Sistemi Operativi L-A

Esercizi
14 Giugno 2007

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 la toliette 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:
   - stato della risorsa:
     - numero di persone (uomini o donne) nella toilette
   - sincronizzazione:
     - 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

```
#include <stdio.h>
#include <pthread.h>
#define MAX 10 /* max capacita toilette */

typedef struct{
   int ND; /* num. donne nella toilette*/
   int NU; /* numero uomini nella toilette*/
   pthread_mutex_t lock;/*lock di mutua esclusione */
   pthread_cond_t codaD; /* var. cond. donne */
   pthread_cond_t codaU; /* var. cond. uomini */
   int sospD;/* numero di donne  sospese */
   int sospU;/* numero di uomini  sospesi*/
}toilette;
```

4.Def. procedure entry

```
/* Accesso alla toilette DONNE*/
void accessoD(toilette *p)
{  pthread_mutex_lock (&p->lock);
/* controlla le condizioni di accesso:*/
   while ((p->NU>0)  || /* ci sono uomini in bagno */
         (p->ND==MAX)) /* il bagno e` pieno */
   {    p->sospD++;
        pthread_cond_wait (&p->codaD, &p->lock);
        p->sospD--;  }
   p->ND++;
   /* catena di risvegli: */
   if ((p->ND<MAX)&&(p->sospD)) /* c'e` ancora posto: */
       pthread_cond_signal (&p->codaD);
   printf("DONNA %lu nella toilette (ci sono %d persone)..\n",
   pthread_self(), p->ND);
   pthread_mutex_unlock (&p->lock);
}
```


4.Def. procedure entry

```
void accessoU(toilette *p)
{       pthread_mutex_lock (&p->lock);
        /* controlla le condizioni di accesso:*/
        while  ((p->ND>0 ) || /* ci sono donne nel bagno*/
                 (p->sospD>0) ||/* ci sono donne in coda*/
                 (p->NU==MAX))  /* il bagno e` pieno */
        {       p->sospU++;
                pthread_cond_wait (&p->codaU, &p->lock);
                p->sospU--;
        }
        p->NU++;
        /* catena di risvegli: */
        if ((p->sospU)&& p->NU<MAX&&p->sospD==0)
                pthread_cond_signal (&p->codaU);
        printf("UOMO %lu nella toilette (ci sono %d
        persone)..\n",pthread_self(), p->NU);
        pthread_mutex_unlock (&p->lock);
}
```

4.Def. procedure entry

```
void uscitaD (toilette *p)
{     pthread_mutex_lock (&p->lock);
      p->ND--;
      if (p->sospD)
            pthread_cond_signal (&p->codaD);
      else
            if ((p->sospU) && (p->ND==0))
                  pthread_cond_signal (&p->codaU);
      pthread_mutex_unlock (&p->lock);
}
```


4.Def. procedure entry

```
void uscitaU (toilette *p)
{     pthread_mutex_lock (&p->lock);
      p->NU--;
      if ((p->sospD) && (p->NU==0))
            pthread_cond_signal (&p->codaD);
      else
            if ((p->sospU) )
                  pthread_cond_signal (&p->codaU);
      pthread_mutex_unlock (&p->lock);
}
```

5. Definizione del programma concorrente

```
/* Programma di test: genera un numero arbitrario di
   thread */
#define MAXT 100 /* num. max di thread  */

/* Inizializzazione monitor:*/
void init (toilette *p)
{  pthread_mutex_init (&p->lock, NULL);
   pthread_cond_init (&p->codaD, NULL);
   pthread_cond_init (&p->codaU, NULL);
   p->ND=0;  p->NU=0;  p->sospD= 0;  p->sospU=0;
   return;
}
/* definizione istanza del monitor: */
toilette p;
```



```
  void *uomo (void *arg) /*codice del thread "uomo" */
  { accessoU(&p);
  /* permanenza nel bagno: */
    sleep(3);
    uscitaU(&p);
    return NULL;
  }

  void *donna (void *arg) /*codice del thread "donna" */
  { accessoD(&p);
    /* permanenza nel bagno: */
    sleep(3);
    uscitaD(&p);
    return NULL;
  }
```

```
/* programma principale: codice del thread iniziale:*/
main ()
{ pthread_t thU[MAXT], thD[MAXT];
  int NU, ND,i;
  void *retval;
  init (&p); /* inizializzazione monitor:*/
  printf("\nquanti uomini? ");  scanf("%d", &NU);
  printf("\nquante donne? ");  scanf("%d", &ND);
  /*CREAZIONE threads:*/
  for (i=0; i<NU; i++)
      pthread_create (&thU[i], NULL, uomo, NULL);
  for (i=0; i<ND; i++)
      pthread_create (&thD[i], NULL, donna, NULL);
  /*ATTESA: */
  for (i=0; i<NU; i++)  pthread_join(thU[i], &retval);
  for (i=0; i<ND; i++)  pthread_join(thD[i], &retval);
  return 0;}
```


Esercizio monitor

Una società di noleggio di automobili offre ai propri clienti **tre tipi di automobili**: piccole, medie, grandi. Ogni tipo di auto è disponibile in numero **limitato** (Npiccole, Nmedie, Ngrandi).

