Capitolo 4 : I Threads

Table of contents

Molte delle applicazioni moderne sono **multi-thread**, ovvero senza che l'utente si accorga di nulla, all'interno dell'applicazione sono presenti più attività che vengono eseguite simultaneamente. Vengono eseguiti quindi compiti diversi mediante esecuzioni separate.

Questo è possibile grazie all'hardware dei moderni calcolatori, dotati di processori multicore; per sfruttare la potenza di calcolo di questi sistemi, si è sviluppato, appunto, la programmazione multi-thread.

L'idea è quella di generare un qualcosa che possa essere eseguito **separatamente**, che sia più leggero della creazione dei processi. Questo significa che se voglio creare dei processi **figli** da un processo padre, ogni volta devono essere eseguite delle syscalls come **fork(), exec(), ecc**, e sono quindi costretto a creare più processi. Questa è una cosa più che fattibile, ma qualora avessi a disposizione un hardware **multicore**, potrei sfruttarlo al meglio facendo eseguire un diverso processo per core.

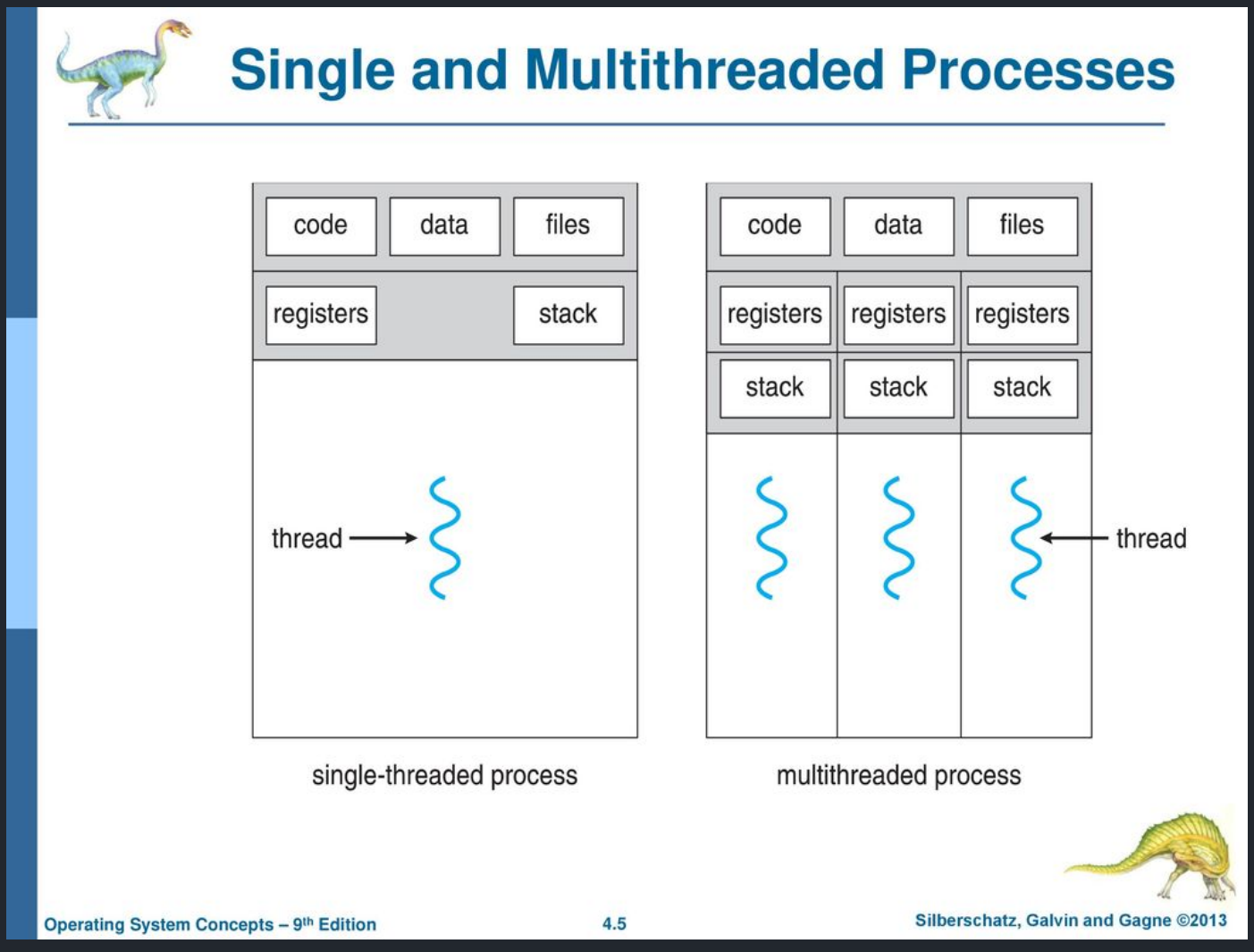
Il problema è che ogni processo riceve un proprio spazio indirizzi, quando voglio saltare da un processo all'altro, il processo deve eseguire un gran numero di operazioni, andando quindi a **"perdere tempo"**. Questo perchè il SO va a salvare lo stato del processo (ed altre operazioni come abbiamo visto nel CH 3), ogni volta che passa da un processo all'altro.

