CAPITOLO 5

THREAD

Un’applicazione può avere più compiti, detti comunemente **task**, da portare avanti contemporaneamente. Esempio dei task di un text editor:

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

Se il linguaggio offre i **costrutti 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, a patto che i vari task eseguano in parallelo:
  - parallelismo tra lavoro di CPU di un task e lavoro di I/O di un altro task;
  - parallelismo tra lavoro di CPU di due task, possibile solo se esistono più processori.

Esempio: il **C** offre le funzioni di libreria che consentono di **realizzare i flussi mediante processi**.

Però, se ogni flusso viene realizzato con un processo:

- i **context switch sono frequenti**, l'overhead può essere pesante;

- **ogni processo accede solo alla propria memoria**, ma i vari flussi di esecuzione devono condividere dati: con le comunicazioni esplicite tra processi (scambio di messaggi) il problema è risolvibile, ma in modo non semplice e non naturale;

- il S.O. deve tener traccia di numerosi processi, lo **scheduling risulta più complesso** ed aumenta ulteriormente l'**overhead**.

In sintesi, programmando l'applicazione con più processi:

- l'attività di programmazione è più semplice, ma

- si rischia di non guadagnare in efficienza.

Il secondo aspetto non sorprende, perché l'ambiente a processi è stato pensato per processi associati ad applicazioni diverse (multiprogrammazione), e non per realizzare applicazioni con processi concorrenti.

Idea: avere gruppi di “processi light” che condividano memoria (testo e dati), file, dispositivi ed altre risorse, e che differiscano solo per il CPU state.

Definizione: **Un thread è un’esecuzione di un programma che usa le risorse di un processo**.

Un processo può avere più thread. Possibile implementazione:

1. Aree testo e dati e risorse logiche e fisiche (file, dispositivi,…) sono condivise;
2. Ogni thread ha i propri identificatore (**Thread IDentifier, TID**), CPU state, stack e stato (running, waiting, ready,……).

Per ogni thread è necessario avere un **Thread Control Block** (**TCB**) con le informazioni elencate al punto 2. I TCB vengono organizzati in una **Thread Table**.

Il **thread switching** tra due thread dello stesso processo introduce meno overhead rispetto al classico context switch, a vantaggio dell'**efficienza**. Per esempio:

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

La creazione di un thread è più veloce della creazione di un processo, perché il TCB ha meno dati del PCB e perché non va allocata memoria per testo e dati.

Sempre a proposito di efficienza:

- due thread dello stesso processo possono scambiarsi informazioni con semplici letture/scritture di variabili dell'area dati, che è condivisa;

- due processi possono scambiarsi informazioni mediante invio e ricezione di messaggi, implementabili con system call che richiedono l'intervento del S.O., con tanto di TRAP, IRET e due salvataggi/settaggi dei registri generali.

Vediamo ora un esempio di un programma in **C** dove definiamo più thread che mandiamo in esecuzione concorrentemente.

Il programma usa funzioni di libreria, quali **pthread_create** e **pthread_exit**, che sono previste nello **standard POSIX**, cioè lo standard IEEE 1003.1.c.

Lo standard POSIX è supportato da quasi tutti i sistemi UNIX.

```
#include <pthread.h>  /* Esempio di thread in POSIX */
#include <stdio.h>
#include <stdlib.h>
void *PrintHello(void *threadid){     /* "programma" per i thread */
   long tid = (long)threadid;
   printf("Ciao, sono la thread numero #%ld!\n", tid);
   pthread_exit(NULL);
}
int main(  ){
   pthread_t th[5];
   long t;
   for(t=0; t<5; t++){  /* creo 5 thread, ognuno esegue PrintHello*/
      printf("Creo la thread numero %ld\n", t);
      pthread_create(&th[t], NULL, PrintHello, (void *)t);
   }
   exit(NULL);
}
```

- **pthread_t** è un tipo di dato definito nel **pthread.h**. L'idea è che le sue istanze siano TID.

- **pthread_create** è la funzione di libreria per creare i thread. Breve descrizione dei 4 parametri:
  - memory address ove memorizzare il TID del thread creato;
  - attributi (con **NULL** si forza l'uso degli attributi di default);
  - funzione di tipo ***void** → ***void** che deve essere eseguita dal thread creato;
  - parametro di tipo ***void** da passare a tale funzione;

- **pthread_exit** è la funzione di libreria per terminare i thread. Lo stack viene liberato.

