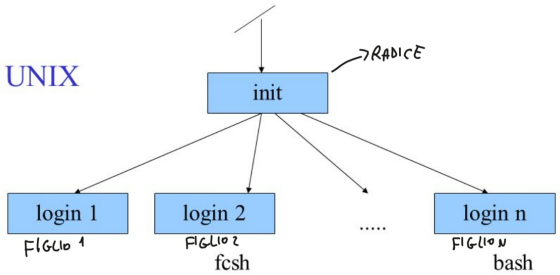
23-03-2023

Riassunto veloce

In informatica, un sistema operativo (abbreviato comunemente con i termini SO oppure OS, dall' inglese operating system, pronunciato operèiting sìstem) è un software di base adibito a gestire le risorse hardware e software di un computer
Ad ogni processi è associato il suo spazio degli indirizzi:
• codice eseguibile;
• dati del programma;
• stack;
• copia dei registri della CPU;
• file aperti;
• allarmi pendenti;
• processi imparentati.

- **Processo**: **porzione di programma** caricata in memoria. E' proprio quella che serve per essere eseguita
- E' possibile **scambiare il tempo di esecuzione** fra processi.
- Ogni processo ha il proprio **spazio degli indirizzi** che serve per interscambiare i processi fra loro
- Nella **tabella dei processi** (*Process Control Block*) ci sono le informazioni necessarie per la **cronologia di attività** svolta da quel determinato processo e serve anche per sapere da dove ripartire se questo è stato cambiato con un altro processo
- Durante lo scambio fra processi serve tenere conto del ***Program Counter*** (PC). Quest'ultimo è **unico** e ogni processo ha un suo PC logico per ogni processo. Poi questo dovrà essere caricato nel PC fisico in modo da continuare da dove si è fermato.
  - Quindi il *PC* logico viene **caricato nel *PC fisico***
  - In caso di **STOP**, il *PC fisico* viene **memorizzato sul *PC logico***
- Un processo può trovarsi in 3 stati diversi: *ready*, *running* e *blocked*.
- **CREAZIONE PROCESSI**. *3 modi*:
  - All'**avvio del sistema** e in questo modo si distinguono:
    - processi **ATTIVI**, consumano
    - processi **INATTIVI**, non appartengono a nessunun utente in particolare e sono in background (***daemon***), ovvero *dormienti* (come *il processo di stampa* che si attiva ogni volta che serve) --> Provare il comando `top` sul *Shell* per vedere i processi con il relativo stato
  - **Su richiesta dell'utente**: doppio click su un software
  - Un processo crea un altro processo tramite chiamate di sistema (`fork`) e ogni figlio, poi, sono due **ENTITA' SEPARATE** per quanto riguarda *l'attività che svolgono*(*spazio degli indirizzi diverso* ecc..)
    - E' sempre possibile trovare la **GERARCHIA DEI PROCESSI** e trovare i relativi padri dei vari processi

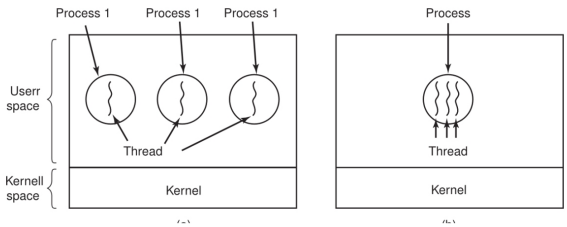


- **TERMINAZIONE PROCESSI**:
  - uscita **normale** (*volontario*) -> si usa la `exit` su UNIX o `ExitProcess` su Win32
  - uscita su **errore** (*volontario*): *per esempio* -> un puntatore punta a una locazione di memoria non istanziata
  - **errore critico**: *per esempio* -> compilazione di un file da terminale, cioè si scrive il nome di un *file inesistente*
  - **terminato da un altro processo**: tramite la chiamata di sistema `kill` (UNIX) oppure `TerminateProcess` (Win32)

- **TABELLA DEI PROCESSI**:
Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		
  - Se un processo ha bisogno di più memoria di quanta ce n'è disponibile, allora deve andare a prendere la memoria dall'insieme dei processi dello stesso utente e non di un altro a caso.
  - Chiamata di sistema **Page Fault**: un processo non ha tutte le informazioni necessarie in memoria e deve accedere al disco fisso per caricare queste informazioni in *RAM*. In questo caso il processo viene **BLOCCATO** perchè l'operazione in memoria è molto lenta.
    - Se **c'è spazio in RAM** allora si carica e basta
    - Se ci sono **più utenti** allora si deve occupare la memoria destinata a tale utente di quel processo in modo da non danneggiare altri utenti.
    - Se si ha poca memoria allora è più alta la probabilità che un processo chieda dei dati non presenti in memoria. Di conseguenza, se si ha più memoria la probabilità di tale chiamata di sistema è minore ma, comunque sia, tale chiamata verrà fatta prima o poi perchè, inizialmente, i processi iniziano a lavorare con dei dati incompleti, parziali.
    - Per questo motivo la tabella dei processi memorizza anche il **proprietario del processo**
- Ogni processo si divide in **mini sottoprocessi** ognuno con un compito diverso e in questo modo si ottimizza a **livello tempistico** e ogni miniprocesso è detto **THREAD**

I thread

Un processo è suddiviso in tanti filamenti destinati ad un'attività specifica che permette di simulare un **PARALLELISMO**.



