Ancilotti's Book 1.0

Esercizi piu' o meno risolti

NOTE:

- Liberamente distribuibile e modificabile basta che riportiate interamente queste note…
- Riguardo a Internet se avete aggiornamenti e correzioni, mandatele a paoloarrigoni@tin.it saranno messe a disposizione di tutti su: www.paoloarrigoni.it. Vi chiedo di non metterlo in Internet, ma di fare riferimento al sito di cui prima. Almeno avro' la soddisfazione di vederne il counter arrivare a dieci.. :>
- In parte tratto dai compiti e dalle soluzioni Unix dei prof Lipari e Ancilotti pubblicate su http://sisop.sssup.it/
- Prima del 13 Gennaio 2000 non esistono i compiti di Unix
- Ver 1.0 a cura di

  Daniele Benedetti
  Fabio Giuliani
  Paolo Arrigoni

Sommario

Esame del 3/2/94

In un sistema strutturato secondo il modello a scambio di messaggi sia definito il seguente tipo di risorse (si supponga noto il tipo *T*):

type *tipo_risorsa* = **class;**

```
    var R :  T;

    procedure entry A(x:T);
        begin
        R:=x
        end;

    procedure entry B(var x:T);
        begin
        x:=R
        end;

begin end;
```

Definire un processo servitore, locale al quale viene dichiarata la variabile *risorsa* del precedente tipo astratto. Tale processo fornisce il servizio relativo alla esecuzione dell'operazione *A* o dell'operazione *B* sulla variabile *risorsa,* su richiesta proveniente da un qualunque processo cliente.

1) Realizzare il processo servitore e specificare quali azioni deve eseguire un cliente per richiedere il servizio *A* o il servizio *B*. Nel realizzare il servitore non specificare nessuna priorita` fra le varie richieste di servizio. Si utilizzino le seguenti primitive di comunicazione sincronizzate e asimmetriche (eventualmente usando comandi con guardie):

send (*m*) to *proc.p*
proc := **receive**(*m*) **from** *p*

dove *p* e' una porta dello stesso tipo della variabile *m* e *proc* e` una variabile del tipo predefinito *process*.

2) Supposto di avere a disposizione la primitiva:

test *p*

che verifica se sulla porta *p* vi sono messaggi in arrivo e restituisce il valore *true* se vi e` almeno un messaggio e il valore *false* in caso contrario, riprogrammare il servitore realizzando una gestione delle richieste che privilegia le richieste di *A* rispetto a quelle di *B*.

OPZIONALE

3) Senza riscrivere il programma indicare brevemente, a parole, come il quesito 2) si sarebbe potuto risolvere in assenza della primitiva *test*.

SOLUZIONE

Punto a)

```
Process server
{
        risorsa r;
        port { signal B, T A } ;
        T mess ;
        signal ok ;
        process p ;

        do

                □ p=receive(mess) from A →          { r.A(mess) }
                □ p=receive(ok) from B →    {
                                                        r.B(&mess);
                                                        send(mess) to p.in;
                                                    }
        od
}

// lato client

process client
{
        port{T in};
        T mess ;
        signal ok ;
        …..
        send(mess) to server.A;
        …..
        send(ok) to server.B;
        receive(mess) from in;
        ….
}
```

punto b)

```
//versione con comandi con guardie

Process server
{
        risorsa r;
        port { signal B, T A } ;
        T mess ;
        signal ok ;
        process p ;

        do
```

```
                □ p=receive(mess) from A →          { r.A(mess) }
                □ (!test(A)) , p=receive(ok) from B →
                                                    {
                                                        r.B(&mess);
                                                        send(mess) to p.in;
                                                    }
        od
}

// versione senza comandi con guardie

process server
{
        risorsa r;
        port {signal B, T A} ;
        T mess ;
        signal ok ;
        process p ;
        while (1)
        {
                if(test(A))
                {
                        p=receive(mess) from A;
                        R.A(mess);
                }
                else
                {
                        if(test(B))
                        {
                                p=receive(ok) from B;
                                r.B(&mess);
                                send(mess) to p.in;
                        }
                }
        }
}
```

Esame del 11/04/96

In un sistema che utilizza i monitor devono essere realizzati due buffer di messaggi tutti dello stesso tipo predefinito *message*: il buffer *INPUT* e il buffer *OUTPUT* che possono contenere al massimo *BI* e *BO* messaggi rispettivamente. In *INPUT* inserisce messaggi il processo *I* e da *OUTPUT* preleva messaggi il processo *O*. Vi sono poi N processi *P1*, *P2*,....*PN*, ciascuno dei quali preleva un messaggio da *INPUT* lo modifica e inserisce il risultato in *OUTPUT*.
Risolvere il problema in modo tale che in *OUTPUT* vengano inseriti i messaggi (modificati) nello stesso ordine con cui sono stati prelevati da *INPUT*, cercando di fornire una soluzione che garantisce il massimo livello di concorrenza fra i processi *Pi* (in particolare una operazione di prelievo da INPUT da parte di un processo *Pi* deve poter essere eseguita concorrentemente con una di inserimento in *OUTPUT* da parte di *Pj*).

```
Monitor Input
{
        mess coda[BI];
        int cont, primo, ultimo; (0)
        int est[N];
        int cont2, primo2, ultimo2 ; (0)
        condition nonpiena, nonvuota, aspetta[N];

Public:
        void inserisci (mess m)
        {
                if(cont==BI)
                        nonpiena.wait();
                coda[ultimo]=m;
                ultimo=(ultimo+1)%BI;
                cont++;
                nonvuota.signal();
        }

        void estrai (mess& m, int p)
        {
                if(cont==0)
                        nonvuota.wait() ;
                m=coda[primo] ;
                primo=(primo+1)%BI;
                cont- -;
                nonpiena.signal();
                est[primo2]=p;
                ultimo2=(ultimo2+1)%N ;
          cont2++;
        }

        void verifica(int p)
        {
                if(est[primo2] != p)
                        aspetta[p].wait() ;
```

```
                primo2=(primo2+1)%N;
                cont2- -;
        }

        void sveglia()
        {
                if(cont2>0)
                        aspetta[est[primo2]].signal();
        }
}

Monitor Output
{
        mess coda[BO];
        int cont, primo, ultimo;(0)
        condition nonpiena, nonvuota;

Public:
        void immetti (mess m)
        {
          if(cont==BO)
                        nonpiena.wait();
                coda[ultimo]=m;
                ultimo=(ultimo+1)%BI;
                cont++;
                nonvuota.signal();
        }

        void preleva(mess& m)
        {
                if(cont==0)
                        nonvuota.wait() ;
                m=coda[primo] ;
                primo=(primo+1)%BO;
                cont- -;
                nonpiena.signal();
        {
}
```

//nota: esempio di esecuzione da parte dei processi Pi

```
Input in; Output out;
…….
In.estrai(m);
//modifico m;
In.verifica(indice);
Out.immetti(m);
In.sveglia();
```

Esame del 6/6/96

Simulare un sistema composto da *N* processi *passeggeri* e da un processo *auto* in cui i *passeggeri* si mettono periodicamente in coda per montare sull'*auto*. L'*auto* può contenere *P passeggari* (*P*<*N*) e non si muove fino a quando non è piena. Una volta piena l'*auto* fa un giro al termine del quale i passeggeri scendono, ne vengono caricati altri P e così via.

a) definire le azioni che i processi *passeggeri* e il processo *auto* devono compiere e realizzare un monitor che le sincronizza.

b) Risolvere il problema nel caso in cui le auto siano in numero *A* (*A*>1) con il vincolo che non siano consentiti sorpassi, cioè ogni auto ancora parte solo quando è piena, le fasi di carico passeggeri e relativo scarico devono avvenire una alla volta e le varie auto devono arrivare nello stesso ordine in cui sono partite.

SOLUZIONE
Punto a)

```
Monitor taxi
{
        int cont; (0)
        condition vuota, piena, fine;

Public:
        void monta()
        {
                if (cont==P)
                        vuota.wait();
                cont++;
                if(cont==P)
                        piena.signal();
                fine.wait();
        }

        void corsa()
        {
                piena.wait();
                for(int i=0, i<P, i++)
                        fine.signal();
                cont=0;
                for(i=0, i<P, i++)
                        vuota.signal();

        }
}
```

//Nota: I passeggeri in coda possono essere anche più di P , ma sarebbe inutile svegliarne di più se tanto poi non entrano nel taxi.

punto b)

//stavolta il P-esimo passeggero non sveglia la corsa ma la funzione carica che //permette al primo taxi della coda di partire. La funzione corsa si limita a svegliare i //passeggeri montati sul taxi e quelli in coda per rimontare.

```
Monitor taxi
{
        int cont; (0)
        int codat; (0)
        condition vuota, piena, fine, stam;

Public:
        void monta()
        {
                if (cont==P)
                        vuota.wait();
                cont++;
                if(cont==P)
                        piena.signal();
                fine.wait();
        }

        void carica()
        {
                codat++;
                if(codat>1)
                        stam.wait();
                piena.wait();
                codat- -;
                stam.signal();
        }

        void corsa()
        {
                for(int i=0, i<P, i++)
                        {
                                cont- -;
                                fine.signal();
                        }
                for(i=0, i<P, i++)
                        vuota.signa();
        }
```

??????????????

Esame del 15/09/97

Un meccanismo di comunicazione tra processi prevede lo scambio di informazioni del seguente tipo:

type *mes* = **array** [1..5] **of** *T*

dove *T* rappresenta un generico tipo. Un processo *R* riceve messaggi del precedente tipo *mes* tramite una mailbox di dimensione *N* utilizzando la procedura:

procedure entry *receive* (**var** *y*:*mes*);

Vi sono poi 5 processi mittenti *P1*, *P2*, ..., *P5*, ciascuno dei quali invia una informazione di tipo *T* tramite la procedura:

procedure entry *sendinf* (*x*:*T*; *p*:*proc*);

dove p denota il nome del mittente (con: **type** *proc* = 1..5.)
Ogni processo *Pi* (*i*=1...5) invia, indipendentemente dagli altri, le proprie informazioni nella mailbox dove i messaggi vengono assemblati e quindi ricevuti da *R* (il j_{esimo} messaggio ricevuto da *R* sarà costituito dalle j_{esime} informazioni che ciascuno dei 5 mittenti avrà rispettivamente inviato.

a) utilizzando il meccanismo semaforico, realizzare la mailbox e le due procedure entry precedentemente specificate.

b) modificare la precedente soluzione in modo tale da consentire, oltre che la ricezione dei messaggi completi da parte di *R*, anche la ricezione delle singole informazioni inviate dai processi Pi da parte di cinque processi *Qi* (i=1,2,..5). In particolare, ogni informazione inviata da *Pi* deve essere ricevuta da *Qi*, per esempio tramite la procedura:

procedure entry *receiveinf* (**var** *x*:*T*; *p*:*proc*);

In questo caso, una informazione di tipo T (inviata da qualche processo *Pi*) deve rimanere nella mailbox fino a quando la stessa non sia stata ricevuta da *Qi* e anche da *R* (che la riceve come parte di un messaggio contenente anche le corrispondenti informazioni inviate dagli altri mittenti.

SOLUZIONE
Punto a)

```
Class server
{
        semaforo mutex; (1)
        mess buf[N];  //ogni mess è composto da 5 dati T, uno per ogni processo
        int riempi[N]; //di ogni mess del buffer fornisce lo stato di riempimento

        int primo ; (0) // è unico per ogni processo
        int ultimo[5];(0)        //ogni processo è indipendente dagli altri ed ha un un suo..
        int con[5];(0)  //…..puntatore quindi anche un suo contatore
```

```
        semaforo attesa;(0)
        int cont_att; (0)
        semaforo ricezione ; (0)
        bool blocric ; (false)

Public:

        Void sendinf(T x, proc p)
        {
                mess appoggio;
                mutex.wait();
                if(cont[p]==N)
                        {
                                cont_att++;
                                mutex.signal();
                                attesa.wait();
                                cont_att- -;
                        }
                appoggio=buff[ultimo[p]];    //uso appoggio perché devo modificare..
                appoggio[p]=x;                          //…solo una parte di mess
                buff[ultimo[p]]=appoggio;
                ultimo[p]=(ultimo[p]+1)%N;
                cont[p]++;
                riempi[ultimo[p]]++;
                if(cont_att>0)
                        attesa.signal();
                else
                        if(riempi[primo]==5)
                                {
                                        if(blocric)
                                                ricezione.signal();
                                        else
                                                mutex.signal();
                                }
        }

        void receive (mess & y)
        {
                mutex.wait();
                if(riempi[primo] < 5)
                        {
                                blocric=true;
                                mutex.signal();
                                ricezione.wait();
                                blocric=false;
                        }
                y=buff[primo];
                primo=(primo+1)%N;
                for(int i=0, i<5, i++)
                        cont[I]- -;
```

```
                if(cont_att>0)
                        attesa.signal();
                else
                        mutex.signal();
        }
}
```

Esame del 25/09/97

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

SOLUZIONE

Punto a)

```
Enum sex{m,f,ind}

Class gabinetto
{
        semaforo mutex; (1)
        int cont; (0)
        sex stato;(ind)
        semaforo attesa[2];
        cont bloc[2];

Public:

        void entra(sex s)
        {
                mutex.wait();
                if ((stato!=s)||(cont==5))
                        {
                                bloc[s]++;
                                mutex.signal();
                                attesa[s].wait();
                                bloc[s]- -;
                        }
                stato=s;
                cont++;
                if ((cont<5)&&(bloc[s]>0))
                        attesa[s].signal();
                else
                        mutex.signal();
        }

        void esci()             //non importa passare  il sesso: è quello dello stato attuale
        {
                mutex.wait();
```

```
                cont- - ;
                if (cont>0)
                        {
                                if(bloc[stato]>0)
                                        attesa[stato].signal();
                                else
                                        mutex.signal();
                        }
                else
                        {
                                if(bloc[(stato+1)%2]>0)
                                        attesa[(stato+1)%2].signal();
                                else
                                        {
                                                stato=ind;
                                                mutex.signal();
                                        }
                        }
        }
}
```

punto b)

// supponiamo L il limite di ingressi consecutivi di persone dello stesso sesso,… //….superato il quale entra in funzione il meccanismo anti-starvation

```
Enum sex{m,f,ind}

Class gabinetto
{
        semaforo mutex; (1)
        int cont; (0)
        int star; (0)
        sex stato;(ind)
        semaforo attesa[2];
        cont bloc[2];

Public:

        void entra(sex s)
        {
                mutex.wait();
                if ((stato!=s)||(cont==5))
                        {
                                bloc[s]++;
                                mutex.signal();
                                attesa[s].wait();
                                bloc[s]- -;
                        }
                stato=s;
                cont++;
```

```
            star+=1;
            if ((cont<5)&&(bloc[s]>0))
                  attesa[s].signal();
            else
                  mutex.signal();
      }

      void esci()
      {
            mutex.wait();
            cont- - ;
            if (cont>0)
                  {
                        if ((bloc[stato]>0) && !(star<L  &&  bloc[(stato+1)%2]))
                              attesa[stato].signal();
                        else
                              mutex.signal();
                  }
            else
                  {
                        if(bloc[(stato+1)%2]>0)
                              {
                                    star=0;
                                    attesa[(stato+1)%2].signal();
                              }
                        else
                              {
                                    stato=ind;
                                    mutex.signal();
                              }
                  }
      }
}
```

//Nota:

la funzione esci si comporta nel seguente modo:

- fa uscire una persona dal bagno
- se ci sono persone del suo stesso sesso in coda e non è vero che contemporaneamente ci sono anche persone dell'altro sesso con limite antistarvation superato, fa entrare una pesona del suo sesso.
- Altrimenti fa entrare (se c'è) una persona dell'altro sesso, azzerando il contatore antistarvation.

Per come è strutturata la "entra" ovvero tramite catena di S.Antonio, il limte massimo di ingressi consecutivi di un certo sesso (con presenza di elementi dell'altro sesso in coda) è L+4

Esame del 15/04/98

Si risolva il seguente problema di allocazione di risorse in un sistema strutturato secondo il modello a memoria comune:
nel sistema sono presenti *P* processi che competono tra loro per l'uso di *N* risorse equivalenti (per esempio pagine di memoria o blocchi fisici di un disco). Ogni processo può richiedere (o rilasciare) al gestore una qualunque quantità *x* di risorse contemporaneamente ($x \leq N$) tramite la procedura *Request* (o *Release*). Il processo richiedente si blocca se le *x* risorse non sono disponibili. Quando sono disponibili, la procedura *Request* restituisce al processo il puntatore *y* (passato come parametro) ad un vettore di indici di risorse contenente gli indici delle risorse allocate. Un processo che rilascia delle risorse dovrà passare alla procedura *Release* il puntatore al vettore contenente gli indici delle risorse rilasciate.

a) realizzare il gestore implementando le procedure in modo tale da privilegiare, fra le varie richieste, quella che richiede il numero più basso di risorse (strategia shortest-job-first);

b) riscrivere la soluzione adottando la strategia FIFO fra le varie richieste (ciò significa che una richiesta può essere ritardata anche quando le risorse richieste sono disponibili.

Si adotti il meccanismo semaforico (non è noto il criterio con cui è gestita la coda semaforica). Se servono, si possono utilizzare i seguenti tipi: considerandoli predefiniti

type *indice_ris* = 1 . . *N*;
numero_ris= 0 . . *N*;
indice_proc = 1 . . *P*;
numero_proc = 0 . . *P*;
vettore_indici = **array** [*indice_ris*] **of** *indice_ris* ;
-- nei primi *x* elementi di una variabile di questo tipo un processo che richieda *x* risorse si aspetta di ricevere dal gestore gli indici delle *x* risorse allocate. Analogamente, nei primi *x* elementi di una variabile di questo tipo sono contenuti gli indici delle *x* risorse che un processo rilascia al gestore.
pun = *vettore_indici*;
-- tipo del parametro passato alle procedure *Request* e *Release* per indicare il puntatore al vettore degli indici delle risorse richieste o rilasciate.

SOLUZIONE
Punto a)

```
Class gestore
{
        semaforo mutex; (1);
        bool ris[N]; (true)
        int nris; (N)
        semaforo priv[P] ;
        int bloc;(0)
        int richieste[P]; (0)
        int minrich; (N)
        int minproc;
```

```
Public:
        void Request(int proc, int x, int* y)
        {
                mutex.wait() ;
                if(n>nris)
                        {
                                if(x<richmin)
                                        richmin=x;
                                richieste[proc]=x;
                                bloc++;
                                mutex.signal();
                                priv[proc].wait();
                                bloc-- ;
                                richieste[proc]=0 ;
                                minrich=N ;
                                for(int i=0; i<P; i++)
                                        if(richieste[i]!=0 && richieste[i]<minrich)
                                {
                                minrich=richieste[i];
                                minproc=i;
                                }
                        }
                nris-= x;
                int j=0;
                for(i=0; i<x; i++)
                        {
                                while(!ris[j])
                                        j++;
                                y[i]=j;
                        }
                if (bloc>0 && minrich<=nris)
                        priv[minproc].signal() ;
                else
                        mutex.signal() ;
        }

        void Release (int x, int* y)
        {
           mutex.wait();
           nris+= x;
           for(int i=0; i<x, i++)
                ris[y[i]]=true;
           if(bloc>0 && minrich<=nris)
                priv[minproc].signal();
           else
                mutex.signal() ;
        }
}
```

punto b)

```
struct req
        {
                int processo ;
                int quante ;
        }

class gestore
{
        semaforo mutex; (1);
        bool ris[N]; (true)
        int nris; (N)
        semaforo priv[P] ;
        int cont;(0)
        req coda [P];
        int primo, ultimo; (0)

Public:
        Void Request(int proc, int x, int* y)
        {
                mutex.wait() ;
                if( n>nris  ||  cont>0)
                        {
                                cont++;
                                coda[ultimo].processo=proc;
                                coda[ultimo].quante=x;
                                ultimo=(ultimo+1)%P;
                                mutex.signal();
                                priv[proc].wait;
                                cont- - ;
                                primo=(primo+1)%P ;
                        }
                nris -= x;
                int j=0;
                for(i=0; i<x; i++)
                        {
                                while(!ris[j])
                                        j++;
                                y[i]=j;
                        }
                if(cont>0  &&  coda[primo].quante<=nris)
                        priv[coda[primo].processo].signal;
                else
                        mutex.signal();
        }

        void Release(int x, int*y)
        {
                mutex.wait();
        nris+= x;
        for(int i=0; i<x, i++)
                ris[y[i]]=true;
```

```
                if(cont>0 && coda[primo].quante<=nris)
                        priv[coda[primo].processo].signal;
                else
                        mutex.signal();
        }
}
```

Esame del 14-09-98

Un'istanza *M* del seguente monitor e' condivisa tra un insieme di processi:

```
type T = monitor
   var a,b : integer;
       c1,c2 : condition;

   procedure entry P1 (x in integer)
      begin
        if a>b then c1.wait;
        a :=x;
        SA;
        c2.signal;
        SB;
      end;

   procedure entry P2 (x in integer, y out integer)
      begin
        if b<0 then c2.wait;
        b:=x;
        y:=E;
        SC;
        c1.signal;
      end;

   begin  SD  end;
```

dove gli statement *SA*,*SB*, *SC, SD* e l'espressione *E* sono lasciati non specificati.

a) scrivere il codice di un processo server *M* che simuli il comportamento del precedente monitor nei confronti dei processi clienti.
   1 - Effettuare tale simulazione usando le seguenti primitive asincrone ed asimmetriche:

send (*m*) **to** *proc.port*
proc:= **receive**(*m*) **from** *port*

   2 - riscrivere la soluzione con le precedenti primitive ma considerandole sincronizzate;
   3 - riscrivere la soluzione con le chiamate di procedure remote.

   Indicare anche le operazioni che i processi devono eseguire per simulare le chiamate delle due procedure *P1* e *P2* nei tre casi.

b) Indicare quale semantica della *signal* e' stata simulata dalle tre soluzioni precedenti.

