ESD Electronic Systems Design
Synthesis Verification Testing Power Communication

Programmazione di sistema in Unix - Thread Posix

N. Drago, M. Lora, G. Di Guglielmo, L. Di Guglielmo,
V. Guarnieri, G. Pravadelli, F. Stefanni

May 11, 2017

Agenda

- Introduzione
- Libreria thread
  - Creazione e terminazione di thread
  - Sincronizzazione tra thread
  - Variabili condition - cenni
- Esempi ed esercizi

Fonti

- "POSIX Thread Programming"
  www.llnl.gov/computing/tutorials/pthreads/
- "Pthreads Programming"
  B. Nichols et al. O'Reilly and Associates.
- "Threads Primer"
  B. Lewis and D. Berg. Prentice Hall
- "Programming With POSIX Threads"
  D. Butenhof. Addison Wesley

Introduzione: Thread vs. Processi

- `Processo`
  - unità di possesso di risorse creato dal S.O.
- Possiede
  - PID, process group ID, UID, and GID
  - Ambiente
  - Text, Dati, Stack, Heap
  - Registri
  - Descrittori di file
  - Gestore dei segnali
  - Meccanismi di IPC (code di messaggi, pipe, semafori, memoria condivisa)

Introduzione: Thread vs. Processi

- `Thread`
  - Unità di esecuzione
  - Esiste all'interno di un processo
- Possiede
  - Stack pointer
  - Registri
  - Proprietà di scheduling (algoritmo, priorità)
  - Set di segnali pendenti o bloccati
  - Dati specifici a una thread

Thread

- Thread multiple eseguono all'interno dello stesso spazio di indirizzamento
- Conseguenze
  - Modifiche effettuate da una thread ad una risorsa condivisa sono visibili da tutte le altre thread
  - Possibile lettura e scrittura sulla stessa locazione di memoria
  - È necessaria esplicita sincronizzazione da parte del programmatore
  - Thread sono paritetiche (no gerarchia)

Perché thread?

- Motivo essenziale: *prestazioni*
- Costo rispetto alla creazione di un processo molto inferiore
- Costo di IPC molto inferiore
- Aumento del parallelismo nelle applicazioni

Quali thread?

- Numerose implementazioni
- Implementazione standard
  - `POSIX thread (pthread)`
    - Definite come API in C
    - Implementate nella libreria `libpthread.a` e utilizzabili includendo lo header file `pthread.h`
  - In Linux
    - `gcc <file.c> -lpthread`
    - `<file.c> deve includere pthread.h`

Programmi multi-threaded

- Per sfruttare i vantaggi delle thread, un programma deve poter essere organizzato in una serie di task indipendenti eseguibili in modo `concorrente`
- Esempi
  - Procedure che possono essere sovrapposte nel tempo o eseguite in un qualunque ordine
  - Procedure che si bloccano per attese potenzialmente lunghe
  - Procedure che devono rispondere ad eventi asincroni
  - Procedure che sono più/meno importanti di altre

Programmi multithreaded

- Modelli tipici di programmi multithreaded
  - `Manager/worker`
    - Thread manager gestisce gli input e assegna operazioni a altre thread worker
  - `Pipeline`
    - Un task viene spezzato in una serie di sottooperazioni (implementate da thread distinte) da eseguire in serie e in modo concorrente
  - `Peer`
    - Simile al modello manager/worker
    - Dopo che la thread principale crea le altre, partecipa al lavoro

API Pthread

- Tre categorie di funzioni
  - Gestione thread
  - Sincronizzazione (mutex) tra thread
  - Comunicazione tra thread

Prefisso della routine	Utilizzo
pthread_	Subroutine generica per gestione thread
pthread_attr	Oggetto attributo
pthread_mutex	Mutex
pthread_mutexattr	Oggetto attributo per mutex
pthread_cond	Variabile condition
pthread_condattr	Oggetto attributo per variabile condition
pthread_key	Chiave specifica per la thread

Creazione di thread

```
#include <pthread.h>

int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine) (void *),
                    void *arg);
```

- `pthread_t *thread`: thread ID (TID)
  - Necessario allocare in memoria un vettore di pthread_t
- `pthread_attr_t *attr`: per settare attributi della thread
  - NULL= valori di default
  - Per altre modalità si vedano le slide successive
- `void*(*start_routine)(void*)`
  - la funzione che verrà eseguita dalla thread
- `void *arg`: argomento per la start_routine
  - Se necessari più parametri
    - Costruire una struct con i parametri desiderati
    - Passare un puntatore alla struct (cast a (void*))
- `Valore di ritorno:` diverso da 0 $\rightarrow$ errore

Terminazione di thread

```
int pthread_exit (void* status)
```

- Termina una thread
- status: valore di terminazione
  - Tipicamente = NULL
- Non chiude eventuali file aperti

Modi alternativi per terminare una thread

- `exit()` sul processo
- Termine della `start_routine`

Terminazione di thread (Cont.)