I 5 thread creati dal **main** vanno in esecuzione in ordine non prevedibile. Vediamo due esecuzioni possibili nella prossima slide.

```
simone$a.out
Creo la thread numero 0
Creo la thread numero 1
Creo la thread numero 2
Creo la thread numero 3
Creo la thread numero 4
Ciao, sono la thread numero #0!
Ciao, sono la thread numero #3!
Ciao, sono la thread numero #1!
Ciao, sono la thread numero #2!
Ciao, sono la thread numero #4!
```

```
simone$a.out
Creo la thread numero 0
Creo la thread numero 1
Creo la thread numero 2
Creo la thread numero 3
Creo la thread numero 4
Ciao, sono la thread numero #2!
Ciao, sono la thread numero #4!
Ciao, sono la thread numero #1!
Ciao, sono la thread numero #3!
Ciao, sono la thread numero #0!
```

Un'osservazione. La **pthread_create** restituisce:

- 0 se è tutto ok
- un codice di errore altrimenti

Sarebbe opportuno controllare il risultato:

```
int x;
x = pthread_create(&th[t], NULL, PrintHello, (void *)t);
if(x!=0){
   printf("Errore: %d",x);
   exit(-1);
}
```

Vediamo ora due esempi:

- nel primo i thread condividono variabili;
- nel secondo i thread stampano i propri PID e TID.

```
#include <pthread.h>  /* Piccola modifica: definiamo una variabile x  */
#include <stdio.h>       /* che viene condivisa tra i vari thread */
#include <stdlib.h>
int x;  /* il main e tutti i thread condividono x */

void *PrintHello(void *threadid){
   long tid = (long)threadid;
   printf("Ciao, sono la thread numero #%ld!\n", tid);
   printf("x vale %d\n", x);
   x=x+1;
   pthread_exit(NULL);
}

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

Esecuzione possibile (nel Cap. 6 parleremo delle ***race condition***, alla luce delle quali questa esecuzione andrebbe commentata con attenzione):

```
simone$a.out
Creo la thread numero 0
Creo la thread numero 1
Creo la thread numero 2
Creo la thread numero 3
Creo la thread numero 4
Ciao, sono la thread numero #0!
x vale 10
Ciao, sono la thread numero #3!
x vale 11
Ciao, sono la thread numero #1!
x vale 12
Ciao, sono la thread numero #2!
x vale 13
Ciao, sono la thread numero #4!
x vale 14
```

```
#include <pthread.h> /* Esempio con stampa di PID e TID */
#include <stdio.h>
#include <stdlib.h>
int x;
void *PrintHello(void *threadid){
   pid_t pid = getpid( );
   pthread_t tid = pthread_self( ); /*chiamata POSIX per chiedere il TID*/
   printf("Ciao, pid = %d, tid = %d \n", pid, tid);
   pthread_exit(NULL);
}
int main(  ){
   pthread_t th[5];
   long t;
   for(t=0;t<5;t++){
        printf("Creo la thread numero %ld\n", t);
        pthread_create(&th[t], NULL, PrintHello, (void *)t);
        printf("Il suo tid e' %d\n",th[t]);
    }
}
```

Ecco una possibile esecuzione:

