AXO

Architettura dei Calcolatori e Sistemi Operativi

modello thread

programma e parallelismo

- il tipo di parallelismo dipende dal grado di cooperazione (scambio di informazione) necessario tra attività svolte in parallelo
- processo – una macchina virtuale indipendente con meccanismi *ad hoc* per scambiare informazione tra processi (*IPC*)
  - attività parallele indipendenti
- thread – un'attività parallela che consente strutturalmente un grado di cooperazione elevato tra attività parallele
  - lo scambio di informazione è facilitato rispetto ai meccanismi *IPC*
- talvolta il thread è chiamato processo leggero
  - la gestione del thread da parte del *SO* è assai meno onerosa rispetto alla gestione del processo (quanto a strutture dati e funzioni del *SO*)

concetto di thread

- thread – è un flusso (attività) di controllo che può essere svolto in parallelo con altri flussi (thread) nell'ambito di uno stesso processo
  - flusso di controllo = esecuzione sequenziale di istruzioni (macchina)
- il flusso di controllo del thread è una funzione (sottoprogramma), che viene messa in esecuzione alla creazione del thread
- poiché i thread sono attività parallele in uno stesso processo, essi condividono lo spazio di indirizzamento (quasi tutto – non la pila)
- i thread realizzano un'implementazione efficiente dello scambio di informazioni tra attività parallele
- scopo dello studio del thread: capirne il principio di funzionamento per comprendere / affrontare alcuni dei problemi principali che si incontrano nella programmazione concorrente

thread POSIX – pthread

- esistono svariati modelli implementativi di thread
- qui si considera lo standard *POSIX*
  - Portable Operating System Interface for Computing Environment
- standard *POSIX*
  - insieme di standard di interfacce applicative (*API*) di *SO*
  - utilizzando le funzioni e i servizi delle *API* di *POSIX*, è garantita la portabilità dell'applicazione su tutti i *SO* conformi a *POSIX*
  - lo standard definisce le *API*, ma non stabilisce quale debba essere la loro implementazione in uno specifico *SO*
    - *NPTL* (Native *POSIX* Threads Library): ultima implementazione, più efficiente, più aderente allo standard (sul modello "da uno a uno")
    - oggi in Linux si utilizza esclusivamente ***NPTL***
- pthread = thread di *POSIX*

```
thread: libreria + funzioni di libreria
processo: SO + system call
```

pthread e processo – 1

- un thread può essere attivato nell'ambito di un processo
- il processo costituisce l'ambiente di esecuzione del thread
- un generico processo può attivare uno, due o più thread
- quando un processo termina, terminano forzatamente anche tutti i suoi thread, qualunque sia il loro punto di esecuzione
- è necessario garantire la terminazione coerente dei thread
  - cioè garantire che quando il processo termina, tutti gli eventuali thread da esso attivati siano già terminati

pthread e processo – 2

- ciascun thread ha un identificatore di thread (*TID* – Thread IDentifier), di tipo **`pthread_t`**
  - l'identificatore di thread identifica il thread univocamente
  - l'identificatore di thread è diverso dall'identificatore di processo *PID*, che invece è di tipo **`pid_t`**
  - la funzione **`getpid`**, eseguita all'interno dei thread di un medesimo processo, restituisce sempre il *PID* del processo stesso
- ogni thread può essere posto in attesa di un evento
- pertanto l'esecuzione di un thread può essere sospesa in modo totalmente indipendente rispetto agli altri thread
  - per esempio per attendere la fine di un'operazione di ingresso / uscita

pthread – creazione

- primitiva (chiamata di sistema – *system call*) per creare un thread

  `pthread_create (...)`

- la primitiva **`pthread_create`** è simile alla primitiva **`fork`**
- con *create*, un thread ne crea un altro (nello stesso processo)
- bisogna passare a *create* come argomento il nome della funzione che il thread creato deve eseguire (funzione di thread)
- il thread inizia a eseguire il suo codice sempre dallo stesso punto (cioè dall'inizio della funzione di thread)
- qualunque sia il punto di esecuzione dove si trova il thread che lo crea (cioè ovunque *create* venga invocata)

pthread – attesa di terminazione

- primitiva (chiamata di sistema – *system call*) per attendere (cioè per sincronizzarsi con) la terminazione di un thread

