Azzolini Riccardo 2019-10-24

Thread

1 Task e programmazione concorrente

Un'applicazione può avere più compiti, detti comunemente **task**, da portare avanti contemporaneamente. Ad esempio, i task di un editor di testo potrebbero essere:

- gestione degli eventi generati dal mouse;
- gestione dell'input da tastiera;
- gestione dell'output a video;
- gestione della struttura dati del testo;
- controllo ortografico;
- salvataggio periodico dei dati.

Implementare singolarmente ciascuna di queste attività è relativamente semplice, ma scrivere un programma sequenziale che le faccia tutte è molti più difficile, perché bisogna fare in modo che esso "salti" da un'attività all'altra e determinare i momenti in cui ciò avviene.

Se, invece, il linguaggio utilizzato offre i **costrutti per la programmazione concorrente**, cioè i costrutti per

- definire flussi di esecuzione sequenziale indipendenti per i vari task
- eseguire i flussi concorrentemente

si può guadagnare in termini di:

- semplicità di programmazione;
- efficienza di esecuzione, se si ha parallelismo:
  - tra il lavoro di CPU di un task e il lavoro di I/O di un altro task;
  - tra il lavoro di CPU di due o più task (possibile solo su sistemi con processori multipli).

2 Programmazione concorrente con i processi

Il linguaggio C (ad esempio) offre funzioni di libreria che consentono di *realizzare i flussi mediante processi*. Questa soluzione ha però alcuni svantaggi:

- siccome ci sono numerosi processi, i context switch sono frequenti e lo scheduling è più complesso, quindi aumenta l'*overhead*;
- ogni processo accede solo alla propria memoria, però i flussi di esecuzione hanno bisogno di condividere dati: il problema è risolvibile (mediante le comunicazioni esplicite tra processi, cioè lo scambio di messaggi), ma in modo non semplice e non naturale.

In sintesi, programmando l'applicazione con più processi:

- l'attività di programmazione è più semplice;
- si rischia di non guadagnare in efficienza.

Infatti, i processi sono pensati per eseguire programmi indipendenti, non per "suddividere in parti" un singolo programma.

3 Thread

Una soluzione più adatta alla programmazione concorrente sarebbe avere gruppi di "processi light" che condividano memoria (testo e dati), file, dispositivi, e altre risorse, e che differiscano solo per lo stato della CPU.

Definizione: Un **thread** è un'esecuzione di un programma[1] che usa le risorse di un processo.

Ci possono essere più thread che sono associati a uno stesso processo e ne usano le risorse.

3.1 Possibile implementazione

- Le aree di testo e dati, e le risorse logiche e fisiche, sono condivise tra tutti i thread di uno stesso processo.
- Ogni thread ha i propri
  - identificatore (**TID**, **Thread IDEntifier**);
  - stato della CPU;

[1] In questa definizione, il "programma" può essere anche solo un sottoprogramma (ad esempio, una procedura).

- stack;
- stato (running, waiting, ready, ecc.);

Queste informazioni sono contenute nel **Thread Control Block** (**TCB**) corrispondente al thread, che è situato in una **thread table**.

4 Esempio di thread POSIX

Il programma in C riportato in seguito usa funzioni di libreria definite nello *standard POSIX* (supportato da quasi tutti i sistemi UNIX) per creare più thread e mandarli in esecuzione concorrentemente:

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

void *print_hello(void *num) {
    printf("Ciao, sono il thread numero %ld!\n", (long)num);
    pthread_exit(NULL);
}

int main(void) {
    pthread_t th[5];
    for (long t = 0; t < 5; t++) {
        printf("Creo il thread numero %ld\n", t);
        pthread_create(&th[t], NULL, print_hello, (void *)t);
    }
}
```

- `pthread_t` è un tipo che, intuitivamente, rappresenta un TID. L'array `th` dichiarato all'interno di `main` può quindi memorizzare 5 TID.
- `pthread_create` è la funzione di libreria per la creazione dei thread. I suoi 4 parametri sono:
  1. l'indirizzo di memoria nel quale memorizzare il TID del thread creato;
  2. gli attributi del thread (`NULL` per usare gli attributi di default);
  3. la funzione di tipo `void*` → `void*` che verrà eseguita dal thread creato;
  4. il parametro di tipo `void*` da passare a tale funzione.[2]
- `pthread_exit` è la funzione di libreria per terminare il thread corrente.

[2] Questo programma usa dei cast per passare come parametro un numero invece di un puntatore.

L'ordine di esecuzione dei thread non è prevedibile. Un esempio di output del programma è:

```
Creo il thread numero 0
Creo il thread numero 1
Ciao, sono il thread numero 0!
Creo il thread numero 2
Ciao, sono il thread numero 1!
Creo il thread numero 3
Creo il thread numero 4
Ciao, sono il thread numero 3!
Ciao, sono il thread numero 2!
Ciao, sono il thread numero 4!
```

4.1 Gestione degli errori

La `pthread_create` restituisce:

- 0 se la creazione del thread è avvenuta con successo;
- un codice di errore altrimenti.

Sarebbe opportuno controllare il risultato:

```
int x = pthread_create(&th[t], NULL, print_hello, (void *)t);
if (x != 0) {
    printf("Errore: %d\n", x);
    exit(EXIT_FAILURE);
}
```

5 Thread switching

Il **thread switching** si basa sullo stesso principio del context switching, ma ha effetti diversi. In particolare, esso introduce meno overhead, a vantaggio dell'*efficienza*, perché, ad esempio:

- non vanno aggiornati i dati della MMU relativi alle aree di testo e dati;
- grazie alla condivisione di testo e dati, è possibile che la cache della memoria richieda meno aggiornamenti;
- grazie alla condivisione dei file, è possibile che la cache del disco richieda meno aggiornamenti.

6 Efficienza

Oltre al thread switching, anche la creazione di un thread è più veloce rispetto alla creazione di un processo, perché:

- il TCB ha meno dati del PCB;
- non va allocata memoria per testo e dati.

Inoltre, due thread dello stesso processo possono scambiarsi informazioni con semplici letture/scritture di variabili condivise, situate nell'area dati. Invece, due processi possono comunicare mediante l'invio e la ricezione di messaggi, che avvengono attraverso delle system call, e quindi introducono overhead.

7 Esempio di variabili condivise

Il programma seguente mostra l'uso di una variabile condivisa:

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

int x; // condivisa dal main e da tutti i thread

void *print_hello(void *num) {
    printf("Ciao, sono il thread numero %ld!\n", (long)num);
    printf("x vale %d\n", x);
    x++;
    pthread_exit(NULL);
}

int main(void) {
    pthread_t th[5];
    x = 10;
    for (long t = 0; t < 5; t++) {
        printf("Creo il thread numero %ld\n", t);
        pthread_create(&th[t], NULL, print_hello, (void *)t);
    }
}
```

Un esempio di output è:

```
Creo il thread numero 0
Creo il thread numero 1
Creo il thread numero 2
Creo il thread numero 3
Ciao, sono il thread numero 1!
x vale 10
Creo il thread numero 4
Ciao, sono il thread numero 0!
x vale 11
Ciao, sono il thread numero 2!
x vale 12
Ciao, sono il thread numero 4!
x vale 13
Ciao, sono il thread numero 3!
x vale 14
```

Nota: questo programma non è scritto "correttamente", perché è soggetto a *race condition*: a seconda di come si alterna l'esecuzione dei vari thread, potrebbe accadere che alcuni incrementi della variabile `x` vengano persi.

8 Implementazione dei thread

Le funzioni di libreria `pthread_create` e `pthread_exit` sembrano wrapper di system call, cioè funzioni che invocano le (solitamente omonime) system call, ma *non è detto* che lo siano.

Infatti, i thread possono essere:

- **implementati nello spazio kernel**, cioè gestiti dal SO (quindi queste funzioni sarebbero wrapper di system call);
- **implementati nello spazio user** (quindi queste funzioni sarebbero semplici procedure, eseguite interamente in modalità user).

8.1 Implementazione nello spazio kernel

Con un'implementazione nello spazio kernel, i thread sono gestiti dal SO:

- La thread table contenente i TCB è una struttura dati situata nella kernel area della memoria.
- Lo scheduler deve schedulare i singoli thread, anziché interi processi.

- Il SO offre system call per creare/terminare thread, ecc., che il programmatore usa direttamente (se programma in assembly) o tramite le funzioni di libreria wrapper (se programma, ad esempio, in C).

8.2 Implementazione nello spazio user

- Il SO non sa nulla dei thread, ma gestisce (e in particolare schedula) solo processi.
- I thread sono gestiti da una libreria di procedure (**sistema runtime**) che eseguono in *modalità user*.
- Lo scheduler dei thread è una delle procedure della libreria, e la thread table (che contiene i TCB) è situata nell'area di memoria assegnata al processo (quindi nella user area).
- Il programmatore maneggia i thread usando le procedure della libreria, che permettono, ad esempio, di:
  - creare/terminare thread;
  - invocare lo scheduler per "cedere" il controllo agli altri thread (in POSIX, questa è la procedura `pthread_yield`).

Vantaggi:

- funziona anche su SO che non implementano i thread;
- il thread switching è più veloce, perché non richiede l'intervento del SO;
- l'algoritmo di scheduling può essere personalizzato, perché la libreria può mettere a disposizione diverse procedure di scheduling e consentire al programmatore di scegliere quale utilizzare.

Svantaggi:

- in sistemi con una CPU multithreading (che può eseguire più thread in parallelo), o in sistemi multiprocessore (che possono eseguire anche più processi in parallelo), il parallelismo è possibile solo tra i kernel thread;
- se un thread fa una system call che lo manda in waiting, il realtà per il SO va in waiting l'intero processo, quindi si bloccano anche gli altri thread.

9 Thread nei linguaggi di programmazione

Alcuni linguaggi supportano direttamente i thread, permettendo al programmatore di usarli senza bisogno di ricorrere esplicitamente alle funzioni di libreria per la gestione dei thread.

Ad esempio, Java permette di definire i thread come istanze di classi che implementano l'apposita interfaccia `Runnable`. Essa prevede un metodo, `run`, che contiene il codice eseguito dal thread (analogamente al `main` dei programmi classici).

Inoltre, è disponibile la classe `Thread`, che implementa `Runnable`, quindi un thread può anche essere un'istanza di una sottoclasse di `Thread`. Tale sottoclasse deve ridefinire il metodo `run`, dato che l'implementazione di default fornita da `Thread` non fa niente.

Nel codice di un thread è possibile usare:

- le variabili dichiarate nella classe che definisce tale thread;
- le variabili statiche di qualsiasi classe.

I thread Java possono essere user thread (implementati dalla JVM) o kernel thread (se il SO su cui gira la JVM supporta i thread).

9.1 Esempio

```
class ThreadUno extends Thread {
    private String threadName;

    public ThreadUno(String name) {
        threadName = name;
    }

    public void run() {
        while (true) {
            System.out.println(threadName + " " + MainThread.x);
        }
    }
}

class MainThread {
    static int x = 123;

    public static void main(String[] args) {
        ThreadUno t = new ThreadUno("EsempioThread");
        t.start();
```

```
    }
}
```

L'output (infinito) di questo programma è:

```
EsempioThread 123
EsempioThread 123
EsempioThread 123
EsempioThread 123
EsempioThread 123
...
```