28-03-2023

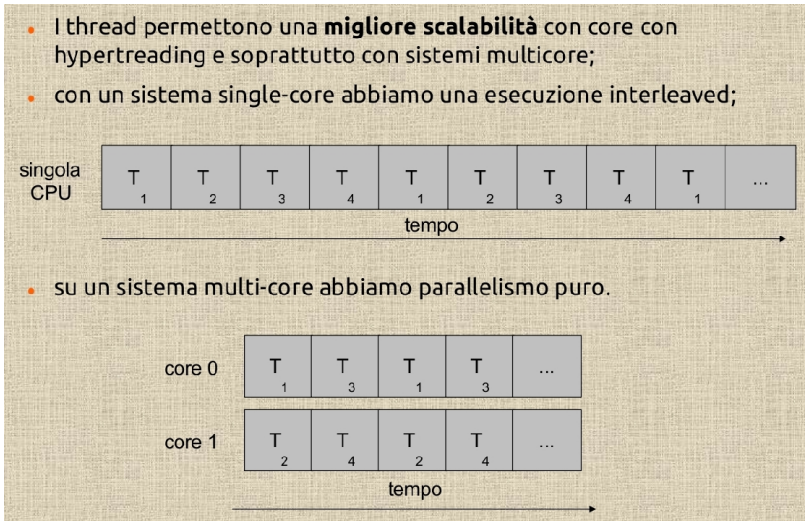
Operazioni sui Thread

In un sistema **MultiThread** si hanno almeno 2 thread per processo, **ogni processo inizia la sua esecuzione con UN SINGOLO THREAD** e gli altri thread verranno creati man mano.

Attraverso **funzioni di libreria** si possono avere:

- **CREAZIONE** di thread (`thread_create`): (un thread ne crea un altro) avviene **mediante una procedura** che avrà come **parametro di input** il **task che il NUOVO thread dovrà eseguire**.
  - Quindi si avrà una **rappresentazione GERARCHICA** proprio come i processi solo che, in questo caso, il thread figlio condivide le stesse informazioni dello stesso processo (*quindi c'è una sorta di legame diretto*)
- **TERMINAZIONE** dei thread (`thread_exit`): termina l'esecuzione di quel thread
- **LEGATURA** di thread (`thread_join`): un thread si sincronizza con la fine di un altro thread
- **RILASCIO** della CPU (`thread_yield`): i thread *rilasciano l'occupazione della CPU* attraverso questa procedura. Il thread si ferma in caso necessario visto che ogni processo non è dotato di `interrupt` di sistema.

Efficienza nell'utilizzo del multithread su un singolo core e su più core



Nel secondo caso si ha un parallelismo perchè si usano i thread in parallelo e il tempo di esecuzione si dimezza e quindi si **guadagna in velocità**.

Programmazione multicore

Si ha la possibilità di eseguire diversi thread su diversi processori (**core**).

> *La situazione è molto delicata*

Esistono dei **PRINCIPI** che permettono di avere le **migliori performance** in questo tipo di programmazione:

- **separazione dei task**: esaminare l'applicazione per individuare le aree che possono essere **separate** e quindi **raggruppate in task distinte** e **diverse** fra loro in modo da essere eseguite in **modalità parallela** su diversi core
- **bilanciamento**: individuare i task da poter essere eseguiti su core distinti in modo che ogni processore possa avere lo **stesso carico di lavoro**.
  - *Tutte le attività devono avere lo stesso carico di lavoro in modo tale che ogni core possa essere eseguito al massimo della propria potenzalità*.
- **suddivisione dei dati**: dividere i dati per essere eseguiti su core separati. Ogni task ha un suo compito e deve accedere ad una **PARTE LIMITATA DI DATI**.
- **dipendenze dei dati**: sincronizzare correttamente i dati. Può capitare che un'attività abbia bisogno di output dato da altre attività e quindi questo problema deve essere *evitato*. (*l'output diventa input di altre task fra diversi task*)
- **test e debugging**: non è fatto più su un singolo core ma su flussi di esecuzione che vanno su distinti core

