LinuxThreads

Variabili Condizione

LinuxThreads: monitor & variabili condizione

Lo standard POSIX 1003.1c (libreria <**pthread.h**>) **non implementa il costrutto Monitor**

ma

- **implementa i mutex**
- **implementa le variabili condizione**

➔ **mediante mutex e variabili condizione e` possibile** realizzare meccanismi di accesso alle risorse equivalenti a quelli forniti dal concetto di **monitor** [Hoare'74].

Variabili Condizione

- Le variabili condizione (condition) sono uno strumento di sincronizzazione che premette ai threads di sospendere la propria esecuzione in attesa che siano soddisfatte alcune condizioni su dati condivisi.
- ad ogni condition viene associata una coda nella quale i threads possono sospendersi (tipicamente, se la condizione non e` verificata).

Definizione di variabili condizione:

`pthread_cond_t`: è il tipo predefinito per le variabili condizione

Operazioni fondamentali:

- **inizializzazione**: `pthread_cond_init`
- **sospensione**: `pthread_cond_wait`
- **risveglio**: `pthread_cond_signal`

NB: non e` prevista la wait con priorita`, ne` l'operazione queue.

Variabili Condizione: inizializzazione

- **L'inizializzazione di una condition si puo`realizzare con:**

```
int pthread_cond_init(pthread_cond_t* cond,
    pthread_cond_attr_t* cond_attr)
```

dove

- `cond` : individua la condizione da inizializzare
- `attr` : punta a una struttura che contiene gli attributi della condizione; se NULL, viene inizializzata a default.

NB: linux non implementa gli attributi ! -> attr=NULL

- in alternativa, una variabile condizione può essere inizializzata staticamente con la costante:

  `PTHREAD_COND_INIZIALIZER`

- **esempio**: `pthread_cond_t C= PTHREAD_COND_INIZIALIZER;`

Variabili condizione: wait

- Un thread puo` sospendersi su una variabile condizione, se la condizione non e` verificata:
  - ad esempio:

```
pthread_cond_t C= PTHREAD_COND_INIZIALIZER;
int bufferpieno=0;
...
if (bufferpieno) <sospensione sulla cond. C>;
```

  - La verifica della condizione e` una sezione critica!
  - Necessita` di garantire la mutua esclusione:

    e` necessario associare ad ogni variabile condizione un mutex :

```
pthread_cond_t C= PTHREAD_COND_INIZIALIZER;
pthread_mutex_t M=PTHREAD_MUTEX_INITIALIZER;
int bufferpieno=0;
...
pthread_mutex_lock(&M);
if (bufferpieno) <sospensione sulla cond. C>
pthread_mutex_unlock(&M);
```


Variabili condizione: wait

- La sospensione su una condizione si ottiene mediante:

```
int pthread_cond_wait(pthread_cond_t* cond,
                      pthread_mutex_t* mux);
```

dove:

- cond: e` la variabile condizione
- mux: e` il mutex associato ad essa

Effetto:

- il thread chiamante si sospende sulla coda associata a cond, e il mutex mux viene liberato

➔ Al successivo risveglio (provocato da una signal), il thread rioccupera` il mutex automaticamente.

Variabili condizione: signal

- **Il risveglio di un thread sospeso su una variabile condizione puo`essere ottenuto mediante la funzione:**

```
int pthread_cond_signal(pthread_cond_t* cond);
```

dove **cond** e` la variabile condizione.

Effetto:

- se esistono thread sospesi nella coda associata a cond, ne viene risvegliato uno (non viene specificato quale).
- se non vi sono thread sospesi sulla condizione, la signal non ha effetto.
- realizzazione "signal_and_continue"

- Per risvegliare tutti i thread sospesi su una variabile condizione(v. signal_all):

```
int pthread_cond_broadcast(pthread_cond_t* cond);
```

N.B. non e` prevista una funzione ("queue") per verificare lo stato della coda associata a una condizione.