```
simone$a.out
 Creo la thread numero 0
 Il suo tid e' 1082132800
 Creo la thread numero 1
 Il suo tid e' 1090525504
 Creo la thread numero 2
 Il suo tid e' 1098918208
 Creo la thread numero 3
 Il suo tid e' 1107310912
 Creo la thread numero 4
 Il suo tid e' 1115703616
 Ciao, pid = 5424, tid = 1098918208
 Ciao, pid = 5424, tid = 1082132800
 Ciao, pid = 5424, tid = 1090525504
 Ciao, pid = 5424, tid = 1107310912
 Ciao, pid = 5424, tid = 1115703616
 simone$a.out
```

Domanda. Nei programmi **C** abbiamo usato funzioni di libreria, precisamente **pthread_create** e **pthread_exit**, che sembrano **wrapper** di system call, cioè funzioni di libreria che invocano le (solitamente omonime) system call.

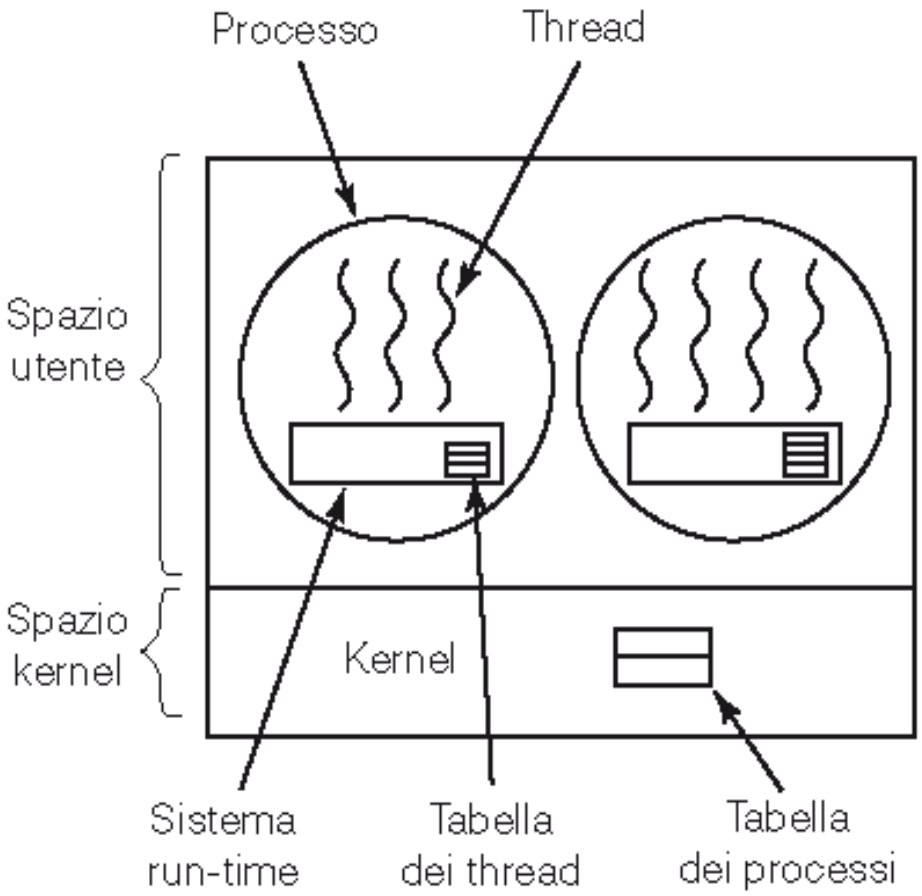
E' veramente così?

Risposta: dipende! Infatti, è possibile:

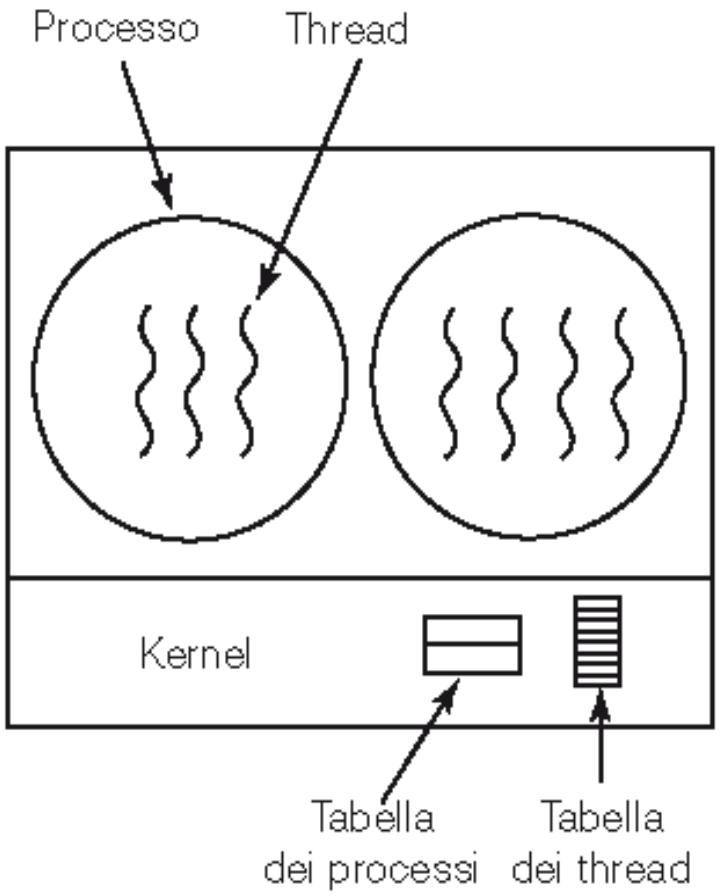
- **implementare i thread nello spazio kernel**. In questo caso la risposta sarebbe si;

- **implementare i thread nello spazio user**. In questo caso la risposta sarebbe no.

Fig. 2.16 Tanenbaum: thread nello spazio user e kernel.



USER THREAD



KERNEL THREAD

Thread implementati nello **spazio kernel**, probabilmente quel che ci si aspetta che venga considerato in un corso di S.O.:

1. il S.O. gestisce i thread;
2. i TCB sono strutture dati gestite dal S.O.;
3. il S.O. offre system call per creare/terminare thread, per far cambiare loro il programma come avviene per i processi, ecc;
4. lo scheduler del S.O. deve schedulare thread anziché processi;
5. il programmatore maneggia i thread chiamando le funzioni di libreria wrapper di system call.

Per esempio, **Native Posix Thread Library** (**NPTL**) è l'implementazione Linux, quindi a livello kernel, dei thread che rispetta lo standard POSIX.

Thread implementati nello **spazio user**:

1. il S.O. non sa nulla dei thread, gestisce solo processi, mentre **i thread sono gestiti da una libreria di procedure (sistema runtime) che eseguono in modalità user**;
2. i TCB sono strutture dati gestite dalla libreria di procedure;
3. esistono procedure per creare/terminare thread, per far cambiare loro il programma, ecc. Esiste anche una procedura (**pthread_yeld** in POSIX) per invocare lo scheduler al fine di "cedere" il controllo agli altri thread;
4. lo scheduler del S.O. schedula processi, una delle procedure della libreria è lo scheduler delle thread;
5. il programmatore maneggia i thread usando le procedure della libreria.

Per esempio, **GNU Portable threads** (**GNU Pth**) è una libreria di procedure che implementano lo standard POSIX e che sono invocabili da programmi C per sistemi UNIX.