- NOTE
  - Se il `main()` termina prima delle thread che ha creato, ed esce con `pthread_exit()`, le altre thread continuano l'esecuzione
  - In caso contrario (no `pthread_exit()`) le thread vengono terminate insieme al processo
  - `pthread_exit()` non chiude i file
    - File aperti dentro la thread rimarranno aperti dopo la terminazione delle thread

Esempio

```c
/* hello.c
Creazione di 5 thread che stampano "Hello World!" */
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

/* Routine per la thread */
void *PrintHello(void *threadID)
{
   printf("\n%d: Hello World!\n", threadID);
   pthread_exit(NULL);
}
```

Esempio (cont.)

```
1  int main(){
2     pthread_t threads[NUM_THREADS];
3     int rc, t;
4     for(t=0;t<NUM_THREADS;t++){
5        printf("Creating thread %d\n", t);
6        rc = pthread_create(&threads[t], NULL,
7                                    PrintHello, (void *)t);
8        if (rc){
9          printf("ERROR; return code from pthread_create()
10                  is %d\n", rc);
11          exit(-1);
12       }
13    }
14    pthread_exit(NULL);
15 }
```

Esempio: parametri

```
1  /* hello_arg2.c
2  Il puntatore (void *) consente di passare
3  più argomenti in una struct */
4  /* inserire include necessari, compreso pthread.h*/
5  #define NUM_THREADS 8
6  char *messages[NUM_THREADS];
7
8  /* struct per il passaggio di parametri alla thread */
9  struct thread_data {
10     int thread_id;
11     int sum;
12     char *message;
13 };
14
15 /* definizione dei parametri per le thread */
16 struct thread_data thread_buf[NUM_THREADS];
```

Esempio: parametri (cont.)

```
1  /* start routine per le thread */
2  void *PrintHello(void *thread_arg){
3      struct thread_data *my_data;
4      int taskid, sum;
5      char* hello_msg;
6
7      sleep(1);
8      my_data   = (struct thread_data *) thread_arg;
9      taskid    = my_data->thread_id;
10     sum       = my_data->sum;
11     hello_msg = my_data->message;
12     printf("Thread %d: %s sum=%d\n",taskid,hello_msg,sum);
13     pthread_exit(NULL);
14 }
```

Esempio: parametri (cont.)

```
1  int main(){
2      pthread_t threads[NUM_THREADS];
3      int rc, t, sum=0;
4      messages[0] = "English: Hello world!";
5      ...
6      messages[7] = "Latin: Orbis, te saluto!";
7      for(t=0; t < NUM_THREADS, t++){
8          sum = sum + t;
9          thread_buf[t].thread_id = t;
10         thread_buf[t].sum = sum;
11         thread_buf[t].message = messages[t];
12         printf("Creating thread %d\n", t);
13         rc=pthread_create(&threads[t],NULL,
14               printHello, (void*) &thread_buf[t]);
15         if (rc){
16            printf("ERROR:return code is %d", rc);
17            exit(-1);
18     }
19     pthread_exit(NULL);
20 }
```

Identificazione di thread

```c
#include <pthread.h>

pthread_t pthread_self(void);
```

- ritorna alla thread chiamante l'ID assegnato dal sistema

```c
#include <pthread.h>

int pthread_equal(pthread_t thread1, pthread_t thread2);
```

- Confronta due thread ID, se diversi ritorna 0, altrimenti un valore diverso da 0

Sincronizzazione tra thread

- Il "join" di thread è uno dei modi per realizzare la sincronizzazione tra thread

```
#include <pthread.h>

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

  - Blocca la thread chiamante fino a che la thread specificata da `th` termina
  - L'argomento `thread_return` è un puntatore alla variabile `status` della thread `th` passata come parametro di `pthread_exit()`

Sincronizzazione tra thread (Cont.)

- Quando una thread viene creata, il campo `attr` definisce alcuni parametri della thread. Ad esempio, se:
  - se ne può fare il join (`joinable` – default)
  - non se ne può fare il join (`detached`)

Join/detach di thread: Funzioni

