Thread POSIX

Insegnamento di Sistemi di Elaborazione

Corso di L.M. In Ing. Elettronica

Ing. Vincenzo Maffione

E-mail: vincenzo.maffione@ing.unipi.it
Dipartimento di Ingegneria dell'Informazione, Università di Pisa

Diapositive derivate dal materiale didattico fornito dall'Ing. Simone Brienza

Thread POSIX

- Introduzione ai thread POSIX
  - operazioni elementari sui thread
- Sincronizzazione
  - Semafori
    - ⇨ semafori di mutua esclusione
    - ⇨ semafori generali
    - ⇨ utilizzo ed esempi
  - Variabili condition
    - ⇨ generalità
    - ⇨ utilizzo ed esempi

Thread POSIX: aspetti preliminari

Programmazione concorrente

- La programmazione concorrente è l'insieme di tecniche e di strumenti necessari a poter supportare più attività simultanee in una applicazione software.
- Possibile su sistemi multiprogrammati.
- Consente ad un singolo programma di scomporre la propria attività in più flussi di esecuzioni concorrenti

Programma vs Processo

- Programma: è un'entità statica che rimane immutata durante l'esecuzione ed è costituita dal codice oggetto generato dalla compilazione del codice sorgente.

- Processo: è l'entità con cui il sistema operativo rappresenta una specifica esecuzione di un programma. E' un'**entità dinamica**, con uno stato che si evolve nel tempo e dipende dai dati che vengono forniti in ingresso.

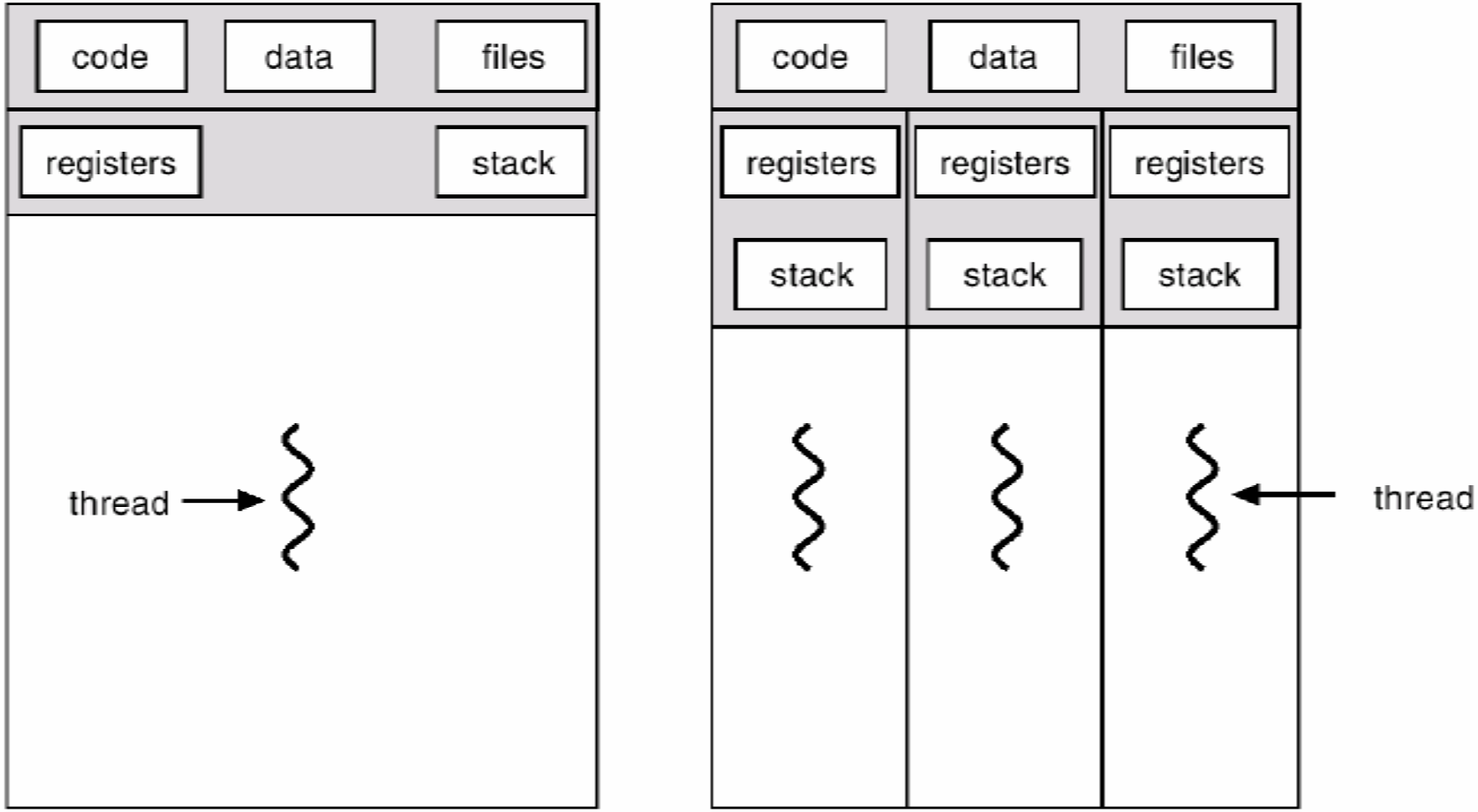
  Ogni processo ha un *contesto*:
  - Il suo codice eseguibile (il programma)
  - il suo stato
    - ⇨ Spazio di indirizzamento (memoria)
    - ⇨ I thread del processo
    - ⇨ I file descriptors (che riferiscono file e dispositivi di IO).

