Sistemi Operativi T

Esercizi

Esercizio monitor

Si consideri la toilette di un ristorante. La toilette è unica per uomini e donne.

Utilizzando la libreria pthread, si realizzi un'applicazione concorrente nella quale ogni utente della toilette (uomo o donna) è rappresentato da un processo e il bagno come una risorsa.

La politica di sincronizzazione tra i processi dovrà

garantire che:

- nella toilette non vi siano contemporaneamente uomini e donne
- nell'accesso alla toilette, le donne abbiano la priorità sugli uomini.

Si supponga che la toilette abbia una capacità limitata a N persone.

Impostazione

1. Quali processi?
2. Qual è la struttura di ogni processo?
3. Definizione del monitor per gestire la risorsa
4. Definizione delle procedure "entry"
5. Definizione del programma concorrente

Impostazione

1. Quali processi?
   - uomini
   - donne
2. Quale struttura per i processi ?

Uomo:

```
entraU(toilet);
<usa il bagno>
esciU(toilet);
```

Donna:

```
entraD(toilet);
<usa il bagno>
esciD(toilet);
```

Soluzione

3. Definizione del monitor per gestire la risorsa:
   - uomini e donne sono soggetti a vincoli di sincronizzazione diversi
   - possibilita` di attesa in ingresso per uomini e donne

   ➔ prevedo 2 condition (1 per uomini, 1 per donne)

3. Definizione del monitor

```
public toilet{
   private final int MAX=10; /* max capacita */

   int ND; /* num. donne nella toilette*/
   int NU; /* numero uomini nella toilette*/
   private Lock lock = new ReentrantLock();
   condition codaD; /* var. cond. donne */
   condition codaU; /* var. cond. uomini */
   int sospD;/* numero di donne  sospese */
   int sospU;/* numero di uomini  sospesi*/
```

```
public toilet()
{ ND=0; /* num. donne nella toilette*/
  NU=0; /* numero uomini nella toilette*/
  codaD=lock.newCondition();
  codaU=lock.newCondition();
  sospD=0;/* numero di donne  sospese */
  sospU=0;
  }
```

4.Def. procedure entry

```
/* Accesso alla toilette DONNE*/
public void accessoD() throws InterruptedException
{  lock.lock();
try { while ((NU>0)  || /* ci sono uomini in
  bagno */                  (ND==MAX)) /* il
  bagno e` pieno */
      {    sospD++;
           codaD.await();
           sospD--;  }
     ND++;
  }finally{lock.unlock();}
}
```

4.Def. procedure entry

```
public void accessoU() throws InterruptedException
{  lock.lock();
try { while ((ND>0)  || /* ci sono donne in
  bagno */                  (NU==MAX)||    /* il
  bagno e` pieno */          (sospD>0))
  /* ci sono donne in attesa*/
      {    sospU++;
           codaU.await();
           sospU--;  }
     NU++;
  }finally{lock.unlock();}
}
```

4.Def. procedure entry

```
public void uscitaD ()
{  lock.lock();
  ND--;
  if (sospD)
     codaD.signal();
  else     if ((sospU) && (ND==0))
   codaU.signalAll();
  lock.unlock()
}
```

4.Def. procedure entry

```
public void uscitaU ()
{lock.lock();
  NU--;
  if ((sospD)&& (NU==0))
     codaD.signalAll();
  else      if (sospU)
   codaU.signal();
  lock.unlock()
}
}
```

Definizione main

```
import java.util.concurrent.*;
public class Bagno {
public static void main (String args[]) {
int i;
toilet p;
Uomo []U= new Uomo[100];
Donna []D= new Donna[100];
for (i=0; i<100; i++)
{  U[i]=new Uomo(i);
   D[i]=new Donna(i);
}
for (i=0; i<100; i++)
{  U[i].start();
   D[i].start();
}}
```

Esercizio Unix

Si scriva un programma in C che, utilizzando le *system call* di unix, preveda la seguente sintassi:

```
esame N N1 N2 C
```

dove:

esame è il nome dell'eseguibile da generare

- `N, N1, N2` sono interi positivi
- `C e`` il nome di un file eseguibile (presente nel PATH)

Il comando dovrà funzionare nel modo seguente:

- il processo 'padre' P0 deve creare 2 processi figli: P1e P2;

Il comando dovrà funzionare nel modo seguente: il processo 'padre' P0 deve creare 2 processi figli: P1e P2;

- il figlio P1 deve aspettare N1 secondi e successivamente eseguire il comando C;
- il figlio P2 dopo N2 secondi dalla sua creazione dovra` provocare la terminazione del processo fratello P1 e successivamente terminare; nel frattempo P2 deve periodicamente sincronizzarsi con il padre P0 (si assuma la frequenza di 1 segnale al secondo).
- il padre P0, dopo aver creato i figli, si pone in attesa di segnali da P1: per ogni segnale ricevuto, dovra` stampare il proprio pid; al N-simo segnale ricevuto dovra` attendere la terminazione dei figli e successivamente terminare.