I clienti del noleggio possono essere di due tipi: **convenzionati**, **non convenzionati**.

Ogni cliente richiede una macchina di un particolare tipo (piccola, media, o grande); il noleggio, sulla base della propria disponibilità corrente :

- assegnerà a chi ha richiesto una macchina piccola, preferenzialmente una macchina piccola (se è disponibile), oppure una media (solo se vi è **immediata** disponibilità di medie) .
- assegnerà a chi ha richiesto una media, preferenzialmente una media (se è disponibile), oppure una grande (solo se vi è **immediata** disponibilità di grandi).
- a chi richiede una macchina grande, deve essere comunque assegnata una grande.

..continua

Il cliente, dopo aver **ritirato** la macchina presso la sede del noleggio, la usa per un intervallo di tempo arbitrario, e successivamente procede alla **restituzione** della macchina. Le regole del noleggio prevedono che **ritiro** e **restituzione** delle auto avvengano sempre presso la stessa sede: non è prevista la restituzione in una sede diversa da quella del ritiro.

Per la gestione del noleggio, vale inoltre il seguente vincolo:

- nel ritiro delle auto, i clienti **convenzionati** devono essere prioritari rispetto ai non convenzionati.

Definire una politica di gestione del noleggio, e la si realizzi, utilizzando la libreria pthread.

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?
   - clienti convenzionati
   - clienti non convenzionati
2. Quale struttura per i processi ?

Convenzionato:

```
A=richiesta (nolo, Auto, Conv);
<usa l'auto A>
restituzione(nolo,A);
```

Non Convenzionato:

```
A=richiesta (nolo, Auto, NonConv);
<usa l'auto A>
restituzione(nolo,A);
```


3. Definizione del monitor noleggio:

- lo stato del noleggio e` definito da:
  - numero auto piccole, medie e grandi disponibili.
- lo stato e` modificabile dalle operazioni di:
  - richiesta: acquisizione di un'auto da parte di un cliente
  - restituzione di un auto da parte di un cliente
- ogni cliente e` rappresentato da un thread, caratterizzato dal tipo dell'auto richiesta (P, M, G) e dal tipo del cliente (Convenzionato o No):
  - una coda per ogni tipo di cliente (Conv, NonConv) e per ogni tipo di auto (P, M, G).

➔Incapsulo il tutto all'interno del tipo struct `noleggio`

Tipo associato al noleggio

```
typedef struct
{ int disp[3]; /* numero auto disp.
                  (per ogni cat.)*/
  pthread_mutex_t MUX; /*mutex */
  pthread_cond_t Q[3][2]; /* Q[tipoauto][tipocliente] */
}noleggio;
```


4.Definizione delle operazioni entry:

- **richiesta(noleggio, auto, cliente):** operazione eseguita da ogni thread per ottenere un'auto a nolo; e` possibile ottenere un'automobile di categoria diversa -> funzione
- **restituzione(noleggio, auto):** operazione eseguita dai thread per restituire un'auto.