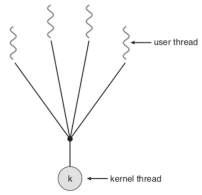
Gestione dei thread

I thread rilasciano in maniera volontaria la *CPU*. Se è il kernel che gestisce, allora serve un sistema hardware che consenta la possibilità di usare i thread ma non tutti gli hardware permettono tale procedura. Per bypassare questo problema i thread possono essere implementati in diversi modi:

- Se il kernel non consente di usare i thread, è possibile sviluppare i thread a **livello software** (le librerie e le operazioni accennate prima)
- Se il kernel lo consente allora si usano thread a **livello utente**

Thread a livello utente (modelli uno a molti)

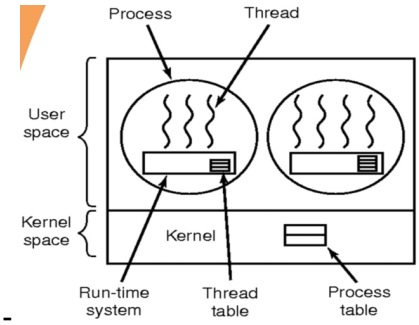
Si ha: *Un kernel -> molti thread*



Vengono implementati a livello software usando apposite librerie. Il kernel **non è a conoscenza dell'esistenza dei thread**. Il thread gestisce i processi, il processo crea e gestisce i thread (*tramite le librerie*).

Vantaggi

- SI possono usare quando il kernel **NON LO CONSENTE**
- l'esecuzione dei thread è **gestita dal processo** ed è il processo a gestire l'ordine di esecuzione dei thread (*scheduling*). In questo caso, l'ordine è scelto appositamente dallo stesso processo in base alle situazioni. Si parla allora di **scheduling personalizzato**.



- i thread vengono eseguiti su un sistema *run-time*
- ogni thread sono è eseguito in **modalità utente** e ad ognuno di essi viene associata una **tabella dei thread gestita dal sistema run-time** e tiene traccia delle informazioni relative ai thread
- *quando un thread esegue la **yield**, la gestione passa a run-time il quale decide quale altro thread mandare in esecuzione* e tale scambio è **IMMEDIATO** (altro vantaggio). Quindi **non ho bisogno** di fare `TRAP` al kernel

Quindi:

1. thread usabili dove fisicamente non è possibile
2. **velocità** nel cambio fra thread
3. possibilità di poter **gestire l'ordine di esecuzione dei thread (scheduling)**

