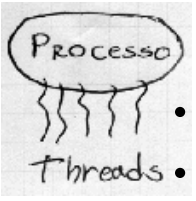
Appunti di sistemi operativi

Appunti per il corso universitario di sistemi operativi, riferito a sistemi Unix/Windows.
Si discute su problemi di sincronizzazione, memoria e scheduling dei processi.

THREAD

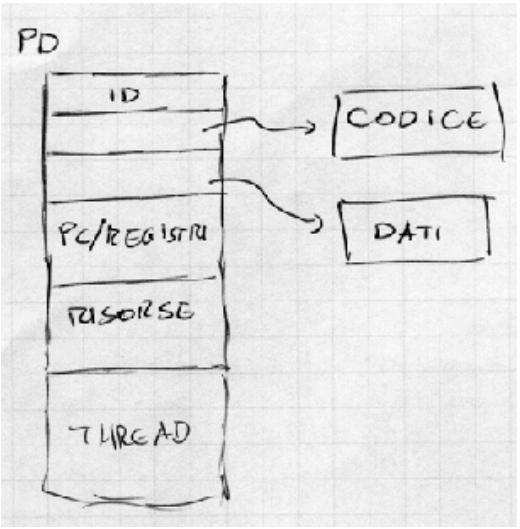


anche chiamati **processi leggeri** (**lightweight process**); un thread comprende:

- un identificatore di thread (ID)
- un contatore di programma
- un insieme di registri
- una pila (stack)

un processo al suo interno può avere più thread, ossia più programmi in esecuzione simultanea all'interno dello stesso processo

si può definire come parte dell'esecuzione di un processo

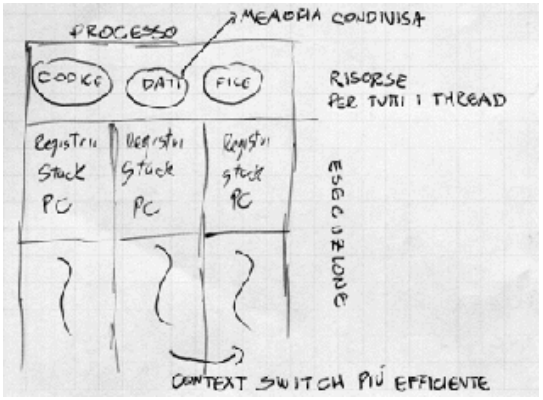


nel PD ci sono dati strettamente legati all'esecuzione del processo (codice, dati); altre no, come tutta la parte dell'allocazione delle risorse

possiamo pensare che all'interno di un processo ci siano più programmi, con diversi valori per i registri, per lo stack e per il PC, ma che attingono dallo stesso codice e dagli stessi dati, anche se possono avere delle variabili locali

la memoria tra i vari thread è condivisa

è più vantaggioso avere tanti thread che



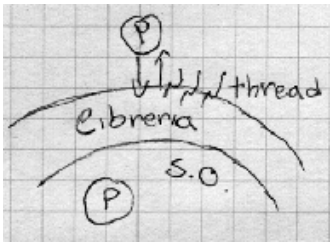
e **più vantaggioso avere tanti thread che tanti processi**, perché il thread **condivide i vari dati**, quindi ne ho meno da memorizzare; inoltre nello scheduling se **cambio thread** ho **meno dati da salvare** e ripristinare rispetto al context switch di processi

IMPLEMENTAZIONE:

- thread a livello **utente** (gestiti da librerie)
- thread a livello **kernel** (gestiti direttamente dal sistema operativo)

nella pratica abbiamo un ibrido tra le due soluzioni viste sopra

IMPLEMENTAZIONE THREAD TRAMITE LIBRERIE



con questo approccio, **dal punto di vista del sistema operativo non ci sono thread**: c'è un solo processo che al suo interno gestisce dei thread utilizzando una libreria

posso immaginare di avere dei meccanismi utente che permettano di passare l'esecuzione a altre parti di codice utente (un po' come nello scheduling tra processi)

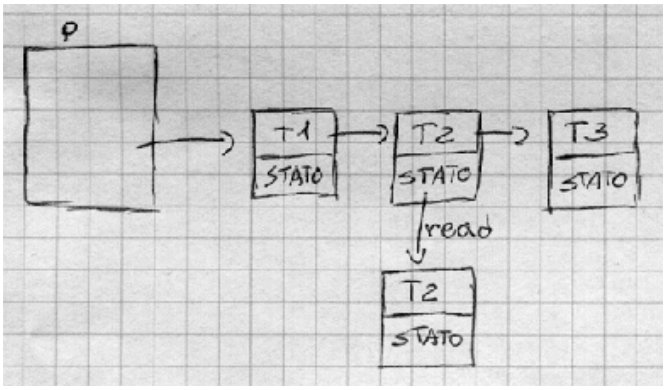
vantaggi:

- **kernel più semplice**
- **più efficiente**: non devo disturbare il sistema per cambiare thread (niente interrupt, niente cambio di contesto di fatto)

svantaggi:

- **scheduling complicato** (non so bene su quali eventi passare il controllo da un thread all'altro, perchè non ho una visione globale come il S.O.)
- supponiamo che ci sia un thread che effettua una read da una periferica di I/O: l'intero processo aspetta e quindi viene messo nella lista dei processi in attesa (il sistema non manda in esecuzione un altro thread dello stesso processo, non sa nemmeno che ci sono, in quanto vede solo un unico PD, quindi mette in attesa l'intero processo)
- se ho una macchina multi-processore, non posso parallelizzare i thread di uno stesso processo

IMPLEMENTAZIONE THREAD SU KERNEL



i thread sono gestiti direttamente dal S.O.