Perchè cambiare lo spazio degli indirizzi, e non far avvenire tutto all'interno di un unico spazio indirizzi?

La differenza fondamentale tra un'applicazione composta da **più processi separati** ed un'applicazione composta da **più thread**, è che i **processi** hanno un proprio spazio degli indirizzi,
i **thread**, invece, condividono lo spazio degli indirizzi del processo .

Questo significa che la creazione dei processi è detta **light-weight**, ovvero **leggera** (a differenza della creazione dei processi che è detta **heavy-weight**), siccome avviene in tempi anche di 100 volte minori rispetto ai processi.

Cosa significa lavorare all'interno dello stesso spazio degli indirizzi?



Processo a singolo thread

In un processo a singolo thread ho a disposizione i registri della CPU ed il Program Counter (che mi ricorda dove è arrivata l'esecuzione), ho inoltre uno **stack** di appoggio, che mi conserva tutto ciò che le varie funzioni hanno ritornato. Nel momento in cui il processo viene interrotto (per passare ad un altro) vengono salvati PC e Registri.

Sistema multi threaded

Utilizzo lo stesso codice, quindi la **stessa area text, stessa area dati inizializzata, stessi file aperti**, ma i percorsi di esecuzione diventano **multipli**, ovvero, mentre un un thread esegue qualocosa, ne è presente un altro che esegue un'altra cosa.

È quindi necessario che ogni thread (che fa un qualcosa) abbia uno **stack privato**, significa che **l'area stack** viene partizionata, in modo che ogni thread abbia uno stack diverso dagli altri. Non si può tassativamente mischiare i diversi stack, perchè, giustamente, ogni thread ha un processo che può potenzialmente eseguire un processo diverso dagli altri.

Quindi, ricapitolando, in un sistema multi-threaded abbiamo:

- Area Text **comune** (stesso codice)
- Area dati globali **comune**
- File aperti **comuni**
- **Diversi** percorsi di esecuzione, dove ogni percorso di esecuzione ha uno **stack privato**.

Cosa significa che l'area text (codice) è lo stesso per ogni thread?

Vuol dire che, essendo il codice uguale, ogni thread esegue la stessa operazione ? Ovviamente no.

Il concetto è che quando parte l'esecuzione, il processo è tradizionale, quindi a **singolo thread**, dopodiché ci saranno delle **particolari chiamate** che permettono di attivare ulteriori thread. Nel momento in cui questi nuovi thread vengono attivati, viene specificata la funzione che devono eseguire.

Di conseguenza, il codice è **unico**, ma ogni thread esegue una funzione diversa del codice comune. La funzione che il thread eseguirà viene specificata nel momento in cui esso viene attivato.

Cosa si guadagna con la programmazione multi-threaded?

Con questo sistema il guadagno evidente consiste nel fatto che diversi thread lavorano sullo stesso **spazio indirizzi**, e quindi la loro creazione è **velocissima**; il sistema non ha bisogno di riservare della memoria, siccome è già presente.

I thread vengono inoltre computati molto velocemente: Se un thread viene fermato ma ne viene avviato un altro, non viene sprecato del tempo, siccome si resta sempre alla stessa area di memoria, ho quindi la **massima velocità** nella computazione.