```c
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

- Per inizializzare l'oggetto attributo puntato da `attr` con i valori di default

```c
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
```

- Libera la memoria associata all'oggetto attributo `attr`

Join/detach di thread: Funzioni (Cont.)

```c
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
```

- Per settare l'oggetto attributo `attr` al valore `detachstate`
  - PTHREAD_CREATE_JOINABLE oppure
  - PTHREAD_CREATE_DETACHED

```c
#include <pthread.h>

int pthread_attr_getdetachstate(pthread_attr_t *attr,
                                int *detachstate);
```

- Recupera il valore corrente dell'attributo `attr`
- Memorizza il valore dell'attributo `attr` nell'argomento `detachstate`

Join/detach di thread: Funzioni (Cont.)

```c
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

- Per mettere la thread `thread` nello stato detached
- Non si può fare il join della thread

Join/detach di thread

1. Dichiarare una variabile attributo di tipo `pthread_attr_t`
2. Inizializzare la variabile attributo con la funzione `pthread_attr_init()`
3. Settare il valore desiderato della variabile attributo tramite `pthread_attr_setdetachstate()`
4. Creare la thread con la variabile attributo
5. Cancellare le risorse usate dalla variabile attributo con la funzione `pthread_attr_destroy()`

- La funzione `pthread_detach()` permette di fare il detach esplicito di una thread anche se era stata creata come joinable

Esempio

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #define NUM_THREADS 3
4
5  void *BusyWork(void *) {
6     int i;
7     double result=0.0;
8     for (i=0; i<100000; i++) {
9       result = result/2 + (double)random();
10    }
11    printf("Thread result = %d\n",result);
12    pthread_exit((void *) 0);
13 }
```

Esempio (Cont.)

```
1  int main()
2  {
3      pthread_t thread[NUM_THREADS];
4      pthread_attr_t attr;                                              (1)
5      int rc, t, status;
6
7      /* Initialize and set thread detached attribute */
8      pthread_attr_init(&attr);                                         (2)
9      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);      (3)
```

Esempio (Cont.)

```
1    /* creation of threads, they are joinable */
2    for(t=0; t < NUM_THREADS; t++) {
3       printf("Creating thread %d\n", t);
4       rc = pthread_create(&thread[t], &attr,                    (4)
5                           BusyWork, NULL);
6       if (rc != 0) {
7         printf("ERROR; return code from
8                 pthread_create() is %d\n", rc);
9         exit(-1);
10      }
11   }
```

Esempio (Cont.)

```
1     /*Free attribute and wait for the other threads*/
2     pthread_attr_destroy(&attr);                                    (5)
3     for(t=0;t<NUM_THREADS;t++) {
4       rc = pthread_join(thread[t],(void**)&status);
5       if (rc) {
6         printf("ERROR return code from
7                 pthread_join() is %d\n", rc);
8         exit(-1);
9       }
10      printf("Completed join with thread %d
11              status= %d\n", t, status);
12    }
13    pthread_exit(NULL);
14  }
```

Mutex

- Meccanismo base per l'accesso protetto ad una risorsa condivisa (`sezione critica`)
- Una sola thread alla volta può fare il lock di un mutex
- Le altre thread in competizione sono messe in attesa
- Variabili `mutex`
  - Tipo `pthread_mutex_t`
  - Devono essere inizializzate prima dell'uso
    - Staticamente all'atto della dichiarazione

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER
```

    - Dinamicamente con opportuna funzione

```
pthread_mutex_init (mutex,attr)
```

Mutex - Funzioni

```c
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

- Per inizializzare una variabile `mutex` come specificato dall'attributo `mutexattr`

```c
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Per distruggere una variabile `mutex`

Mutex - Funzioni (Cont.)

- Il parametro `mutex` rappresenta l'indirizzo del mutex che si inizializza/distrugge
- Il parametro `attr` viene usato per definire proprietà di un mutex
  - Default = NULL
  - Attributi definiti per usi avanzati
- Tipica invocazione

```
pthread_mutex_t s;
...
pthread_mutex_init(&s, NULL);
```

Mutex - Attributi per usi avanzati

- Gli attributi determinano il tipo di mutex
  - `Fast mutex (default)`
  - `Recursive mutex`
  - `Error checking mutex`
- Il tipo determina il comportamento del mutex rispetto a operazioni di lock da parte di una thread su un mutex già posseduto dalla thread stessa
  - Fast: thread viene bloccata fino a quando il lock è disponibile
  - Recursive: thread incrementa numero di lock e prosegue
    - Per sbloccare il mutex bisogna chiamare la unlock un numero di volte pari al numero di lock effettuati
  - Error checking: thread ritorna errore e prosegue

Mutex - Attributi per usi avanzati (Cont.)

- È possibile definire mutex dei 3 tipi tramite costanti di inizializzazione
  - Recursive