Vantaggi dell'implementazione nello spazio user:

- funziona anche su S.O. che non implementano i thread.
- il thread switching non richiede l'intervento del S.O., quindi è più veloce;
- l'algoritmo di scheduler può essere personalizzato, cioè potremmo avere più procedure di scheduling.

Svantaggi dell'implementazione nello spazio user:

- in CPU multiprocessor o multithreading il parallelismo tra thread è possibile solo con kernel thread;
- se un thread fa una system call che manda la thread in waiting, quale un I/O su disco, in realtà per il S.O. va in waiting il processo, quindi si bloccano tutte i thread.

Spesso le implementazioni sono **miste** (non indaghiamo oltre).

Alcuni linguaggi, quali **Java**, anziché offrire le chiamate alle funzioni che gestiscono i thread (user o kernel), supportano direttamente i thread.

Vediamo ora un esempio di uso di thread in **Java**, precisando che l'implementazione può essere:

- con user thread (**green thread** in terminologia **Java**), dove la **JVM** offre la libreria di procedure (ormai obsoleta, vecchie implementazioni Solaris)
- con kernel thread, a patto che il S.O. su cui gira la **JVM** supporti i thread.
- mista, con impiego dei **lightweigth processes** (LWP): più user-level thread sono implementate col medesimo LWP, i LWP sono gestiti dal kernel.

```
class ThreadUno extends Thread{   \\ ESEMPIO DI THREAD IN JAVA
   private String threadName;
   ThreadUno(String name){ threadName = name; \\ costruttore banale}
   public void run( ) { \\ metodo run, obbligatorio per le thread
       for( ; ; ) {
       System.out.println(threadName+" "+MainThread.x);
          \\ usiamo la variabile x della classe MainThread
       }
   }
}
class MainThread {
   protected static int x = 123;
   public static void main(String[ ] args) {
   ThreadUno t = new ThreadUno("EsempioThread");
              \\ t è un oggetto di classe ThreadUno, cioè una thread
   t.run( ); \\ invocando il metodo run, la thread t viene eseguita
   }
}
```

- **Thread** è una classe che implementa l'interfaccia **Runnable**;

- Il metodo **run** è previsto nell'interfaccia **Runnable**, quindi le classi che implementano **Runnable** devono averlo. Corrisponde al **main** dei programmi classici;

- I thread Java possono essere:
  - istanze di sottoclassi di **Thread**, come in questo esempio,
  - istanze di classi che implementano **Runnable**;

- Un thread può usare le variabili dichiarate nella classe in cui il thread viene creato (come nel caso della variabile **x** dell'esempio).