  `pthread_join (...)`

- la primitiva **`pthread_join`** è simile alla primitiva **`waitpid`**
- con *join*, un thread si può mettere in attesa della terminazione di un qualsiasi altro thread dello stesso processo (ma non di un thread di un altro processo)
- similmente a *waitpid*, bisogna passare a *join* come argomento il thread la cui terminazione si vuole attendere
- la primitiva *join* è d'obbligo per garantire che, al termine di un processo, i suoi thread siano già terminati coerentemente

pthread – terminazione

- dato che il thread esegue una funzione, esso termina quando la funzione esegue **`return`** o, se **`return`** non è specificato, alla fine del codice eseguibile della funzione
- tramite **`return`**, il thread che termina può passare un codice di terminazione (codice di uscita) a un thread che ne attenda la terminazione
- il passaggio del codice di terminazione avviene se e quando un altro thread utilizza la primitiva **`pthread_join`**
- il codice di terminazione viene ricevuto dal thread in attesa tramite un argomento della primitiva **`pthread_join`**
- il codice di terminazione ricevuto è il valore restituito tramite **`return`** da parte della funzione eseguita dal thread

terminologia – 1

- esecuzione sequenziale
  - date due attività A e B nel codice di un programma, esse sono sequenziali se – esaminando il codice del programma stesso – si può prevedere se A verrà svolta ***sempre*** prima di B oppure se B verrà svolta ***sempre*** prima di A
- esecuzione concorrente
  - date due attività A e B nel codice di un programma, esse sono concorrenti se – esaminando il codice del programma stesso – NON si può prevedere se A verrà svolta sempre prima di B oppure se B verrà svolta sempre prima di A

terminologia - 2

- esecuzione parallela di processi / thread: simulata o reale
- simulata: sistema di tipo mono-processore, i processi / thread sono in esecuzione a turno sul processore (condivisione di tempo - *time-sharing*)
- reale: sistema di tipo multi-processore (*multi-core*), i processi / thread sono in esecuzione contemporaneamente su processori diversi
- in entrambi i casi i processi / thread sono concorrenti e per l'osservatore (utente) il modello del sistema è sempre lo stesso
  - naturalmente un sistema multi-processore potrà avere prestazioni superiori
- se il sistema è mono-processore, il termine concorrente è sinonimo del termine parallelo
- nota bene: anche su un sistema multi-processore, spesso il numero di processi / thread eccede il numero di processori, pertanto quasi sempre il parallelismo è in parte simulato
- tutte le considerazioni che verranno fatte nel seguito sono indipendenti dal numero di processori (uno o più)

pthread – esempio 1

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

// testata della funzione di thread - tf
void * tf (void * tID)

// variabili globali
pthread_t tID1, tID2;

// thread principale
void main ( ) {
   pthread_create (&tID1, NULL, tf, (void *) 1);
   pthread_create (&tID2, NULL, tf, (void *) 2);
   pthread_join (tID1, NULL);
   pthread_join (tID2, NULL);
 } /* main */
```

il thread principale (*main*) crea i thread secondari

il thread principale attende la fine dei thread secondari

cast per convertire il tipo da intero a puntatore

```
void * tf (void * tID) {
   // variabile locale di tf
   int conta = 0;

   conta++;
   printf (
       "sono il thread n: %d;
       conta = %d\n",
      (int) tID,
      conta
   );
   return NULL;
} /* tf1 */
```

cast per (ri)convertire il tipo in intero

`pthread_create` e parametri

- parametri di **`pthread_create (...)`**

per passare l'indirizzo della funzione di thread *tf*, il linguaggio C ammette le due scritture equivalenti **`tf`** ed **`&tf`**

