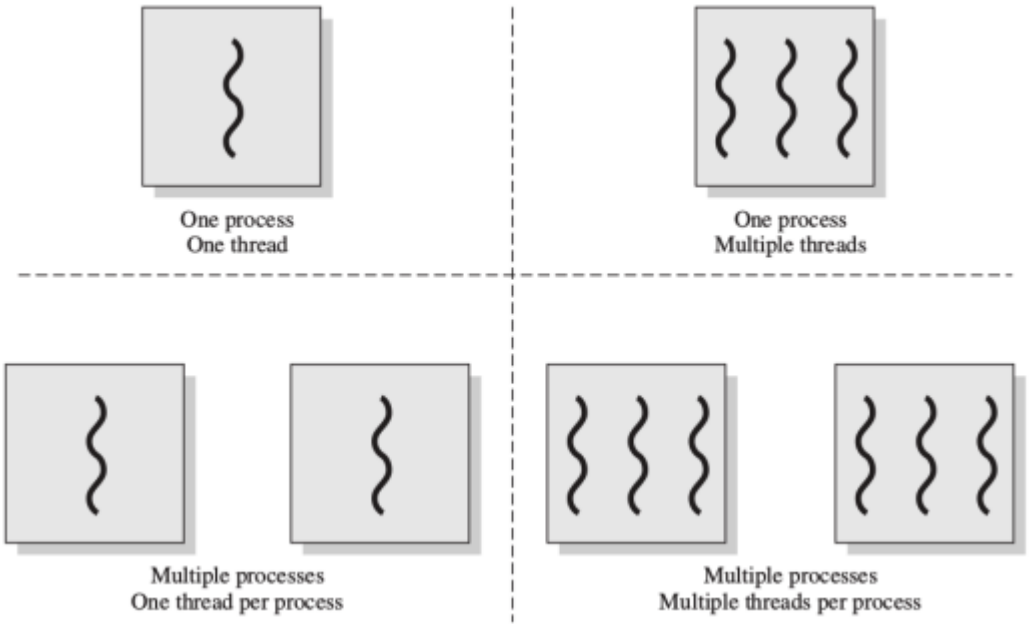
Thread

Index

Introduction

Finora abbiamo visto che ciascun processo compete con tutti gli altri alternando la loro esecuzione, tuttavia non è sempre così.Ci sono infatti delle applicazioni particolare (ad es. la maggior parte delle applicazioni GUI) che sono a loro volta organizzate in modo parallelo.
Infatti il programmatore dell'applicazione la ha suddivisa in diverse esecuzioni e ciascuna esecuzione è chiamata **thread**. Si tratta però di un processo, ma all'interno del processo è necessario di solito avere tre computazioni diverse (es. una monitora gli input, una che di conseguenza agli input ridisegni la finestra e una terza che faccia i calcoli richiesti) che devono poter avvenire contemporaneamente.

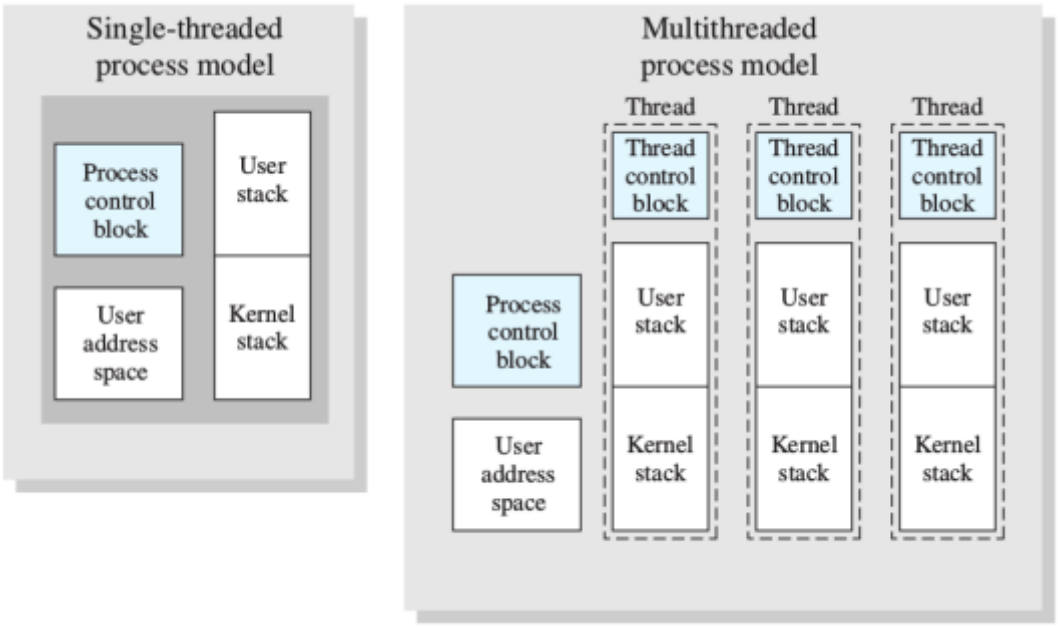


Diversi thread di uno stesso processo condividono le risorse **tranne** lo **stack** delle chiamate e il **processore**. Vengono quindi condivise le risorse input (un file aperto da un thread è disponibile anche a tutti gli altri thread), il codice sorgente ecc.