SOLUZIONE

punto a) primitive asincrone ed asimmetriche

// prima versione…..è la più semplice ma non sembra rispettare alcuna semantica

```
process Server_M
{
        port {int P1, int P2}
        int a,b;
        process proc;
        int x,y;
        SD;
        do
                ■ (a<=b) , proc=receive(x) from P1 → {
                                                        a=x;
                                                        SA;
                                                        SB;
                                                }

                ■ (b>=0) , proc=receive(x) from P2 → {
                                                        b=x;
                                                        y=E;
                                                        SC;
                                                 }
}
```

// seconda versione……consideriamo l'esistenza della primitiva test(porta)

```
process Server_M
{
        port {int P1, int P2}
        int a,b;
        process proc;
        int x,y;
        int t1,t2 ; (0)
        SD;

        do
                ■ (a<=b) &&  !(t2==1 ), proc=receive(x) from P1 → {
                                                        t1=0;
                                                        a=x;
                                                        SA;
                                                        if(test(P2)
                                                                t2=1;
                                                        SB;
                                                        if  (b<0)  t2 = 0;

                                                }

                ■ (b>=0) && !(t1==1 ) , proc=receive(x) from P2 → {
                                                        t2=0;
                                                        b=x;
                                                        y=E;
                                                        SC ;
```

```
                    if (test(P1))
                        t1=1
                }
}
```

Esame del 4/2/99

In un sistema, *N* processi ciclici operano sulla risorsa condivisa *deposito* contenente, inizialmente, *M* dati di tipo *T*. Durante ogni periodo ogni processo preleva dal *deposito* uno degli *M* dati e lo consuma secondo il seguente schema:

```
process Pi (1≤i≤N)
   begin
     while (true) do
       begin
          .................
          preleva (var x:T);
          < consuma x >;
          .................
       end;
   end;
```

Quando un processo, invocando la procedure *preleva*, trova che il *deposito* è ormai vuoto, sveglia il processo *fornitore* attendendo che quest'ultimo riempia di nuovo *deposito* e solo dopo che il *deposito* è di nuovo pieno, il processo può ripartire. Il comportamento del processo *fornitore*, anch'esso ciclico, è illustrato dal seguente schema:

```
process fornitore
   var D : array [1 .. M] of T;
   begin
     while (true) do
       begin
          .................
          attesa;
          <prepara gli M dati nella variabile locale D>
          deposita;
          .................
       end;
   end;
```

Con la procedura *attesa*, il *fornitore* attende di essere attivato e con la procedura *deposita* trasferisce gli *M* dati preparati nel *deposito*.

a) utilizzando il meccanismo semaforico, realizzare la risorsa astratta *deposito* con le tre procedure *preleva*, *attesa* e *deposita*, che sincronizzano i processi coinvolti come sopra specificato (il *fornitore* deve essere attivato solo quando il *deposito* è vuoto).

b) ripetere la soluzione ipotizzando che vi siano, non uno ma $F > 1$ *fornitori* ciascuno dei quali però in ogni ciclo è in grado di produrre un solo dato (ricordando che un processo che durante l'esecuzione di *preleva* trova il *deposito* vuoto può ripartire solo dopo che il *deposito* è stato completamente riempito):

```
process fornitorej  (1≤j≤F)
   var D :T;
```

```
begin
  while (true) do
    begin
      .....
      attesa;
      <prepara il dato nella variabile locale D>
      deposita;
      .....
    end;
end;
```

c) ripetere la soluzione b) utilizzando il costrutto monitor.

SOLUZIONE
Punto a)

```
Class deposito
{
        T dep[M];
        semaforo mutex; (0)
        int lib; (0)
        semaforo attendi; (0)
        semaforo attmit; (0)
        semaforo att_for; (0)
        int cont_for; (0)
        bool rif; (false)

Public:
        void preleva(T& x)
        {
                semaforo mutex() ;
                if(lib==M)
                {
                        attendi.signal();
                        attmit.wait();
                        lib=0;
                }
                x=dep[lib];
                lib++;
                mutex.signal();
        }

        void attesa()
        {
                attendi.wait();
        }

        void deposita(T* vett)          //lavora in mutua in quanto unico attivo
        {
                for(int i=0 ; i<M ; i++)
```

```
                dep[i]=vett[i];
            attmit.wait();
        }
}
```

Punto b)

```
Class deposito
{
        T dep[M];
        semaforo mutex;(1)
        int lib; (0)
        intcontdep;(0)
        int cont; (0)
        semaforo attendi; (0)
        semaforo attmit; (0)

Public:
        void preleva(T& x)
        {
                mutex.wait() ;
                if(rif)
                        {
                                cont_for++ ;
                                mutex.signal() ;
                                att_for.wait() ;
                                cont_for- - ;
                        }
                if(lib==M)
                        {
                                cont=1;
                                rif=true;
                                contdep=0;
                                attendi.signal();
                                mutex.signal();
                                attmit.wait();
                                lib=0;
                        }
                x=dep[lib];
                lib++;
                if(cont_for>0)
                        att_for.signal();
                else
                        mutex.signal();
        }

        void attesa()
        {
                attendi.wait();
                mutex.wait();
```

```
            if(cont<M)
                {
                    cont++;
                    attendi.signal();
                }
            mutex.signal();
        }

        void deposita(T dato)
        {
            mutex.wait() ;
            dep[contdep]=dato ;
            contdep++;
            rif=false;
            if(contdep==M)
                attmit.signa();
            else
                mutex.signal();
        }
}
```

//nota:

- rif diventa true quando entriamo in una fase di rifornimento
- nella fase di rifornimento i processi si accodano su att_rif
- quando l'M+1-esimo processo va a prelevare scatta il rifornimento
- a fine rifornimento viene svegliato quel M-esimo processo fermo
- dopo vengono svegliati col testimone quelli fermi in attesa di fine rifornimento

Esame del 18/02/99

Ciascuno degli *N* processi *P1*, *P2*, ..., *PN* è costituito da due parti eseguite in sequenza: la prima parte, denotata come *PARTEAi* ($1 \le i \le N$), è seguita da una seconda parte che può essere, in alternativa, o lo statement *PARTEBi* o lo statement *PARTECi*. La scelta tra le due alternativa da eseguire nella seconda parte dipende dai risultati delle prime parti di tutti i processi. In particolare i processi dovranno concordare se eseguire **tutti** la *PARTEB* o **tutti** la *PARTEC*. Per questo motivo, alla fine della prima parte ciascun processo invoca la procedura *accordo* (procedura entry di un risorsa condivisa utilizzata per garantire l'accordo su quale seconda parte eseguire):

procedure *accordo* (*i*: *process*, *x*:*esito*, **var** *y*:*esito*)

con **type** *process* =1..*N*;
esito = (*B*,*C*)

Alla procedura il processo chiamante passa il propio nome *i* e, tramite il parametro *x* il suo desiderio ad eseguire la *PARTEB* (*x*=*B*) oppure la *PARTEC* (*x*=*C*) e si aspetta, tramite il parametro *y* , l'indicazione di quale parte dovrà effettivamente eseguire: la *PARTEB* se *y*=*B* oppure la *PARTEC* se *y*=*C*.
La procedura deve garantire che:

- a risposta sia la stessa per tutti i processi;
- la comuna risposta sia *B* **se e solo se** tutti i processi hanno espresso il desiderio di eseguire la *PARTEB* ;
- la comune risposta sia *C* se **almeno uno** dei processi ha espresso il desiderio di eseguire la *PARTEC*;
- ciascun processo, invocando quindi la procedura *accordo*, esprime il proprio desiderio e attende l'esito della consultazione. Tale attesa, qualora tutti esprimano il proprio desiderio come *B*, potrà terminare solo dopo che tutti si sono espressi. Ciò è necessario perchè per rispondere *B* dobbiamo avere la garanzia che tutti abbiano espresso *B* come desiderio. Se però qualcuno esprime il desiderio *C*, allora abbiamo la certezza che *C* dovrà essere la risposta per tutti. Quindi, in questo caso, la procedura restituisce subito *C* al chiamante e, se ci sono processi in attesa dell'esito li deve risvegliare rispondendo *C* , oltre che rispondere *C* ad ogni ulteriore processo chiamante, senza nessuna attesa e qualunque sia il desiderio da lui espresso.

a) utilizzando il meccanismo semaforico, realizzare la struttura dati della risorsa condivisa e la procedura *accordo*. Supporre, come indicato nello schema seguente, che ogni processo esegua la procedura *accordo* una sola volta nella sua vita:

```
process Pi  (1≤i≤N)
var desiderio, risultato: esito;
begin
        PARTEAi;
        accordo (i, desiderio, risultato);
        if risultato =B then PARTEBi
        else PARTECi;
end;
```

b) riscrivere la soluzione ipotizzando che i processi siano ciclici:

process *Pi* (1≤*i*≤*N*)

```
var desiderio, risultato: esito;
begin
        while (true) do
        begin
                PARTEAi;
                accordo (i, desiderio, risultato);
                if risultato =B then PARTEBi
                else PARTECi;
        end;
end;
```

Punto a)

```
Class votazione
{
        semaforo mutex; (1)
        esito es; (B)
        int cont; (0)
        semaforo attesa; (0)
        int bloc; (0)

Public:
        void accordo (proc i, esito x, esito& y)
        {
                mutex.wait();
                cont++;
                if ((x==c) && (es==B))
                        es=c;
                if(cont<N && es==B)
                        {
                                bloc++;
                                mutex.signal();
                                attesa.wait();
                                bloc- -;
                        }
                y=es;
                if(bloc>0)
                        attesa.signal();
                else
                        mutex.signal();
        }
}
```

punto 2)

```
Class votazione
{
        semaforo mutex; (1)
        esito es; (B)
        int cont; (0)
```

```
        semaforo attesa; (0)
        int bloc; (0)
        semaforo priv[N];(1)

Public:
        void accordo (proc i, esito x, esito& y)
        {
                priv[i].wait() ;
                mutex.wait();
                cont++;
                if ((x==c) && (es==B))
                        es=c;
                if(cont<N && es==B)

                        {
                                bloc++;
                                mutex.signal();
                                attesa.wait();
                                bloc- -;
                        }
                y=es;
                if(bloc>0)
                        attesa.signal();
                else
                        {
                                if(cont==N)
                                        {
                                                cont=0;
                                                es=B;
                                                for(int i=0, i<N, i++)
                                                        priv[i].signal();
                                        }
                                else
                                        mutex.signal();
                        }
        }
}
```

Esame del 7/4/99

In un sistema in cui le interazioni tra processi sono realizzate tramite il meccanismo delle chiamate di procedure remote e` presente un processo *server* che mette a disposizione dei propri clienti due entry con gli stessi parametri:

process *server*
entry *E1* (*x*: **in** *T1*; *y*: **out** *T2*);
E2 (*x*: **in** *T1*; *y*: **out** *T2*);
var .....
begin
.....
end;

I processi clienti sono esattamente *N*: *P1*, *P2*, ..., *PN*.

a) Utilizzando il meccanismo dei monitor, simulare il precedente meccanismo delle chiamate remote con riferimento al precedente esempio. Cioe` simulare quanto necessario per fornire il supporto alle due entry *E1*, ed *E2* oltre che gli statement **accept** delle due entry eseguiti dal *server*:

accept *E1* (*x*: **in** *T1*; *y*: **out** *T2*) **do** *S1* **end**;

e

accept *E2* (*x*: **in** *T1*; *y*: **out** *T2*) **do** *S2* **end**;

e gli statement di entry call eseguiti da uno qualunque dei processi clienti:

call *server.E1*(*a*,*b*);

e

call *server.E2*(*a*,*b*);

b) Potrebbe essere semplificata (ed eventualmente indicare come) la soluzione nel caso in cui le entry prevedessero solo parametri di modo **in** oppure nel caso in cui non fossero presenti parametri e il corpo dello statement **accept** fosse nullo?

SOLUZIONE
Punto a)

```
struct mes
{
     int proc;   //nome del processo
        T1 inf;
};

Monitor coda_entry
{
```

```
    mes coda[N];        //visto che le call sono bloccanti, se ho N processi clienti posso      // avere
                        max N chiamate; dimensionando la coda con N elementi //non devo più
                        occuparmi della coda piena (sparisce una              // condition dalla mailbox)
        int primo, ultimo, cont;(0)
        condition NonVuota;

        void ins (mes X)
        {
                //come detto, il controllo "coda piena" non serve
                coda[ultimo]=x;
                cont++;
                NonVuota.signal();
        }

        mes estrai()
        {
                mes x;
                if (cont==0)
                NonVuota.wait();
                x=coda[primo];
                primo=(primo+1)%N;
                cont--;
                return x;
        }

};
```

Ora ci serve un altro monitor per la sincronizzazione server/client per il passaggio dei parametri di tipo out:

```
Monitor aspetta
{
        T2 buffer;

//mailbox lunga 1 in cui solo il cliente può bloccarsi: il server inserisce
//alla fine di una accept, quando il cliente è sicuramente bloccato
//anche qui serve una sola condition

        condition attesa;
        bool risultato_pronto(false);

//il bool si potrebbe eliminare se si usasse la queue()

        T2 aspetta()
        {
                if(!risultato_pronto)
                attesa.wait();
```

```
                risultato_pronto = false;
                return buffer;
        }

        void rispondi(T2 x)
        {
                buffer = x;
                risultato_pronto = true;
                attesa.signal();
        }
};
```

Ora dobbiamo dichiarare le strutture dati che ci servono:

```
coda_entry coda1,coda2;
aspetta risultato[N];   //un monitor per ogni cliente
```

Implementiamo le accept e le chiamate (solo per la entry E1, per E2 è uguale):

```
accept E1 (T1 in x, T2 out y) {S1;}
                ↓
process server
{       int cliente;
        mes messaggio;
        T1 x; T2 y;
        ......
        ......
        messaggio = coda1.estrai();
        cliente = messaggio.proc;
        x = messaggio.inf;
        S1; //ci sarà sicuramente un assegnamento a y del valore di ritorno
        risultato[cliente].rispondi(y);
        ......
        ......
}

server.E1 (a,b)
        ↓
process clientei
{
        mes messaggio;
        T1 a; T2 b;
        .....
        .....
        messaggio.proc = PE; //processo in esecuzione
        messaggio.inf = a;
        coda1.ins(messaggio);
        b=risultato[PE].aspetta();
        ......
        ......
```

}

Punto b)

Esame del 03-06-99

Realizzare un canale **simmetrico** e **sincronizzato** da cui un ricevente *R* puo` estrarre messaggi a lui inviati (messaggi di un generico tipo *T*).
Il mittente viene realizzato scrivendo tre copie (programmate in modo diverso) dello stesso processo. Ogni processo della terna invia la sua copia del messaggio e la copia che verra` ricevuta da *R* sara` scelta mediante una tecnica di voting (quella corrispondente alle tre copie inviate dai tre processi se queste sono identiche, se due sono identiche ed una diversa la copia diversa viene scartata come erronea, se le tre copie sono tutte diverse la trasmissione fallisce).

Le procedure da realizzare sono:

procedure *send* (*messaggio*:*T*; *p*: *process*; **var** *ris*: *risultato*)

con **type** *process* = 1..3; e **type** *risultato* = (*ok*, *errore*)

e dove il parametro *p* identifica il processo mittente fra i tre della terna e il parametro *ris* restituisce *ok* se il messaggio inviato era corretto ed *errore* se considerato sbagliato.

procedure *receive* (**var** *messaggio*: *T*; **var** *ris*: *risultato*)

dove *ris* restituisce *ok* se la trasmissione e` andata a buon fine ed *errore* se fallita.

a) utilizzare il meccanisnmo semaforico.

b) indicare come dovrebbe essere modificata la soluzione (se lo deve essere) se il canale fosse asincrono (in pratica una mailbox unitaria).

SOLUZIONE DANIELE

Parte a)

```
enum risultato {ok, errore}

class canale
{
        T buf[3];
        int cont; (0)
        T  m;
        bool fine; (true)               //se false, la trasmissione è fallita
        semaforo mutex; (1)
        semaforo attesaM; (0)
        semaforo attesaR; (0)

Public:
        void send (T mess, proc p, risultato& ris)
        {
                mutex.wait();
                buf[p-1]=mess;
                cont++;
```

```
            if(cont==3)
            attesaR.signal();
        mutex.signal();
        attesaM.wait();
        cont- -;
        if ((mess==m)&&(fine))
            ris=ok;
        else
            ris=errato;
        if(cont!=0)
            attesaM.signal();
        else
            mutex.signal();
    }

    void receive (T& mess, risultato& ris)
    {
        attesaR.wait()
        mutex.wait();
        if((buf[0]==buf[1]) || (buf[1]==buf[2]) || (buf[0]==buf[2]))
            {
                mess = (buf[0]==buf[1])? buf[0] : buf[2];
                m=mess;
                ris=ok;
            }
        else
            {
                ris=errato;
                fine=false;
            }
        attesaM.signal();
    }


// note:
```

- ciascun processo mittente invia un messaggio e si blocca
- i processi sono considerati NON ciclici. In caso contrario ci dovrebbero essere ulteriori controlli.
- il terzo mittente sveglia il ricevente il quale si era bloccato subito in partenza su di un semaforo di sincronizzazione.
- il ricevente sceglie il messaggio e sveglia un mittente passandogli il testimone (sa che tutti i mittenti sono bloccati)
- tutti i mittenti vengono svegliati tramite catena di S.Antonio.

parte b)

Se vogliamo che la send restituisca il risultato questa deve comunque aspettare la ricezione dei messaggi, perciò va mantenuta la soluzione sincronizzata.

SOLUZIONE PAOLO

Parte a)

```
class canale{

    enum stati {ERRORE,OK}
    sem mutex       (1)
    sem ric         (0)
    sem mit   [3]   (0)
    int num_msg     (0)
    T msg[3]
    T msgScelto

    send (T messaggio, int p, stati *risultato){
        mutex.wait
        num_msg++
        msg[num_msg -1]=messaggio
        if(num_msg==3){
            ric.signal
        }
        mutex.signal
        mit[p].wait
        if (msgScelto==messaggio)
            *risultato=OK
        else
            *risultato=ERRORE
    }

    receive(T *messaggio, stati *risultato){
      ric.wait
      mutex.wait
      if(msg[0]== msg[1]||msg[1]==msg[2]||msg[2]==msg[0]){
            *risultato=OK
            *messaggio=(msg[0]==msg[1])?msg[1]: msg[2]
            msgScelto=messaggio
      }else{
            *risultato=ERROR
            msgScelto=NULL
      }

        num_msg=0
        mutex.signal
        mit[0].signal
        mit[1].signal
        mit[2].signal
    }
}
```

Esame del 24/06/99

Utilizzando i semafori, realizzare un meccanismo di comunicazione tra processi con le seguenti caratteristiche:

a) Il meccanismo e` utilizzato da un solo processo mittente *P* che invia messaggi di tipo *T* . Il meccanismo che si vuole realizzare ha una capacita` di bufferizzazione di *N* messaggi.

b) Ogni messaggio inviato da *P* deve essere ricevuto esattamente da tre processi identici di cui esistono *n* (*n*>3) istanze tutte uguali (il messaggio sara` quindi eliminato dal buffer del meccanismo di comunicazione solo dopo che e` stato ricevuto da tre processi).

Ciascun processo ricevente deve sapere, quando riceve un messaggio, se e` il primo, il secondo oppure il terzo a riceverlo, poiche` le operazioni che il ricevente dovra` eseguire sono diverse nei tre casi.

Definendo:

type *n_ricevente* = 1..3;

le procedure di comunicazione da realizzare sono quindi le seguenti:

procedure *send* (*m*:*T*);
procedure *receive* (**var** *m*:*T*; **var** *numero*:*n_ricevente*);

I processi coinvolti sono ciclici:

```
process P
var     m:T;
begin
  while (true) do
    begin
      <produce un messaggio m>;
      send (m);
    end;
end;

process riceventei  [1≤i≤n]
varnr:n_ricevente;
        m:T;
begin
  while (true) do
    begin
      receive (m,nr);
      case nr of
        1: S1;
        2: S2;
        3: S3;
      end;
end;
```

1) realizzare il precedente meccanismo senza stabilire nessuna priorita` tra i processi riceventi.

2) riscrivere la soluzione ipotizzando che ricevente1 abbia maggiore priorita` di ricevente2 e cosi` via. In questo caso si puo` usare la primitiva PE (Processo in Esecuzione) che restituisce l'indice del processo in esecuzione (da 1 a *n* nel caso dei processi riceventi).

SOLUZIONE

Punto a)

```
Class mailbox
{
        semaforo mutex; (1)
        T buf[N];
        int cont; (0)
        int primo, ultimo; (0)
        int ric; (1)
        semaforo nonpieno, nonvuoto; (0)
        bool np; (true)
        int nv; (0)

Public:

        Void send (T m)
        {
                mutex.wait();
                if (cont == N)
                        {
                                np=false;
                                mutex.signal();
                                nonpieno.wait();
                                np=true;
                        }
                buf[ultimo]=m;
                ultimo=(ultimo+1)%N;
                cont++;
                if (nv>0)
                nonvuoto.signal;
                else
                        mutex.signal();
        }


        void receive (T & m, int & NR)
        {
                mutex.wait();
                if (cont ==0)
                        {
                                nv++;
```

```
                        mutex.signal();
                        nonvuoto.wait();
                        nv--;
                }
        m=buf[primo] ;
        NR=ric;
        ric++;
        if(ric<=3)
                {
                        if(nv>0)
                                nonvuoto.signal();
                        else
                                mutex.signal();
        else
                {
                        ric=1;
                        cont--;
                        primo=(primo+1)%N;
                        if(!np)
                                nonpieno.signal();
                        else
                                mutex.signal();
                }
        }
}
```

punto b)

```
Class mailbox
{
        semaforo mutex; (1)
        T buf[N];
        int cont; (0)
        int primo, ultimo; (0)
        int ric; (1)
        semaforo nonpieno; (0)
        semaforo priv[N]; (0)
        bool bloccati [N]; (false)
        bool np; (true)
        int nv; (0)

Public:

        Void send (T m)
        {
                mutex.wait();
                if (cont == N)
                        {
                                np=false;
                                mutex.signal();
```

```
                nonpieno.wait();
                np=true;
        }
    buf[ultimo]=m;
    ultimo=(ultimo+1)%N;
    cont++;
    if (nv>0)
        {
            int i=0;
            while(!bloccati) i++;
            priv[i];
        }

    else
        mutex.signal();
}

void receive (T & m, int & NR)
{
    mutex.wait();
    int x;
    if (cont ==0)
        {
            nv++;
            x=PE();
            bloccati[x]=true;
            mutex.signal();
            priv[x].wait();
            bloccati[x]=false ;
            nv--;
        }
    m=buf[primo] ;
    NR=ric;
    ric++;
    if(ric<=3)
        {
            if(nv>0)
                {
                    int i=0;
                    while(!bloccati) i++;
                    priv[i];
                }

            else
                mutex.signal();
    else
        {
            ric=1;
            cont--;
            primo=(primo+1)%N;
            if(!np)
```

```
                    nonpieno.signal();
                else
                    mutex.signal();
            }
        }
    }
```

Esame del 22/07/99

Utilizzando il meccanismo dei monitor, simulare il seguente sistema:
un contenitore *C* di gettoni puo` contenere al massimo *max* gettoni. Vi sono *N1* persone (simulate mediante processi ciclici) che periodicamente desiderano depositare un certo numero *n* di gettoni nel contenitore (dove *n* puo` essere anche maggiore di *max*) e *N2* persone che periodiocamente desiderano estrarre dal contenitore *m* gettoni (anche *m* puo` essere maggiore di *max*). I due valori *n* ed *m* costituiscono, rispettivamente, il parametro della procedura *deposita* (chiamata dai processi che inseriscono gettoni) e della procerdura *estrai* (chiamata dai processi che estraggono gettoni).
Quando un processo chiede di depositare (estrarre) piu` gettoni di quanto e` possibile depositarne (estrarre), in base anche alla quantita` presente nel contenitore, il processo deposita (estrae) tutto cio` che puo` depositare (estrarre) quindi si blocca in attesa di poter continuare a depositare (estrarre) fino a quando non ha completato il deposito (l'estrazione) desiderato.
Non deve mai accadere che il deposito (l'estrazione) di un processo venga iniziato mentre e` in corso e non ancora completato il deposito (l'estrazione) da parte di un altro processo.

a) realizzare il contenitore *C* con le due procedure *deposita* ed *estrai* utilizzando la semantica signal-and-urgent.

b) cambia la soluzione se viene utilizzata la semantica signal-and-wait? (e se si com e?)