Pthread Condition & Monitor

- La condition permette di implementare politiche di sincronizzazione mediante funzioni/procedure "entry", realizzando meccanismi di accesso alle risorse equivalenti a quelli forniti dal concetto di monitor [Hoare'74]. Tuttavia, non e` previsto il costrutto linguistico "monitor", e pertanto:
  - i dati "interni al monitor" sono potenzialmente accessibili direttamente da tutti i processi;
  - la mutua esclusione delle funzioni/procedure entry deve essere garantita esplicitamente dal programmatore mediante lock/unlock su un mutex associato al "monitor".

➔necessita` di autodisciplina da parte del programmatore !

Esempio: produttore e consumatore

Si vuole risolvere il classico problema del produttore e consumatore.

Progetto della risorsa (prodcons):

- buffer circolare di interi, di dimensione data (ad esempio, 16) il cui stato e` dato da:
  - numero degli elementi contenuti: `cont`
  - puntatore alla prima posizione libera: `writepos`
  - puntatore al primo elemento occupato : `readpos`
- il buffer e`una risorsa da accedere in modo mutuamente esclusivo:
  - predispongo un mutex per il controllo della mutua esclusione nell'accesso al buffer: `lock`
- i thread produttori e consumatori necessitano di sincronizzazione in caso di :
  - buffer pieno: definisco una condition per la sospensione dei produttori se il buffer e` pieno (`notfull`)
  - buffer vuoto: definisco una condition per la sospensione dei produttori se il buffer e` vuoto (`notempty`)

Incapsulo il tutto all'interno di un tipo struct associato al buffer: `prodcons`

Produttori & Consumatori: tipo di dato associato al buffer (dati del "monitor")

```
typedef struct
{
  int buffer[BUFFER_SIZE];
  pthread_mutex_t lock;
  int readpos, writepos;
  int cont;
  pthread_cond_t notempty;
  pthread_cond_t notfull;
}prodcons;
```

Questa struttura rappresenta l'insieme dei dati incapsulati dal "monitor".

Produttore e consumatore

Operazioni sul "monitor": `prodcons`:

- `Init`: inizializzazione del buffer.
- `Inserisci`: operazione "entry" eseguita da ogni produttore per l'inserimento di un nuovo elemento.
- `Estrai`: operazione "entry" eseguita da ogni consumatore per l'estrazione di un elemento dal buffer.

Esempio: produttore e consumatore

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

#define BUFFER_SIZE 16


typedef struct
{ int buffer[BUFFER_SIZE];
  pthread_mutex_t lock;
  int readpos, writepos;
  int cont;
  pthread_cond_t notempty;
  pthread_cond_t notfull;
}prodcons;
```

Esempio:Operazioni sul buffer

```
/* Inizializza il buffer */
void init (prodcons *b)
{
  pthread_mutex_init (&b->lock, NULL);
  pthread_cond_init (&b->notempty, NULL);
  pthread_cond_init (&b->notfull, NULL);
  b->cont=0;
  b->readpos = 0;
  b->writepos = 0;
}
```


Operazioni sul buffer

```
/* Inserimento: */
void inserisci (prodcons *b, int data) /*entry*/
{  pthread_mutex_lock (&b->lock);
  /* controlla che il buffer non sia pieno:*/
  while ( b->cont==BUFFER_SIZE)
      pthread_cond_wait (&b->notfull, &b->lock);
  /* scrivi data e aggiorna lo stato del buffer */
  b->buffer[b->writepos] = data;
  b->cont++;
  b->writepos++;
  if (b->writepos >= BUFFER_SIZE)
    b->writepos = 0;
  /* risveglia eventuali thread (consumatori) sospesi: */
  pthread_cond_signal (&b->notempty);
  pthread_mutex_unlock (&b->lock);
}
```

Operazioni sul buffer

```
/*ESTRAZIONE:  */
int estrai (prodcons *b)/*entry*/
{  int data;
  pthread_mutex_lock (&b->lock);
  while (b->cont==0) /* il buffer e` vuoto? */
      pthread_cond_wait (&b->notempty, &b->lock);
  /* Leggi l'elemento e aggiorna lo stato del buffer*/
  data = b->buffer[b->readpos];
  b->cont--;
  b->readpos++;
  if (b->readpos >= BUFFER_SIZE)
    b->readpos = 0;
  /* Risveglia eventuali threads (produttori):*/
  pthread_cond_signal (&b->notfull);
  pthread_mutex_unlock (&b->lock);
  return data;
}
```


Produttore/consumatore: programma di test