```
pthread_create (&tID, NULL, tf, (void *) n);
```

cast per garantire che il tipo passato coincida con quello specificato nella testata della *tf*

1) indirizzo di una variabile di tipo **`pthread_t`**
   - per contenere l'identificatore del thread creato
2) puntatore agli attributi del thread creato
   - se NULL il thread avrà gli attributi di default
3) indirizzo della funzione eseguita dal thread creato (**`thread_function`**)
4) indirizzo dell'argomento che si vuole passare alla thread_function
   - si può passare un solo argomento, che deve essere di tipo **`void *`**
     - in linguaggio C il tipo **`void *`** è il puntatore universale, cioè è un indirizzo di memoria generico, e pertanto è anche assimilabile a un intero positivo o nullo
   - se si vogliono passare più parametri, è necessario creare una **`struct`** contenente tutti i parametri, e passare l'indirizzo della **`struct`**

`pthread_join` e codice di terminazione

- parametri di **`pthread_join (...)`**

```
pthread_join (tID, (void *) &thread_exit);
```

1) variabile di tipo **`pthread_t`** che contiene l'identificatore del thread la cui terminazione si vuole attendere
2) puntatore a una variabile che conterrà il codice terminazione (codice di uscita) passato dalla funzione di thread – se è NULL il codice non viene passato

```
// thread_function - tf
void * tf (void * arg) {
   // codice della funzione
   return  (void *) valore_da_restituire
} / * tf */
```

cast per garantire che il tipo puntatore coincida con quello specificato nell'argomento formale di *join*

cast per garantire che il tipo restituito coincida con quello specificato nella testata della *tf*

thread principale e thread secondari – 1

- un thread viene creato nell'ambito di un processo che ha già un suo flusso di controllo: **`main`**
  - si prende **`main`** come thread principale o di default
- pertanto quando si lancia un codice eseguibile, nel processo viene automaticamente creato un thread, chiamato principale
  - il thread principale è sempre il modulo **`main`** del processo
- tramite la primitiva **`pthread_create`**, in seguito il thread principale può creare altri thread, chiamati secondari
- un thread secondario può a sua volta creare altri thread secondari, idealmente senza limite di numero o annidamento
- quando un thread ne crea un altro, si ha esecuzione concorrente dei due flussi di controllo
- a differenza dei processi, i thread sono tutti pari tra di loro, cioè non conta chi ha creato chi (i thread non hanno relazione padre-figlio)

thread principale e creato – 2

- il thread viene eseguito nello stesso ***spazio di indirizzamento di memoria*** del processo che lo ha creato
- dunque tutti i thread di un processo condividono lo stesso segmento dati
- ciascun thread in un processo ha una sua pila, indipendente dalle pile degli altri thread e allocata in un (sotto)spazio di indirizzamento diverso
- pertanto le variabili locali della funzione eseguita da un certo thread appartengono solo a quel thread, poiché in C esse sono allocate in pila
  - in particolare, due thread che eseguno lo stessa funzione di thread hanno variabili locali con gli stessi nomi, ma i valori delle rispettive variabili locali possono differire
- pertanto le variabili globali (e le variabili statiche), che in C non sono allocate in pila ma nel segmento dati statici, sono condivise tra tutti i thread
  - le modifiche fatte da un thread a una variabile globale sono visibili a tutti gli altri thread
  - questa condivisione facilita lo scambio di informazione tra thread (che è immediato)
  - ma rende più difficile garantire la correttezza di un programma con thread

ancora esempio 1 – variabile locale

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

// testata della funzione di thread - tf
void * tf (void * tID)

// variabili globali
pthread_t tID1, tID2;

void main ( ) {
   pthread_create (&tID1, NULL, tf, (void *) 1);
   pthread_create (&tID2, NULL, tf, (void *) 2);
   pthread_join (tID1, NULL);
   pthread_join (tID2, NULL);
} /* main */
```

```
void * tf (void * tID) {
   // variabile locale di tf
   int conta = 0;

   conta++;
   printf (
        "sono il thread n: %d;
        conta = %d\n",
      (int) tID,
      conta
   );
   return NULL;
} /* tf1 */
```

esempio 1 – possibile sequenza di esecuzione

```
sono il thread n: 1; conta = 1
sono il thread n: 2; conta = 1
```

oppure

```
sono il thread n: 2; conta = 1
sono il thread n: 1; conta = 1
```

il risultato dipende dall'ordine di esecuzione della *printf* nei due thread

ancora esempio 1 – variabile globale

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

// testata della funzione di thread - tf
void * tf (void * tID)