SOLUZIONE
Punto a)

```
Monitor contenitore
{
        int cont; (0)
        condition nv, np;

Public :
        void deposita(int n)
        {
                int residuo=n ;//gettoni ancora da inserire
                int capacità;    //gettoni che posso ancora essere messi

                while(residuo>0)
                {
                        capacità=N-cont;
                        if(capacità>=residuo)
                        {
                                cont=cont+residuo;
                                residuo=0 ;
                        }
                        else
                        {
                                residuo=residuo-capacità ;
                                cont=N;
                        }
```

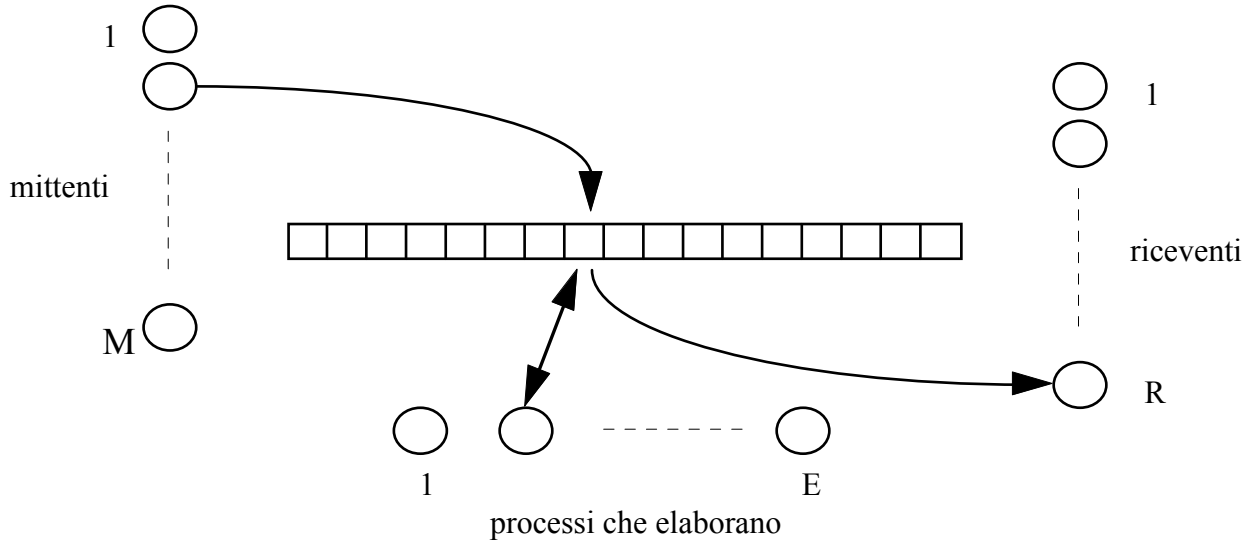
```
                nv.signal();
                if(residuo>0)
                        np.wait();
        }
    }


    void estrai(int n)
    {
        int residuo=n ;         //gettoni ancora da prelevare

        while(residuo>0)
        {
            if(cont>=residuo)
            {
                cont=cont-residuo;
                residuo=0 ;
            }
            else
            {
                residuo=residuo-cont ;
                cont=0;
            }
            np.signal();
            if(residuo>0)
                nv.wait();
        }
    }
}
```

Esame del 9/9/99

Realizzare il seguente meccanismo di comunicazione tra processi:
ciascun processo appartenente ad un insieme di *M* processi mittenti invia messaggi di tipo *T* in una mailbox di dimensione *N*.
Ogni messaggio inviato, una volta inserito nella mailbox, viene elaborato direttamente nella mailbox (senza essere estratto) da uno dei processi appartenenti ad un secondo insieme di *E* processi.
I messaggi, una volta elaborati, vengono ricevuti da uno qualunque dei processi appartenenti ad un terzo insieme di *R* processi



Lo schema che ciascun processo (ciclico) di ogni gruppo esegue è il seguente:

```
Process mittentei (1 ≤ i ≤ M)
var inf:T;
begin
  while true do
        begin
        .............
        send (inf) ;
        .............

        end;

end;
```

```
Process elaborai (1 ≤ i ≤ E)
        var j: indice;
begin
  while true do
        begin
              inizioE (j);
               <elabora l'informazione
                contenuta nel jesimo
                elemento della mailbox>;
              fineE (j);
               end;
        end;
```

```
Process riceventei (1 ≤ i ≤ R)
var inf:T;
begin
  while true do
        begin
        .............
```

```
        receive (inf) ;
        .............
        end;
end;
```

dove il tipo *indice* è definito come sottorange degli interi tra 0 ed n-1 (oppure tra 1 ed n).
Ciascun processo mittente si limita, quindi, ad inviare in ogni ciclo un messaggio nella mailbox mediante la procedura *send*.
Ogni processo che elabora, in ogni ciclo, eseguendo la procedura *inizioE* ottiene, tramite il parametro *j*, l'indice di un elemento della mailbox contenente un messaggio gia` inviato ma che deve essere elaborato. Quindi elabora il messaggio contenuto nel jesimo elemento della mailbox, e quindi, eseguendo la procedura *fineE* dichiara che il jesimo elemento della mailbox contiene un messaggio già elaborato che può quindi essere ricevuto. Infine ciascun processo ricevente si limita ad eseguire durante ogni ciclo la procedura *receive* con cui estrae dalla mailbox, se disponibile, un messaggio già elaborato.

a) realizzare, utilizzando il meccanismo semaforico, la mailbox con le quattro procedure:

   procedure entry *send* (*inf*:*T*);
   procedure entry *receive* (**var** *inf*:*T*);
   procedure entry *inizioE* (**var** *j* : *indice*);
   procedure entry *fineE* (*j* : *indice*);

   tenendo conto che i messaggi devono essere ricevuti nello stesso ordine con cui sono stati inviati e che l'elaborazione dei singoli messaggi da parte dei processi che elaborano, può essere di durata diversa (per esempio fra l'inizio dell'elaborazione di un messaggio e la fine dell'elaborazione dello stesso può essere completamente elaborato un altro messaggio da parte di un altro processo.
   Come è ovvio, un mittente si deve bloccare solo se, eseguendo la *send*, non ci sono elementi vuoti nella mailbox; un processo che elabora si blocca, eseguendo la *inizioE*, se nella mailbox non ci sono messaggi già inviati da elaborare; infine un ricevente si blocca, eseguendo la *receive* se non ci sono messaggi nella mailbox o, anche se presenti, il primo di essi non è stato ancora elaborato.

b) senza riscrivere la soluzione indicare se, abilitando i riceventi a ricevere i messaggi nello stesso ordine in cui vengono terminate le loro elaborazioni, la soluzione si semplificherebbe o se risulterebbe più complessa, e perchè.

SOLUZIONE
Punto a) … prima versione

```
Class Mailbox
{
        semaforo mutex; (1)
        T coda [N];
        int cont, primo, ultimo; (0)
        semaforo attEI, attR, attS; (0)
        int blocEI, blocR, blocS; (0)
        bool elab[N]; (false)   //memorizza le terminazioni delle elaborazioni
```

```
        int contEF;
        int primoNE; (0)        //primo mess inviato ma la cui elaborazione non è ancora
                iniziata

Public:
        Void send(T inf)
        {
                mutex.wait();
                if(cont==N)
                        {
                                blocS++;
                                mutex.signal();
                                attS.wait();
                                blocs- - ;
                        }
                coda[ultimo]=inf;
                cont++;
                ultimo= (ultimo+1)%N;
                if(blocEI>0)
                        attEI.signal();
                else
                        mutex.signal();
        }


        void inizioE(int& ind)
        {
                mutex.wait();
                if(primoNE==ultimo)
                        {
                                blocEI++;
                                mutex.signal();
                                attEI.wait();
                                blocEI- -;
                        }
                ind=primoNE;
                primoNE=(primoNE+1)%N;
                mutex.signal();
        }

        void fineE(int ind)
        {
                mutex.wait();
                elab[ind]=true;
                contEF++;
                int i=0;
                if(blocR>0)
                {
                        int i=0;
                        while(!elab[i])
                                i++;
```

```
                if(i = primo)
                        attR.signal();
                else
                        mutex.signal();
        }
        else
                mutex.signal();
}

void Receive(T& inf)
{
        mutex.wait();
        int i=0;
        while(!elab[i])
                        i++;
        if(i != primo)
        {
                blocR++;
                mutex.signal();
                attR.wait();
                blocR- -;
        }
        inf=coda[primo];
        elab[primo]=false;
        primo=(primo+1)%N;
        cont--;
        if(elab[primo] && blocR>0)
                attR.signal();
        else
                if(blocS>0)
                        attS.signal();
                else
                        mutex.signal();
}
```

punto a) … seconda versione

```
Class Mailbox
{
        semaforo vuota; (N)
        semaforo piena; (0)
        semaforo ini_elab; (0)
        T coda[N];
        int primo_elab, ultimo_elab; primo, ultimo; (0)
        bool elab_finita[N]; (false)
        semaforo mutex; (1)

Public:
        void send(T inf)
        {
                vuota.wait();
```

```
        mutex.wait();
        coda[ultimo]=inf;
        ultimo=(ultimo+1)%N;
        ini_elab.signal();
        mutex.signal();
}

void Receive(T& inf)
{
        piena.wait();
        mutex.wait();
        inf=coda[primo];
        primo=(primo+1)%N;
        vuota.signal();//non faccio alcun controllo in quanto c'è stata almeno
                una wait per ogni send effettuata
        mutex.signal();
}

void inizioE(int& ind)
{
        ini_elab.wait();
        mutex.wait();
        ind=ultimo_elab;
        ultimo_elab=(ultimo_elab+1)%N;
        mutex.signal();
}

void fineE(int ind)
{
        mutex.wait();
        elab_finita[ind]=true;
        if(ind==primo_elab)
        {
                while(elab_finita[ind]==true)        //sveglio un numero di riceventi pari al
                                                     n° di elementi consecutivi (dall'inizio
                                                     coda) già elaborati
                {
                        elab_finita[ind]=false;
                        ind=(ind+1)%N;
                        piena.signal();
                        primo_elab=(primo_elab+1)%N;
                }
        }
        mutex.signal();
}
```

punto B)

La soluzione si complicherebbe non tanto per la determinazione del momento in cui un ricevente può togliere un certo messaggio dalla mailbox, ma quanto perché tale estrazione determina una

frammentazione dello spazio vuoto all'interno della coda, la quale, dal punto di vista dei mittenti, non sarebbe più tale.

Esame del 23/09/99

Una risorsa condivisa *R* è costituita da un record con *N* campi: *r1*, *r2*, ... , *rN*. Per operare su *R* sono definite *N* procedure *P1*, *P2*, ... , *PN*, ciascuna delle quali manipola esclusivamente uno dei campi del record: *P1* opera solo su *r1*, *P2* solo su *r2*, e così via.
Per garantire la consistenza di *R* è necessario imporre i seguenti vincoli di sincronizzazione:

> "inizialmente una qualunque delle *N* procedure può essere eseguita senza nessun vincolo ma la stessa, una volta terminata, non può essere eseguita di nuovo se non dopo che anche tutte le altre sono state eseguite. Terminata l'esecuzione di tutte le *N* procedure, in un qualunque ordine, lo stato di *R* è di nuovo consistente e le procedure possono essere rieseguite con le stesse regole".

a) Utilizzando il meccanismo semaforico, scrivere il prologo e l'epilogo delle *N* procedure, identificando la stuttura dati di *R* necessaria per garantire i precedenti vincoli di sincronizzazione.

b) Supponiamo di avere a disposizione una ulteriore procedura per operare su *R*, la procedura *RESET*, che quando eseguita blocca l'esecuzione di ulteriori procedure *Pi* e, se ci sono procedure in esecuzione, attende la loro terminazione, quindi esegue la procedura predefinita *restore* per riportare *R* in stato consistente.
Implementare la procedura RESET modiificando, se del caso. la soluzione a).

SOLUZIONE

punto a) //prima versione: passaggio del testimone

```
class Risorsa
{
        semaforo mutex; (1)
        semaforo blocca; (0)
        int terminati; (0)
        bool iniziati [N]; (false)
        int bloccati; (0)

Public:
        void prologo (int i)
        {
                mutex.wait();
                if (iniziati[i]==true)
                        {
                                bloccati++;
                                mutex.signal();
                                blocca.wait();
                                bloccati--;
                        }
                iniziati[i]=true;
```

```
            if (bloccati>0)
                  blocca.signal();
            else
                  mutex.signal();
      }

      void epilogo (int i)
      {
            mutex.wait();
            terminati++;
            if (terminati==N)
                  {
                        terminati=0;
                        for (int i=0; i<N; i++)
                              iniziati[i]=false;
                        blocca.signal();
            else
                  mutex.signal
      }
}
```

punto a) //seconda versione: semafori privati

```
class risorsa
{
      int cont;
      semaforo priv[N]; (1)
      semaforo mutex; (1)

Public:
      void prologo(int ind)
      {
            priv[ind];
      }

      void epilogo()
      {
            mutex.wait();
            cont++;
            if(cont==N)
                  {
                        cont=0;
                        for(int i, i<N, i++;)
                              priv[I].signal();
```

```
                }
            else
                mutex.signal();
        }
}
```

punto b)

```
class risorsa
{
        semaforo mutex; (1)
        int ini;
        bool iniziati[N]; (false)
        bool R; (false)
        int bloc_R; (0)
        semaforo resettati; (0)
        int bloc_att; (0)
        semaforo attesa; (0)
        int fine; (0)

Public:
        void prologo(int ind)
        {
            mutex.wait();
            if(R)
                {
                    bloc_R++;
                    mutex.signal();
                    resettati.wait();
                    mutex.wait();
                }
            if(iniziati[ind])
                {
                    bloc_att++;
                    mutex.signal()
                    attesa.wait();
                    bloc_att- -;
                }
            iniziati[ind]=true;
            ini++;
            if(bloc_att>0)
                attesa.signal();
            else
                {
```

```
                if ((bloc_att==0)&& R)
                        reset();
                    mutex.signal();
                }
        }

        void epilogo()
        {
            mutex.wait();
            fine++;
            if ( (fine==N) || (R && (fine==ini))  )
                {
                    fine=0;
                    ini=0;
                    for(int i=0; i<N, i++)
                        iniziati[i]=false;
                    if(bloc_att>0)
                        attesa.signal();
                }
            else
                mutex.signal();
        }

        void R()
        {
            mutex.wait();
          R=true;
          mutex.signal()
        }

        void restore()
        {
            R=false;
            while(bloc_r>0)
                {
                    bloc_r- -;
                    resettati.signal();
                }
        }
}

//note:
```

- Ho considerato l'epilogo come parte integrante della procedura

- Quando scatta R lascio terminare gli iniziati e tutti quelli in attesa, i quali avevano iniziato il prologo, cioè la procedura, senza R attivo)
- Reset fa tutto in mutua esclusione
- I resettati quando si svegliano devono riconquistarsi la mutua esclusione, senza tener conto dell'ordine in cui erano arrivati, questo perché mi dà l'idea di un resettaggio ☺

Esame del 13/01/00

Un insieme di processi mittenti *P1*, *P2*, ..., *PN* inviano messaggi di un tipo *T* predefinito ad un processo ricevente *Q* tramite una risorsa condivisa *M* da realizzare mediante un monitor.

La risorsa *M* deve contenere due code di messaggi in cui i messaggi inviati da qualunque processo mittente sono inseriti. I processi mittenti selezionano la coda in cui inserire il messaggio invocando la procedura *send1* del monitor o la procedura *send2* rispettivamente. Entrambe le code vengono definite di dimensione *M*<*N*.

Il processo ricevente puo` richiedere di ricevere o dalla *coda1* (tramite la procedura di monitor *receive1*) o dalla *coda2* (tramite la procedura di monitor *receive2*) o da una qualunque delle due (tramite la procedura di monitor *receiveM*). In quest'ultimo caso la procedura è bloccante se entrambe le code sono vuote e, in tal caso, il processo *Q* resta in attesa del primo messaggio inviato su una qualunque delle due code. Se una coda è vuota e l'altra no, la *receiveM* estrae e restituisce un messaggio dalla coda non vuota. Se entrambe le code sono non vuote la *receiveM* estrae il messaggio dalla coda diversa da quella da cui è stato estratto l'ultimo messaggio (sia con la *receive1* o con la *receive2* o con la *receiveM*). Per evitare ambiguità, nel caso in cui la prima receive eseguita sia la *receiveM* e entrambe le code siano non vuote si riceve da *coda1*.

a) Realizzare il precedente meccanismo.

b Definendo diversamente le dimensioni delle due code, riscrivere la soluzione, ed in particolare le procedure del monitor, in modo tale che le due send funzionino con la semantica della send sincronizzata.

c) utilizzando il meccanismo realizzato con la soluzione a), oppure la b), come potrebbe essere tradotto il seguente comando con guardie?

do

□ *B1*; *receive1*(*m*) **do** *S1*;

□ *B2*; *receive2*(*m*) **do** *S2*;

od

Non so se e' la soluzione di questo o del 13-1-03
Punto a)

```
Monitor Mailbox
{
        mes buffer[N];
        int primo1, primo2, ultimo ; (0)
        bool ricevuto[N];(false)
        condition pieno1, pieno2; vuoto;
        int quanti1, quanti2; (0)

Public:
        void send(mes m)
        {
                if(quanti1==N || quanti2==N)
```

```
                vuoto.wait();
        buffer[ultimo]=m;
        ultimo=(ultimo+1)%N;
        quanti1++;
        quanti2++;
    }

    void receive1(mess& m)
    {
        if(quanti1==0)
            piena1.wait();
        m=buffer[primo1];
        quanti1- -;
        if(ricevuto[primo1]= !ricevuto[primo1];
        primo1=(primo1+1)%N;
    }

    void receive2(mess& m)
    {
        if(quanti2==0)
            piena2.wait();
        m=buffer[primo2];
        quanti2- -;
        if(ricevuto[primo2]= !ricevuto[primo2];
        primo2=(primo2+1)%N;
    }
}
```

punto b)

```
Monitor Mailbox
{
    mes buffer[N];
    int primo1, primo2, ultimo ; (0)
    condition pieno1, pieno2; vuoto;
    int quanti1, quanti2; (0)

Public:
    void send(mes m)
    {
        if(quanti1+quanti2==N)
            vuoto.wait();
        buffer[ultimo]=m;
        ultimo=(ultimo+1)%N;
        quanti1++;
        pieno1.signal();
    }

    void receive1(mes& m)
    {
        if(quanti1==0)
```

```
                pieno1.wait();
            m=buffer[primo1];
            primo1=(primo+1)%N;
            quanti1- -;
            quanti2++;
            pieno2.signal();
        }

        void receive2(mes& m)
        {
            if(quanti2==0)
                pieno2.wait();
            m=buffer[primo2];
            primo2=(primo2+1)%N;
            quanti2- -;
            vuoto.signal();
        }
};
```

//**nota:** nel punto è stata adottata la semantica “signal&urgent”

UNIX

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 e 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).

SOLUZIONE UNIX

```
/* Traccia per il compito del 13 Gennaio
 * E' il classico problema dei filosofi a tavola ...
 */

#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <signal.h>
#include <string.h>

/* Macro che dice quanti sono i filosofi */
#define NUM_FILOSOFI 100

/* Massimo numero per cui eseguo un ciclo a vuoto */
#define DELAY 50000000

/* Struttura condivisa che contiene i semafori che rappresentano
 * le risorse condivise (forchette)
 */
struct {
  sem_t forchetta[NUM_FILOSOFI];
  /* Array dei parametri che vengono passati ad ogni thread */
  int par[NUM_FILOSOFI];
} shared;

/* Generico filosofo */
void *filosofo(void *);
/* Ultimo filosofo che invece inverte l'ordine delle wait */
void *ultimo(void *);

int main(void) {
  /* ID dei vari thread ... */
  pthread_t filosofoID[NUM_FILOSOFI];
  /* Argomenti passati in input ai vari thread */
  int i, res;

  /* Inizializzo il generatore di numeri casuali */
  srand(time(NULL));

  /* Adesso inizializzo tutti i semafori */
  for (i=0; i<NUM_FILOSOFI; i++) {
    if (sem_init(&shared.forchetta[i], 0, 1) == -1) {
      printf("sem_init forchetta[%d] failed!!!\n", i);
      exit(-1);
    }
    shared.par[i] = i;
  }

  /* Numero di thread che creero' .... */
  /* pthread_setconcurrency(NUM_FILOSOFI); */

  /* A questo punto posso creare i thread dei filosofi generici */
  for (i=0; i<NUM_FILOSOFI-1; i++) {
    res = pthread_create(&filosofoID[i], NULL,
```

```
                    filosofo, &shared.par[i]);
      printf("Sto per creare il thread %d \n", i);
      if (res != 0) {
        printf("pthread_create filosofo[%d] failed!!!!\n", i);
        exit(-1);
      }
    }

    /* Ora creo il thread dell'ultimo filosofo */
    res = pthread_create(&filosofoID[NUM_FILOSOFI-1], NULL,
                    filosofo, &shared.par[NUM_FILOSOFI-1]);
    if (res != 0) {
      printf("pthread_create filosofo[%d] failed!!!!\n", NUM_FILOSOFI-1);
      exit(-1);
    }

    for(i=0; i<NUM_FILOSOFI; i++) {
      pthread_join(filosofoID[i], NULL);
    }

    printf("Fine del programma ... \n");
    return 0;
}

/* Thread che rappresenta un generico filosofo.
 */
void *filosofo(void *in)
{
    int posizione = *((int *)in);
    int i;

    /* Prima fase ... il filosofo pensa .... */
    for (i=0; i<DELAY; i++);

    /* Inizio ad acquisire le risorse .... */
    sem_wait(&shared.forchetta[posizione]);
    sem_wait(&shared.forchetta[(posizione+1)%NUM_FILOSOFI]);

    /* Seconda fase ... il filosofo mangia */
    for (i=0; i<DELAY; i++);

    /* A questo punto rilascio le "risorse" */
    sem_post(&(shared.forchetta[posizione]));
    sem_post(&(shared.forchetta[(posizione+1)%NUM_FILOSOFI]));

    printf("Filosofo %d ... ho finito di mangiare \n", posizione);

    return NULL;
}

/* Thread che rappresenta l'ultimo filosofo (quello che inverte
 * l'ordine di acquisizione delle forchette).
 */
void *ultimo(void *in)
{
    int posizione = *((int *)in);
    int i;

    /* Prima fase ... il filosofo pensa .... */
    for (i=0; i<DELAY; i++);

    /* Inizio ad acquisire le risorse .... */
    sem_wait(&shared.forchetta[(posizione+1)%NUM_FILOSOFI]);
```

```
  sem_wait(&shared.forchetta[posizione]);

  /* Seconda fase ... il filosofo mangia */
  for (i=0; i<DELAY; i++);

  /* A questo punto rilascio le "risorse" */
  sem_post(&(shared.forchetta[(posizione+1)%NUM_FILOSOFI]));
  sem_post(&(shared.forchetta[posizione]));

  printf("Filosofo %d ... ho finito di mangiare \n", posizione);

  return NULL;
}
```

Esame del 03/02/00 (No Ancilotti)

UNIX

Si richiede al candidato di modellare ed implementare un pianoforte *multitasking*. L'architettura richiesta si compone delle seguenti parti :

- Sette task, uno per ogni nota musicale.
- Un task **pianista**.
- Una coda FIFO usata per comunicare dai task **note** al task **pianista**.