Soluzione dell'esercizio

```
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
int PID1, PID2,N,esci=0;
int cont=0; /* cont. dei segnali ricev. da P0*/
void gestore_P(int sig); /* gestore di SIGUSR1
                                per P0*/
void timeout(int sig); /* gestore timeout P2*/

main(int argc , char *argv[])
{
  int  N1, N2, pf, status, i;
  char com[20];
```

```
if (argc!=5)
  { printf("sintassi sbagliata!\n");
        exit(1);
  }

 N=atoi(argv[1]);
 N1=atoi(argv[2]);
 N2=atoi(argv[3]);
 strcpy(com, argv[4]);
 signal(SIGUSR1, gestore_P);
 PID1=fork();
```

```
if (PID1==0) /*codice figlio P1*/
 {       sleep(N1);
        execlp(com,com,(char *)0);
         exit(0);
 }
 else if (PID1<0) exit(-1);

PID2=fork();
 if (PID2==0)
 {/*codice figlio P2*/
    int pp=getppid();
    signal(SIGALRM, timeout);
    alarm(N2);
    for(;;)
    { sleep(1); kill(pp, SIGUSR1);}
    exit(0);
 }
```

```
else if (PID2<0) exit(-1);
/* padre */
while(1) pause();


exit(0);

 }
```

```
void gestore_P(int sig)
{   int i, status, pf;
    cont++;
  printf("padre %d: ricevuto %d (cont=%d)!\n",
  getpid(), sig, cont);
   if (cont==N)
   {    for (i=0; i<2; i++)
     {    pf=wait(&status);
             if ((char)status==0)
          printf("term. %d con stato%d\n", pf,
          status>>8);
             else
          printf("term. %d inv. (segnale %d)\n",
          pf, (char)status);       }
          exit(0);
  }    return;
}
```

```
void timeout(int sig)
{  printf("figlio%d: scaduto
  timeout!\n");
   kill(PID1, SIGKILL);
   exit(0);
}
```

Esercizio Unix (esame giugno 2007)

Si realizzi un programma, che, utilizzando le system call del sistema operativo UNIX, soddisfi le seguenti specifiche:

Sintassi di invocazione: `Esame C N`

Significato degli argomenti:

- Esame è il nome del file eseguibile associato al programma.
- N e` un intero non negativo.
- C e` una stringa che rappresenta il nome di un file eseguibile (per semplicita`, si supponga che il direttorio di appartenenza del file C sia nel PATH).

Specifiche:

Il processo iniziale (P0) deve creare 1 processo figlio P1 che, a sua volta crea un proprio figlio P2 . Si deve quindi ottenere una gerarchia di 3 processi: P0 (padre), P1 (figlio) e P2 (nipote).

- **Il processo P2**, una volta creato, passa ad eseguire il comando *C*.

- **Il processo P1** , dopo aver generato P2 , deve mettersi in attesa di uno dei 2 eventi seguenti:
  1. la ricezione di un segnale dal padre, oppure
  2. la terminazione di P2.

  Nel primo caso (ricezione di un segnale da P0) P1 termina forzatamente l'esecuzione di P2 e poi termina.

  Nel secondo caso (terminazione di P2), P1 invia un segnale al padre P0 e successivamente termina trasferendo a P0 il pid di P2

- **Il processo P0**, dopo aver generato P1, entra in un ciclo nel quale, ad ogni secondo, incrementa un contatore K; se K raggiunge il valore N, P0 invia un segnale al figlio P1 e termina. Nel caso in cui P0 riceva un segnale da P1 durante l'esecuzione del ciclo, prima di terminare dovra` stampare lo stato di terminazione del figlio e successivamente terminare.

Soluzione dell'esercizio

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#define dim 10

int P1, P2;
int   status;
void GP(int sig); //gestore padre
void GF(int sig); //gestore segnali figlio P1

main(int argc , char *argv[])
{       int N,  K=0;
        char C[dim];
        if( argc != 3 ) {
                printf( "sintassi: %s   <nome_comando>  <N>
   \n", argv[0] );
                exit(2);
           }
     strcpy(C,argv[1]);
   N=atoi(argv[2]);
```

```
signal(SIGUSR1, GP);
 P1=fork();
 if (P1==0)//figlio
 {       printf("sono il figlio ..\n");
          signal(SIGUSR1, GF);
            signal(SIGCHLD, GF);
        P2=fork();
              if (P2==0) //nipote
       {      execlp(C, C, NULL);
              perror("attenzione: exec fallita!");
              exit(1);      }
       pause();      //P1: attesa evento
       exit(0);
     }
     else            //padre
     { for (K=0; K<N; K++)
              sleep(1);
       printf("padre P0: esaurito ciclo di conteggio -
  segnale a P1 ..\n");
                kill(P1, SIGUSR1);
          }
  exit(0);} /* fine padre */
```

```
 void GP(int sig)
{  int pf, status;
   printf( "P0 (PID: %d):RICEVUTO SIGUSR1\n", getpid());
   pf=wait(&status);
   if ((char)status==0)
              printf("PADRE: valore trasferito da P1 (pid
   di P2):  %d\n",  status>>8);
   else
      printf("PADRE: la terminazione di %d involontaria
    (per
      segnale %d)\n", pf, (char)status);
   exit(1);
}
```

```
void GF(int sig)
{  int pf, status;
   if (sig==SIGCHLD) //P2 terminato
   {   printf( "P1 (PID: %d):RICEVUTO SIGCHLD-> esecuzione
   di P2
                terminata\n", getpid());
       pf=wait(&status);
       if ((char)status==0)
                     printf("P1: terminazione di %d con
   stato %d\n", pf, status>>8);
       else          printf("P1: terminazione di %d
   involontaria (segnale %d)\n", pf, (char)status);
       kill(getppid(), SIGUSR1);
       exit(pf); //trasferimento pid di P2 al padre P0
   }
   else //segnale da P0
   {   printf( "P1 (PID: %d):RICEVUTO SIGUSR1-> uccido
   P2\n", getpid());
       kill(P2, SIGKILL);
       exit(0);
   }
}
```