Processo vs Thread

- Il contesto del processo è un insieme di risorse, alcune delle quali ereditate dal padre.
  - Spazio di indirizzamento del processo:
    - ⇨ Dati statici (variabili globali)
    - ⇨ Stack per supportare le chiamate di funzione
    - ⇨ Heap per supportare la memoria dinamica
  - Due processi non condividono lo spazio di indirizzamento

- Thread: è un flusso di esecuzione indipendente interno ad un processo
  - Tutti i thread di uno stesso processo condividono le sue risorse
  - In particolare, condividono lo spazio di indirizzamento (la memoria)
  - I thread sono anche chiamati *lightweight process* o *processi leggeri* perchè possiedono un contesto più piccolo rispetto ai processi

Processi con uno o più thread



single-threaded

multithreaded

Thread

- Thread
  - è uno dei flussi di esecuzione di un processo e può essere eseguito **concorrentemente** ad altri thread
  - se la macchina ha più CPU, i thread di un processo possono essere eseguiti **in parallelo**
  - *main thread:* thread principale che parte per primo ed esegue la funzione main()
- Stato di un thread → definito da:
  - Stack
  - Registri del processore
  - Heap
  - Dati statici

Esempi di processi multi-thread

- Un browser Web, che usa un thread diverso per scaricare ogni immagine in una pagina Web contenente più immagini.

- Un processo server che risponde concorrentemente alle richieste provenienti da più utenti.

Vantaggi

- Più flussi di esecuzione per fare più cose concorrentemente
- Condivisione di dati tra thread è immediata
  - Tutti i thread di un processo condividono lo stesso spazio di indirizzamento, quindi le comunicazioni tra thread sono <u>più semplici</u> delle comunicazioni tra processi.
- Context switch veloce
  - Nel passaggio da un thread ad un altro di uno stesso processo viene <u>mantenuta</u> buona parte del contesto.

Svantaggi

- Concorrenza causa problemi di consistenza
  - gestire la mutua esclusione
- I thread di un programma usano il sistema operativo mediante system call che usano dati e tabelle di sistema dedicate al processo →
  - Le syscall devono essere costruite in modo da poter essere utilizzate da più thread contemporaneamente.

Standard

- Standard ANSI/IEEE POSIX 1003.1 (1990)
  - Lo standard specifica l'interfaccia di programmazione (Application Program Interface - API) dei thread.
  - I thread POSIX sono noti come PThread.