La comunicazione fra tutti i vari task avviene nel seguente modo :

- Il task **pianista** segnala a un task **nota** tramite un signal di tipo SIGUSR1 che la corrispondente nota deve essere suonata.
- Un task **nota**, ogni volta che riceve un signal SIGUSR1, stampa a video la nota corrispondente e quindi segnala che ha finito il suo compito inviando il suo PID sulla coda FIFO.
- Il task **pianista** resta in attesa sulla coda FIFO per sincronizzarsi con la fine di ogni task **nota**.

Tale architettura è esemplificata in figura 1.

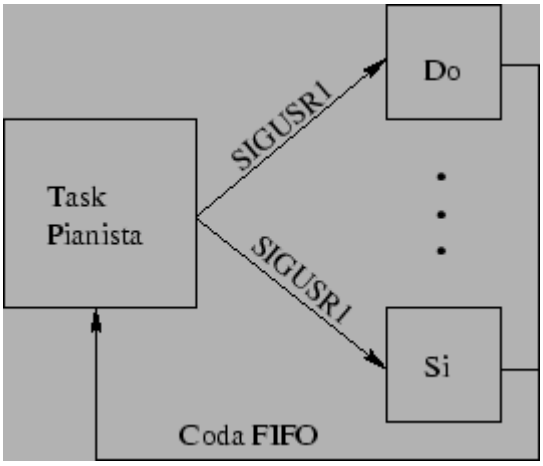


Figura 1: Architettura da implementare

Il codice dei task **nota** deve essere unico. Ogni task **nota** riceve fra i suoi parametri di input la nota che deve *impersonare*.

Schematicamente il task **nota** compie i seguenti passi :

- Installa la signal SIGUSR1.
- Apre in scrittura la coda FIFO.
- Segnala che ha finito la fase di inizializzizione inviando il proprio PID sulla coda FIFO.
- Entra in un ciclo in cui attende l'arrivo delle signal tramite l'uso della primitiva **pause()**

Il task **pianista** compie invece i seguenti passi :

1

Crea su disco la coda FIFO.

2

Crea tutti i task **nota** tramite la coppia di primitive **fork()/exec(...)**.

3

Apre in scrittura la coda FIFO.

4

Attende che tutti i task abbiano finito la fase di inizializzazione aspettando i PID sulla coda FIFO.

5

Suona i task **nota** inviando delle opportune signal.

6

Uccide tutti i task **nota** ed esce.

Domanda Facoltativa :

Cosa succede se si invertono i passi 2 e 3?

SOLUZIONE UNIX

```
/* Programma per il compito del 3 Febbraio 2000
 * Processi "pianisti" ...
 */

#include <stdio.h>
#include <unistd.h>
#define __USE_POSIX
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <time.h>

/* Numero delle note musicali*/
#define NUM_NOTE 7
/* Nome della pipe usata per la risposta */
#define RISPOSTA "risposta"
/* Numero delle note della "sinfonia" */
#define NUM_SINF 7

int main(void)
{
      /* PID dei task che "impersonano" le note */
      pid_t PIDnote[NUM_NOTE];
      /* Identificatori della FIFO per la risposta ... */
      int answer;
      /* Stringa con i nomi delle note */
      char *nomiNote[NUM_NOTE] = {"do", "re", "mi", "fa", "sol", "la", "si"};
      /* Array che contiene l'identificatore dei processi della sinfonia */
      int sinfonia[NUM_SINF] = {2, 1, 2, 1, 2, 1, 0};
      /* "buffer" contenente il dato inviato sulla FIFO */
      int i;
      int dato;
```

```
	/* Creo la fifo per la risposta di ogni task .... */
	if (mkfifo(RISPOSTA, S_IRUSR | S_IWUSR) < 0) {
		/* C'e' stato un errore nell'apertura della pipe */
		if (errno != EEXIST) {
			perror("Errore nella creazione della fifo di risposta");
			exit(-1);
		}
	}

	/* A questo punto inizio a creare i processi "nota" */
	for (i=0; i<NUM_NOTE; i++) {
		if ((PIDnote[i] = fork()) < 0) {
			perror("Errore nella fork");
			exit(-1);
		}
		else if (PIDnote[i] != 0) {
			/* Processo Padre ... */
			/* --------------- Non fa niente ---------------- */
		}
		else {
			/* Processo Figlio ... */
			if (execlp("nota", "nota", nomiNote[i], (char *)0) < 0){
				/* E' fallita la exec ... */
				perror("E' fallita le exec di una delle note");
				exit(-1);
			}
			/* A questa linea non si dovrebbe MAI arrivare .. */
			exit(0);
		}
	}

	/* Apro in lettura la pipe per la risposta */
	if ((answer = open(RISPOSTA, O_RDONLY)) == -1) {
		perror("Errore nell'apertura della pipe di risposta");
		exit(-1);
	}

	/* Mi sincronizzo con la fine dell'inizializzione di ogni task */
	for (i=0; i<NUM_NOTE; i++) {
		while (read(answer, &dato, sizeof(int)) == -1)
			;
	}

	/* A questo punto inizio a suonare la "sinfonia" */
	for (i=0; i<NUM_SINF; i++) {
		if (kill(PIDnote[sinfonia[i]], SIGUSR1) == -1) {
			perror("Errore durante la kill");
			exit(-1);
		}
		/* Aspetto che la nota sia finita ... */
		while (read(answer, &dato, sizeof(int)) == -1)
			;
	}

	/* A questo punto uccido tutti i task */
	for(i=0; i<NUM_NOTE; i++) {
		kill(PIDnote[i], SIGKILL);
	}

	close(answer);
	return 0;
}
```

```
/*
/* Codice del task "nota" ...
 */

#include <stdio.h>
#include <unistd.h>
#define __USE_POSIX
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <time.h>

/* Nome della pipe usata per la risposta */
#define RISPOSTA "risposta"

/* Coda FIFO per comunicare che si e' pronti */
int fifo;
/* Indice della fifo che serve per la risposta */
int risposta;
/* Stringa contenente la nota .... */
char nota[5];
/* PID del task .... */
int pid;

/* Forward declaration */
void handler (int signo);

int main(int argc, char *argv[])
{
      struct sigaction nuova;

      /* Come prima cosa apro la coda fifo in scrittura */
      if ((risposta = open(RISPOSTA, O_WRONLY)) == -1) {
            perror("Errore nell'apertura della pipe di risposta");
            exit(-1);
      }
      /* .. quindi copio la stringa della nota */
      strcpy(nota, argv[1]);
      /* Ottengo il mio PID */
      pid = getpid();

      /* Installo l'handler della signal ... */
      nuova.sa_handler = handler;
      /* Quando l'handler scatta maschera TUTTI i signal */
      sigemptyset(&nuova.sa_mask);
      /* Nessun flag viene settato */
      nuova.sa_flags = 0;

      /* Installo l'handler per il signal di codice SIGUSR1 */
      if (sigaction(SIGUSR1, &nuova, NULL) == -1) {
            perror("Non ho potuto allocare il signal SIGUSR1 ....\n");
            exit(-1);
      }

      /* A questo punto segnalo che ho finito le inizializzazioni */
```

```
	write(risposta, &pid, sizeof(int));

	/* Mi metto in attesa di signal .... */
	for(;;)
		pause();

	return 0;
}

/* Codice dell'handler della signal .... */
void handler(int signo)
{
	printf("Ho suonato la nota %s\n", nota);
	/* A questo punto segnalo che ho finito la nota */
	write(risposta, &pid, sizeof(int));
}
```

Esame del 17/02/00 (No Ancilotti)

UNIX

In molte applicazioni ad alta velocità il costo di una fork() può essere considerato inaccettabile per i ritardi che può introdurre.

In ambito multitasking si risolve questo problema tramite la tecnica dei server pre-forkati, ossia al momento dell'inizializzazione si creano più istanze di un server e non solo una.

Quando un client richiede un servizio il sistema guarda prima se uno dei server pre-forkati è pronto per essere utilizzato. In questo caso ``sveglia'' tale copia ed è quindi quest'ultima a svolgere il servizio, altrimenti ne crea una nuova istanza tramite una fork().

Si chiede in questo compito di replicare in ambiente multi-threading questo tipo di architettura.

Nella tipologia di connessione client-server in esame il server risponde al client con il suo identificatore di thread che il client stampa a video.

In particolare si richiede di scrivere le strutture dati per la gestione ed il codice relativo ai seguenti thread :

server

- È il server pre-forkato. Al momento dell'inizializzazione ne vengono create NUM_SERVERS istanze che si bloccano, in maniera opportuna, fino all'arrivo di una richiesta. Da ora in poi si farà riferimento a questi thread come thread *pronti*, per significare che non vengono generati all'arrivo di ogni richiesta, ma una volta terminato il loro servizio (che consiste, lo ricordiamo, nel passare il loro thread ID al client) diventano nuovamente disponibili.

masterServer

- Il comportamento di questo thread ciclico può essere così esemplificato :

  - All'arrivo di una richiesta controlla se c'è un thread server *pronto*, se ciò è vero allora lo attiva.
  - Se non ci sono thread pronti stampa un messaggio di errore ed abortisce il programma (si può tranquillamente usare una exit()).

client

- Al momento dell'inizializzazione vengono create NUM_CLIENTS istanze di questo thread che richiedono in modo ciclico per CLIENT_TIMES volte un servizio al masterServer che provvede a smistare la loro richiesta in maniera opportuna. Come prova dell'avvenuto servizio il thread client stampa a video l'ID del thread che ha fornito il servizio stesso.

Trattandosi di un ambiente multithreaded, tutte le comunicazioni *devono* avvenire a memoria comune, e le sincronizzazioni devono essere effettuate tramite semafori (o mutex e variabili condition).

Domanda facoltativa

Implementare la seguente modifica al thread masterServer: se il thread masterServer non trova thread pronti genera al volo un nuovo thread di tipo newServer.

I thread newServer sono i thread che vengono creati quando non ci sono più thread pronti: eseguono le stesse funzionalità dei thread server, ma terminano dopo aver servito una richiesta.

SOLUZIONE UNIX

```
/* Programma per l'esame del 17 Febbraio 2000.
 * Un server pre-forkato con i thread ...
 */

#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <signal.h>
#include <string.h>

/* Numero di thread "pre-threadati" */
#define NUM_SERVERS 5
/* Numero di thread client */
#define NUM_CLIENTS 10
/* Cicli a vuoto che fa un server */
#define DELAY_SERVER 5000000
/* Cicli a vuoto che fa un client */
#define DELAY_CLIENT 1000000
/* Numero di volte per cui un client esegue ... */
#define CLIENT_TIMES 5

/* Tipo enumerato per dire se un server e' libero o occupato */
typedef enum {BUSY, FREE} onoff;

/* Definizione di un tipo di dato */
struct {
      /* Semaforo di mutua esclusione */
      sem_t mutex;
      /* Numero dei server liberi */
      int freeServers;
      /* Semafori su sui si bloccano i vari server */
      sem_t wait[NUM_SERVERS];
      /* Stato di ogni server */
      onoff status[NUM_SERVERS];
} serversData;

struct {
  /* Semaforo di mutua esclusione fra i client */
  sem_t mutex;
  /* Indirizzo del semaforo privato su cui si blocca il client */
  sem_t *private;
      /* Indirizzo della variabile in cui il server
       * andra' a scrivere i risultati */
      pthread_t *result;
} clientData;

/* Semaforo su sui si mette in attesa il master Server */
```

```
sem_t richiesta;

/* Forward declaration .. */
void *masterServer(void *);
void *server(void *);
void *client(void *);

int main (void)
{
  /* IDs dei thread server e client */
  pthread_t serversID[NUM_SERVERS], clientsID[NUM_CLIENTS];
  /* IDs del thread master */
  pthread_t masterID;
  /* Indice da passare ai thread client */
  int input[NUM_CLIENTS];
  int i, res;

  /* Inizializzazione delle strutture dati */

      /* Inizializzo i dati della struttura servers */
  for(i=0; i<NUM_SERVERS; i++) {
    if (sem_init(&serversData.wait[i], 0, 0) == -1) {
      printf("NON sono riuscito ad inizializzare il semaforo %d\n", i);
      exit(-1);
    }
    serversData.status[i] = FREE;
  }
      if (sem_init(&serversData.mutex, 0, 1) == -1) {
    printf("NON sono riuscito ad inizializzare servers.mutex \n");
    exit(-1);
  }
      serversData.freeServers = NUM_SERVERS;

  /* Inizializzo i dati della struttura client */
  if (sem_init(&clientData.mutex, 0, 1) == -1) {
    printf("NON sono riuscito ad inizializzare master.mutex \n");
    exit(-1);
  }

  if (sem_init(&richiesta, 0, 0) == -1) {
    printf("NON sono riuscito ad inizializzare request \n");
    exit(-1);
  }

  /* A questo punto inizia la fase di creazione dei thread */
/*  pthread_setconcurrency(1+NUM_CLIENTS+NUM_SERVERS);*/

  /* Prima creo il thread del master server */
  res = pthread_create(&masterID, NULL, masterServer, NULL);
  if (res != 0) {
    printf("Non ho potuto creare il master server \n");
    exit(-1);
  }

  /* A questo punto creo i server normali .... */
  for (i=0; i<NUM_SERVERS; i++) {
    input[i] = i;
    res =  pthread_create(&serversID[i], NULL, server, &input[i]);

    if (res != 0) {
      printf("Non ho potuto creare il client %d-esimo \n", i);
      exit(-1);
    }
```

```
  }

  /* A questo punto creo i client */
  for (i=0; i<NUM_CLIENTS; i++) {
    res =  pthread_create(&clientsID[i], NULL, client, NULL);
    if (res != 0) {
      printf("Non ho potuto creare il client %d-esimo \n", i);
      exit(-1);
    }
  }

      /* Mi sincronizzo con la fine dei servers .... */
      for (i=0; i<NUM_CLIENTS; i++) {
            pthread_join(clientsID[i], NULL);
      }

      return 0;
}


/* Codice del master server */
void *masterServer(void *in)
{
      /* Mi dice quale e' il server pronto */
      int ready;
      /* ID del thread che viene eventualmente creato */

      for(;;) {
            /* Inizio bloccandomi in attesa di una richiesta */
            sem_wait(&richiesta);
            /* ... a questo punto c'e' stata una richiesta.
             * Come prima cosa controllo se c'e' un server
             * disponibile altrimenti lo devo creare ...
             */
            sem_wait(&serversData.mutex);
            if (serversData.freeServers != 0) {
                  /* a questo punto ho un server pronto ... devo solo
                   * scoprire quale ....*/
                  ready = 0;
                  while (ready <= NUM_SERVERS) {
                        if (serversData.status[ready] == FREE) {
                              break;
                        }
                        ready++;
                  }

                  /* Aggiorno le strutture dati */
                  serversData.freeServers--;
                  serversData.status[ready] = BUSY;
                  /* Rilascio la mutua esclusione. */
                  sem_post(&serversData.mutex);
                  /* .... e sveglio il server .... */
                  sem_post(&serversData.wait[ready]);
            }
            else {
                  printf("Sto creando un nuovo server ... \n");
                  printf("ARGGG!!!! Questa parte e' stata tolta!!!\n");
                  exit(-1);
            }
      }
      return NULL;
}
```

```
/* Codice del server normale .... */
void *server(void *in)
{
      int indice = *((int *)in), j;
      /* Zona dove andro' a scrivere il risultato */
      pthread_t *result;
      /* Semaforo privato del tack client */
      sem_t *private;

      for(;;) {
            /* Sono bloccato fino a che non mi arriva una richiesta. */
            sem_wait(&serversData.wait[indice]);
            /* A questo punto sono stato svegliato ... e, come
             * prima cosa, copio i dati dalla zona comune. */
            result = clientData.result;
            private = clientData.private;

            /* A questo punto rilascio la mutua esclusione ... */
            sem_post(&clientData.mutex);

            /* Do il risultato */
            *result = pthread_self();

            /* Perdo un poco di tempo .... */
            for(j=0; j<DELAY_SERVER; j++);

            /* Sveglio il thread client */
            sem_post(private);

            /* Segnalo che sono di nuovo libero */
            sem_wait(&serversData.mutex);
            serversData.freeServers++;
            serversData.status[indice] = FREE;
            sem_post(&serversData.mutex);

      }
      return NULL;
}

/* Codice del thread client ... accede al servizio per
 * un certo numero di volte ... */
void *client(void *in)
{
      int i, j;
      sem_t private;
      pthread_t result;

      /* Inizializzo il semaforo di attesa ... */
      if (sem_init(&private, 0, 0) == -1) {
            printf("NON sono riuscito ad inizializzare il semaforo private\n");
            exit(-1);
      }

      /* Ciclo del client .... */
      for(i=0; i<CLIENT_TIMES; i++) {
            result = 0;
            /* Ottengo la mutua esclusione ... */
            sem_wait(&clientData.mutex);
            /* Copio i dati .... */
            clientData.private = &private;
            clientData.result = &result;
            /* A questo punto sveglio il server ... */
            sem_post(&richiesta);
```

```
            /* e mi metto in attesa sul semaforo privato ... */
            sem_wait(&private);
            printf("Thread %ld -- servito dal server %ld \n", pthread_self(),
result);
            /* Aspetto un poco prima di fare una nuova richiesta */
            for(j=0; j<DELAY_CLIENT; j++);
      }

      return NULL;
}
```

Esame del 31/05/00 (No Ancilotti)

UNIX

Si richiede al candidato implementare l'operazione $y = (x + 1) * 2$ tramite un pipeline di due processi: il primo processo effettua l'operazione $x + 1$, passando il risultato al secondo processo che lo moltiplica per 2.

In particolare, realizzare il seguente programma. Il task padre:

- crea un `pipe`;
- crea due processi figli;
- attende la loro terminazione;

Il **primo processo figlio** ripete ciclicamente i seguenti passi:

1. legge un intero dal file ``in.txt''
2. ci somma 1
3. scrive il risultato sul pipe
4. torna al punto 1

Il **secondo processo figlio** esegue ciclicamente i seguenti passi:

1. legge un intero dal pipe
2. lo moltiplica per 2
3. scrive il risultato nel file ``out.txt''
4. torna al punto 1

Domanda Facoltativa :

Dopo aver processato tutti i dati in ingresso, i due task figli terminano. Il padre attende la loro morte, dopo un messaggio di commiato, termina. (**Ricordarsi di chiudere i descrittori di pipe in modo adeguato**)

SOLUZIONE UNIX

```
/* ------------------------------------------------------------
 * Compito del 31 Maggio 2000
 * Si tratta di implementare, tramite il multitasking, un semplice
 * esempio di pipeline.
 * ------------------------------------------------------------*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Scommentare la seguente linea e vedere cosa succede ...
 * Perche'????
 */
```

```
/* #define WRONG */

int main(int argc, char *argv[])
{
      /* Descrittori dei due processi figli creati */
      pid_t child1, child2;
  /* Descrittore del pipe usato per la comunicazione */
      int pipedescr[2];
      /* Variabile usata per leggere i risultati */
      int res;
      /* Descrittori dei file */
      FILE *infile, *outfile;
      /* ... un po' di variabili di appoggio .... */
      int invalue, outvalue;
      int status;

      /* Come prima cosa creo il pipe .... si noti come il pipe
       * viene creato dal processo padre ma viene utilizzato
       * dai due processi figli che lo ereditano tramite la fork()!! */
      if (pipe(pipedescr) < 0) {
            perror("Errore nell'apertura della pipe");
            return -1;
      }

      /* Creo il primo figlio (il processo che legge i dati dal file) */
      child1 = fork();
      if (child1 < 0) {
            perror("Errore nelle fork del primo figlio");
            return -1;
      }

      if (child1 == 0) {
            /* Processo figlio .... */

            /* Chiudo il pipe in lettura, perche' viene utilizzato
             * solo in scrittura. */
            close(pipedescr[0]);
            /* Apro il file contenente i numeri da leggere */
            infile = fopen("in.txt", "r");
            if (infile == 0) {
                  perror("Errore durante l'apertura di in.txt in lettura");
                  return -1;
            }

            /* Ciclo che legge i numeri e li passa all'altro processo */
            while(!feof(infile)) {
                  /* Leggo un numero, ci sommo uno e lo scrivo sul pipe */
                  if (fscanf(infile, "%d", &invalue) > 0) {
                        printf("Child1: read %d\n", invalue);
                        outvalue = invalue + 1;
                        printf("Child1: writing %d\n", outvalue);

                        res = write(pipedescr[1], &outvalue, sizeof(outvalue));
                  }
            }

            /* E' buona educazione chiudere il pipe prima di uscire!!! :-) */
            close(pipedescr[1]);

            printf("Figlio 1 -- Finito!!! \n");
            return 1;
      }
```

```
	/* Creo il secondo figlio (il processo che scrive i dati su file) */
	child2 = fork();
	if (child2 < 0) {
		perror("Errore nella fork del secondo figlio");
		return -1;
	}

	if (child2 == 0) {
		/* Processo figlio .... */

		/* Chiudo il pipe in scrittura, perche' viene utilizzato
		 * solo in lettura. */
		close(pipedescr[1]);
		/* Apro il file in cui verranno scritti i risultati */
		outfile = fopen("out.txt", "w");
		if (outfile == 0) {
			perror("Errore durante l'apertura di out.txt in scrittura");
			return -1;
		}

		/* Ciclo che legge i dati dal pipe, calcola i risultati e li
		 * scrive su file. */
		while(1) {
			/* Leggo il numero dal pipe, lo moltiplico per due
			 * e lo scrivo su file!! */
			res = read(pipedescr[0], &invalue, sizeof(invalue));
			if (res == 0) {
				/* Se res == 0 significa che il buffer del pipe e'
				 * vuoto e non ci sono piu' processi scrittori. */
				break;
			}
			printf("Figlio 2: letti %d bytes: %d\n", res, invalue);
			outvalue = invalue * 2;
			printf("Figlio 2: scritto il valore %d\n", outvalue);
			fprintf(outfile, "%d\n", outvalue);
		}

		/* E' buona educazione chiudere il pipe prima di uscire!!! :-) */
		close(pipedescr[0]);

		printf("Figlio 2 -- Finito!!! \n");
		return 1;
	}

	/* ... siamo nel processo padre ... */

	/* Chiudiamo il pipe ... visto che non viene usato dal processo
	 * padre .... */
	close(pipedescr[0]);
#ifndef WRONG
	close(pipedescr[1]);
#endif

	/* ... aspetto che terminino i due processi figlio */
	printf("processo padre... aspetto che i processi figli terminino \n");
	res = wait(&status);
	res = wait(&status);

	printf("Finished!!!\n");
	return 1;
}
```

Esame del 22/06/00

Due risorse *RA* ed *RB* sono allocate dinamicamente ad un gruppo di processi clienti *P1*, *P2*, ..., *PN* tramite un gestore *G*. Ogni processo puo` richiedere (e rilasciare) una specifica risorsa (*RA* o *RB*), rimanendo bloccato se la risorsa richiesta non e` disponibile, oppure una qualunque delle due risorse, ed in tal caso il processo si blocca solo se nessuna delle due e` disponibile.

a) Supponendo di operare un un sistema organizzato secondo il modello a scambio di messaggi, realizzare il codice di un processo server *S* che implementa il gestore delle due risorse ed indicare lo operazioni che ogni processo cliente deve eseguire sia per richiedere e rilasciare una specifica risorsa sia per richiedere e rilasciare una qualunque delle due. Nel realizzare il server non specificare nessuna priorita` fra le richieste

a1) per prima cosa realizzare il server utilizzando le seguenti primitive di comunicazione asimmetriche e sincronizzate:

send (*mes*) **to** *proc.porta*;

proc:= **receive**(*mes*) **from** *porta*

a2) successivamente ripetere la realizzazione del server utilizzando la entry call e lo statement accept tipici delle chiamate di procedure remote.

b) utilizzando le chiamate remote, riscrivere la soluzione ipotizzando di dare priorita` alle richieste specifiche rispetto alle richieste generiche.

unix

Per molte operazioni di tipo intrinsecamente parallelo si usa un paradigma di tipo *processor farm*. Ossia un'unità centrale smista il carico computazionale a diverse unità di elaborazione, rimanendo poi in attesa dei risultati e riallocando le unità che abbiano, nel frattempo finito il loro compito. Un semplice esempio di questa architettura si trova nei moderni processori in cui sono presenti più code di pipeline per l'aritmetica intera, in virgola mobile, l'accesso alla memoria, ecc.