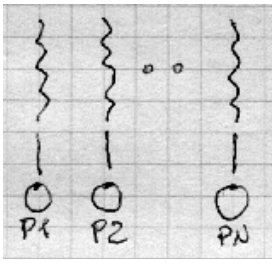
lo scheduling diventa più complesso, in quanto oltre a cambiare processo deve anche scegliere quale thread eseguire

MODELLI MULTITHREAD

MOLTI-A-1

ci sono più thread mappati su un unico processo

1-A-1

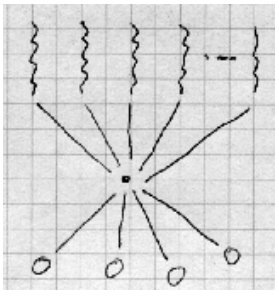


tipo di linux

non voglio complicare il kernel

quindi ogni thread lo vedo come processo, poi implementerò nello scheduler una gestione più efficiente

MOLTI-A-MOLTI



m x n (m >= n)

abbiamo m thread mappati su n processi

supponiamo di avere molti thread mappati su 2 processi: se ho in esecuzione un thread e si blocca, siccome il mapping è dinamico, posso cambiare l'associazione

questo sistema è tipico di solaris

mi permette di mantenere il sistema leggero, il cambio tra thread viene gestito da librerie

POSIX (PORTABLE OPERATING SYSTEM INTERFACE)

posix = standard di interfaccia al sistema operativo

```
#include <pthread.h>

// per compilarlo gcc program.c -lpthread


void *runner(void *param){

<codice del thread>

}
```

```
main(){

pthread_t tid; // id del thread

pthread_attr_t attr; // attributi

pthread_attr_init(&attr);

pthread_create(&tid,&attr,runner,Arg);

<......>

pthread_join(tid, NULL);

}
```

THREAD IN JAVA

in origine la JVM gestiva i thread con librerie

ora la JVM è eseguita in più thread a livello del S.O e quindi poi mappa i thread del programma eseguito

nei thread implementati come libreria avevo il problema delle chiamate bloccanti; qui no, es.

se ho una read su disco, viene interpretata dalla JVM, che si rende conto che la chiamata è bloccante e valuta come ottimizzare l'esecuzione

procedura per la creazione di un thread in java (utilizza l'**extends Thread**):

- si estende la classe thread e si sovrascrive il metodo run
- si crea un nuovo oggetto
- si invoca il metodo start, che si occupa di:
    - inizializzare il thread
    - richiamare run

c'è un altro metodo che utilizza l'**implements Runnable**

PRODUTTORE / CONSUMATORE IN JAVA

```
import java.io.*;

class Produttore extends Thread {
   public void run() {
      int temp=0; // metto il numero che produco
      while(true) {


         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

         String input = "" ;
         try {
            input = in.readLine() ;
         } catch (IOException e) { }
         if (input != null) {
            try {
               temp = Integer.parseInt(input) ;
            } catch (NumberFormatException e) { }
         }



         // attendo fino a che il consumatore non legge (busy waiting)
         while ((PC.inserisci + 1) % PC.dimensione == (PC.preleva)) {
            // qui dentro frego cicli alla cpu
```

```
            }

            // inserisco nel buffer il numero prodotto
            PC.buffer[PC.inserisci] = temp;

            // stampo il prodotto
            System.out.println("Produco " + PC.buffer[PC.inserisci]);

            // aggiungo 1 ad inserisci
            PC.inserisci = (PC.inserisci + 1) % PC.dimensione;
         }
      }
}

class Consumatore extends Thread {
   public void run() {
      while(true) {

         // attendo fino a che il produttore non produce (busy waiting)
         while ((PC.preleva) % PC.dimensione == (PC.inserisci)) {
            // qui dentro frego cicli alla cpu
         }

         // stampo il numero consumato
         System.out.println("Consumo " + PC.buffer[PC.preleva % PC.dimensione]);

         // aggiungo 1 a preleva
         PC.preleva = (PC.preleva + 1) % PC.dimensione;
      }
   }
}

public class PC {
   // memoria condivisa
   public static int dimensione = 10;
   public static int buffer[];
   public static int inserisci = 0, preleva = 0;

   public static void main(String [] args){
      buffer = new int [dimensione];
      Produttore p = new Produttore();

      Consumatore c = new Consumatore();
      p.start();
      c.start();
   }
}
```

continua..