Funzioni delle API per Pthread

- Le API per PThread distinguono le funzioni in 3 gruppi:
  - Thread management
    - ⇨ funzioni per creare, eliminare, attendere la fine dei pthread
  - Mutex:
    - ⇨ oggetti per supportare la mutua esclusione («mutex», o «lock»)
    - ⇨ funzioni per creare ed eliminare mutex, acquisire e rilasciare il lock
  - Variabili condition:
    - ⇨ oggetti a supporto di una sincronizzazione più generica, dipendente dal valore di variabili definite dal programmatore
    - ⇨ funzioni per creare ed eliminare variabili condition, per attendere eventi e notificare thread in attesa

Utilizzo

- Utilizzo
  - includere l'header della libreria che contiene le definizioni dei pthread

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

    - ⇨ Per interpretare correttamente i messaggi di errore è necessario anche includere l'header `<errno.h>`
  - compilare specificando la libreria

```
gcc <opzioni> -lpthread
```

    - ⇨ Libreria pthread (libpthread) → `lpthread`
    - ⇨ Per ulteriori informazioni sulla compilazione fare riferimento alla documentazione della piattaforma utilizzata `man pthread` o `man pthreads`

Convenzione sui nomi delle funzioni

- Gli identificatori della libreria dei Pthread iniziano con `pthread_`
  - `pthread_`
    - ⇨ indica la gestione dei thread in generale
  - `pthread_attr_`
    - ⇨ funzioni per gestire proprietà dei thread
  - `pthread_mutex_`
    - ⇨ gestione della mutua esclusione
  - `pthread_mutexattr_`
    - ⇨ proprietà delle strutture per la mutua esclusione
  - `pthread_cond_`
    - ⇨ gestione delle variabili di condizione
  - `pthread_condattr_`
    - ⇨ proprietà delle variabili di condizione
  - `pthread_key_`
    - ⇨ dati speciali dei thread

Risorse

- POSIX Threads Programming Tutorial
  - http://www.llnl.gov/computing/tutorials/pthreads/
- Libri (consultazione)
  - B. Lewis, D. Berg, "*Threads Primer*", Prentice Hall
  - D. Butenhof, "*Programming With POSIX Threads*", Addison Wesley
  - B. Nichols et al, "*Pthreads Programming*", O'Reilly

Risorse

- **Manpages**
  - **pacchetto** `manpages-posix-dev` (**Debian**)
  - `man pthread.h`
  - `man <nomefunzione>`
- **Manuale GNU libc**
  - http://www.gnu.org/software/libc/manual/html_node/POSIX-Threads.html

Gestione dei thread

Tipi definiti nella libreria pthread

- All'interno di un programma un thread è rappresentato da un identificatore
  - tipo opaco `pthread_t`
- Attributi di un thread
  - tipo opaco `pthread_attr_t`

Tipo Opaco: si definiscono così strutture ed altri oggetti usati da una libreria, la cui struttura interna non deve essere visibile dall'utente (da cui il nome) che li deve utilizzare solo attraverso alcune opportune funzioni di gestione.

Identificatore del thread

- **Processo:** **process id (pid)** `pid_t`
- **Thread:** **thread id (tid)** `pthread_t`

```
pthread_t pthread_self( void )
```

- **restituisce il tid del thread chiamante**

Confronto tra thread

```
int pthread_equal( pthread_t t1, pthread_t t2 )
```

- confronta i due identificatori di thread.
  - `1` se i due identificatori sono uguali

Creazione di un thread (1 di 2)

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

- crea un thread e lo rende eseguibile, cioè lo mette a disposizione dello scheduler che prima o poi lo farà partire

Creazione di un thread (2 di 2)

- `pthread_t *thread`
  - puntatore ad un identificatore di thread in cui verrà scritto l'identificatore univoco del thread creato (se creato con successo)
- `const pthread_attr_t *attr`
  - attributi del processo da creare: può indicare le caratteristiche del thread riguardo alle operazioni di join o allo scheduling
  - se `NULL` usa valori di default
- `void *(*start_routine)(void *)`
  - è il nome (indirizzo) della funzione da eseguire alla creazione del thread
- `void *arg`
  - puntatore che viene passato come argomento a `start_routine`.
- Valore di ritorno
  - `0` in assenza di errore
  - diverso da zero altrimenti
    - ⇨ attributi errati
    - ⇨ mancanza di risorse