L'esempio che sarà preso in considerazione in questo compito riguarda l'elaborazione di dati grafici. Alcuni tipi di filtraggio possono essere rappresentati come l'applicazione ripetuta di un prodotto fra matrici. Ogni pixel della nuova immagine viene infatti ottenuto sommando tutti gli elementi di una matrice ottenuta moltiplicando la matrice che rappresenta il filtro per la porzione di immagine, avente le stesse dimensioni del filtro, centrata sulle coordinate del pixel. Il numero di volte per cui il processo di moltiplicazione fra matrici e somma degli elementi è quindi uguale al numero di pixel che devono essere generati. Ad esempio, per un'immagine quadrata di 16x16 pixel esso deve essere ripetuto per 256 volte.

Si richiede di implementare una tale architettura di elaborazione usando i thread, definendo tutte le opportune strutture dati. In particolare si supponga che l'immagine sia quadrata e di lato DIM.

Il main deve svolgere i seguenti compiti :

1. Inizializza le strutture dati.
2. Crea NUM_THREAD thread.
3. Cerca un thread che attualmente non stia compiendo elaborazioni.
4. Notifica al thread le coordinate del pixel intorno a cui il filtraggio deve essere effettuato.
5. Ritorna al punto 3 fino a che tutti i pixel dell'immagine non siano stati elaborati.

6. Aspetta la fine di tutte le elaborazioni ed uccide i thread inviando un signal di tipo SIGKILL.

Ogni thread compie ciclicamente le seguenti azioni :

1. Aspetta di essere ``risvegliato''.
2. Stampa a video il suo identificatore e le coordinate del centro del filtro.
3. Aspetta per un numero casuale di cicli fra 0 e MAX_ATTESA.
4. Segnala la fine della sua elaborazione.
5. Ritorna al punto 1.

Domanda facoltativa

Discutere brevemente se, e per quali motivi, un'implementazione basata sui task sia più o meno vantaggiosa di quella basata sui thread.

Soluzione

Punto a1:

```
Process Server
{
        port{signal RicA, signal RicA, signal RicQ, int Ril }
        bool libera[2]; (true)
    process p;
    int ind;
    signal s;

    do
       ■ (libera[0]), p=receive(s) from RicA → { libera[0]=false}
       ■ (libera[1]), p=receive(s) from RicB → { libera[0]=false}
       ■ (libera[0] || libera[1]), p=receive(s) from RicQ →
                                                    {
                                          ind= (libera[0])?0:1;
                                          libera[ind]=false;
                                          send(ind) to p.ris;    // ris è la porta sul client
                                      }
       ■ p=receive(ind) from Ril → { libera[ind]= true}
    od

    // lato client

    send(s) to server.RicA;                  //in caso di richiesta della risorsa 1
       :
    send(s) to server.Ril(0);
       :
    send(s) to server.RicB;                  //in caso di richiesta della risorsa 2
       :
    send(s) to server.Ril(1);
```

```
    :
send(s) to server.RicQ;                    //in caso di richiesta di una risorsa qualunque
receive(ind) from ris;
    :
send(s) to server.Ril(ind);
```

//nota: è possible accorpare le funzioni di rilascio, ma non quelle di richiesta perchè non si può accettare una richiesta prima di saperne il tipo. In questo modo accodiamo le richieste sulle relative porte.

Punto a2:

```
process Server
{
        entry { RicA(), RicB(),  RicQ(int out r), Ril(int in r) }
        bool libera[2]; (true)
        int ind;

        do
                ■ (libera[0]), accept RicA() {} → { libera[0]=false;}
                ■ (libera[1]), accept RicB() {} → { libera[1]=false;}
                ■ (libera[0] || libera[1]), accept RicQ (int out r)              {

                        r=(libera[0])?0:1;

                        ind=r;

                } →

                {libera[ind]=false; }
                ■ accept Ril (int int r) { libera[r]=false;}      → {}
od
```

//nota: non c'è nessuna priorità fra le richieste ma in questa implementazione si dà priorità alla scelta della risorsa A nel caso di richiesta RicQ.
Se volessimo togliere anche questa priorità dovremmo suddividere la richiesta generica in due comandi con guardie:

```
    ■ (libera[0]), p=receive(s) from RicQ → {
                                                libera[0]=false;
                                                send(0) to p.ris;
                                            }
    ■ (libera[1]), p=receive(s) from RicQ → {
```

```
                                        libera[0]=false
                                        send (1) to p.ris;
                                    }
```

In questo caso se sono disponibili entrambe le risorse la scelta fra le due diventa casuale

Punto B:

```
process Server
{
        entry {RicA() ,RicB(), RicQ(), RisA(), RisB(), RisQ(int out r), Rilascio(int in r) }
        bool libera[2]; (true)
        int ind;
        int ca, cb, cq; (0)

do
        ■ accept RicA() {} → {
                                        if (libera[0])
                                                {
                                                        accept RisA() {}
                                                        libera[0]=false ;
                                                }
                                           else ca++ ;
        ■ accept RicA() {} → {
                                        if (libera[1])
                                                {
                                                        accept RisB() {}
                                                        libera[1]=false ;
                                                }
                                           else cb++ ;
        ■ accept RicQ() {} → {
                                        if (libera[0] || libera[1])
                                                {
                                                        accept RisQ(int out r)
                                    {
                                        r=(libera[0])?0:1;
                                        ind=r;
                                                                }
                                libera[ind]=false ;
                                                }
                                           else cq++ ;

        ■ accept Rilascio(int in r) {ind=r } →
                                           {if (ind=0)
                                                {
                                                        if(ca>0)
                                                                {
                                                                        accept
RisA() {}
                                                                        ca--;
                                                                }
```

```
                                                    else  if (cq>0)
                                                                {

        accept risQ(int out r)

        {r=0}
                                                                        cq--
;
                                                                    else
libera[0]=true;
                                                        if (ind=1)
                                                        {
                                                            if(cb>0)
                                                                {
                                                                    accept
RisB() {}
                                                                    cb--;
                                                                }
                                                            else  if (cq>0)
                                                                {

        accept risQ(int out r)

        {r=1}
                                                                        cq--
;
                                                                    else
libera[1]=true;
                                                    }
od
```

//nota: in caso di rilascio con riassegnamento della risorsa, quest'ultima non viene nemmeno rilasciata.

UNIX

```
/*************************************************
 * Soluzione per il compito del 22 Giugno 2000
 *************************************************
 */

#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <signal.h>
#include <math.h>

/* Dimensione dell'immagine (si suppone quadrata!!) */
#define DIM 2
```

```
/* Numero di thread che agiscono sull'immagine */
#define NUM_THREAD 4

/* Massimo indice per il ciclo di attesa */
#define MAX_ATTESA 100000000
/* #define MAX_ATTESA 0 */

/* Thread che fa il filtraggio */
void *compute(void *in);

/* Tipo enumerato booleano */
typedef enum {false, true} bool;

/* Struttura da passare ad ogni thread */
typedef struct {
      int x;
      int y;
} data;

struct {
      /* Numero di task liberi */
      sem_t numLiberi;
      /* Quali task sono liberi */
      bool liberi[NUM_THREAD];
      /* Semaforo privato per ogni task */
      sem_t private[NUM_THREAD];
      /* Dati da passare ad ogni thread */
      data dataThread[NUM_THREAD];
} shared;

/* Here we go .... :-) */
int main(void)
{
      /* .. un poco di variabili locali */
      int i, x, y, res, semValue;
      pthread_t threadID[i], inputID[NUM_THREAD];

      /* Inizializzo i dati */
      for(i=0; i<NUM_THREAD; i++) {
            shared.liberi[i] = true;
            shared.dataThread[i].x = 0;
            shared.dataThread[i].y = 0;
            if (sem_init(&shared.private[i], 0, 0) == -1) {
                  printf("sem_init shared.private[%d] failed!!!\n", i);
                  exit(-1);
            }
      }

      if (sem_init(&shared.numLiberi, 0, NUM_THREAD) == -1) {
            printf("sem_init numLiberi failed!!!\n");
            exit(-1);
      }

      /* Creo i 5 thread .... */
      for (i=0; i<NUM_THREAD; i++) {
            inputID[i] = i;
            res = pthread_create(&threadID[i], NULL, compute, &inputID[i]);
            if (res != 0) {
                  printf("pthread_create thread %d-esimo failed!!!!\n", i);
                  exit(-1);
            }
      }
```

```
	/* Ora inizio ad assegnare i compiti .... */
	for (x=0; x<DIM; x++) {
		for (y=0; y<DIM; y++) {
			sem_wait(&shared.numLiberi);
			/* A questo punto cerco quale thread e' libero */
			for (i=0; i<NUM_THREAD; i++) {
				if (shared.liberi[i] == true) {
					printf("indice libero = %d \n", i);
					/* Ho trovato un thread libero per cui gli passo i
dati */
					shared.dataThread[i].x = x;
					shared.dataThread[i].y = y;
					/* segnalo che il thread non e' piu' attivo */
					shared.liberi[i] = false;
					/* ...  e lo sveglio */
					sem_post(&shared.private[i]);
					break;
				}
			}
		}
	}

	/* Controllo che tutti i task abbiano finito */
	semValue = 0;
	while (semValue != NUM_THREAD)
		sem_getvalue(&shared.numLiberi, &semValue);

	/* A questo punto fermo tutti i thread */
	for (i=0; i<NUM_THREAD; i++) {
		pthread_kill(threadID[i], SIGKILL);
	}

	return 0;
}

/* Codice del thread */
void *compute(void *in)
{
	int i, attesa;
	int indice = *((int *)in);

	while(1) {
		sem_wait(&shared.private[indice]);
		attesa = (int)(((float)rand()/RAND_MAX)*MAX_ATTESA);
		printf("-------------- Thread %ld ----------\n", pthread_self());
		printf("attesa = %d \n", attesa);
		printf("x = %d\n", shared.dataThread[indice].x);
		printf("y = %d\n", shared.dataThread[indice].y);
		printf("-------------------------------------\n");
		for(i=0; i<attesa; i++);

		shared.liberi[indice] = true;
		sem_post(&shared.numLiberi);
	}

	return NULL;
}
```

Esame del 14/07/00 (no Ancilotti)

UNIX
In molti esempi di programmazione concorrente, diverse attività che eseguono in parallelo devono accedere ad una risorsa (software o hardware) condivisa, preservandone la consistenza logica.

Si considerino N attività concorrenti, che accedono alternativamente ad una risorsa condivisa, in questo caso rappresentata dal video del computer, effettuando 2 operazioni in maniera atomica. Ogni attività effettua ciclicamente m volte le seguenti operazioni:

1. scrive a video il proprio ID
2. scrive a video un progressivo rappresentante il numero del ciclo attuale
3. attende tramite un ciclo vuoto

Le operazioni 1) e 2) si intendono eseguite in maniera atomica, in modo da evitare ``interlacciamenti'' nelle scritte delle varie attività (ogni sequenza 1 - 2 eseguita da una singola attività deve essere non interrompibile).

Si implementi tale struttura prima tramite processi, ossia utilizzando un processo gestore per accedere al video, e quindi tramite thread, ossia utilizzando le primitive fornite dai pthread per assicurare l'atomicità della sequenza 1 - 2.

Domanda facoltativa

Discutere brevemente le due implementazioni, in termini di semplicità, di scalabilità e di efficienza.

SOLUZIONE UNIX
File 1
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

struct d {
      pid_t pid;
      int val;
};

int fd[2];

#define N 3
#define M 4

void childbody(void)
{
      struct d data;
      int i, j;

      close(fd[0]);
      for (i = 0; i < M; i++) {
            data.pid = getpid();
            data.val = i;
            write(fd[1], &data, sizeof(data));
            for (j = 0; j < 100000000; j++);
```

```
	}
	close(fd[1]);
}

int main(int argc, char *argv[])
{
	int i;
	pid_t pid;
	int res, ok;
	struct d data;

	/* Creiamo un pipe... */

	res = pipe(fd);
	if (res < 0) {
		perror("Error creating pipe");
		exit(-1);
	}

	for (i = 0; i < N; i++) {
		pid = fork();
		if (pid < 0) {
			perror("Error forking");
			exit(-1);
		}
		if (pid == 0) {
			/* Figlio... */
			childbody();
			exit(1);
		}
		/* Padre; continua a proliferare */
	}
	close(fd[1]);

	/* E ora, serviamo i figli!!! */
	while(ok) {
		ok = read(fd[0], &data, sizeof(data));

		if (ok != 0) {
			printf("TASK %d\n", data.pid);
			printf("\t count %d\n", data.val);
		}
	}
}
```

FILE 2

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>

#define N 3
#define M 4

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *threadbody(void *p)
{
	int i, j;
```

```
      for (i = 0; i < M; i++) {
            pthread_mutex_lock(&mutex);
                  printf("THREAD %d\n", pthread_self());
                  printf("\t count %d\n", i);
            pthread_mutex_unlock(&mutex);
            for (j = 0; j < 100000000; j++);
      }
      return NULL;
}


int main(void)
{
        pthread_t child[N];
        int res, i;

        /* Setto il concurrency level a 3 */
//        pthread_setconcurrency(N);

        /* A questo punto posso creare i thread .... */
      for (i = 0; i < N; i++) {
            res = pthread_create(&(child[i]), NULL, threadbody, NULL);
            if (res != 0) {
                  printf("Errore nella creazione del thread %d\n", i);
                  return -1;
            }
        }

      /* Beh.... Ho finito: basta aspettare i threads... */
      for(i = 0; i < N; i++) {
            pthread_join(child[i], NULL);
      }
}
```

Esame del 07-09-00

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:

procedure entry *parere* (*i*:*indice*, *voto*:*boolean*)

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

function *esito*: *boolean*;

funzione invocata da ogni processo, dopo aver espresso il proprio voto, per sapere l'esito della consultazione. Se la funzione restituisce il valore *true* l'esito e` positivo e il processo continua indipendentemente dal suo specifico risultato. Se il valore restituito e` *false* 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.

process *Pi* (1≤i≤*N*)
.....
begin
Fi;
voto(*i*,*risultato*);
if not *esito* **then** *Gi*;
.....
.....
end;

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.

b) ripetere la soluzione ipotizzando di essere in un sistema orientato ai messaggi ed utilizzando il meccanismo delle chiamate di procedure remote.

Unix

Nella figura 1 ogni cerchio rappresenta un processo ed ogni freccia un pipe unidirezionale. Il processo i-esimo si comporta nel seguente modo per NUM_PASSI volte:

1. Estrae un numero casuale compreso fra 0 e MAX_i con $i \in \{1,2,3\}$
2. Invia il numero estratto al processo successivo in figura.
3. Riceve il numero estratto dal processo precedente nella figura.
4. Se il numero estratto è più alto di quello ricevuto allora $MAX_i --$, altrimenti $MAX_i ++$.
5. Ritorna al passo 1.

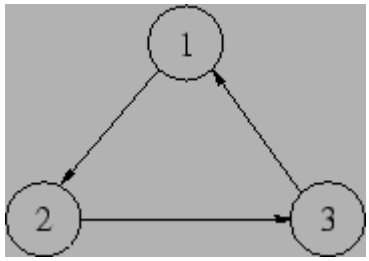


Figura 1: Struttura dei processi

Si chiede al candidato di scrivere il programma che implementi la struttura mostrata in figura ed il comportamento dei vari processi.

Domanda facoltativa
Si esamini il seguente listato e si scriva uno dei possibili output a video, spiegandone le ragioni.

SOLUZIONE

Parte a)

```
Class votazione
{
        semaforo mutex; (1)
        semaforo attesa; (0)
        int numpos; (0)
        int cont; (0)
        int bloc; (0)
        bool fine; (false)        // se vero indica che l'esito è definitivo
        bool ris; (false)                 // indica l'esito della consultazione
Public:
        void voto (int ind, bool voto)
        {
```

```
        mutex.wait();
        cont ++;
        if(!ris)
                {
                        if(voto)
                                numpos++;
                        if((numpos==M) || ((cont-numpos)>(N-M)))
                                {
                                        ris=true;
                                        fine=(numpos==M)?true:false;
                                        while(bloc>0)
                                                {
                                                        attesa.signal();
                                                        bloc- - ;
                                                }
                                }
                }
      mutex.signal();
}

void esito ()
{
        mutex.wait();
        if(!ris)
                {
                        bloc++;
                        mutex.signal();
                        attesa.wait();
                        mutex.wait();
                        bloc- -;
                }
        return fine;
}
```

//nota: non ho fatto uso del parametro "i" della chiamata a Voto, in quanto mi sembra superfluo.

//nota: I processi non sono ciclici, quindi le funzioni Voto e Esito vengono chiamate una sola volta per ogni processo

//nota: Il testo non lo specifica quando comunicare l'esito della votazione ai processi: secondo questa implementazione l'esito viene comunicato non a fine votazione ma appena è possibile saperlo e solo tramite la\1

//nota: In caso di esito noto la funzione Voto sveglia tutti i processo bloccati su Esito contemporaneamente, in quanto non si pone il problema delle corse critiche.

Parte b)

```
Process Votazione
{
        entry { voto(int in ind, bool in voto), esito(bool out end) }
        bool x;
```

```
        bool ris;(false)
        bool fine; (false)
        int numpos; (0)

        do
                ■ accept voto(int in ind, bool in voto) {x=voto;} →

                                                        {
                                        if(!ris)
                                                {
                                                        if(x) numpos++;
                                                        if((numpos==M) || ((cont-numpos)>(N-M)))
                                                                {
                                                                        ris=true;
                                                                        fine=(numpos==M)?true:false;
                                                                }
                                                        }
                ■ (ris), accept esito(out bool end) { end=fine; }
        od
}
```

SOLUZIONE UNIX
File 1

```
/* Compito per l'appello di Sistemi Operativi del 7 Settembre.
 * Si tratta in pratica di un esercizio sulla fork e sulle
 * pipe come mezzo di comunicazione fra un task padre ed un
 * figlio.
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

#define MAX_MODULO 2000

/* File del main .... */
int main(void)
{
      /* Dichiarazione dei pipe che saranno usati per l'esempio */
      int unodue[2], duetre[2], treuno[2], num, i;
      pid_t child1, child2;
      int seed1, seed2;
      /* Inizializzo il modulo */
      int modulo = rand()%MAX_MODULO, estratto;

      /* Inizializzo il generatore di numeri casuali nel processo padre.*/
      srand(time(NULL));
      /* Valori iniziali per settare i generatori di numeri casuali.
       * Perche' si deve settare il generatore di numeri casuali anche
       * nei processi figli??? Non basta farlo solo nel processo padre????
       */
      seed1 = rand();
      seed2 = rand();

      /* Apertura del primo pipe */
```

```
if(pipe(unodue) < 0) {
      printf("Errore nell'apertura del pipe unodue \n");
      exit(-1);
}

/* Apertura del secondo pipe */
if(pipe(duetre) < 0) {
      printf("Errore nell'apertura del pipe duetre \n");
      exit(-1);
}

/* A questo punto posso iniziare con la fork */
child1 = fork();
if (child1 < 0) {
      printf("errore nella prima fork \n");
      exit(-1);
}
else if (child1 == 0) {
      /* Sono nel processo figlio */
      close(unodue[1]);
      close(duetre[0]);
      /* Setto il generatore di numeri casuali */
      srand(seed1);
      modulo = rand();
      modulo = modulo%MAX_MODULO;
      num = 0;
      while (1) {
            read(unodue[0], &num, sizeof(int));
            /* Se riceve -1 sul pipe allora il processo figlio termina
             * le sue elaborazioni */
            if (num == -1) {
                  write(duetre[1], &num, sizeof(int));
                  return 0;
            }
            estratto = rand()%modulo;
            if (num > estratto)
                  modulo++;
            else
                  modulo--;
            printf("processo figlio 1 -- modulo = %d \n", modulo);
            write(duetre[1], &estratto, sizeof(int));
      }
      return 0;
}

/* Apertura del terzo pipe. Perche' viene aperto solo a questo
 * punto e non prima???? */
if(pipe(treuno) < 0) {
      printf("Errore nell'apertura del pipe treuno \n");
      exit(-1);
}

child2 = fork();
if (child2 < 0) {
      printf("errore nell'apertura del pipe \n");
      exit(-1);
}
else if (child2 == 0) {
      /* Sono nel processo figlio */
      close(unodue[0]);
      close(unodue[1]);
      close(duetre[1]);
      close(treuno[0]);
```

```
            /* Setto il generatore di numeri casuali */
            srand(seed2);
            modulo = rand();
            modulo = modulo%MAX_MODULO;
            num = 0;
            while (1) {
                  read(duetre[0], &num, sizeof(int));
                  /* Se riceve -1 sul pipe allora il processo figlio termina
                   * le sue elaborazioni */
                  if (num == -1) {
                        write(treuno[1], &num, sizeof(int));
                        return 0;
                  }
                  estratto = rand()%modulo;
                  if (num > estratto)
                        modulo++;
                  else
                        modulo--;
                  printf("processo figlio 2 -- modulo = %d \n", modulo);
                  write(treuno[1], &estratto, sizeof(int));
            }
      }

      /* Sono nel processo padre */
      close(duetre[0]);
      close(duetre[1]);
      close(unodue[0]);
      close(treuno[1]);

      for (i=0; i<3*MAX_MODULO; i++) {
            estratto = rand()%modulo;
            write(unodue[1], &estratto, sizeof(int));
            read(treuno[0], &num, sizeof(int));
            if (num > estratto)
                  modulo++;
            else
                  modulo--;
            printf("processo padre -- modulo = %d\n", modulo);
      }

      /* Segnalo ai processi figli che devono terminare inviando
       * il valore -1 sul pipe */
      estratto = -1;
      write(unodue[1], &estratto, sizeof(int));
      read(treuno[0], &num, sizeof(int));
      printf("inviato %d -- letto %d \n", estratto, num);

      /* Finisco di chiudere le pipe ancora aperte!! */
      close(unodue[1]);
      close(treuno[0]);

      return 0;
}
```

File 2

```
/* Quesito facoltativo dell'appello del 7Settembre 2000
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
```

```
int main(void)
{
      pid_t child[2];
      pid_t child2nd[4];
      int val = 0, i, j;

      for (i=0; i<2; i++) {
            child[i] = fork();
            if (child[i] == -1) {
                  printf("Errore nella fork!! \n");
                  exit(-1);
            }
            else if (child[i] == 0) {
                  printf("Valore = %d \n", val);

                  for (j=0; j<2; j++) {
                        child2nd[i*2+j] = fork();
                        if (child2nd[i*2+j] == -1) {
                              printf("Errore nella fork!! \n");
                              exit(-1);
                        }
                        else if (child2nd[i*2+j] == 0) {
                              printf("-- Valore = %d\n", val);
                              return 0;
                        }
                        val++;
                  }
                  return(0);
            }
            val++;
      }

      for (i=0; i<2; i++)
            waitpid(child[i], NULL, 0);
      for(j=0; j<4; j++)
            waitpid(child2nd[j], NULL, 0);

      return(0);
}
```

Esame del 21-09-00

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 sincronmizzazione:

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 può seguire esclusivamente l'esecuzione di una o più attivazioni concorrenti della procedura *B;*

- la seconda sequenza è costituita dall'esecuzione della procedura *C* a cui può 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.

a) utilizzando il meccanismo delle chiamate di procedure remote, realizzare un processo server che fornisca le entry*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 è richiesta la soluzione ad eventuali problemi di starvation.

b)Ripetere la soluzione adottando il costrutto semaforico.