I thread, oltre ad eseguire compiti diversi, sono **singolarmente schedulabili**: se ho diverse CPU, ognuno di questi thread può andare in esecuzione insieme agli altri thread sui **core disponibili**, in modo da sfruttare l'architettura **multi-core**.

Diversi anni fa, la programmazione multi-thread non era molto diffusa, siccome la maggior parte dei processori non supportava più core (più CPU), ma con i sistemi attuali, praticamente tutti multi-core, la programmazione multi-thread è **basilare**.

La programmazione **multi-thread è il futuro della programmazione.**

Quali programmi sfruttano il multi-threading?

Solitamente i programmi più **CPU intensive**, come quelli di modellazione 3D, Matlab, programmi di grafica, sfruttano il multi-threading. Anche alcuni browser, come Chrome, utilizzano questo tipo di programmazione.

Programmi più leggeri come quelli che gestiscono le **e-mail** non sfruttano, quasi mai, il multi-thread.

Esempio ampiamente utilizzato per il multi-threading

L'esempio più comunemente utilizzato per spiegare il funzionamento dei thread è quello del **server** che deve servire delle **richieste**. Se ad esempio mi collego ad un server, questo riceverà delle richieste, da tutto il mondo, che deve "servire" nel minor tempo possibile.

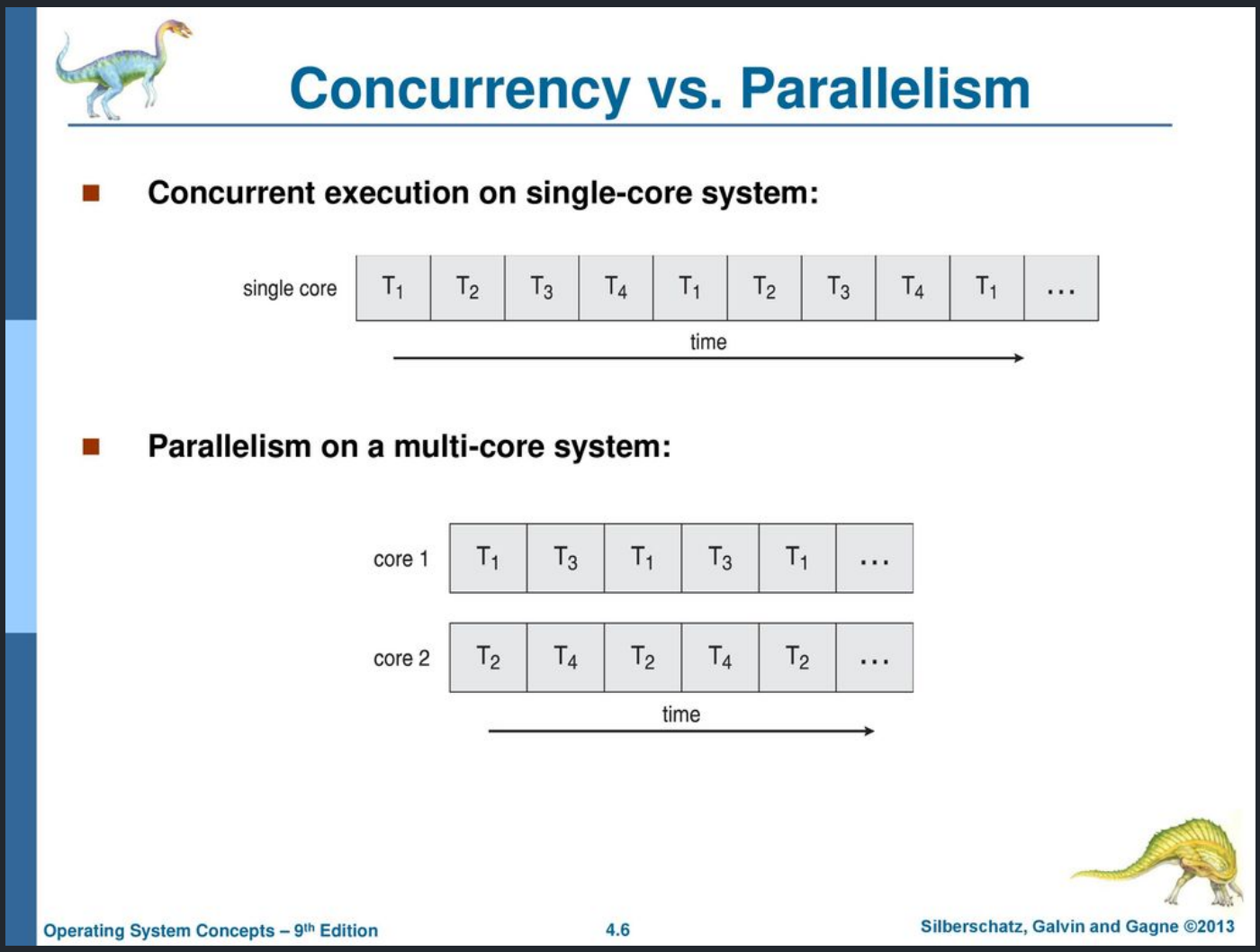
Se questo server fosse un processo a **singolo thread**, potrebbe servire una singola richiesta alla volta, ignorando tutte le richieste arrivate dopo quella corrente.