Terminazione di un thread

```
void pthread_exit( void *value_ptr )
```

- Termina l'esecuzione del thread da cui viene chiamata
- Il sistema libera le risorse allocate al thread.
- Se il `main` termina prima che i thread da lui creati siano terminati e non chiama la funzione `pthread_exit`, allora <u>tutti i thread vengono terminati</u>. Se invece il `main` chiama `pthread_exit` allora i thread possono continuare a vivere fino alla loro terminazione.
- `void *value_ptr`
  - valore di ritorno del thread consultabile da altri thread attraverso la funzione `pthread_join` (che vedremo in seguito)

Scheduling dei PThread

- In generale i Pthreads sono schedulati dal kernel in maniera *arbitraria* e *non predicibile*

- Qualsiasi operazione che richieda sincronizzazione tra thread va progettata tenendo presente che l'ordine di creazione dei thread **non ha alcuna relazione** con la loro schedulazione

Scheduling dei PThread

- In un qualsiasi momento, lo scheduler può revocare la CPU al thread in esecuzione.
- E' dunque compito del programmatore far sì che le strutture dati condivise fra i thread siano sempre consistenti.

Esempio 1: creazione e terminazione (1 di 3)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NUM_THREADS     5

void *PrintHello(void *num) { /* Corpo del thread */
   int num_par = *((int*)num);
   printf("%d: Hello World!\n", num_par);
   pthread_exit(NULL);   }
```

Continua ⇨

Esempio 1: creazione e terminazione (2 di 3)

```
int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int ids[NUM_THREADS];
   int rc, t;
   for(t=0; t<NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        ids[t] = t;
        rc = pthread_create(&threads[t], NULL, PrintHello, (void*)
&ids[t]);
        if (rc){
           printf("ERROR; return code from pthread_create() is %d\n", rc);
           exit(-1);
        }
    }
   pthread_exit(NULL);
}
```

Esempio 1: creazione e terminazione (3 di 3)

- Eseguire più volte il programma
  - Cosa accade?

- Provare a commentare in main() la funzione
  `pthread_exit(NULL)`

  - Cosa accade?

Passaggio parametri (1 di 4)

- La `pthread_create` prevede un puntatore per passare una struttura dati al thread nel momento in cui comincia la sua esecuzione.
- Attenzione al caso in cui il *thread* e/o il chiamande debbano modificare la struttura dati

Passaggio parametri (2 di 4)

- Per riferimento con un cast a `void*`
- Esempio (errato) 1/2

```
int rc, t;

for(t=0; t<NUM_THREADS; t++) {
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
   ...
}
```

Passaggio parametri

- **Esempio (errato) 2/2**

```
/* Include */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS     5

/* Corpo del thread */

void *PrintHello(void *num) {
   printf("\n%d: Hello World!\n", *(int *) num);
   *(int *)num = 100;
   pthread_exit(NULL);
}
```

Passaggio parametri

- Quali sono gli errori in questo programma?

Passaggio parametri

- Errori e conseguenze:
  - Si passa come parametro al thread un valore che viene concorrentemente modificato all' interno del main(), per cui il valore stampato non è deterministico
  - Più grave: i thread creati modificano l'indice del ciclo for nel main(), e il programma va in crash, non deterministicamente

Passaggio parametri (3 di 4)

- Esempio (corretto)
  - Per ogni thread viene allocata una struttura dati privata

```
int *taskids[NUM_THREADS];
for(t=0; t<NUM_THREADS; t++){
   taskids[t] = (int *) malloc(sizeof(int));
   *taskids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
   ...
}
```

Passaggio parametri (4 di 4)

- Esempio (corretto) – Versione alternativa
  - Il valore della variabile *t* è copiato nell'array *taskids* nella cella di indice *t*: il thread riceve l'indirizzo della cella dell'array associata al proprio indice ed è l'unico thread ad usarlo.