UNIX

Il fattoriale di un numero è definito come il prodotto dei primi n numeri naturali.

Si richiede al candidato di dare un'implementazione del calcolo del fattoriale in ambiente multi-tasking. Il processo principale ha una variabile locale di nome *num* contenente il numero di cui si vuole calcolare il fattoriale.

Se $num \neq 0$ allora il processo padre fa nell'ordine i seguenti passi :

1. crea un pipe.
2. crea un figlio, tramite una *fork()*, il quale dovrà calcolare $(num-1)!$.
3. aspetta di ricevere, tramite il pipe, il risultato dell'elaborazione del figlio.

Ogni processo figlio esegue i seguenti passi :

- Se il numero di cui deva calcolare il fattoriale è 1 allora restituisce 1 sul pipe e termina.
- altrimenti :
    1. crea un pipe
    2. crea un figlio, tramite una *fork()*, il quale dovrà calcolare $(num-1)!$.
    3. aspetta di ricevere, tramite il pipe, il risultato dell'elaborazione del figlio.
    4. calcola $num!$, lo invia sul pipe e termina.

Suggerimento
Si consiglia di implementare le elaborazioni del processo figlio come una funzione

SOLUZIONE

Punto a)

```
Enum permessi{AC,B,DE,nessuno}

// AC: posso eseguire A o C
// B: posso eseguire solo B
// DE : posso eseguire D o E
// nessuno: stato intermedio, per le inconsistenze

process server
{
        permessi stato=AC;
        int proc=0;
        entry SA(),EA(),SB(),EB(),SC(),EC();SD(),ED();SE(),EE();

        do
                ■ (stato==AC),accept SA(){} → stato=nessuno;

                ■ accept EA(){} → stato=B;

                ■ (stato==B),accept SB(){} →{
                                                        proc++;
                                                }
                ■ accept EB(){} → {
                                                                proc- -;
                                                                if(proc==0)
                                                                        stato=AC;
                                                        }
                ■ (stato==AC),accept SC(){} → stato=nessuno;

                ■ accept EC(){} → stato=DE;

                ■ (stato==DE),accept SD(){} → stato=nessuno;

                ■ accept ED(){} → stato=nessuno;

                ■ (stato==DE),accept SE(){} → stato=nessuno;

                ■ accept EE(){} → stato=AC;
od

}
```

punto b)

```
Enum permessi{AC,B,DE,nessuno}
Class gestore
{
        permesso stato; (AC)
```

```
        int contB, blocA, blocB, blocC, blocD, blocE; (0)
        semaforo mutex; (1)
        semaforo privA, privB, privC, privD, privE; (0)

Public:
        void SA()
        {
                mutex.wait();
                if(stato!=AC)
                        {
                                blocA++;
                                mutex.signal();
                                privA.wait();
                                blocA--;
                        }
                stato=nessuno;
                mutex.signal();
        }

        void EA()
        {
                mutex.wait();
                stato=B;
                if(blocB>0)
                        privB.signal();
                else
                        mutex.signal();
        }

        void SB()
        {
           mutex.wait();
                 if(stato!=B)
                        {
                                blocB++;
                                mutex.signal();
                                privB.wait();
                                blocB--;
                        }
                contB++;
                if (blocB>0)
                        privB.signal();
                else
                mutex.signal();
        }

        void EB()
        {
                mutex.wait();
                contB- -;
                if(contB==0)
                        {
                                stato=AC;
                                if(blocC>0)
                                        privC.signal();
                                else
```

```
                                        if(blocA>0)
                                                privA.signal();
                                        else
                                                mutex.signal();
                        }
                else
                        mutex.signal();
}

void SC()
{
        mutex.wait();
        if(stato!=AC)
                {
                        blocC++;
                        mutex.signal();
                        privC.wait();
                        blocC--;
                }
        stato=nessuno;
        mutex.signal()
}

void EC()
{
        mutex.wait();
        stato=DE;
        if(blocD>0)
                privD.signal();
        else
                if(blocE>0)
                        privE.signal();
                else
                        mutex.signal();
}

void SD()
{
        mutex.wait();
        if(stato!=DE)
                {
                        blocD++;
                        mutex.signal();
                        privD.wait();
                        blocD--;
                }
        stato=nessuno;
        mutex.signal()
}

void ED()
{
        mutex.wait();
        stato=AC;
        if(blocA>0)
                privA.signal();
```

```
            else
                if(blocC>0)
                    privC.signal();
                else
                    mutex.signal();
        }

        void SE()
        {
            mutex.wait();
            if(stato!=DE)
                {
                    blocE++;
                    mutex.signal();
                    privE.wait();
                    blocE--;
                }
            stato=nessuno;
            mutex.signal()
        }

        void EE()
        {
            mutex.wait();
            stato=AC;
            if(blocA>0)
                privA.signal();
            else
                if(blocC>0)
                    privC.signal();
                else
                    mutex.signal();
        }
```

//Note:

-Lo stato nessuno è utilizzato quando una funzione è iniziata ma non è ancora finita, tranne in B dove è possibile che si abbiano + istanze della procedura attive in contemporanea

-la startB è l'unica che non va in stato=nessuno, per permettere l'attivazione di più istanze di B

- Ho dato una qualche priorità:
  -a fine B prima controllo i C bloccati poi gli A
  -a fine D o E prima controllo gli A bloccati poi i C

SOLUZIONE UNIX

```
/* Compito per l'appello del 21 Settembre.
 * Si tratta di un'implementazione del fattoriale tramite
 * una serie di processi multitasking.
 */

#include <stdio.h>
#include <unistd.h>
```

```
#include <sys/wait.h>
#include <stdlib.h>

/* Forward Declaration */
void funzione(int, int *);

int main(void)
{
      int prima[2], numero=0;
      pid_t child;

      /* Apro il pipe prima */
      if(pipe(prima) < 0) {
            printf("Errore nell'apertura del pipe prima \n");
            exit(-1);
      }

      child = fork();
      if (child < 0) {
            printf("Errore nella fork!!");
            exit(-1);
      }
      else if (child == 0) {
            /* Processo figlio */
            funzione(10, prima);
      }
      else {
            /* Qui invece sono nel processo padre */
            close(prima[1]);
            read(prima[0], &numero, sizeof(int));
            printf("Il numero e' %d \n", numero);
            exit(0);
      }
}

/* Funzione che implementa il fattoriale */
void funzione(int numero, int *ritorno)
{
      /* Pipe usati per la comunicazione */
      int input[2], ricevuto;
      /* Identificativo del processo figlio */
      pid_t child;

      /* Chiudo le estremita' del pipe che non mi servono */
      close(ritorno[0]);

      if (numero == 1) {
            write(ritorno[1], &numero, sizeof(int));
      }
      else {
            /* A questo punto apro i pipe */
            if(pipe(input) < 0) {
                  printf("Errore nell'apertura del pipe input \n");
                  exit(-1);
            }

            /* Ora faccio la fork */
            child = fork();
            if (child < 0) {
                  printf("errore sulla fork\n");
                  exit(-1);
            }
            else if (child == 0) {
```

```
                /* Sono nel processo figlio */
                funzione(numero-1, input);
            }
            else {
                close(input[1]);
                /* Ora mi metto in ascolto sul pipe */
                read(input[0], &ricevuto, sizeof(int));
                /* kill(child); */
                ricevuto *= numero;
                write(ritorno[1], &ricevuto, sizeof(int));
            }
        }
}
```

Esame del 12/01/01 (No Ancilotti)

UNIX

Il sistema operativo Unix viene di solito fornito con un insieme di programmi di utilità. Tra questi, il programma grep serve per cercare una stringa in un insieme di file specificato sulla linea di comando.

Il candidato deve scrivere una versione semplificata e multi-tasking del comando grep che deve fare le seguenti cose:

- prende in ingresso sulla linea di comando una stringa e un insieme di file;
- per ogni file, crea una pipe e fa la fork() di un figlio;
- ogni figlio legge il proprio file una riga per volta: se la riga contiene la stringa specificata, il figlio scrive la riga sulla pipe; alla fine del file, il figlio semplicemente chiude la pipe ed esce.
- Dopo aver creato tutti i figli, il padre si mette in attesa sulle pipe (una per volta). Quando riceve qualcosa, stampa a video il nome del file seguito dal carattere ':' e dalla riga letta dalla pipe.

Tenere presente le seguenti cose:

- assumere che la lunghezza massima di una riga sia data da una costante MAX_ROW_LEN (per esempio pari a 1000 caratteri)
- assumere che il numero massimo di file specificabili sulla linea di comando sia dato da una costante MAX_FILES (per esempio pari a 100)
- utilizzare le seguenti funzioni di libreria:

```
#include<stdio.h>

char *fgets(char *s, int size, FILE *stream);
```

- la funzione fgets() legge una riga (della lunghezza massima di size caratteri) dal file puntato da stream e lo mette in s.

```
#include<string.h>

size_t strlen(const char *s);
char *strstr(const char *s, const char *substring);
```

- La funzione strlen() restituisce la lungezza della stringa puntata da s; la funzione strstr() restituisce il puntatore alla prima occorrenza della sottostringa substring nella stringa s, oppure 0 se la substring non e' contenuta in s.

SOLUZIONE UNIX

```
#include <stdio.h>
#include <string.h>

#define MAX_ROW_LEN 1000
#define MAX_FILES 1000

void sys_err(char *s)
{
  perror(s);
  exit(-1);
}

void cerca(char *str, char *filename, int fd)
{
  FILE *file;
  char row[MAX_ROW_LEN];
  char *pointer;

  file = fopen(filename, "r");

  if (!file) sys_err("Error in opening file");

  while (!feof(file)) {
    fgets(row, MAX_ROW_LEN, file);
    pointer = strstr(row, str);
    if (pointer != NULL)
      write(fd, row, strlen(row));
  }

  close(fd);
}


int main(int argc, char *argv[])
{
  int nfiles;
  int i;
  int child[MAX_FILES];
  int chpipe[MAX_FILES][2];
  int nbytes;
  char buf[MAX_ROW_LEN];

  if (argc < 3) {
    printf("Uso: %s STRING [FILE]\n", argv[0]);
    exit(-1);
  }

  nfiles = argc - 2;

  for (i = 0; i < nfiles; i++) {
    pipe(&(chpipe[i][0]));
    if ( (child[i] = fork()) < 0 ) sys_err("Errore nella fork!");
    if (child[i] == 0) {
      close(chpipe[i][0]); // chiude la pipe in lettura
      cerca(argv[1], argv[i+2], chpipe[i][1]);
      exit(0);
    }
    else close(chpipe[i][1]); // chiude la pipe in scrittura
```

```
  }

  for (i = 0; i < nfiles; i++) {
    nbytes = read(chpipe[i][0], buf, MAX_ROW_LEN);
    while (nbytes != 0) {
      buf[nbytes] = 0;
      printf("%s : %s \n", argv[i+2], buf);
      nbytes = read(chpipe[i][0], buf, MAX_ROW_LEN);
    }
    waitpid(child[i],0,0);
  }
}
```

Esame del 01/02/01

UNIX

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!");
   else printf("Ho perso!");

   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.

SOLUZIONE UNIX

```
#include <pthread.h>
#define NUM_THREADS 11

struct {
  int voti_si;
  int voti_no;
  int blocked;
  pthread_cond_t attesa;
  pthread_mutex_t mutex;
} urna = {0,0,0,PTHREAD_COND_INITIALIZER,PTHREAD_MUTEX_INITIALIZER};

void vota(int v)
{
  pthread_mutex_lock(&urna.mutex);
  if (v) urna.voti_si++;
  else urna.voti_no++;

  if ((urna.voti_si > (NUM_THREADS / 2)) ||
      (urna.voti_no > (NUM_THREADS / 2))) {

    while (urna.blocked > 0) {
      pthread_cond_signal(&urna.attesa);
      urna.blocked--;
    }
  }
  pthread_mutex_unlock(&urna.mutex);
}

int risultato()
{
  pthread_mutex_lock(&urna.mutex);
  if ((urna.voti_si <= (NUM_THREADS / 2)) &&
      (urna.voti_no <= (NUM_THREADS / 2))) {
    urna.blocked ++;
    pthread_cond_wait(&(urna.attesa), &(urna.mutex));
  }

  if (urna.voti_si > urna.voti_no) {
    pthread_mutex_unlock(&urna.mutex);
    return 1;
  }
  else {
    pthread_mutex_unlock(&urna.mutex);
    return 0;
  }
}

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);
}


int main()
{
  int i;
  pthread_t tid[NUM_THREADS];

  pthread_setconcurrency(NUM_THREADS);

  for (i=0; i < NUM_THREADS; i++)
    pthread_create(&(tid[i]), NULL, thread, NULL);

  for (i=0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
```

Esame del 15/02/01 (Solo testo Unix)

UNIX

Dati i due vettori x[] e y[], il prodotto vettoriale z[] si calcola nella seguente maniera:

$$\begin{aligned} z[0] &= x[1] * y[2] - x[2] * y[1]; \\ z[1] &= x[2] * y[0] - x[0] * y[2]; \\ z[2] &= x[0] * y[1] - x[1] * y[0]; \end{aligned}$$

Realizzare un programma concorrente che esegua il prodotto vettoriale fra x[] e y[] come segue:

- Il programma principale (padre), esegue la fork() per 3 volte, producendo 3 figli;
- Il figlio i-esimo si occupa di calcolare la variabile z[i];
- Quando tutti i figli hanno finito, il padre stampa a video il risultato, cioe' il contenuto del vettore z[].

Per semplicita', si supponga che i vettori x[] e y[] vengano inizializzati tramite una funzione esterna gia' esistente (il candidato non deve implementare tale funzione!):

```
void init(double x[], double y[]);
```

SOLUZIONE UNIX

Esame del 07/06/01 (Solo testo Unix)

UNIX

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''};
```

DOMANDA FACOLTATIVA: Se si volesse implementare lo stesso programma tramite dei processi invece che dei thread, e si volessero usare le pipe() come meccanismo di comunicazione, quante pipe in totale sarebbero necessarie?

SOLUZIONE UNIX

Esame del 13/07/01 (No Ancilotti)

UNIX

Scrivere un programma con due thread che interagiscono tramite la seguente struttura dati:

```
struct {

pthread_mutex_t mutex;

pthread_cond_t cond1;

pthread_cond_t cond2;

int cnt1;

int cnt2;

} s = {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER,
PTHREAD_COND_INITIALIZER};
```

Il thread numero 1 esegue il seguente ciclo infinito:

- se il contatore cnt1 e' maggiore o uguale al contatore cnt2, si mette in attesa sulla condizione cond1;
- incrementa il contatore cnt1 di una unita';
- se il contatore cnt1 e' uguale al contatore cnt2, sblocca la condizione cond2;
- stampa a video il valore di entrambi i contatori.

Il thread numero 2 esegue il seguente ciclo infinito:

- Se il contatore cnt2 e' maggiore di cnt1 si mette in attesa sulla condizione cond2;
- incrementa il contatore cnt2 di 3 unita';
- sblocca la condizione cond1.

Scrivere il codice del main() e di ognuno dei due thread, stando ben attenti ad evitare condizioni di deadlock e a proteggere la struttura s con il mutex.

SOLUZIONE UNIX

```
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>

struct {
  pthread_mutex_t mutex;
  pthread_cond_t cond1;
  pthread_cond_t cond2;
  int cont1;
  int cont2;
} s = {PTHREAD_MUTEX_INITIALIZER,
```

```
       PTHREAD_COND_INITIALIZER,
       PTHREAD_COND_INITIALIZER};

pthread_t thread_id[2];       /* thread id                                    */

void sys_err(char *s)
{
  perror(s);
  exit(-1);
}

/* codice dei due thread*/
void *thread1(void *arg)
{
  while (1) {
    pthread_mutex_lock(&s.mutex);
    if (s.cont1 >= s.cont2)
      pthread_cond_wait(&s.cond1, &s.mutex);
    else {
      s.cont1++;
      if (s.cont1 == s.cont2)
      pthread_cond_signal(&s.cond2);
    }
    printf("cont1 = %d    cont2 = %d\n", s.cont1, s.cont2);
    pthread_mutex_unlock(&s.mutex);
  }
}

void *thread2(void *arg)
{
  while (1) {
    pthread_mutex_lock(&s.mutex);
    if (s.cont2 > s.cont1)
      pthread_cond_wait(&s.cond2, &s.mutex);

    s.cont2 += 3;
    pthread_cond_signal(&s.cond1);

    pthread_mutex_unlock(&s.mutex);
  }
}

int main()
{

  s.cont1 = 0;
  s.cont2 = 0;

  thread_id[0] = pthread_create(&(thread_id[0]), 0, thread1, 0);
  thread_id[1] = pthread_create(&(thread_id[0]), 0, thread2, 0);

  while (1);
}
```

Esame del 13/09/01 (Solo soluz Unix)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>

void sys_err(char *str)
{
  perror(str);
  exit(-1);
}

struct data {
  char pipename[12];
  int num;
};

#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

int main()
{
  int fifo_server, fifo_client;
  int fifo2;
  int len;
  int res;
  struct data req;

  if (mkfifo("FIFOServer", FILE_MODE) < 0) {
    if (errno != EEXIST)
      sys_err("errore nella creazione della fifo");
  }

  if ((fifo_server = open("FIFOServer", O_RDONLY)) < 0)
    sys_err("Non riesco ad aprire la fifo del server");
  if ((fifo2 = open("FIFOServer", O_WRONLY)) < 0)
    sys_err("Non riesco ad aprire la fifo del server");

  printf("Il server e' pronto!\n");

  while (1) {
    len = read(fifo_server, &req, sizeof(struct data));
    if (len < sizeof(struct data))
      sys_err("Wrong data size");

    printf("SERVER: Dati ricevuti!!\n");

    fifo_client = open(req.pipename, O_WRONLY);
    res = req.num % 2;
    len = write(fifo_client, &res, sizeof(int));
    close(fifo_client);
  }
}
```

Esame del 04/10/01 (No Ancilotti)

UNIX

Scrivere un programma il cui unico scopo e' forkare un certo numero di processi in una struttura ad albero, nel modo seguente:

- Il processo padre forka due figli, e subito dopo si mette in attesa che essi terminino;
- ricorsivamente, ognuno dei figli forka altri due figli, e si mette in attesa della loro terminazione.
- La ricorsione termina quando il numero COMPLESSIVO di processi e' superiore alla variabile NPROC (definita a priori): in questo caso ognuno dei figli, invece di forkare, stampa la stringa "fatto!" sullo schermo ed esce.

DOMANDA FACOLTATIVA: quante volte viene stampata la stringa "fatto!" e quanti processi esattamente vengono creati complessivamente?

SOLUZIONE UNIX

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>

#define N_PROC     12

void child(int lev, int tot)
{
  int i;
  int ch;

  printf("lev --> %d /// tot --> %d\n", lev, tot);
  if (tot >= N_PROC) {
    printf("fatto!\n");
    exit(0);
  }

  if ((ch = fork()) == 0) {
    child(lev*2, lev*2 + tot);
    exit(0);
  }

  if ((ch = fork()) == 0) {
    child(lev*2, lev*2 + tot);
    exit(0);
  }

  wait(0);
  wait(0);
}

int main()
{
  child(1,1);
}
```

Esame del 11/01/02 (Solo traccia Unix)

UNIX

Un programma concorrente e' composto da 2 thread e da una risorsa condivisa che consiste della seguente struttura dati:

```
struct coppia {
  int a;
  int b;
} dato;
```

Tutti e due i thread hanno lo stesso codice, che e' il seguente:

```
void *thread(void *arg)
{
  int c,d;

  while (1) {
    /* prologo */
    ...

    c = dato.a + dato.b;
    d = dato.a - dato.b;
    dato.a = c;
    dato.b = d;
    printf("a = %d, b = %d\n", dato.a, dato.b);

    /* epilogo */
    ...
  }
}
```

La funzione main() crea i 2 thread e poi li coordina, facendo in modo che vada in esecuzione prima il thread1 e poi il thread2, poi ancora il thread1 e poi il thread2, e cosi' via all'infinito.

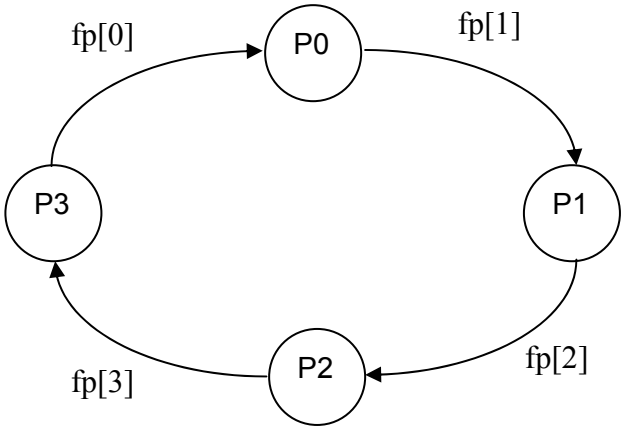
Per far questo, ogni thread nel prologo controlla che una condizione sia verificata, e nell'epilogo comunica al main() di aver terminato il suo compito.

Il candidato deve scrivere il codice del prologo e dell'epilogo e il codice del main in maniera che tutto funzioni correttamente.

SOLUZIONE UNIX

Esame del 31/01/02 (Solo traccia Unix)

UNIX
Un programma concorrente è composto da 4 processi, P0, P1, P2 e P3. I processi sono tutti collegati fra loro



tramite delle pipe, come mostrato in figura: il processo P0 legge dalla pipe fp[0] e scrive sulla pipe fp[1], il processo P1 legge dalla pipe fp[1] e scrive sulla pipe fp[2], e così via.

Ognuno dei processi compie un ciclo infinito in cui svolge i seguenti passi:

a) si mette in attesa di ricevere un intero sulla pipe di ingresso;
b) stampa a video la scritta "Sono passati xx minuti e yy secondi", dove xx e yy sono degli opportuni contatori;
c) si blocca per 15 secondi (per far questo usa la primitiva alarm())
d) incrementa opportunamente i contatori;
e) manda un intero qualsiasi sulla pipe di uscita;

L'output del programma è quindi:

Sono passati 0 minuti e 0 secondi
Sono passati 0 minuti e 15 secondi
Sono passati 0 minuti e 30 secondi
Sono passati 0 minuti e 45 secondi
Sono passati 1 minuti e 0 secondi
….

Il candidato deve scrivere;
- il programma main in cui vengono create le pipe, installato l'handler, e forkati i processi;
- il codice dei processi.

SOLUZIONE UNIX

Esame del 14/02/02 (Solo traccia Unix)

UNIX
Un programma multi-threaded è composto da alcuni thread che, nel corso della loro esecuzione, accedono ripetutamente a una mailbox che può contenere al massimo 3 messaggi. La struttura dati che implementa la mailbox è la seguente:

```
struct Msg {…};

struct Mailbox {
  struct Msg array[3];
  /* (1) strutture dati per implementare la
     sincronizzazione */
  …
} mbox;
```

Inoltre, sono definite due funzioni per accedere alla variabile *mbox*: la funzione

```
void sendMsg(struct Msg *m)
{
  // (2)
}
```

copia il messaggio contenuto nella zona di memoria puntata da *m* nella prima posizione libera dentro *mbox*; se le 3 posizioni sono tutte occupate, il thread che ha invocato la *sendMsg* deve bloccarsi, e sarà risvegliato solo quando una posizione libera si sarà resa disponibile.

La funzione

```
void receiveMsg(struct Msg *m)
{
  // (2)
}
```

copia il primo messaggio disponibile nella zona di memoria puntata da *m*. Se non c'è nessun messaggio disponibile, il thread che ha chiamato la funzione *receiveMsg* deve bloccarsi, e sarà risvegliato non appena un messaggio sarà disponibile.

Il candidato deve:
- completare la dichiarazione della struttura dati Mailbox con le variabili necessarie ad assicurare una corretta sincronizzazione del programma;
- Il codice delle funzioni *sendMsg()* e *receiveMsg()*;

NB: per copiare una struttura dati a partire da un puntatore è possibile usare la funzione

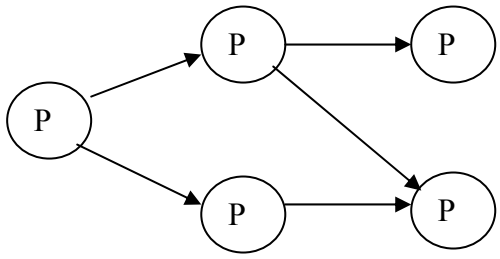
```
int memcpy(void *dest, void *source, size_t nbytes);
```

SOLUZIONE UNIX

Esame del 17/07/02 (Solo traccia Unix)

UNIX

Un processo genera 5 processi figli, indicati in figura con P1…P5, e poi si mette in attesa che essi terminino. I processi figli sono in relazione fra di loro tramite il seguente grafo di precedenza:



Ognuno dei processi è indicato in figura da un nodo: un arco dal nodo P_i al nodo P_j indica che vi è una comunicazione sincrona fra i due processi tramite *pipe*. In particolare, ogni processo P_j esegue un ciclo **for** per 3 volte, in cui come prima cosa aspetta sulle *pipe* in ingresso, quindi invoca la funzione fun_j(), e infine manda un dato qualunque sulle *pipe* in uscita. Il processo P1 non aspetta niente in ingresso, mentre i processi P4 e P5 non mandano niente in uscita.

Scrivere un programma UNIX che realizzi lo schema sopra descritto.

Domanda facoltativa: supponendo che ognuna delle funzioni fun_j() abbia un tempo di elaborazione di un secondo, e che il ritardo di comunicazione fra due processi sia trascurabile, qual'è il tempo minimo fra la prima esecuzione di P1 e l'ultima esecuzione di P5 in un sistema uni-processore? E in un sistema bi-processore?

SOLUZIONE UNIX

Esame del 12/09/02

PARTE PRIMA

In un sistema organizzato secondo il modello a memoria comune *N* diversi processi (*P1*, *P2*, …, *PN*) competono per l'uso di *M* risorse equivalenti (*R1*, *R2*, …., *RM*) allocate dinamicamente ai processi richiedenti mediante un gestore *G*. Ogni processo può chiedere (e successivamente rilasciare) una sola risorsa alla volta o due risorse contemporaneamente tramite le seguenti funzioni:

richiesta1, *richiesta2*, *rilascio1* e *rilascio2*

La politica di allocazione delle risorse è la seguente:

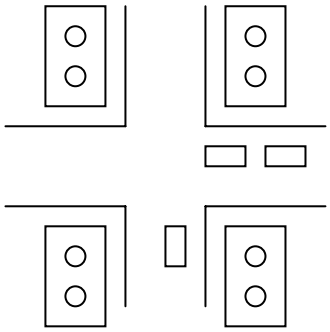
a) le richieste di due risorse devono essere privilegiate nei confronti delle richieste di una sola risorsa. In particolare nessuna richiesta di una sola risorsa può essere accolta se ci sono richieste pendenti di due risorse.
b) fra i processi richiedenti deve essere prevista una priorità: privilegiare *P1* su *P2*, *P2* su *P3* , ecc.

Realizzare il codice del gestore *G* tramite il meccanismo semaforico.

PARTE SECONDA

Tramite un programma concorrente multi-thread si vuole realizzare la simulazione di un incrocio stradale regolato da un semaforo. Le automobili sono rappresentate da dei thread, mentre il semaforo è rappresentato dalla struttura dati semaforo:

```
struct semaforo {
      ...
};
```



ed è regolato tramite un thread manager, che si occupa di regolare il traffico. Il semaforo può assumere soltanto i valori rosso o verde.

Quando un thread arriva a uno dei quattro lati dell'incrocio, chiama la funzione:

```
  void arriva(int lato);
```

dove lato può assumere i valori 0,1,2,3. Se il semaforo dal lato in cui è arrivata l'automobile è rosso, la funzione si blocca in attesa che diventi verde, altrimenti ritorna immediatamente. Quando l'auto lascia l'incrocio, il thread chiama la funzione:

```
  void lascia();
```

Quindi, la struttura di ogni thread "automobile" è la seguente:

```
void *macchina(void *d)
{
      while (1) {
            ...
            arriva(lato);
            lascia();
            ...;
      }
}
```

Scrivere la struttura dati semaforo, le funzioni arriva e lascia e il thread manager, rispettando i seguenti vincoli:

a) Il semaforo passa da rosso a verde e viceversa ogni 5 secondi;
b) Il semaforo non può cambiare da rosso a verde se ci sono auto che impegnano l'incrocio.

SOLUZIONE
Parte Prima

```
Class sistema
{
        semaforo mutex; (1)
        bool stato1[N]; (false)
        bool stato2[N]; (false)
        int c1,c2; (0)
        bool ris[M]; (true)
        int cont; (0)
        semaforo  priv[N]; (0)

Public:
        void ric1(int p, int&ind)
        {
                if( (cont==M) || (c2>0))
                {
                        c1++;
                        stato1[p]=true;
                        mutex.signal();
                        stato1[p]=false;
                        c1- -;
                }
                int i=0;
                while(!ris[i])
                        i++;
                ind=i;
                cont- -;
                if(c1>0)
                {
                        i=0;
                        while(!stato1[i])
                                i++;
                        priv[i].signal();
                }
                else
                        mutex.signal();
        }

        void ric2(int p, int& ind1      , int& ind2)
        {
                mutex.wait();
                if(cont>M-2)
                        {
                                c2++;
                                stato2[p]=true;
                                mutex.signal();
                                priv[p].wait();
                                stato2[p]=false ;
                                c2- -;
                        }
```

```
        int i=0;
        while(!ris[i])
                i++;
        ind1=i;
        i+=1;
        while(!ris[i])
                i++;
        ind2=i;
        cont=cont-2;
        if(c1>0)
                {
                        i=0;
                        while(!stato1[i])
                                i++;
                        priv[i].signal();
                }
        else
                mutex.signal();
}
void RiL1(int ind)
{
        mutex.wait();
        int i;
        ris[ind]=true;
        cont++;
        if( (cont>=(M-2))   &&  (c2>0) )
                {
                        i=0;
                        while(!stato2[i])
                                i++;
                        priv[i].signal();
                }
        else
                if( (c1>0)  &&  (!c2>0) )
                        {
                                i=0;
                                while(!stato1[i])
                                        i++;
                                priv[i].signal();
                        }
                else
                        mutex.signal();
}

void RiL2(int ind1, int ind2)
{
        mutex.wait();
        int i;
        ris[ind1]=true;
        ris[ind2]=true;
        cont+=2;
```

```
            if(c2>0)
                {
                    i=0;
                    while(!stato2[i])
                        i++;
                    priv[i].signal();
                }
            else
                if(c1>0)
                    {
                        i=0;
                        while(!stato1[i])
                            i++;
                        priv[i].signal();
                    }
                else
                    mutex.signal();
        }
}
```

Parte seconda

Sol con semafori

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

struct semaforo {
  pthread_mutex_t m;
  sem_t coda[2];
  sem_t sm;
  int bm;
  int bloc[2];
  int rosso[2];
  int impegno;
} s = {PTHREAD_MUTEX_INITIALIZER};

void arriva(int lato)
{
  pthread_mutex_lock(&s.m);
  lato = lato %2;

  printf("Arrivo lato: %d\n", lato);
  if (s.rosso[lato]) {
    printf("Ho trovato rosso!\n");
    s.bloc[lato]++;
    pthread_mutex_unlock(&s.m);
    sem_wait(&s.coda[lato]);
    pthread_mutex_lock(&s.m);
    printf("Riparto\n");
  }
  printf("In mezzo all'incrocio\n");
  s.impegno++;
  pthread_mutex_unlock(&s.m);
}
```

```
void lascia()
{
  printf("Lascio l'incrocio\n");
  pthread_mutex_lock(&s.m);
  s.impegno--;
  if (s.impegno==0 && s.bm) sem_post(&s.sm);
  pthread_mutex_unlock(&s.m);
}

void* manager(void *dato)
{
  int turno = 0;
  s.rosso[0] = 0;
  s.rosso[1] = 1;
  s.bm = 0;
  s.bloc[0] = 0;
  s.bloc[1] = 0;
  sem_init(&s.coda[0], 0, 0);
  sem_init(&s.coda[1], 0, 0);
  sem_init(&s.sm, 0, 0);
  while(1) {
    sleep(5);
    printf("MANAGER: Cambio!!\n");
    pthread_mutex_lock(&s.m);
    s.rosso[turno] = 1;
    if (s.impegno > 0) {
      printf("Aspetto che le macchine escano...\n");
      s.bm = 1;
      pthread_mutex_unlock(&s.m);
      sem_wait(&s.sm);
      pthread_mutex_lock(&s.m);
      s.bm = 0;
    }
    turno = (turno + 1) % 2;

    printf("Adesso %d e %d sono rossi, %d e %d sono verdi\n", turno + 1, (turno
+ 3) % 4, turno, turno + 2);
    s.rosso[turno] = 0;
    while (s.bloc[turno]) {
      sem_post(&s.coda[turno]);
      s.bloc[turno]--;
    }
    pthread_mutex_unlock(&s.m);
  }
}

void * macchina(void *d)
{
  while (1) {
    arriva(rand() % 4);
    sleep(1);
    lascia();
    sleep(rand() % 5);
  }
}

int main()
{
  int i;
  pthread_t th[10];
  pthread_t man;

  pthread_create(&man, 0, manager, 0);
```

```
  sleep(1);

  for (i=0; i<10; i++) {
    pthread_create(&(th[i]), 0, macchina, 0);
  }

  while(1);
}
```

soluzione con variabili condition

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

struct semaforo {
  pthread_mutex_t m;
  pthread_cond_t coda0;
  pthread_cond_t coda1;
  pthread_cond_t sm;
  int bm;
  int bloc[2];
  int rosso[2];
  int impegno;
} s = {PTHREAD_MUTEX_INITIALIZER,
       PTHREAD_COND_INITIALIZER,
       PTHREAD_COND_INITIALIZER,
       PTHREAD_COND_INITIALIZER};

void arriva(int lato)
{
  pthread_mutex_lock(&s.m);
  lato = lato % 2;

  printf("Arrivo lato %d\n", lato);
  if (s.rosso[lato]) {
    printf("Ho trovato rosso!\n");
    s.bloc[lato]++;
    if (lato == 0) pthread_cond_wait(&s.coda0, &s.m);
    else pthread_cond_wait(&s.coda1, &s.m);
    printf("Riparte\n");
  }
  printf("In mezzo all'incrocio\n");
  s.impegno++;
  pthread_mutex_unlock(&s.m);
}

void lascia()
{
  printf("Lascia l'incrocio\n");
  pthread_mutex_lock(&s.m);
  s.impegno--;
  if (s.impegno==0 && s.bm) pthread_cond_signal(&s.sm);
  pthread_mutex_unlock(&s.m);
}

void* manager(void *dato)
{
  int turno = 0;
  s.rosso[0] = 0;
```

```
  s.rosso[1] = 1;
  s.bm = 0;
  s.bloc[0] = 0;
  s.bloc[1] = 0;
  while(1) {
    sleep(5);
    printf("MANAGER: Cambio!!\n");
    pthread_mutex_lock(&s.m);
    s.rosso[turno] = 1;
    if (s.impegno > 0) {
      printf("MANAGER: Aspetto che le macchine escano...\n");
      s.bm = 1;
      pthread_cond_wait(&s.sm, &s.m);
      s.bm = 0;
    }
    turno = (turno + 1) % 2;

    printf("MANAGER: adesso %d e %d sono rossi, %d e %d sono verdi\n",
         turno + 1, (turno + 3) % 4, turno, turno + 2);
    s.rosso[turno] = 0;
    while (s.bloc[turno]) {
      if (turno == 0) pthread_cond_signal(&s.coda0);
      else pthread_cond_signal(&s.coda1);
      s.bloc[turno]--;
    }
    pthread_mutex_unlock(&s.m);
  }
}

void * macchina(void *d)
{
  while (1) {
    arriva(rand() % 4);
    sleep(1);
    lascia();
    sleep(rand() % 5);
  }
}

int main()
{
  int i;
  pthread_t th[10];
  pthread_t man;

  pthread_create(&man, 0, manager, 0);

  sleep(1);

  for (i=0; i<10; i++) {
    pthread_create(&(th[i]), 0, macchina, 0);
  }

  while(1);
}
```

Esame del 13/01/03

Parte I

Alcuni processi mittenti inviano periodicamente messaggi di un tipo mes (che si suppone già definito), a due processi riceventi (R1 e R2). 1 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 RI 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 monitor, 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.

Parte II

In un sistema client server, il server è un processo multi-thread che riceve richieste da parte dei client su una FIFO di nome "serverFIFO". Le richieste sono del seguente tipo:

```
struct rich {
    int dato;
    char nomefifo[30];
};
```

Il server, appena ricevuta una richiesta, crea un thread che gestirà la risposta e si mette in attesa della richiesta successiva. Naturalmente, ci può essere più di un thread alla volta in esecuzione, ognuno gestirà una richiesta diversa. Assumere che si possano creare al massimo 5 thread contemporaneamente, dopodichè il server si mette in attesa che uno di essi finisca.

Ogni thread, il cui codice si trova nella funzione

```
void *mythread(void *);
```

esegue i seguenti passi:

- Prende il dato di tipo struct rich che gli ha passato il server;
- Invoca la funzione

  ```
  int fun(int dato);
  ```

  per ottenere la risposta da rimandare indietro al client
- Apre la FIFO di risposta, gli scrive sopra il dato ottenuto e la chiude subito dopo;
- Termina l'esecuzione.

Il candidato deve:

1. Scrivere la struttura per passare i dati dal server ai vari thread che vengono via via creati e per sincronizzare i thread con il processo server.
2. Scrivere il codice della funzione mythread()
3. Scrivere il codice del server (funzione main()).

SOLUZIONE PARTE PRIMA
Punto a)

```
Monitor Mailbox
{
        mes buffer[N];
        int primo1, primo2, ultimo ; (0)
        bool ricevuto[N];(false)
        condition pieno1, pieno2; vuoto;
        int quanti1, quanti2; (0)

Public:
        void send(mes m)
        {
                if(quanti1==N || quanti2==N)
                        vuoto.wait();
                buffer[ultimo]=m;
                ultimo=(ultimo+1)%N;
                quanti1++;
                quanti2++;
        }

        void receive1(mess& m)
        {
                if(quanti1==0)
                        piena1.wait();
                m=buffer[primo1];
                quanti1- -;
                if(ricevuto[primo1]= !ricevuto[primo1];
                primo1=(primo1+1)%N;
        }

        void receive2(mess& m)
        {
                if(quanti2==0)
                        piena2.wait();
                m=buffer[primo2];
                quanti2- -;
                if(ricevuto[primo2]= !ricevuto[primo2];
                primo2=(primo2+1)%N;
        }
}
```

punto b)

```
Monitor Mailbox
{
        mes buffer[N];
        int primo1, primo2, ultimo ; (0)
        condition pieno1, pieno2; vuoto;
        int quanti1, quanti2; (0)
```

```
Public:
        void send(mes m)
        {
                if(quanti1+quanti2==N)
                        vuoto.wait();
                buffer[ultimo]=m;
                ultimo=(ultimo+1)%N;
                quanti1++;
                pieno1.signal();
        }

        void receive1(mes& m)
        {
                if(quanti1==0)
                        pieno1.wait();
                m=buffer[primo1];
                primo1=(primo+1)%N;
                quanti1- -;
                quanti2++;
                pieno2.signal();
        }

        void receive2(mes& m)
        {
                if(quanti2==0)
                        pieno2.wait();
                m=buffer[primo2];
                primo2=(primo2+1)%N;
                quanti2- -;
                vuoto.signal();
        }
};
```

//**nota:** nel punto è stata adottata la semantica “signal&urgent”

SOLUZIONE PARTE SECONDA

SERVER

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>



struct rich {
  int dato;
  char nomefifo[30];
};

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
struct rich array[5];
int libero[5];
int occupati = 0;

int fun(int a)
{
  long int i;
  for (i = 0; i < 10000000; i++);
  return 0;
}

void *mythread(void *arg)
{
  int index = (int)arg;
  int risp = fun(array[index].dato);

  int fdes = open(array[index].nomefifo, O_WRONLY);
  write(fdes, &risp, sizeof(int));
  printf("Dato inviato\n");
  close(fdes);

  pthread_mutex_lock(&mutex);
  libero[index] = 1;
  occupati--;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
  return 0;
}

int main()
{
  int fifodes;
  int i;
  pthread_t tid;

  for (i=0; i<5; i++) libero[i] = 1;
  mkfifo("serverFIFO", S_IRUSR | S_IWUSR | S_IXUSR);

  fifodes = open("serverFIFO", O_RDWR);

  printf("Server pronto!!\n");

  while (1) {
    pthread_mutex_lock(&mutex);
    for (i=0; i<5; i++) if (libero[i]) break;
    libero[i] = 0;
    pthread_mutex_unlock(&mutex);

    read(fifodes,&array[i],sizeof(struct rich));
    printf("Dato ricevuto: %d\n", array[i].dato);

    pthread_mutex_lock(&mutex);
    occupati++;

    pthread_create(&tid, 0, mythread, (void *)i);
    if (occupati > 4) pthread_cond_wait(&cond,&mutex);
    pthread_mutex_unlock(&mutex);

  }
}
```

CLIENT

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

struct rich {
  int dato;
  char fifoname[30];
} data;

int main()
{
  char fifoname[10];
  int i;
  int fdes = open("serverFIFO", O_WRONLY);

  for (i = 0; i<20; i++) {
    data.dato = i;
    sprintf(data.fifoname, "clientFIFO%d",i);
    mkfifo(data.fifoname, S_IRUSR | S_IWUSR | S_IXUSR);
    open(data.fifoname, O_RDWR);
    write(fdes, &data, sizeof(data));
  }

  getchar();
}
```

Esame del 31/01/03 (no soluz lipari)

PARTE PRIMA

Ognuno dei tre processi PI, P2 e P3 invia messaggi di un tipo noto mes in una mailbox di dimensione N.
I messaggi sono ricevuti da uno qualunque dei tre processi P4, P5 e P6.
Quando un processo mittente invia un messaggio, se ci sono processi riceventi m attesa, sveglia uno di loro in base alla seguente priorità: P4 da privilegiare su P5 e P5, su P6. Analogamente quando un ricevente riceve un messaggio, se ci sono processi mittenti in attesa, sveglia uno di loro in base alla seguente priorità: P1 da privilegiare su P2 e P2 su P3. Utilizzando il meccanismo delle chiamate remote (entry call e accept), scrivere il codice di un processo server che implementa la mailbox e indicare le operazioni che devono eseguire sia i mittenti che i riceventí rispettivamente per inviare e ricevere messaggi.

PARTE SECONDA

Un programma multi-processo può essere pensato come una serie di blocchi messi in serie. La struttura dì un blocco è mostrata nella figura A:

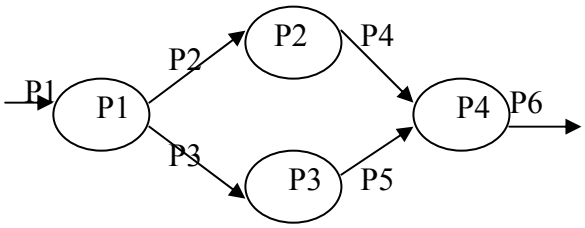


Figura A

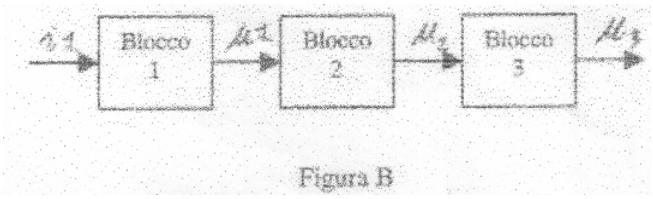


Figura B

Le palle rappresentano i processi, mentre le frecce rappresentano le pipe che connettono ì processi. Il codice dei-processi si trova nelle funzioni processol(int fdin, int fdout1, int fdout2), processol(int fdin, int fdout) processol(int fdin, int fdout) e processol(int fdinl, int fdin2, int fdout), rispettivamente. Il candidato deve scrivere la funzione void crea_blocco(int *b in, int *b out);

che crea i 4 processi tramite fork, crea le 6 pipe, e invoca le rispettive funzioni processoX() per ogni processo. La funzione crea_blocco deve restituire in b _in e b_out i descrittori delle pipe entranti e uscenti dal blocco, rispettivamente.

Successivamente, scrivere la funzione main() che, dopo aver invocato per 3 volte la funzione crea_blocco() per creare 3 blocchi, connette le pipe di entrata e dì uscita di ogni blocco come mostrato in figura B .
PS: Non bisogna scrivere le funzioni processoX() !! Assumere che tali funzioni non ritornino mai.

SOLUZIONE PRIMA PARTE

\\ mes è il tipo noto

```
Process Server
{
        entry {invia [3] (mess in x) , ricevi [3] (mess out x),
                rich_inv (int in proc), rich_ric (int in proc)}
        mess mailbox[N];
```

```
int primo, ultimo; (null)
int count; (0)
mess m;
int p;
bool bloc_ric [3]; (false)
bool bloc_inv [3]; (false)
int sosp_inv, sosp_ric; (0)

do
        ■ accept rich_inv(int in proc) {p=proc}→
                        {if (count<N)
                                        {
                            accept invia[p] (mess in x)
                                                        {m=x;}
                                        mailbox[ultimo]= m;
                                        ultimo=(ultimo+1)%N;
                                        count++;
                                        if (sosp_ric>0)
                                                {
                                                        int i     =0;
                                                        while(!bloc_ric[i] i++;
                                                        accept ricevi (mess out x)
                                                                {x=mailbox[primo];}
                                                        bloc_ric[i]=false;
                                                        primo=(primo+1)%N;
                                                        count--;
                                                }
                                else
                                        {
                                                bloc_inv[p]=true;
                                                sosp_inv++;
                                        }
                                }