// variabili globali
int conta = 0;
pthread_t tID1, tID2;

void main ( ) {
   pthread_create (&tID1, NULL, tf, (void *) 1);
   pthread_create (&tID2, NULL, tf, (void *) 2);
   pthread_join (tID1, NULL);
   pthread_join (tID2, NULL);
} /* main */
```

```
void * tf (void * tID) {
   conta++;
   printf (
        "sono il thread n: %d;
        conta = %d\n",
      (int) tID,
      conta
   );
   return NULL;
} /* tf1 */
```

conta ++ è conta = conta + 1

in linguaggio macchina

- *load* conta in registro
- incrementa registro
- *store* registro in conta

variabile globale – sequenza di esecuzione

```
sono il thread n: 1; conta = 1
sono il thread n: 2; conta = 2
```

oppure

```
sono il thread n: 2; conta = 1
sono il thread n: 1; conta = 2
```

l'uscita mostrata dipende da quale thread esegue per primo la sequenza seguente

```
conta ++
printf
```

tuttavia sono leciti (ottenibili) anche altri risultati ……

```
thread n: 1; conta = 2
thread n: 2; conta = 2
```

e viceversa

MA ANCHE …. !!!

```
thread n: 1; conta = 1
thread n: 2; conta = 1
```

ancora esempio 1 – variabile statica

```
#include <pthread.h> <stdio.h>
// testata di tf
void * tf (void * tID)

// variabili globali
pthread_t tID1, tID2;

void main ( ) {
   pthread_create (
      &tID1, NULL,
      tf, (void *) 1
   );
   pthread_create (
      &tID2, NULL,
      tf, (void *) 2
   );
   pthread_join (tID1, NULL);
   pthread_join (tID2, NULL);
} /* main */
```

```
void * tf (void * tID) {

   // variabile locale statica di tf
   // è dichiarata localmente, ma conserva il
   // valore tra una chiamata della funzione
   // di thread e la chiamata successiva
   // funziona come una variabile globale
   // n.b: però ha visibilità solo locale !

   static int conta = 0;

   conta++;
   printf (
      "sono il thread n: %d; conta = %d \n",
      (int) tID, conta
   );
   return NULL;
} /* tf */
```

`thread_function` – parametri e codice di terminazione

```
#include <pthread.h> <stdio.h>
// testata di tf
void * tf (void * arg)

pthread_t tID1;

void main ( ) {
   int argomento = 10;
   int thread_exit;
   pthread_create (
      &tID1, NULL, tf, (void *) argomento
   );
   pthread_join (tID1, (void *) &thread_exit);
   printf (
      "sono main: codice di uscita thread
       = %d\n", thread_exit
   );
} /* main */
```

```
void * tf (void * arg) {
   // variabile locale di tf
   int i;

   // cast conversione tipo
   i = (int) arg;
   printf (
      "sono thread_function:
       valore argomento = %d \n",
      i
   );
   i++;
   // cast (ri)conversione tipo
   return (void *) i;
} /* tf */
```

`thread_funct` – parametri e codice di terminazione

```
sono thread_function: valore argomento = 10

sono main: codice di uscita thread = 11
```

thread e processo – considerazioni sull'uso

- efficienza: la copia di memoria per un nuovo processo richiede tecniche di gestione e risorse più onerose rispetto a un thread
  - in generale il thread è più efficiente del processo
- protezione: un thread con errori può danneggiare altri thread nello stesso processo; invece un processo non può danneggiarne un altro (poiché i processi hanno spazi di indirizzamento di memoria disgiunti)
  - rispetto ai thread, i processi sono più protetti uno dall'altro
- cambiamento di codice eseguibile: è possibile solo con il processo; il thread può eseguire soltanto il codice della funzione di thread associata, già presente nel codice del processo
  - tramite la chiamata di sistema *exec*, un processo figlio può sostituire l'intero programma eseguibile (flessibilità)
- condivisione dei dati: la condivisione di dati tra due o più thread è molto semplice, mentre tra processi è complicata (richiede meccanismi di comunicazione tra processi – *InterProcess Communication* o *IPC*)