L'idea è quindi quella di avere una gestione concorrente delle richieste, in modo da avere un processo che gestisce le richieste, e ad ogni nuova richiesta assegnarla a diversi agenti, in modo da poter continuare a ricevere nuove richieste.

Conviene quindi realizzare questa struttura a **thread** invece di quella a processi. Anche perchè avremmo bisogno di un'area di memoria comune (per la cache), un compito perfetto per i thread, siccome essi **per definizione** condividono la **data area**.

I benefits (recap)

- **Responsività:** potrebbe permettere la continuazione dell'esecuzione se una parte del processo viene fermata, specialmente importante nelle interfacce grafiche.
- **Condividisione delle risorse:** i thread condividono le risorse del processo, rendendo il tutto molto più semplice della **memoria condivisa o dello scambio dei messaggi.**
- **Economia:** più "economici" della creazione dei processi, inoltre si ha un **overhead** (perdita di tempo) minore rispetto al **context switching** , ovvero lo scambio di contesto dei processi.
- **Scalabilità:** i processi possono prendere vantaggio delle architetture a multi processore.



Distinzione fondamentale

Qual è il modo per generare i thread?

Come faccio passare da un sistema ad ambiente a singolo thread a uno a multi thread? Fino agli anni 90 i SO prevedevano, come **entità minima** un **singolo processo.** Il sistema **UNIX** tradizionale prevede i pid (process identifiers) ma **non prevede i thread.**

Quindi la domanda sorge spontanea: come si è fatto a lanciare un processo multi-thread, quando **lo stesso sistema operativo non prevedeva questa possibilità?**

Fondamentalmente, i programmatori, hanno "inventato" delle **librerie** che in qualche maniera, all'interno del processo, **schedulassero separatamente** i diversi thread:

1. Scrivo il mio codice
2. Mi linko ad una **libreria** che mi fornisce delle **funzioni** per generare **thread multipli** all'interno del processo. Questi thread multipli possono funzionare in diversi modi:
    1. Un thread si sospende chiamando un altro

2. Vengono schedulati, a **time sharing** , da uno scheduler che è quello **interno al processo** , che è eseguito dalle funzioni della libreria utilizzata.

Importante: il Sistema Operativo, non ha idea di cosa stia accadendo, ma vede il tutto come un **semplice processo**!

Questo modo di esecuzione, si chiama **implementazione del thread spazio utente (user threads)**. Quando il SO non aveva il concetto di threads, ma solo di processi, era l'unico modo per mandare in esecuzione dei threads.

Inoltre, questo modo di esecuzione funziona in questo modo:
Se il SO concede 10ms ti tempo CPU al processo, in questo lasso di tempo vengono **alternati** i threads da eseguire.

Pro e contro

PRO

Il pro di utilizzare questo metodo per la programmazione multi-thread, è sicuramente il fatto che il sistema operativo, come già detto, non ha idea di cosa stia accedendo, o meglio, crede di eseguire un semplice processo. Di conseguenza, ho la possibilità di eseguire meno **system calls** (che utilizzano molto tempo di CPU - quindi overhead), e di conseguenza le operazioni divengono molto più veloci.

Contro

Se con questo sistema costruisco un modello simile ad un server che accetta delle richieste, il tutto **non funziona**. Questo perchè, come sappiamo, il SO non è a conoscenza del fatto che ci siano esecuzioni multiple (threads); infatti, quando un thread (utente) effettua una richiesta I/O (ad esempio di un'immagine) il SO **sospende l'intero processo**, e quindi anche gli altri thread, invalidando il tutto.