```
/* Programma di test: 2 thread
   - un thread inserisce sequenzialmente max interi,
   - l'altro thread li estrae sequenzialmente per stamparli */

#define OVER (-1)
#define max 20

prodcons buffer;

void *producer (void *data)
{ int n;
  printf("sono il thread produttore\n\n");
  for (n = 0; n < max; n++)
    { printf ("Thread produttore %d --->\n", n);
      inserisci (&buffer, n);
    }
  inserisci (&buffer, OVER);
  return NULL;
}
```

```
void *consumer (void *data)
{  int d;
  printf("sono il thread consumatore \n\n");

  while (1)
    {
      d = estrai (&buffer);
      if (d == OVER)
        break;
      printf("Thread consumatore: --> %d\n",  d);
    }
  return NULL;
}
```



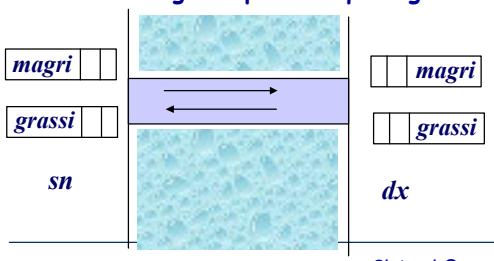
```
  main ()
  {
    pthread_t th_a, th_b;
    void *retval;

    init (&buffer);
    /* Creazione threads: */
    pthread_create (&th_a, NULL, producer, 0);
    pthread_create (&th_b, NULL, consumer, 0);
    /* Attesa teminazione threads creati: */
    pthread_join (th_a, &retval);
    pthread_join (th_b, &retval);
    return 0;
  }
```

Esempio: Ponte con utenti grassi e magri

Si consideri un ponte pedonale che collega le due rive di un fiume.

- Al ponte possono accedere due tipi di utenti: utenti magri e utenti grassi.
- Il ponte ha una capacita` massima MAX che esprime il numero massimo di persone che possono transitare contemporaneamente su di esso.
- Il ponte e` talmente stretto che il transito di un grasso in una particolare direzione d impedisce l'accesso al ponte di altri utenti (grassi e magri) in direzione opposta a d.

Realizzare una politica di sincronizzazione delle entrate e delle uscite dal ponte che tenga conto delle specifiche date e che favorisca gli utenti magri rispetto a quelli grassi nell'accesso al ponte.



➔ *due tipi di thread:*

✓*grassi*

✓*magri*

➔ *una coda per ogni tipo di thread e per ogni direzione*

Progetto della risorsa ponte:

- lo stato del ponte e`definito da:
  - numero magri e di grassi sul ponte (per ogni direzione)
- lo stato e` modificabile dalle operazioni di:
  - accesso: ingresso di un thread nel ponte
  - rilascio: uscita di un thread dal ponte
- il ponte e`una risorsa da acquisire e rilasciare in modo mutuamente esclusivo:
  - ➔predispongo un mutex per il controllo della mutua esclusione nell'esecuzione delle operazioni di accesso e di rilascio: `lock`
- i thread grassi e magri si possono sospendere se le condizioni necessarie per l'accesso non sono verificate :
  - ➔una coda per ogni tipo di thread (grasso o magro)e per ogni direzione
- per ispezionare lo stato delle code introduciamo:
  - ➔un contatore dei thread sospesi per ogni tipo di thread (grasso o magro)e per ogni direzione

➔Incapsulo il tutto all'interno del un tipo struct `ponte`

Grassi & Magri: tipo di dato associato al ponte