Teoricamente viene molto bene: si può dire che il concetto di processo incorpori le seguenti 2 caratteristiche

- gestione delle risorse (memoria, I/O ecc.)
- scheduling/esecuzione (stack e processore)

Dunque per quanto riguarda le risorse i processi vanno presi come un blocco unico, per quanto riguarda lo schedulung, i processi possono contenere diversi thread, e per questo vanno trattati in maniera diversa



Se ci troviamo in un sistema di processo a singolo thread troviamo la situazione affrontata fino ad ora (sinistra); se invece ci troviamo in un sistema a thread multiplo, non si ha solo l'immagine e il control block. Si ha infatti il:

- PCB e imagine comune (variabili globali ecc.) → parti condivise tra i thread
- Thread control block (uno per thread) → gestisce solo la parte di scheduling
- Stack dell'utente e stack di sistema (uno per thread)

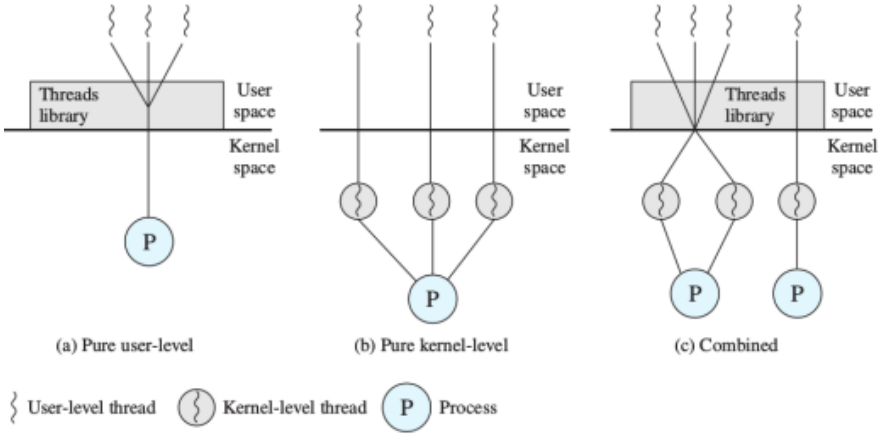
Il vantaggio di usare diversi thread per un processo piuttosto che diversi processi sta nel fatto che con il thread diventa più sempre (e più efficiente) la creazione, la terminazione, fare lo switch tra thread è più efficiente di fare lo switch tra processi e la condivisione di risorse. Dunque ogni processo viene creato con un thread e il programmatore tramite opportune chiamate di sistema può:

- `spawn` → creare un nuovo thread (simile a `fork()` ma più leggera)
- `block` → per far bloccare un thread, non per I/O, ma esplicito (es. aspettare un altro thread)
- `unblock` → sbloccare un thread (es. un thread che ha finito di fare una computazione sblocca un altro thread che era in attesa)

- `finish` → terminare un thread

ULT vs. KLT

I thread possono essere o a **livello utente** (User Level Thread) o a **livello di sistema** (Kernel Level Thread)



Negli ULT, i thread non esistono a livello di sistema operativo (il SO genera il processo ed è totalmente ignaro dell'esistenza dei thread) e opportune librerie (esistenti solo a livello utente) si occupano di gestire i thread
Nei KLT, i thread esistono a livello di kernel e le librerie dei thread si appoggiano direttamente sulle system call del SO

In base a cosa li scelgo?

Pro ULT

Gli ULT sarebbero meglio in quanto per fare lo switch tra due thread dello stesso processo non è necessario fare il mode switch, le librerie necessarie sono infatti tutte contenute dentro la modalità utente. Permettono anche di avere una politica di scheduling diversa per ogni applicazione e di usare i thread sui SO che non li offrono nativamente

Contro ULT

Se un thread si blocca, si bloccano tutti i thread di quel processo (a meno che il blocco non sia dovuto alla chiamata `block` dei thread) in quanto il sistema operativo non sa nulla dei thread e quindi quando questo si blocca il SO blocca tutto il processo.
Se ci sono effettivamente più processori o più core, tutti i thread del processo ne possono usare solamente uno (si tratta sostanzialmente di alternarsi su un core di un processore)

Se il SO non ha i KLT, non possono essere usati i thread per routine del sistema operativo stesso