I thread nel sistema operativo LINUX: *Linuxthreads*

LinuxThreads: Caratteristiche

- Processi leggeri **realizzati a livello kernel**
- System call `clone`:

```
int  clone(int (*fn) (void *arg), void *child_stack, int
      flags, void *arg)
```

➔ **E` specifica di Linux: scarsa portabilita`!**

- Libreria *LinuxThreads*: funzioni di gestione dei threads, in conformita` con lo standard POSIX 1003.1c (*pthreads*):
  - *Creazione/terminazione threads*
  - *Sincronizzazione threads:lock, [semafori], variabili condizione*
  - *Etc.*

➢ Portabilita`

LinuxThreads

Caratteristiche threads:

- Il thread e` realizzato **a livello kernel** (e` l'unita` di schedulazione)
- I thread vengono creati all'interno di un processo (task) per eseguire una funzione
- Ogni thread ha il suo PID (a differenza di POSIX: distinzione tra *task* e *threads*)
- Gestione dei segnali non conforme a POSIX:
  - Non c'e` la possibilita` di inviare un segnale a un task.
  - SIGUSR1 e SIGUSR2 vengono usati per l'implementazione dei threads e quindi non sono piu` disponibili.

Sincronizzazione:

- **Lock**: mutua esclusione (**`pthread_mutex_lock/unlock`**)
- **Semafori**: esterni alla libreria **`pthread`** <semaphore.h> (POSIX 1003.1b)
- **Variabili condizione** : (**`pthread_cond_wait,pthread_cond_signal`**)

Rappresentazione dei threads

Il thread e` l'unita` di scheduling, ed e` univocamente individuato da un indentificatore (intero):

```
pthread_t tid;
```

Il tipo **`pthread_t`** e`dichiarato nell'**header file:**

```
<pthread.h>
```

Creazione di thread: pthread_create

Creazione di thread:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,pthread_attr_t *attr,
             void *(*start_routine)(void *), void * arg);
```

Dove:

- **thread:** e` il puntatore alla variabile che raccogliera` il thread_ID (PID)
- **start_routine:** e` il puntatore alla funzione che contiene il codice del nuovo thread
- **arg:** e` il puntatore all'eventuale vettore contenente i parametri della funzione da eseguire
- **attr:** puo` essere usato per specificare eventuali attributi da associare al thread (di solito: NULL):
  - ad esempio parametri di scheduling: priorita` etc.(solo per superuser!)
  - Legame con gli altri threads (ad esempio: *detached* o no)

Ritorna : 0 in caso di successo, altrimenti un codice di errore (!=0)

LinuxThreads: creazione di threads

Ad esempio:

```
int A, B; /* variabili comuni ai thread che verranno creati*/
void * codice(void *){ /*definizione del codice del thread */ …}
main()
{pthread_t t1, t2;
..
pthread_create(&t1,NULL, codice, NULL);
pthread_create(&t2,NULL, codice, NULL);
..
}
```

- **Vengono creati due thread (di tid t1 e t2) che eseguono le istruzioni contenute nella funzione codice:**
  - **I due thread appartengono allo stesso *task* (processo) e condividono le variabili globali del programma che li ha generati (ad esempio A e B).**

Terminazione di thread: pthread_exit

Terminazione di thread:

```
void pthread_exit(void *retval);
```

Dove:

- **retval**: e` il puntatore alla variabile che contiene il valore di ritorno (puo` essere raccolto da altri threads, v. **pthread_join**).

E` una chiamata senza ritorno.

Alternativa: **return();**

pthread_join

Un thread puo` sospendersi in attesa della terminazione di un altro thread con:

```
int pthread_join(pthread_t th, void **thread_return);
```

Dove:

- **th: e` il pid del particolare thread da attendere**
- **thread_return: e` il puntatore alla variabile dove verra` memorizzato il valore di ritorno del thread (v. pthread_exit)**

Esempio: creazione di thread

```
/*Linuxthreads: esempio 1.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *my_thread_process (void * arg)
{
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
}
```



```
main ()
{
  pthread_t th1, th2;
  int retcode;
  if (pthread_create(&th1,NULL,my_thread_process,"1") < 0)
  { fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
  if (pthread_create(&th2,NULL,my_thread_process,"2") < 0)
  { fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }
  retcode = pthread_join (th1, NULL);
  if (retcode != 0)
    fprintf (stderr, "join fallito %d\n", retcode);
  else printf("terminato il thread 1\n);

  retcode = pthread_join (th2, NULL);
  if (retcode != 0)
    fprintf (stderr, "join fallito %d\n", retcode);
  else printf("terminato il thread 2\n);
  return 0;
}
```

Compilazione

Per compilare un programma che usa i linuxthreads:

```
gcc -D_REENTRANT -o prog prog.c   –lpthread
```

```
[aciampolini@ccib48 threads]$ prog
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
terminato il thread 1
terminato il thread 2
[aciampolini@ccib48 threads]$
```


Sincronizzazione: MUTEX

- Lo standard POSIX 1003.1c (libreria <**pthread.h**>) definisce i **semafori binari** (o lock, mutex, etc.)
  - sono semafori il cui valore puo` essere 0 oppure 1 (*occupato* o *libero*);
  - vengono utilizzati tipicamente per risolvere problemi di **mutua esclusione**
  - **operazioni fondamentali:**
    - **inizializzazione:** `pthread_mutex_init`
    - **locking:** `pthread_mutex_lock`
    - **unlocking:** `pthread_mutex_unlock`
  - **Per operare sui mutex:**

    `pthread_mutex_t` : tipo di dato associato al mutex; esempio:

    ```
    pthread_mutex_t mux;
    ```

MUTEX: inizializzazione

- **L'inizializzazione di un `mutex` si puo`realizzare con:**

```
int pthread_mutex_init(pthread_mutex_t* mutex,
                       const pthread_mutexattr_t* attr)
```

attribuisce un valore iniziale all'intero associato al semaforo (default: *libero*):

- `mutex` : **individua il mutex da inizializzare**
- `attr` : **punta a una struttura che contiene gli attributi del mutex; se NULL, il mutex viene inizializzato a *libero* (default).**

- in alternativa , si puo` inizializzare il mutex a default con la macro:

  `PTHREAD_MUTEX_INIZIALIZER`

- **esempio**: `pthread_mutex_t mux= PTHREAD_MUTEX_INIZIALIZER ;`

MUTEX: lock/unlock

- **Locking/unlocking si realizzano con:**

```
int pthread_mutex_lock(pthread_mutex_t* mux);
int pthread_mutex_unlock(pthread_mutex_t* mux);
```

- **lock: se il mutex `mux` e` occupato,il thread chiamante si sospende; altrimenti occupa il mutex.**
- **unlock: se vi sono processi in attesa del mutex, ne risveglia uno; altrimenti libera il mutex.**

Esempio mutex

```
/*Linuxthreads: esempio2.c – uso dei mutex */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX 10
pthread_mutex_t M; /* def.mutex condiviso */
int DATA=0; /* variabile condivisa */
int accessi1=0; /*num. di accessi thread 1 */
int accessi2=0; /*num. di accessi thread 2 */

void *thread1_process (void * arg)
{   int k=1;
    while(k)
    {
        pthread_mutex_lock(&M); /*prologo */
        accessi1++;
          DATA++;
        k=(DATA>=MAX?0:1);
        printf("accessi di T1: %d\n", accessi1);
        pthread_mutex_unlock(&M); /*epilogo */
    }
    pthread_exit (0);
}
```



```
void *thread2_process (void * arg)
{   int k=1;
   while(k)
    {
        pthread_mutex_lock(&M); /*prologo sez. critica */
        accessi2++;
        DATA++;
        k=(DATA>=MAX?0:1);
        printf("accessi di T2: %d\n", accessi2);
       pthread_mutex_unlock(&M); /*epilogo sez. critica*/
    }
    pthread_exit (0);
}
```

```
main(){ pthread_t th1, th2;
  /* il mutex e` inizialmente libero: */
  pthread_mutex_init (&M, NULL);
   if (pthread_create(&th1, NULL, thread1_process, NULL) < 0)
        { fprintf (stderr, "create error for thread 1\n");
          exit (1);
        }
  if (pthread_create(&th2, NULL,thread2_process,NULL) < 0)
  { fprintf (stderr, "create error for thread 2\n");
    exit (1);
  }
   pthread_join (th1, NULL);
   pthread_join (th2, NULL);
}
```


Test

```
$
$ gcc -D_REENTRANT -o tlock lock.c -lpthread
$ ./tlock
accessi di T2: 1
accessi di T1: 1
accessi di T2: 2
accessi di T1: 2
accessi di T1: 3
accessi di T1: 4
accessi di T1: 5
accessi di T1: 6
accessi di T1: 7
accessi di T1: 8
accessi di T2: 3
$
```

LinuxThreads: Semafori

- **Memoria condivisa**: uso dei semafori (POSIX.1003.1b)
  - Semafori: libreria <semaphore.h>
    - **sem_init: inizializzazione di un semaforo**
    - **sem_wait: *p***
    - **sem_post: *v***
  - **sem_t :** tipo di dato associato al semaforo; esempio:

```
static sem_t my_sem;
```


Operazioni sui semafori

Inizializzazione di un semaforo:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

attribuisce un valore iniziale all'intero associato al semaforo:

- **sem: individua il semaforo da inizializzare**
- **pshared : 0, se il semaforo non e` condiviso tra task, oppure non zero (sempre zero).**
- **value : e` il valore iniziale da assegnare al semaforo.**

- **sem_t** : tipo di dato associato al semaforo; esempio:

```
static sem_t my_sem;
```

- ritorna sempre 0.

Operazioni sui semafori: sem_wait

Operazione p:

