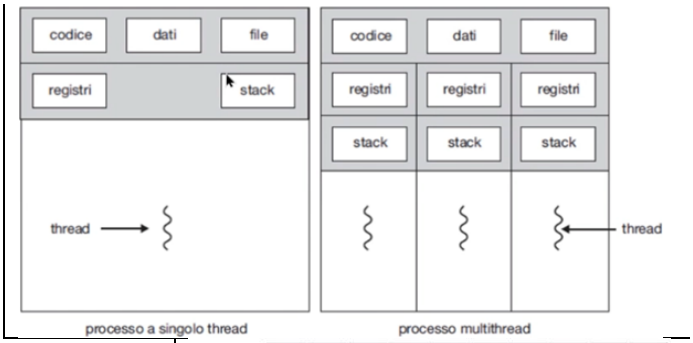
7.0 MULTITHREADING

Il **Multithreading** è l'estensione del concetto di processo. Alla base del concetto di thread sta la constatazione che la definizione del processo è basata su due aspetti: possesso delle risorse ed esecuzione. Questi 2 aspetti sono indipendente e il SO può gestirli in maniera indipendente, dunque l'elemento che viene eseguito è detto **thread** mentre l'elemento che possiede le risorse è il **processo**. Il termine **multithreading** è utilizzato per descrivere la situazione in cui ad un processo sono associati più thread. Supponiamo di mandare in esecuzione un web browser potrebbe avere:

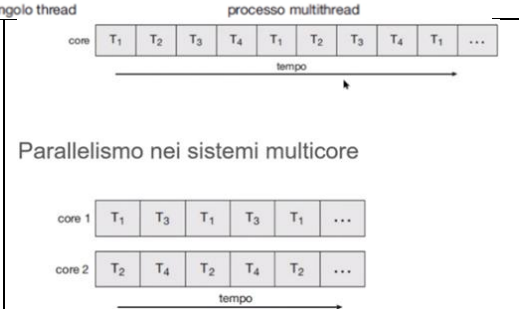
- Un thread per la rappresentazione sullo schermo di immagini e testo.
- Un thread per reperire i dati dalla rete.

Anche un Word processor potrebbe avere un thread per ciascun documento aperto (i thread sono tutti uguali ma lavorano su dati diversi).

Da quest'immagine si vede un processo a singolo thread (quello che abbiamo visto fino ad ora), in cui al processo viene assegnata una certa zona dati che conteneva codice, dati, etc. , e vi era un unico thread di esecuzione.
Per quanto riguarda invece un processo multithread, l'esecuzione viene partizionata in tanti thread, ognuno dei quali si occupa di qualcosa di specifico. Localmente ogni thread ha a disposizione dei registri e uno stack (per favorire l'esecuzione).
Una caratteristica del Multithreading è che si adatta perfettamente alla programmazione nei sistemi multicore, in cui abbiamo 2 CPU posizionati in uno stesso chip. Si può partizionare la computazione sui due Core, in modo da migliorare i tempi di esecuzione.



Vediamo la differenza tra concorrenza e parallelismo.
Piuttosto che partizionare i processi nel tempo, decido di assegnarli ai 2 core, come descritto in precedenza.



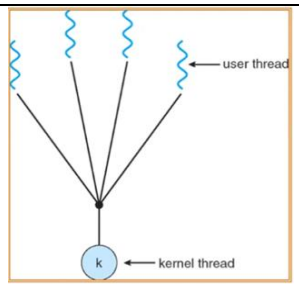
Il supporto del SO ai thread avviene in 2 modi:

- Supporto **user-level**: librerie di funzioni (API) per gestire n thread in esecuzione.
- Supporto **kernel-level**: tabella dei thread del sistema.

I Thread a livello utente sono gestiti come uno strato separato sopra il nucleo del sistema operativo, il kernel non ne è a conoscenza. Vengono realizzati tramite librerie di funzioni per la creazione, lo scheduling e la gestione dei thread. Il vantaggio è che può essere implementato un pacchetto di thread anche su SO che non supportano i thread. Uno svantaggio è che una chiamata di sistema bloccante da parte di un thread bloccherebbe tutti gli altri thread (il kernel blocca il processo). I Thread a livello kernel sono gestiti direttamente dal SO: il nucleo si occupa di creazione, scheduling, sincronizzazione e cancellazione dei thread nel suo spazio di indirizzi. Un vantaggio si ha perché quando un thread si blocca, il kernel può eseguire altri thread (anche dello stesso processo). Vediamo che tipo di relazione può esserci tra i thread a livello utente e i thread a livello kernel:

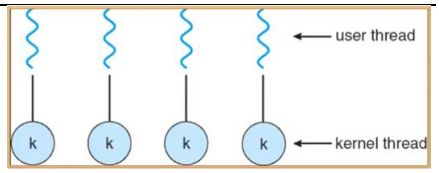
7.1 MODELLO MOLTI-A-UNO (M:1)

In questo tipo, ci sono molti thread a livello utente che però corrispondono ad un unico thread a livello kernel. I thread sono implementati a livello di applicazione, il loro scheduler non fa parte del SO che continua a vedere il tutto come un unico processo.
I vantaggi di questo modello è che la gestione dei thread è efficiente nello spazio utente (scheduling poco oneroso), non richiede un kernel multithread per poter essere implementato.
Gli svantaggi sono che l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante, inoltre, i thread sono legati allo stesso processo a livello kernel e non possono essere eseguiti su processori fisici distinti.



7.2 MODELLO UNO A UNO (1:1)

Ciascun thread a livello utente corrisponde ad un thread a livello kernel. Il kernel "vede" una traccia di esecuzione distinta per ogni thread. I thread vengono gestiti dallo scheduler del kernel (come se fossero processi). I vantaggi sono che lo scheduling è molto efficiente, se un thread effettua una chiamata bloccante, gli altri thread possono proseguire nella loro esecuzione. I thread possono essere eseguiti su processori fisici distinti. Gli svantaggi sono che può esserci inefficienza per il carico di lavoro dovuto alla creazione di molti thread a livello kernel, richiede un kernel multithread per poter essere implementato.



7.3 MODELLO MOLTI A MOLTI (M:M)

Si mettono in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel. In altre parole, il sistema dispone di un insieme ristreetto di thread (detti anche *worker*), ognuno dei quali viene assegnato di volta in volta ad un thread utente. I vantaggi sono che c'è la possibilità di creare tanti thread a livello utente quanti sono necessari per la particolare applicazione (e sulla particolare architettura) ed i corrispondenti thread a livello kernel possono essere eseguiti in parallelo sulle architetture multiprocessore. Inoltre, se un thread invoca una chiamata di un sistema bloccante, il kernel può fare eseguire un altro thread. Uno svantaggio è che c'è difficoltà nel definire la dimensione del pool di worker e le modalità di cooperazione tra i due scheduler.