🏁 00:45 05-12

Distinzione fondamentale

Bisognerebbe distinguere tra la **API**, ovvero le primitive invocate dal programmatore per creare e gestire i thread, e la loro **implementazione**. Questo vuol dire che il programmatore può invocare queste primitive, ma non ha idea di come esse siano implementate, siccome la loro implementazione dipende dalla libreria in uso.

Threads user e kernel

User Threads: La gestione è fatta da librerie di thread a **livello utente**.
Esistono infatti tre librerie primarie dei threads:

- POSIX **Pthreads**
- Windows threads
- Java threads

Kernel Threads: supportati dal kernel. Questo è il caso dei sistemi operativi che supportano nativamente i threads. Tra questi ci sono:

- Windows
- Linux - discorso ampio
- Mac OS
- iOS
- Android - discendente da linux

User and Kernel Threads

Il kernel del sistema operativo, che viene attivato nel momento in cui arriva un **interrupt** o **eccezioni**, deve eseguire delle attività concorrenti. Bisogna inventare un metodo per rendere il kernel del SO più reattivo e sopratutto capace di effettuare più cose contemporaneamente.

Nel momento in cui il concetto di thread è diventato reale, ci si è posti la possibilità di realizzare anche il kernel del SO a thread, ovvero di "spezzettare" tutte le componenti del SO (kernel) in tanti thread, ognuno che facesse un task diverso, allo stesso momento.

Importantissimo: i **kernel threads** di cui parleremo da questo momento in poi, sono **thread all'interno del kernel stesso**.

Librerie di thread

Le librerie dei thread forniscono delle **API**, per creare e gestire dei thread; questi thread potrebbero essere implementati in **spazio utente** o a **livello kernel**.

Pthreads

`disclaimer:` tutte le componenti con la `P` iniziale, appartengono allo standard **POSIX**, che specifica una API per la creazione dei thread. In Linux la libreria è già inclusa.

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

int sum;
void *runner(void *param);

int main(){
  pthread_t tid;
  pthread_attr_t attr;

  // imposta gli attributi di default del thread
  pthread_attr_init(&attr);

  //crea il thread
  pthread_create(&tid, &attr, runner, argv[1]);

  // attende che il thread finisca
  pthread_join(tid, NULL);

  printf("sum = %d\n" , sum);
}
```

Questo nuovo thread, creato con `pthread_create()` eseguirà una funzione chiamata `runner`; questo vuol dire che all'interno del mio programma avrò una funzione **runner** che sarà chiamata per attivare il thread:

```
void *runner(void *param){
  int i, upper = atoi(param);
  sum = 0;

  for(i = i; i <= upper; i++){
    sum += 1;
  }

  pthread_exit(0);
}
```

Dal momento in cui viene eseguito `pthread_create()`, esistono due thread: il **main** ed il nuovo thread (composto da **runner**). Una delle cose che il thread principale potrebbe fare (il main) è chiamare la funzione `pthread_join()`, che permette di attendere la terminazione dell'altro.

Il secondo thread, infatti, quando completa la sua esecuzione, chiama la funzione `pthread_exit()`, che consente di terminare l'esecuzione del thread (e non il programma!).

Quindi, le funzioni vengono chiamate nell'ordine:

1. `pthread_attr_init()`
2. `pthread_create()`
    1. `pthread_exit()`
3. `pthread_join()`

Thread Java

Anche in java è possibile usare i thread. Questo è quanto.

Threading implicito

Esistono altri sistemi in cui l'attivazione dei thread non è **esplicita**, quindi non viene richiesto esplicitamente, ma viene eseguita in maniera **implicita** dal programma?

Il sistema più comunemente utilizzato è **openMP**, che è uno standard abbondantemente utilizzato, che permette di scrivere dei programmi i quali attivano automaticamente (in certi momenti) delle attività concorrenti. In particolare, openMP permette di **parallelizzare** i **loop**, in modo tale che ogni loop faccia un certo numero di iterazioni. In poche parole con openMP non è necessario attivare manualmente i thread, ma questi verranno attivati automaticamente quando serve.

Esempio:

```c
#include <omp.h>
#include <stdio.h>

int main(){
  #pragma omp parallel
  {
    printf("sono in una regione parallela");
  }

  return 0;
}
```

> anche in questo caso omp è incluso nei sistemi linux.