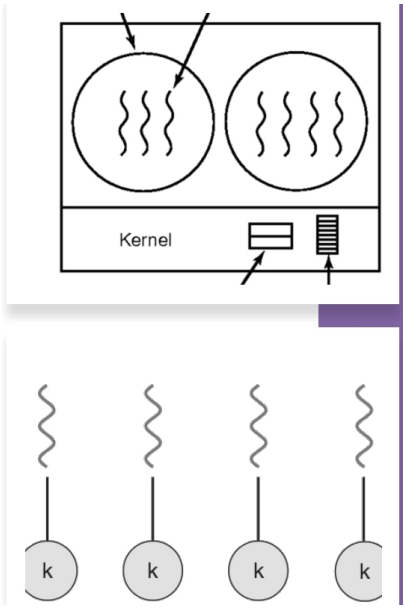
Svantaggi

- **Il kernel riconosce SOLO i processi**. Il processo genera i thread e li esegue.
  - Il kernel non può gestire i thread. Il **"quanto di tempo"** è assegnato al processo proprio per questo motivo. Il processo gestisce internamente la suddivisione fra i vari thread.
  - Quindi il kernel può bloccare il processo (interrupt) e quindi **NON** i thread.
  - Inesistenza di chiamate di interrupt da parte dei processi, quindi i processi non possono bloccare i thread
- **possibilità di non rilascio della CPU**: In caso di esecuzione di thread, non esistono chiamate di interrupt e quindi non può essere interrotto ma si può alternare se il thread rilascia volontariamente la CPU (potrebbe succedere che la CPU non venga rilasciata) e il "quanto di tempo" venga consumato dallo **STESSO THREAD** e viene meno il vantaggio offerto dai thread
  - *Se un **thread fa richiesta I/O**, essa va direttamente al kernel. Ma visto che il **kernel non conosce i thread**, per lui è il **processo stesso che chiede I/O** e quindi **blocca il processo** nonostante **gli altri thread potrebbero andare avanti** nella loro esecuzione e quindi non si sfrutta il vantaggio offerto dai thread.*
  - Una *soluzione*: usare una chiamata di sistema `select` attivata su un'attività: verifica se l'attività risulta essere bloccante o meno. Se non è bloccante viene eseguita, altrimenti sì. E' costosa e richiede l'adattamento dell'intero codice (quindi spesso non è usata)
  - `page-fault`: quando si cerca di accedere a informazioni non caricate in memoria. Il SO deve cercare all'interno del disco fisso. In questo caso un thread specifico fa `page-fault` e **blocca tutto il processo**.

Thread a livello kernel (modello 1 a 1)

Si ha: *Un kernel -> un thread*
Implementati con uno specifico supporto da parte del kernel. Non si ha un sistema run-time ma viene tutto gestito dal kernel. Non si ha più la tabella dei thread ma si ha solo "uno per tutti". Il kernel gestisce un'**unica tabella dei thread**

Il kernel conosce l'esistenza dei thread e quindi la loro gestione.



Vantaggi

- usare i thread così come sono stati pensati. In caso di **chiamte bloccanti** (*I/O*) da parte di un thread, si può **cambiare con un thread dello stesso processo** e il processo, quindi, **NON** si blocca.
  - *Qui non si ha il vantaggio fornito dai thread*
  - il kernel non blocca gli altri thread

> In caso di page-fault da parte di un thread, allora si può cambiare thread

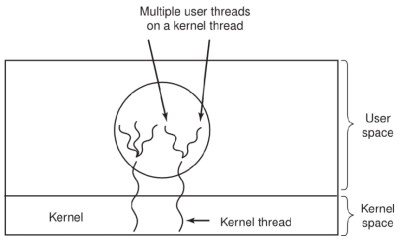
Svantaggi

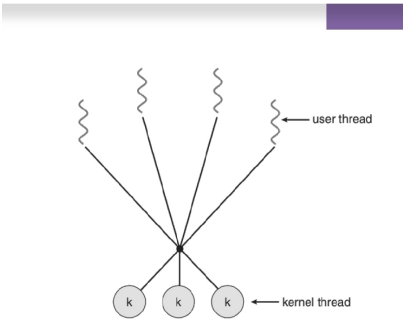
- tutti i thread seguono lo **STESSO SCHEDULING** perchè è *unico per tutti*
- cambio di contesto **più lento**
  - *Si hanno più chiamate di sistema **TRAP**(si ha un **rallentamento**)*

> *I PRO del livello utente corrispondono ai CONTRO del livello kernel e viceversa*

Modello ibrido (molti a molti)

Si ha: *Molti kernel -> molti thread utente*





Questo modello **sfrutta i vantaggi offerti dai 2 modelli descritti precedentemente** e quindi risulta essere molto **più flessibile**.
Consente di aggregare molti thread a *livello utente* e un numero più ridotto di thread a *livello kernel*.

- In questo modello, il kernel riconosce solo i thread implementati a livello kernel.
- i thread a livello utente non sono riconosciuto dal kernel

I thread dei nostri sistemi operativi

- Quasi tutti i sistemi operativi supportano i **thread a livello kernel**;
  - Windows, Linux, Solaris, Mac OS X,...
- Supporto ai **thread utente** attraverso apposite librerie:
  - *green threads* su Solaris;
  - *GNU portable thread* su UNIX;
  - *fiber* su Win32.
- **Librerie di accesso ai thread** (a prescindere dal modello):
  - *Pthreads* di POSIX (Solaris, Linux, Mac OS X, anche Windows);
    - una specifica da implementare sui vari sistemi;
  - threads Win32;
  - thread in Java;
    - wrapper sulle API sottostanti.

Su linux non c'è differenza fra *processo/thread* ma si parla di **attività** (*task*).