```
int taskids[NUM_THREADS];
for(t=0; t<NUM_THREADS; t++){
   taskids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void*) &taskids[t]);
   ...
}
```

Passaggio parametri

```
/* Include */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS     5

/* Corpo del thread */

void *PrintHello(void *num) {
   printf("%d: Hello World!\n", *(int *) num);
   *(int *) num = 100;
   pthread_exit(NULL);
}
```

Esempio 2: passaggio parametri

- Qual è la differenza? Perchè ora la modifica **NON** ha effetto sugli altri thread?

Esempio: Passaggio parametri (1 di 2)

- Passaggio di due parametri

```
struct tipo {
    int par1;
    int par2;
};
...
struct tipo taskpars[NUM_THREADS];
for(t=0; t<NUM_THREADS; t++){
   taskpars[t].par1 = t;
   taskpars[t].par2 = 23 + t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void*)&taskpars[t]);
   ...
}
```

Esempio: Passaggio parametri (2 di 2)

- Passaggio di due parametri

```
void *PrintHello(void *par) {
   int id = ((struct tipo *) par)->par1;
   int num = ((struct tipo *) par)->par2;
   printf("\n%d: Hello World! My number: %d\n", id, num);
   pthread_exit(NULL);
}
```

Sincronizzazione

Join tra thread

- Forma elementare di sincronizzazione
  - il thread che effettua il join si blocca finché uno specifico thread non termina
  - il thread che effettua il join può ottenere lo stato del thread che termina
- Attributo `detachstate` di un thread specifica se si può invocare o no la funzione join su un certo thread
  - un thread è joinable per default

Operazione di join

```
int pthread_join( pthread_t *thread, void **value )
```

- `pthread_t *thread`
  - **identificatore del thread di cui attendere la terminazione**
- `void **value`
  - **valore restituito dal thread che termina**
- **Valore di ritorno**
  - `0` **in caso di successo**
  - `EINVAL` **se il thread da attendere non è joinable**
  - `ERSCH` **se non è stato trovato nessun thread corrispondente all'identificatore specificato**

Esempio di join

```
int main (int argc, char *argv[]) {
   pthread_t thread[NUM_THREADS];
   ...
   for(t=0; t<NUM_THREADS; t++)
   {
      rc = pthread_join(thread[t], (void **)&status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
      }
      printf("Completed join with thread %d status= %d\n",t, status);
   }
   pthread_exit(NULL);
}
```

Impostazione attributo di join (1 di 3)

- Un thread può essere:
  - *Joinable*: i thread non sono rilasciati automaticamente ma rimangono come zombie finchè altri thread non effettuano delle join
  - *Detached*: i thread detached sono rilasciati automaticamente e non possono essere oggetto di join da parte di altri thread.

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

- Detach può essere:
  - PTHREAD_CREATE_DETACHED
  - PTHREAD_CREATE_JOINABLE.

Impostazione attributo di join (2 di 3)

```
int pthread_attr_init( pthread_attr_t *attr )
```

- Inizializza la variabile contenente gli attributi

```
int pthread_attr_destroy ( pthread_attr_t *attr)
```

- Distrugge la variabile

Impostazione attributo di join (3 di 3)

```
/* Attributo */
pthread_attr_t attr;

/* Inizializzazione esplicita dello stato joinable */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
...
pthread_attr_destroy(&attr);
```

Esempio 3: thread join (1 di 3)

```
/* Include */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *num) {
   printf("\n%d: Hello World!\n", num);
   pthread_exit(NULL);
}
```

Continua ⇨

Esempio 3: thread join (2 di 3)

```
int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   void *status;
   int rc, t;
   pthread_attr_t attr;

   /* Inizializzazione esplicita dello stato joinable */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++){
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], &attr, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
```

Continua ⇨

Esempio 3: thread join (3 di 3)

```
   for(t=0; t<NUM_THREADS; t++){
      rc = pthread_join(threads[t], (void **)&status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
      }
      printf("Completed join with thread %d status= %d\n",t, status);
   }

   printf ("Main(): Atteso su %d threads. Fatto \n", NUM_THREADS);

   /*Rimuovi oggetti ed esci*/
   pthread_attr_destroy(&attr);

   pthread_exit(NULL);
}
```

Problemi della Programmazione Concorrente

- La programmazione concorrente è estremamente utile …
- … Ma può comportare notevoli difficoltà per il programmatore
- Il non determinismo insito nella concorrenza causa problemi che non possono esistere in programmi single-threaded:
  - Problema della mutua esclusione
  - Deadlock
  - Livelock (starvation)

Grazie per l'attenzione.