```
typedef struct
{ int nmagri[2]; /* numero magri sul ponte (per ogni dir.)*/
  int ngrassi[2];/* numero grassi sul ponte (per ogni dir.)*/
  pthread_mutex_t lock;/*lock associato alla risorsa "ponte"*/
  pthread_cond_t codamagri[2]; /* var. cond. sosp. magri */
  pthread_cond_t codagrassi[2]; /* var. cond. sosp. grassi */
  int sospM[2];/* numero di processi magri sospesi*/
  int sospG[2];/* numero di processi grassi sospesi*/
}ponte;
```


Produttore e consumatore

Operazioni sulla risorsa ponte:

- `init`: inizializzazione del ponte.
- `accessomagri/accessograssi`: operazione eseguita dai thread (grassi/magri) per l'ingresso nel ponte.
- `rilasciomagri/rilasciograssi`: operazione eseguita dai thread (grassi/magri) per l'uscita dal ponte.

Grassi & Magri: soluzione

```
#include <stdio.h>
#include <pthread.h>
#define MAX 3 /* max capacita ponte */
#define dx 0 /*costanti di direzione*/
#define sn 1

typedef struct
{
  int nmagri[2]; /* numero magri sul ponte (per ogni dir.)*/
  int ngrassi[2];/* numero grassi sul ponte (per ogni dir.)*/
  pthread_mutex_t lock;/*lock associato al"ponte" */
  pthread_cond_t codamagri[2]; /* var. cond. sosp. magri */
  pthread_cond_t codagrassi[2]; /* var. cond. sosp. grassi */
  int sospM[2];/* numero di processi magri sospesi*/
  int sospG[2];/* numero di processi grassi sospesi*/
}ponte;
```


Grassi & Magri: soluzione

```
/* Inizializzazione del ponte */
void init (ponte *p)
{
  pthread_mutex_init (&p->lock, NULL);
  pthread_cond_init (&p->codamagri[dx], NULL);
  pthread_cond_init (&p->codamagri[sn], NULL);
  pthread_cond_init (&p->codagrassi[dx], NULL);
  pthread_cond_init (&p->codagrassi[sn], NULL);
  p->nmagri[dx]=0;
  p->nmagri[sn]=0;
  p->ngrassi[dx]=0;
  p->ngrassi[sn]=0;
  p->sospM[dx] = 0;
  p->sospM[sn] = 0;
  p->sospG[dx] = 0;
  p->sospG[sn] = 0;
  return;
}
```

```
/*operazioni di utilita`: */
int sulponte(ponte p); /* calcola il num. di persone sul ponte */
int altra_dir(int d); /* calcola la direzione opposta a d */

/* Accesso al ponte di un magro in direzione d: */
void accessomagri (ponte *p, int d)
{  pthread_mutex_lock (&p->lock);
   /* controlla le codizioni di accesso:*/
   while ( (sulponte(*p)==MAX) || /* vincolo di capacita` */
           (p->ngrassi[altra_dir(d)]>0) ) /*ci sono grassi in
                                            direzione opposta */
   {      p->sospM[d]++;
          pthread_cond_wait (&p->codamagri[d], &p->lock);
          p->sospM[d]--;
   }
   /* entrata: aggiorna lo stato del ponte */
   p->nmagri[d]++;
   pthread_mutex_unlock (&p->lock);
}
```



```
/*Accesso al ponte di un grasso in dir.d:  */

void accessograssi (ponte *p, int d)
{pthread_mutex_lock (&p->lock);
   /* controlla le codizioni di accesso:*/
   while ( (sulponte(*p)==MAX) ||
           (p->ngrassi[altra_dir(d)]>0)||
          (p->nmagri[altra_dir(d)]>0) ||
           (p->sospM[altra_dir(d)]>0)) /*priorita` ai
  magri*/
   {       p->sospG[d]++;
           pthread_cond_wait (&p->codagrassi[d], &p-
  >lock);
           p->sospG[d]--;
   }
   /* entrata: aggiorna lo stato del ponte */
   p->ngrassi[d]++;
  pthread_mutex_unlock (&p->lock);
}
```

```
/* Rilascio del ponte di un magro in direzione d: */

void rilasciomagri (ponte *p, int d)
{
   pthread_mutex_lock (&p->lock);

   /* uscita: aggiorna lo stato del ponte */
   p->nmagri[d]--;
   /* risveglio in ordine di priorita` */
   pthread_cond_broadcast (&p->codamagri[altra_dir(d)]);
   pthread_cond_broadcast (&p->codamagri[d]);
   pthread_cond_broadcast (&p->codagrassi[altra_dir(d)]);
   pthread_cond_broadcast (&p->codagrassi[d]);
   //printf("USCITA: magro in direzione %d\n", d);
   pthread_mutex_unlock (&p->lock);
}
```