```c
#pragma omp parallel for
for(i = 0; i < N; i++){
  c[i] = a[i] + b[i];
}
```

> programma per la parallelizzazione dei loop

Nel caso della parallelizzazione dei loop, il sistema utomaticamente genera tanti thread quanti sono i core disponibili, e le varie iterazioni dei loop vengono distribuite sui processori disponibili.

Nel corso non vedremo nel dettaglio questo tipo di implementazione, ma serve solo a far capire che i thread non devono per forza essere programmati "a mano", ma esistono anche dei tool appositi per facilitarne la creazione.

Problemi del threading

Nei sistemi **UNIX** i threads si soprappongono a quelli che sono i **processi** normali, questo perchè i sistemi UNIX sono nati per gestire proprio dei processi.

Se ho un processo avente più thread **attivi** in quel momento, ed effettuo una **fork()** che succede?

`Risposta:` bella domanda!
Quando, tanti anni fa, è stata "inventata" la **fork()**, la si intendeva per l'utilizzo solo con i processi, e quindi il processo veniva duplicato. Ora che parliamo di threads, si dovrebbero duplicare anche i thread oppure bisogna avere un unico thread?

Il risultato è che molte semantiche dei sistemi UNIX diventano imprecise nel momento in cui ci sono dei thread.

Infatti, la stessa indecisione avviene nel momento in cui digito `ctrl+C` per effettuare la **kill** del processo in esecuzione (sul terminale): termino solo il thread in esecuzione o tutti?

Thread windows

Windows, sotto il punto di vista dei thread, è completamente fornito, essendo un SO moderno.

Thread Linux

Linux nasce invece come **sistema tradizionale**, dove l'obbiettivo era quello di avere un sistema quanto più monolitico possibile. Il vantaggio diretto di questo approccio è il fatto che linux riesca a girare senza problemi su un gran numero di macchine, anche quelle meno performanti. Tra gli svantaggi, però, è il fatto di non avere assolutamente il **concetto dei threads**.

Nel momento in cui è divenuto necessario l'utilizzo dei threads, anche per una questione di compatibilità, gli sviluppatori hanno dovuto trovare una soluzione, per nulla convenzionale:

Cominciamo con il dire che i threads non sono presenti. Ma se nel momento in cui effettuo una **fork()** do la possibilità di creare un altro processo che condivide l'area dati globale, il codice (area text), i file aperti, ma **non lo stack**, sommariamente sto creando un processo molto simile ad un thread.

Questo perchè non si poteva assolutamente modificare il comportamento della **fork()** classica, quindi solo ed esclusivamente nel sistema **linux**, la **fork()** chiama una system call aggiuntiva chiamata **clone()**.
Questa syscall può specificare cosa clonare e cosa no (non va ad esempio clonata l'area text). la **clone()** crea un nuovo prcesso che duplica **registri, stack e program counter**, ma non duplica **area text, area data e files aperti**.
La **clone()**, quindi, permette al **task** figlio di condividere lo spazio degli indirizzi del **task** padre.

Quindi

Come risultato abbiamo dei **processi** che **sembrano** dei **threads**, ma che non lo sono. La conseguenza è che la loro **computazione** è più lenta, così come la loro **creazione**. Capiamo quindi che i threads linux hanno delle prestazioni a **metà strada** tra i sistemi a **thread nativi** e quelli che utilizzano i **processi**.

Ovviamente la libreria **Pthreads** che viene utilizzata dai sistemi linux, fa utilizzo di questo meccanismo.

È semplice programmare con i threads?

No.

I thread non sono protetti in memoria tra di loro: i thread condividono lo stesso spazio degli indirizzi. Come conseguenza si ha che se un thread "sfonda" un array (aggiunge più elementi di quanto un array può contenere) o utilizza male un puntatore, va a **scrivere** sullo spazio usato da un altro thread, ed il SO non si accorgerà di nulla!

Con i processi, invece, se sfondiamo un array, otteniamo un **segmentation fault** (`Segfaults are caused by a program trying to read or write an illegal memory location`).