        ■ accept rich_ric(int in proc) {p=proc}→
                        {if (count>0)
                                        {
                            accept ricevi[p] (mess out x)
                                                        {x=mailbox[primo];}
                                        primo=(primo+1)%N;
                                        count--;
                                        if (sosp_inv>0)
                                                {
                                                        int i     =0;
                                                        while(!bloc_inv[i] i++;
                                                        accept invia (mess in x)
                                                                {m=x;}
                                                        mailbox[ultimo]= m;
                                                        ultimo=(ultimo+1)%N;
                                                        bloc_inv[i]=false;
                                                        count++;
                                                }
                                else
```

```
                                        {
                                                bloc_ric[p]=true;
                                                sosp_ric++;
                                        }
                                }
```

\\ lato client:

```
process client
{       mess messaggio;
        :
      server.rich_inv(proc);
        server.invia[proc] (messaggio);
        :
        server.rich_ric(proc);
        server.ricevi[proc](messaggio);
        :
}
```

Esame del 06/06/03 (no soluz Ancilotti)

PARTE PRIMA

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 i testaA Altri processi sono preposti a segnalare l'occorrenza dell'evento invocando la funzione segna1aA 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 P], 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.

PARTE SECONDA

Un programma multithread è composto da 3 thread "worker", che ciclicamente svolgono alcuni compiti, più un quarto thread "rnanager" che si occupa della schedulazione e della sincronizzazione. Ognuno dei thread worker esegue un ciclo infinito durante il quale effettua le seguenti operazioni:

invoca la funzione aspetta() che blocca il thread in attesa del via da parte dei thread manager;
esegue la propria funzionalità;

invoca la funzione segnala() con la quale comunica al thread manager di aver finito il proprio lavoro.

Il thread manager ciclicamente attiva un thread workers alla volta, e si mette in attesa della sua segnalazione; quindi attiva il thread successivo, e si mette in attesa della sua segnalazione, e così via, a ciclo. Il thread manager comunica con i thread worker tramite memoria comune. 1 thread usano variabili condition per bloccarsi sulle varie condizioni.

Il candidato deve scrivere le strutture dati necessarie alla sincronizzazione workers-manager, le funzioni aspetta() e segnala(), e il thread manager.

Domanda facoltativa. Descrivere cosa cambia nel caso in cui si debba programmare in una ambiente multiprocesso, in cui il thread manager viene sostituito da un processo manager e i thread worker vengono sostituiti da processi figli del processo manager.

SOLUZIONE UNIX

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t c[3] = {PTHREAD_COND_INITIALIZER,
                       PTHREAD_COND_INITIALIZER,
                       PTHREAD_COND_INITIALIZER};

pthread_cond_t b = PTHREAD_COND_INITIALIZER;

int wblock[3] = {0,0,0};

void aspetta(int i)
{
  pthread_mutex_lock(&m);

  if (wblock[i] == 0) {wblock[i] = 1; pthread_cond_wait(&c[i],&m);}
  wblock[i] = 0;

  pthread_mutex_unlock(&m);
}

void segnala(int i)
{
  pthread_cond_signal(&b);
}

void *worker1(void *x)
{
  while(1) {
    aspetta(0);
    printf("Worker 1\n");
    segnala(0);
  }
}

void *worker2(void *x)
{
  while(1) {
    aspetta(1);
    printf("Worker 2\n");
    segnala(1);
  }
}

void *worker3(void *x)
{
  while(1) {
    aspetta(2);
    printf("Worker 3\n");
    segnala(2);
  }
}

void *manager(void *x)
{
  int i=0;

  while(1) {
    pthread_mutex_lock(&m);
    if (wblock[i] == 0) wblock[i] = 1;
    else pthread_cond_signal(&c[i]);
```

```
    pthread_cond_wait(&b,&m);

    pthread_mutex_unlock(&m);
    i = (i+1)%3;
  }
}

int main()
{
  pthread_t pman, pid[3];

  pthread_create(&pid[0], 0, worker1, 0);
  pthread_create(&pid[1], 0, worker2, 0);
  pthread_create(&pid[2], 0, worker3, 0);

  pthread_create(&pman, 0, manager, 0);

  for(;;);

}
```

Esame del 27/06/03 (NO Ancilotti)

UNIX

Un programma è composto da tre processi, A, B e C. Il processo A crea gli altri due processi, B e C, tramite la fork(). I tre processi comunicano tramite 2 pipe, p1 e p2. Il processo A, ciclicamente compie le seguenti operazioni:

- Estrae un numero intero s a caso tra 1 e 10
- Si sospende per s secondi
- Invia sulla pipe p1 il numero s

Il processo B attende in lettura sulla pipe p1 l'arrivo del numero s. Se il numero arriva prima di 5 secondi, allora si sospende per s secondi e poi lo invia al processo C tramite la pipe p2. Se allo scadere dei 5 secondi il numero non è ancora arrivato, il processo B invia immediatamente al processo C il numero 0 sulla pipe p2. Dopodichè in entrambi i casi si rimette in attesa sulla pipe p1.

Il processo C attende il numero sulla pipe p2 e lo stampa a video. Dopodichè si rimette in attesa del prossimo numero.

Implementare il codice del programma.

SOLUZIONE UNIX

```
#include <unistd.h>
#include <errno.h>
#include <signal.h>

int p1[2];
int p2[2];

void processoA();
void processoB();
void processoC();

void handler()
{
  printf("timeout -- \n");
}

int main()
{
  pipe(p1);
  pipe(p2);

  int ch;

  ch = fork();
  if (ch == 0) processoB();

  ch = fork();
  if (ch == 0) processoC();

  processoA();
}

void processoA()
{
  int s;
```

```
  while (1) {
    s = rand() % 10 + 1;

    sleep(s);
    write(p1[1], &s, sizeof(int));
  }
}

void processoB()
{
  int s;
  int c;

  struct sigaction n,v;

  n.sa_handler = handler;
  n.sa_flag = 0;
  n.sa_mask = 0;

  sigaction(SIGALRM, &n, &v);

  while(1) {
    alarm(5);
    c = read(p1[0], &s, sizeof(int));
    alarm(0);
    if (c <= 0) {
      s = 0;
      write(p2[1], &s, sizeof(int));
    }
    else {
      sleep(s);
      write(p2[1], &s, sizeof(int));
    }
  }
}

void processoC()
{
  int s;

  while(1) {
    read(p2[0], &s, sizeof(int));
    printf("%d\n", s);
  }
}
```

Esame del 17/07/03 (No Ancilotti)

UNIX

```
Un programma consiste di tre processi, un processo padre piu' due
processi figli. I due processi figli comunicano con il padre tramite
una pipe. Ognuno dei due processi figli effettua ciclicamente le
seguenti operazioni:

- manda al processo padre il proprio pid sulla pipe
- si mette in attesa di ricevere un segnale SIGUSR1 da parte del padre
- esegue la procedura proc() (non specificata)
- rimanda al processo padre un segnale SIGUSR1

Il padre esegue il seguente ciclo infinito: aspetta una richiesta da
uno dei figli sulla pipe manda un segnale SIGUSR1 al figlio
corrispondente aspetta di ricevere un segnale SIGUSR1 dal figlio
```

Scrivere il codice dei tre processi.

SOLUZIONE UNIX

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>


void figlio();
void handler(int signo);
int fd[2];
int flag;

int main()
{
  struct sigaction n,v;
  int ch;

  n.sa_handler = handler;
  n.sa_flags = 0;
  sigemptyset(&n.sa_mask);

  sigaction(SIGUSR1, &n, &v);

  pipe(fd);

  // fork dei due processi!
  ch = fork();
  if (ch == 0) figlio();

  ch = fork();
  if (ch == 0) figlio();

  for (;;) {
    read(fd[0], &ch, sizeof(int));

    flag = 0;
    kill(ch, SIGUSR1);

    while (flag == 0);
  }

}
```

```
void figlio()
{
  int n;
  int pid = getpid();
  int ppid = getppid();
  for(;;) {
    flag = 0;
    n = write(fd[1], &pid, sizeof(int));
    while (flag == 0);

    printf("Figlio %d\n", pid);
    sleep(rand()%3);
    printf("Finito!!\n");
    kill(ppid, SIGUSR1);
  }
}


void handler(int signo)
{
  flag = 1;
}
```

Esame del 11/09/03

PARTE PRIMA

In un sistema organizzato secondo il modello a memoria comune due risorse RA e RB sono allocate dinamicamente ai processi P], P2, ... Pn tramite un comune gestore G. Ogni processo può richiedere al a 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 priorítà: 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

PARTE SECONDA

Un sistema di prenotazioni viene realizzato attraverso un programma multi-thread. Nel sistema ci sono N thread utente il cui codice e' il seguente:

```
 void *utente(void *arg){
        int index = (int)arg;
        int k;
        for (k=O;k<5;k++) {
                prenotazione(index);
                servizio(index);
                fine_servizio();
        }
}
```

I thread utente realizzano un ciclo in cui, per prima cosa si prenotano per ricevere un servizio: nel caso ìn cui il servizio sia attualmente utilizzato da un altro thread, il thread in question si blocca in attesa che il servizio sia disponibile. Quindi, invocano la routine servizio0, e infine comunicano la fine del servizio invocando la funzione fine-servizio.

Il candidato deve realizzare le strutture dati necessarie alla gestione della concorrenza e le funzioni prenotazione() e fine_servizio() in modo che :

- massimo un thread alla volta possa essere dentro la funzione servizio
- nel caso in cui ci siano piu' thread bloccati in attesa del servizio, venga servito per prima uno di quelli
  che ha indice dispari. Solo se non ci sono thread con indice dispari in attesa puo' essere servito un
  thread con indice pari.

Scrivere inoltre la funzione maino che inizializza le strutture dati e crea i thread.

SOLUZIONE Parte Prima

```
enum risorsa {a,b};


class gestore
{
        semaforo mutex; (1)
        semaforo sa,sb,sq,s2; (0)
        int ca,cb,cq,c2; (0)
        bool la,lb;(true)
Public:
        void RicA()
        {
                mutex.wait();
                if (!la)
                {
                        ca++;
                        mutex.signal();
                        sa.wait();
                        ca--;
                }
                ra=false;
                if (lb && cb>0)
                        sb.signal();
                else
                        mutex.signal();
        }

        void RicB()
        {
                mutex.wait();
```

```
        if (!lb)
        {
                cb++;
                mutex.signal();
                sb.wait();
                cb--;
        }
        rb=false;
        mutex.signal();
}

void RicQ (risorsa &ris)
{
        mutex.wait();
        if (!la && !lb)
        {
                cq++;
                mutex.signal();
                sq.wait();
                cq--;
        }
        if (la)
        {
                la=false;
                ris=a;
                if (lb)
                {
                        if (cq>0)
                                sq.signal();
                        else
                                if (cb>0)
                                        sb.signal();
                                else
                                        mutex.signal();
                else
                        mutex.signal();

        if (lb)
        {
                lb=false;
                ris=b;
                if (la)
                {
                        if (cq>0)
```

```
                        sq.signal();
                else
                        if (ca>0)
                                sa.signal();
                        else
                                mutex.signal();
        else
                mutex.signal();

}

void Ric2 ()
{
        mutex.wait();
        if (!la || !lb)
        {
                c2++;
                mutex.signal();
                s2.wait();
                c2--;
        }
        la=false;
        lb=false;
        mutex.signal();
}

void RilA()
{
        mutex.wait();
        la=true;
        if(cq>0)        sq.signal();
        else
                if(ca>0)
                        sa.signal();
                else
                        if(c2>0 && lb=true)
                                s2.signal();
                        else
                                mutex.signal();
}

void RilB()
{
        mutex.wait();
```

```
        lb=true;
        if(cq>0)      sq.signal();
        else
                if(cb>0)
                        sb.signal();
                else
                        if(c2>0 && la=true)
                                s2.signal();
                        else
                                mutex.signal();
    }

    void RilQ (risorsa ris)
    {
        mutex.wait();
        if (ris==a)
                la=true;
        else
                lb=true;
        if(cq>0)
                sq.signal();
        else
                if(ca>0 && ris==a)
                        sa.signal();
                else
                        if(cb>0 && ris==b)
                                sb.signal();
                        else
                                if((ris==b && la && c2>0)||(ris==a && lb &&
c2>0))
                                        s2.signal();
                                else
                                        mutex.signal();
    }

    void Ril2 ()
    {
        mutex.wait();
        la=true;
        lb=true;
        if(cq>0)
                sq.signal();
        else
                if (cb>0)
```

```
                    sb.signal();
                else
                    if (c2>0)
                        s2.signal();
                    else
                        mutex.signal();
        }
}
```

SOLUZIONE UNIX

```
#include <pthread.h>
#include <stdio.h>

//#define ATTESA_ATTIVA(x) {long long int j=0 ; for (j=0;
j<(x)*1000000001;j++);}

#define ATTESA_ATTIVA(x) {sleep(x);}

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int libero = 1;
pthread_cond_t cond_pari = PTHREAD_COND_INITIALIZER;
int num_pari = 0;
pthread_cond_t cond_dispari = PTHREAD_COND_INITIALIZER;;
int num_dispari = 0;

void prenotazione(int i)
{
  printf("Thread %i si sta prenotando...\n", i);
  pthread_mutex_lock(&mutex);
  if (libero) libero = 0;
  else {
    if (i%2) {
      num_dispari++;
      pthread_cond_wait(&cond_dispari, &mutex);
      num_dispari--;
    }
    else {
      num_pari++;
      pthread_cond_wait(&cond_pari, &mutex);
      num_pari--;
    }
  }
  pthread_mutex_unlock(&mutex);
}

void servizio(int i)
{
  printf("Thread %d viene servito...\n", i);
  ATTESA_ATTIVA(rand()%4);
  printf("Fatto!\n");
}

void fine_servizio(void)
{
  pthread_mutex_lock(&mutex);
  if (num_dispari>0) pthread_cond_signal(&cond_dispari);
  else if (num_pari>0) pthread_cond_signal(&cond_pari);
  else libero=1;
  pthread_mutex_unlock(&mutex);
```

```
}

void *utente(void * arg)
{
  int index = (int)arg;
  int k;

  for (k=0;k<5;k++) {
    prenotazione(index);
    servizio(index);
    fine_servizio();
    ATTESA_ATTIVA(rand()%4);
  }
}

#define N 4

int main(void)
{
  pthread_t th[N];
  int i;

  for (i=0; i<N; i++) {
    pthread_create(&th[i], 0, utente, (void *)i);
  }

  for (i=0; i<N; i++) {
    pthread_join(th[i], 0);
  }
}
```

Esame del 18/09/2003 (no soluz lipari)

Parte Prima

Utilizzando 1 meccanismo delle chiamate remote di procedure scrivere il codice di un processo server S che fomisce, ad un insieme di processi clienti i servizi relativi all'allocazione di tre risorse RO, RI ed R2. 1 processi clienti possono richiedere al server o l'allocazione di una qualunque risorsa o l',allocazione di due risorse contemporaneamente fra quelle disponibili. Quando la (o le) risorse non sono più necessarie ogni cliente chiede al server di rilasciare la o le risorse precedentemente richiesto.

- Indicare, quali sono le cntry che il server deve mettere a disposizione dei clienti.

- Scrivere il codice relativo al processo server nell'ipotesi che lo stesso dia priorità alle richieste singole rispetto alle richieste di due risorse.

- Indicare quali operazioni deve eseguire un processo cliente per richiedere e rilasciare, rispettivamente, una , due risorse.

Parte Seconda

In un sistema multi-thread ci sono M thread che accedono a N risorse condivise equivalenti fra loro. Ogni thread puo' richiedere, tramite l'uso della funzione richiedi(), l'uso esclusivo di un numero variabile di risorse:

int *richiedi(int index, int num);

dove index e' l'indice del thread (da 0 a MA) e num e' il numero di risorse che il thread vuole accedere (compreso tra 0 e N-l). La funzione restituisce un array di num interi contenente gli indici delle risorse che sono state allocate. Naturalmente, se non ci sono abbastanza risorse libere, la funzione richiedio blocca il thread in attesa che ci siano almeno num risorse libere.

Dopo aver acceduto alle risorse, il thread le rilascia con la seguente funzione:

void rilascia(int *vett, int num)

dove vett e' un array di num interi contenente gli indici che delle risorse che si vogliono rilasciare. Nel caso ci siano dei thread bloccati in attesa delle risorse necessarie, la funzione rilasciao sveglia per primi i thread che hanno richiesto un minor numero di risorse.

Scrivere le strutture dati necessarie e le funzioni richiedi() e rilascia().

DOMANDA FACOLTATIVA: Spiegare cosa cambia se invece si svegliano per prima i thread che hanno richiesto il maggior numero di risorse fra quelli bloccati in attesa.

SOLUZIONE Parte Prima

```
Process Server
{
        entry { RicQ(), Ric2(), RisQ(int out r), Ris2(int out r1, int out r2),
                        Ril(int in r), Ril2(int in r1, int in r2) }
        bool libera[3]; (true)
        int disp; (0)
        int cq, c2; (0)
        int ind, ind2 ; (0)

        do
        ■  accept RicQ() {}→{
                                                        if(disp>0)
                                                                {
                                                                        int i=0 ;
                                                                        while( !libera[i]) i++ ;
                                                                        libera[i]=false;
                                                                        accept RisQ(int out r)
                                                                                {r=i;}
                                                                }
                                                        else
                                                                cq++;
                                                }

        ■ accept Ric2() {} → {
                                                        if(disp>1)
                                                                {
                                                                        int i=0;
                                                                        int j=0;
                                                                        while(!libera[i]) i++;
                                                                        libera[i]       =false;
                                                                        j=i+1;
                                                                        while(!libera[j]) j++;
                                                                        libera[j]       =false;
                                                                accept Ris2(int out r1,int out r2)
                                                                        {
                                                                                r1=i;
                                                                                r2=j;
                                                                        }
                                                        else
                                                                c2++;
                                                }
        ■ accept Ril(int in r) {ind=r}→
                                                {
                                                if(cq>0)
                                                        {
                                                        accept RisQ(int out r)
                                                                {r=ind;}
                                                        cq--;
                                                else
```

```
                        if(disp>1 && c2>0)
                                {
                                        int i=0;
                                        while(!libera[i]) i++;
                                        accept Ris2(int out r1, int out r1)
                                                {
                                                        r1=ind;
                                                        r2=i;
                                                }
                                        libera[i]=false;
                                        disp--;
                                        c2—
                        else
                                {
                                        libera[ind]=true;
                                        disp++;
                                }
■ accept Ril2 (int in r1, int in r2) {
                                                ind=r1;
                                                ind2=r2;
                                         } →

                {
                        if(cq>0)
                                {
                                        accept RisQ(int out r)
                                                {r=ind;}
                                        cq - -;
                                        if(cq>0)
                                                accept(RisQ(int out r)
                                                        {r=ind2;}

                                                cq - -;
                                        else
                                                {
                                                        libera[ind2]=true;
                                                        disp++;
                                                }
                        else
                                {
                                        libera[ind]=true;    //vedi nota
                                        libera[ind2]=true;
                                        disp+=2;
                        if((c2>0) && (disp>1))
                                {
                                        int i    =0;
                                        int j=0;
                                        while (!libera[i]) i++;
                                        j=i+1;
                                        while (!libera[j]) j++;
                                        accept Ris2(int out r1, int out r2)
```

```
                                                                {
                                                                        r1=i;
                                                                        r2=j;
                                                                }
                                                        c2- -;
                                                        libera[i]=false;
                                                        libera[j]=false;
                                                        disp-=2;
                                                {
        od                                      {
```

// nota:
nel rilascio di due risorse non si è rispettato lo schema classico che prevede il riassegnamento automatico delle risorse in caso di processi bloccati, ma si è provveduto ad un rilascio senza sapere ancora se ci sono o meno processi bloccati, per esigenze di programmazione.

// lato client

```
process client
{
        :
    server. RicQ();
    server.RisQ(i);
        :
    server.Ric2();
        :
}
```

Esame 14/01/03 (no soluz)

PARTE PRIMA

In un sistema organizzato secondo il modello a memoria comune, una risorsa R è condivisa fra un insieme di processi. Su R i processi operano mediante le funzioni FA(), FB() ed FC(). Per coordinare gli accessi ad R, deve essere programmato il suo gestore GR che fornisce le sei procedure: RicA() e RiLA(), (invocate da ciascun processo rispettivamente per richiedere o rilasciare l'accesso a R tramite FA), RicB0 e RilB(), RicC() e RilC() (per le analoghe operazioni di accesso a R tramite FB ed FC).
Usando il meccanismo semaforico realizzare il gestore GR in modo tale che:

1) non siano consentite esecuzioni concorrenti di funzioni diverse mentre devono essere consentite esecuzioni concorrenti di FA o di FB o di FC ma, per ciascuna di esse, di non più di 10 attivazioni concorrenti- Senza preoccuparsi della starvation, realizzare il gestore in modo tale che la richiesta di eseguire una delle tre funzioni n-on sia bloccante se la funzione può essere eseguita subito rispettando i precedenti vincoli. Se la funzione richiesta non è immediatamente eseguibile, la richiesta è bloccante e allora, nei risvegli garantire che le richieste di FA abbiano priorità su quelle di FB e queste su quelle di FC.

2) ripetere la soluzione senza specificare nessuna priorità ma evitando i problemi di starvation. Indicare, in questo caso, quali condizioni devono essere imposte per evitare la starvation.

PARTE SECONDA

Un programma multithread simula il comportamento di un porto e di N navi. Ogni nave è simulata da un tmead. Tutti i thrcad hanno lo stesso codice che è il seguente:

```
void *nave(void *arg){
int indice = *ffint *)arg);
int i;
for (i=O; i<3; i++){
        richiesta_ingresso();
        ingresso();
        rilascio_ingresso();
        staziona();
        richiesta_uscit();
        uscita();
        rilascio_uscita();
        }
}
```

Inizialmente tutte le navi si trovano fuori dal porto. Al massimo M navi possono stazionare in porto allo stesso tempo. Inoltre, l'imboccatura dei porto è molto stretta e solo una nave alla volta può impegnare l'imboccatura in ingresso oppure in uscita.

Ouando una nave vuole entrare in porto, deve prima chiedere il permesso invocando la funzione bloccante richiesta_ingressoo. Se c'è posto nel porto (cioè sono ancorate meno di M navi) e se l'mboccatura è libera, allora la nave impegna l'imboccatura chiamando la funzione ingressoo. Alla fine chiamerà la funzione rilascio_ingressoo per liberare l'imboccatura.

Stesso procedimento quando una nave vuole uscire: innanzitutto invoca la procedura richiesta_uscitao. Se l'imboccatura non è impegnata da un'altra nave, essa impegna l'imboccatura chiamando la funzione uscitao. Infine quando la nave è uscita, libera l'imboccatura chiamando la funzione rilascio-uscitao.

Il candidato deve scrivere la struttura dati gestore del porto e le funzioni richiesta_ingressoo, rilascio_ingressoo, richiesta_uscitao e rilascio_uscitao, usando i mutex e le variabili condition. Inoltre, scrivere la funzione main che inizializza le strutture dati e crea tutti i thread.