Soluzione

```
#include <stdio.h>
#include <pthread.h>
#define TotP 10 /* numero totale auto piccole */
#define TotM 10 /* numero totale auto medie */
#define TotG 10 /* numero totale auto grandi */
#define P 0 /*indice auto piccole*/
#define M 1 /*indice auto medie*/
#define G 2 /*indice auto grandi*/
#define Conv 0
#define NonConv 1

typedef struct
{ int disp[3];            /* numero auto disp.(per cat.)*/
   pthread_mutex_t MUX;    /*mutex */
  pthread_cond_t Q[3][2]; /* Q[tipoauto][tipocliente] */
}noleggio;
```



```
/* funzione entry richiesta auto: */
int  richiesta (noleggio *n, int A, int cl)
{ int ottenuta;
   pthread_mutex_lock (&n->MUX);
   switch(A){
   case P:     if ((n->disp[P]==0)&&(n->disp[M]==0))
               {      while(!n->disp[P])
                       pthread_cond_wait (&n->Q[P][cl], &n->MUX);
                      ottenuta=P;
               }
               else if (n->disp[P]!=0) ottenuta=P;
               else   ottenuta=M;
               break;
   case M:     if ((n->disp[M]==0)&&(n->disp[G]==0))
               {       while(!n->disp[M])
                         pthread_cond_wait (&n->Q[M][cl], &n->MUX);
                      ottenuta=M;
               }
               else if (n->disp[M]!=0)       ottenuta=M;
               else   ottenuta=G;
               break;  /* continua..*/
```

```
/* ..continua richiesta: */
case G:     while (n->disp[G]==0)
                  pthread_cond_wait(&n->Q[G][cl],&n->MUX);
            ottenuta=G;
            break;
  }
  n->disp[ottenuta]--;
  pthread_mutex_unlock (&n->MUX);
  return ottenuta;
}
```



```
/* procedure entry restituzione auto: */
void restituzione (noleggio *n, int A)
{
  pthread_mutex_lock (&n->MUX);
  /*  aggiorna lo stato del noleggio */
  n->disp[A]++;
  /* risveglio in ordine di priorita` */
  pthread_cond_signal (&n->Q[A][Conv]);
  pthread_cond_signal (&n->Q[A][NonConv]);
  pthread_mutex_unlock (&n->MUX);
}
```

```
/* Inizializzazione del noleggio */
void init (noleggio *n)
{ int i, j;
  pthread_mutex_init (&n->MUX, NULL);
  for (i=0; i<2; i++)
     for (j=0; j<3; j++)
          pthread_cond_init (&n->Q[i][j], NULL);
  n->disp[P]=TotP;
  n->disp[M]=TotM;
  n->disp[G]=TotG;
  return;

}
```



```
/* Programma di test: genero MAXT thread convenzionati  e non
   convenzionati per ogni tipo di auto */
#define MAXT 50 /* num. di thread per tipo e per auto */

noleggio n;

void *clienteConv (void *arg) /*cliente "convenzionato"*/
{ int A, i;
  printf("thread C. %d: richiedo un'auto di tipo %s\n\n",
             pthread_self(),(char *)arg);
  A=atoi((char *)arg); /*A auto richiesta */
  A=richiesta (&n, A, Conv);
  printf(" thread C.%lu: ottengo un'auto %d\n\n",
       pthread_self(),A);
   /* USO..: */
  restituzione(&n,A);
  printf("thread C:%lu: ho restituito l'auto.\n\n",
       pthread_self());
  return NULL;}
```

```
void *clienteNonConv (void *arg) /* cliente non convenzionato*/
{ int A, i;
  printf("thread NC. %d: richiedo un'auto di tipo %s\n\n",
             pthread_self(),(char *)arg);
  A=atoi((char *)arg); /*A auto richiesta */
  A=richiesta (&n, A, NonConv);
  printf(" thread NC. %lu: ottengo un'auto %d\n\n",
             pthread_self(),A );
   /* USO..: */
  restituzione(&n,A);
  printf("thread NC.%lu: ho restituito l'auto.\n\n",
             pthread_self());
  return NULL;
}

main ()
{  pthread_t th_C[3][MAXT], th_NC[3][MAXT];
  int i,j;
  void *retval;
  init (&n);
```



```
/*CREAZIONE thread: */
  for (i=0; i<MAXT; i++)
   {    pthread_create (&th_C[0][i], NULL, clienteConv, "0");
        pthread_create (&th_C[1][i], NULL, clienteConv, "1");
        pthread_create (&th_C[2][i], NULL, clienteConv, "2");
   }
for (i=0; i<MAXT; i++)
   {    pthread_create (&th_NC[0][i], NULL, clienteNonConv, "0");
        pthread_create (&th_NC[1][i], NULL, clienteNonConv, "1");
        pthread_create (&th_NC[2][i], NULL, clienteNonConv, "2");
   }
/* Attesa teminazione threads :*/
  for (i=0; i<MAXT; i++)
   {    pthread_join (&th_C[0][i], &retval);
        pthread_join (&th_C[1][i], &retval);
        pthread_join (&th_C[2][i], &retval);
   }
   for (i=0; i<MAXT; i++)
   {    pthread_join (&th_NC[0][i], &retval);
        pthread_join (&th_NC[1][i], &retval);
        pthread_join (&th_NC[2][i], &retval);
   }
}
```

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 P2: 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; /* contatore dei segnali ricevuti da P2*/
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)
   {    signal(SIGUSR1, SIG_IGN);
       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", getpid());
     kill(PID1, SIGKILL);
     exit(0);
  }
```