```
pthread_mutex_t m = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
```

  - Error checking

```
pthread_mutex_t m = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

  - Fast

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Mutex - Attributi per usi avanzati - Funzioni

```c
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

- Per inizializzare un oggetto attributo per una variabile mutex con i valori di default

```c
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

- Per distruggere un oggetto attributo usato per una variabile mutex

Mutex - Attributi per usi avanzati - Funzioni (Cont.)

```
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t *attr,
                              int type);
```

- Per specificare il tipo dell'oggetto attributo per una variabile mutex
- I tipi validi sono:
  - PTHREAD_MUTEX_RECURSIVE
  - PTHREAD_MUTEX_ERRORCHECK
  - PTHREAD_MUTEX_DEFAULT

Mutex - Attributi per usi avanzati - Funzioni (Cont.)

```c
#include <pthread.h>

int pthread_mutexattr_gettype(
                const pthread_mutexattr_t *restrict attr,
                int *restrict type);
```

- Per recuperare il tipo corrente dell'oggetto attributo `attr` e copiarlo nella locazione puntata da `type`
- La keyword `restrict` è utilizzata per allineare la funzione allo standard ISO/IEC 9899:1999

Mutex - Attributi per usi avanzati - Funzioni (Cont.)

```c
#include <pthread.h>

...

pthread_mutexattr_t mattr1;

/* initializzare un attributo ai valori di default */
pthread_mutexattr_init(&mattr1);

/* settare il tipo di un attributo
                        usando la funzione set */
pthread_mutexattr_settype(&mattr1,PTHREAD_MUTEX_ERRORCHECK);
```

Lock/unlock di mutex

```c
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Acquisisce un lock attraverso la variabile `mutex`
- Se il mutex è già posseduto da un'altra thread, la chiamata si blocca fino a che il lock è disponibile

```c
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Tenta di acquisire un mutex
- Se il mutex è posseduto da un'altra thread, ritorna immediatamente con un codice di errore "busy"
  - In Linux: codice errore `EBUSY`

Lock/unlock di mutex (Cont.)

```c
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Sblocca il lock posseduto dalla thread chiamante
- Lo sblocco del mutex dipende dal suo tipo. Per esempio, in caso di mutex di tipo `PTHREAD_MUTEX_RECURSIVE`, il mutex è disponibile quando il suo counter è decrementato a zero

Esempio

```c
/* mutex1.c
   utilizzo di mutex */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
/* This is the lock for thread synchronization */
pthread_mutex_t my_sync;
/* starting routine */

void* compute_thread(void* dummy) {
/* Lock the mutex when its our turn */
    pthread_mutex_lock(&my_sync);
    sleep(5);
    printf("%s", dummy);
    fflush(stdout);
    pthread_mutex_unlock(&my_sync);
    return NULL;
}
```

Esempio (cont.)

```
1  int main() {
2      pthread_t tid;
3      pthread_attr_t attr;
4      char hello[] = "Hello, ";
5      /* Initialize the thread attributes */
6      pthread_attr_init(&attr);
7      /* Initialize the mutex (default attributes) */
8      pthread_mutex_init(&my_sync,NULL);
9      /* Create another thread. ID is returned in &tid */
10     pthread_create(&tid, &attr, compute_thread, hello);
11     sleep(1);    /* Let the thread get started */
12     /* Lock the mutex when it's our turn to do work */
13     pthread_mutex_lock(&my_sync); /*change it with trylock!*/
14     sleep(2);
15     printf("thread");
16     printf("\n");
17     pthread_mutex_unlock(&my_sync);
18     exit(0);
19 }
```

Esempio (cont.)

- Nell'esempio precedente provare a sostituire nel main

```
pthread_mutex_lock(&my_sync);
```

con

```
pthread_mutex_trylock(&my_sync);
```

- Cosa succede?

Condition Variable - cenni

- Meccanismo addizionale di sincronizzazione
- Permettono di sincronizzare thread in base ai valori dei dati senza busy waiting
- Senza condition variable, le thread devono testare in busy waiting (eventualmente in una sezione critica) il valore della condizione desiderata
- Le variabili condition sono sempre associate all'uso di un mutex