```
int sem_wait(sem_t *sem);
```

dove:

- `sem`: **individua il semaforo sul quale operare.**

e` la *p* di Dijkstra:

- se il valore del semaforo e` uguale a zero, sospende il thread chiamante nella coda associata al semaforo; altrimenti ne decrementa il valore.

Operazioni sui semafori: sem_post

***Operazione v* :**

```
int sem_post(sem_t *sem);
```

dove:

- `sem`: **individua il semaforo sul quale operare.**

e` la *v* di Dijkstra:

- se c'e` almeno un thread sospeso nella coda associata al semaforo sem, viene risvegliato; altrimenti il valore del semaforo viene incrementato.

Semafori: esempio

```
/* esempio3.c - tre processi che competono per
  incrementare: la variabile V */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX 13

static sem_t m; /* semaforo per la mutua esclusione
            nell'accesso alla sezione critica */

int V=0,F=0;
```



```
void *thread1_process (void * arg)
{   int k=1;
    while(k)
    {
        sem_wait(&m);
  printf("thread1: dentro la sezione critica
  (V=%d)\n",V);
        if (V<MAX)
                V++;
        else
        {       k=0;
                printf("T1: %d (V=%d)\n",++F, V);
        }
        sem_post(&m);
  sleep(1);
    }
  pthread_exit (0);
}
```

```
void *thread2_process (void * arg)
{   int k=1;
    while(k)
    {
        sem_wait(&m);
  printf("thread2: dentro la sezione critica
  (V=%d)\n",V);
        if (V<MAX)
                V++;
         else
        {       k=0;
                printf("T2: %d (V=%d)\n",++F, V);
        }
        sem_post(&m);
  sleep(1); /* per rallentare...*/
    }
  pthread_exit (0);
}
```



```
void *thread3_process (void * arg)
{   int k=1;
    while(k)
    {
        sem_wait(&m);
  printf("thread3: dentro la sezione critica
  (V=%d)\n",V);
        if (V<MAX)
                V++;
         else
        {       k=0;
                printf("T3: %d (V=%d)\n",++F, V);
        }
        sem_post(&m);
  sleep(1); /* per rallentare...*/
    }
  pthread_exit (0);
}
```

```
main ()
{ pthread_t th1, th2,th3;

  sem_init (&m, 0, 1);

  if (pthread_create(&th1, NULL, thread1_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 1\n");
     exit (1);
  }
  if (pthread_create(&th2, NULL,thread2_process,NULL) < 0)
  { fprintf (stderr, "pthread_create error for thread 2\n");
     exit (1);
  }
  if (pthread_create(&th3,NULL,thread3_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 3\n");
     exit (1);
}
```

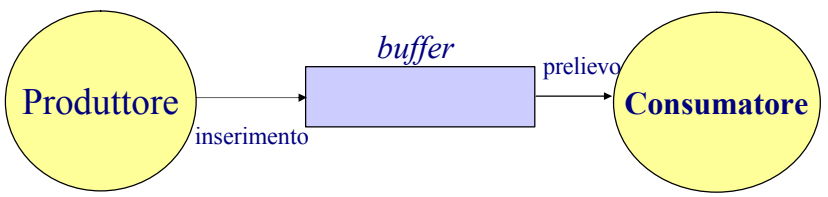

Esercizi Proposti

Esercizi proposti

Facendo uso degli strumenti di sincronizzazione disponibili nella libreria LinuxThreads, risolvere il problema "produttori&consumatori" nei seguenti casi:

1. comunicazione uno-a-uno con buffer di capacita` 1
2. comunicazione uno-a-uno con buffer di capacita` n

Esercizio 1: impostazione



Impostazione (1 e 2)

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

<...variabili globali: buffer, semafori o mutex, etc.>

void *thread_produttore(void * arg)/*codice produttore*/
{  messaggio M;
   while(1)
     {   M=produci();
         <... sincronizzazione (prologo) ...>
         inserimento(M);
         <....sincronizzazione (epilogo)...>
     }
     pthread_exit (0);
}
```



```
void *thread_consumatore(void * arg)/*codice consumatore*/
{  messaggio M;
   while(1)
     {
         <... sincronizzazione (prologo) ...>
         M=prelievo(..);
         <....sincronizzazione (epilogo)...>
         consuma(M);
     }
     pthread_exit (0);
}
```

```
main ()
{ pthread_t P, C;
  void *ret;
  <inizializzazione strumenti di sincronizzazione (semafori
   e/o mutex)..>

if (pthread_create (&th1, NULL, thread.produttore, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 1\n");
     exit (1);
  }

if (pthread_create(&th2,NULL, thread_consumatore, NULL) < 0)
   {fprintf (stderr, "pthread_create error for thread \n");
     exit (1);
  }

  pthread_join (th1, &ret);
  pthread_join (th2, &ret);
}
```


Osservazioni/suggerimenti

- Cercare, quando possibile, di usare gli strumenti di sincronizzazione della libreria pthread (mutex).