```
/* Rilascio del ponte di un grasso in direzione d: */

void rilasciograssi (ponte *p, int d)
{
   pthread_mutex_lock (&p->lock);

   /* uscita: aggiorna lo stato del ponte */
   p->ngrassi[d]--;

   /* risveglio in ordine di priorita` */
   pthread_cond_broadcast (&p->codamagri[altra_dir(d)]);
   pthread_cond_broadcast (&p->codamagri[d]);
   pthread_cond_broadcast (&p->codagrassi[altra_dir(d)]);
   pthread_cond_broadcast (&p->codagrassi[d]);
   pthread_mutex_unlock (&p->lock);
}
```

```
/* Programma di test: genero un numero arbitrario di
   thread magri e grassi nelle due direzioni */
#define MAXT 20 /* num. max di thread per tipo e per
                   direzione */

ponte p;

void *magro (void *arg) /*codice del thread "magro" */
{ int d;

   d=atoi((char *)arg); /*assegno la direzione */
   accessomagri (&p, d);
   /* ATTRAVERSAMENTO: */
   printf("Magro in dir %d: sto attraversando..\n", d);
   sleep(1);
   rilasciomagri(&p,d);
   return NULL;
}
```



```
void *grasso (void *arg) /*codice del thread "grasso" */
{ int d;
  d=atoi((char *)arg); /*assegno la direzione */
   accessograssi (&p, d);
   sleep(1);
   printf("Grasso in dir %d: sto attraversando\n", d);
   rilasciograssi(&p,d);
   return NULL;
}


main ()
{
   pthread_t th_M[2][MAXT], th_G[2][MAXT];
   int NMD, NMS, NGD, NGS, i;
   void *retval;

   init (&p);
```

```
/* Creazione threads: */
   printf("\nquanti magri in direzione dx? ");
   scanf("%d", &NMD);
   printf("\nquanti magri in direzione sn? ");
   scanf("%d", &NMS);
   printf("\nquanti grassi in direzione dx? ");
   scanf("%d", &NGD);
   printf("\nquanti grassi in direzione sn? ");
   scanf("%d", &NGS);

   /*CREAZIONE GRASSI IN DIREZIONE DX */
   for (i=0; i<NGD; i++)
           pthread_create (&th_G[dx][i], NULL, grasso, "0");
    /*CREAZIONE GRASSI IN DIREZIONE SN */
   for (i=0; i<NGS; i++)
           pthread_create (&th_G[sn][i], NULL, grasso, "1");
   /*CREAZIONE MAGRI IN DIREZIONE DX */
   for (i=0; i<NMD; i++)
           pthread_create (&th_M[dx][i], NULL, magro, "0");
    /*CREAZIONE MAGRI IN DIREZIONE SN */
   for (i=0; i<NMS; i++)
           pthread_create (&th_M[sn][i], NULL, magro, "1");
```



```
    /* Attesa teminazione threads creati: */

      /*ATTESA MAGRI IN DIREZIONE DX */
      for (i=0; i<NMD; i++)
              pthread_join(th_M[dx][i], &retval);

       /*ATTESA MAGRI IN DIREZIONE SN */
      for (i=0; i<NMS; i++)
              pthread_join(th_M[sn][i], &retval);

      /*ATTESA GRASSI IN DIREZIONE DX */
      for (i=0; i<NGD; i++)
              pthread_join(th_G[dx][i], &retval);

       /*ATTESA GRASSI IN DIREZIONE SN */
      for (i=0; i<NGS; i++)
              pthread_join(th_G[sn][i], &retval);

      return 0;
   }
```

```
 /* definizione funzioni utilita`:*/
int sulponte(ponte p)
{ return p.nmagri[dx] + p.ngrassi[dx] +
  p.nmagri[sn]+ p.ngrassi[sn];
}

int altra_dir(int d)
{ if (d==sn) return dx;
  else return sn;
}
```