Un thread è caratterizzato da:
• PC, registri, stack, stato;
• condivide tutto il resto;
non protezione di memoria.

> *Entità indipendente da altri processi che esegue dei dati e per fare questo si tiene traccia di molte informazioni. Quindi un processo è un entità che raggruppa risorse utili al processo.

- - Un processo, in generale, gestisce le risorse e poi le esegue.
- Le **risorse sono sempre uguali** ma il **flusso si può spezzare** in diversi (*almeno 2*) **sotto-flussi** di uno **STESSO PROCESSO** e ognuno di essi è detto **THREAD**

> *In caso si abbia solo **un processore/un solo core** allora si parla sempre di **PSEUDO-PARALLELISMO***

Vantaggi nell'uso dei thread

Un programma è fatto dal gruppo di risorse e dalla parte di esecuzione. Se la parte di **esecuzione si divide** su più attività distinte, le **risorse** rimangono le **stesse**. Quindi ogni thread usa le stesse risorse.

> Esempio: *I **file globali** sono **uniche** per tutti i thread*
> **Cambiano** solo **dati specifici** per quel thread, come le *variabili locali*

- **Si ha tutto caricato in memoria** relativa al processo e ogni thread si alterna fra loro e ogni scambio fra thread **non c'è bisogno di ricaricare dal disco nuovi dati** (che sono **pesanti**).
- In sintesi: uso **thread diversi** ma uso gli **stessi dati** che il processo ha caricato in memoria.
- Si ricava **un particolare guadagno di prestazioni** quando le diverse attività si alternano e quindi quando i processi sono ***I/O bound*** (fanno SPESSO richieste I/O)
  - Se un thread fa richieste I/O allora si blocca ***SOLO QUEL THREAD*** e gli altri vanno avanti

> *Rendono un sistema più veloce e riducono i tempi di attesa:*

Esempi

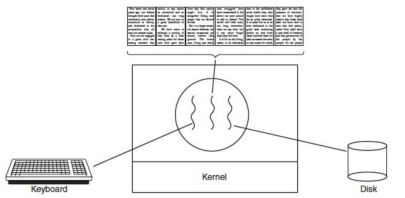


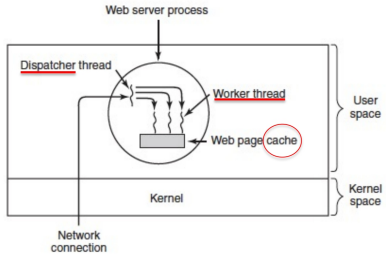
Figure 2-7. A word processor with three threads.

Un esempio può essre un **programma di video-scrittura** con un *testo già scritto* (il testo è già formattato)

- A partire dalla prima pagina costruisce la prima pagina per capire e formattare le successive
- In caso di *modifica di testo*, allora cambia tutto l'intero testo.
- Se ho **un solo processo**:
  - Esso si occupa dell'input/output e allo stesso tempo formattare
  - Se si cerca una parola, allora si deve analizzare tutto il testo e valutare il punto in cui si è arrivati in un certo istante di ricerca
- Se **usassi i thread**:
  - ognuno si potrebbe occupare di un'attività diversa.
  - Potrei usare 2 thread: **ricevere I/O** e **formattare**
  - Potrei usarne un altro thread che si occupa di salvare su disco fisso il file (*daemon*)

Un altro esempio:

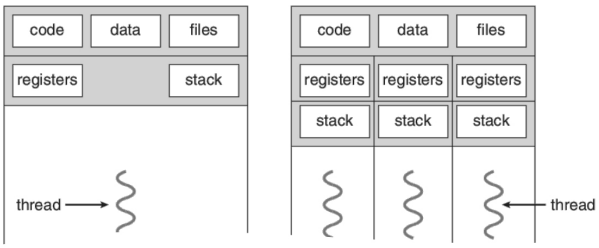
Un **web server process** usa un sistema **CACHE** in modo tale da fornire *più velocemente possibile* le pagine *più spesso visualizzate*



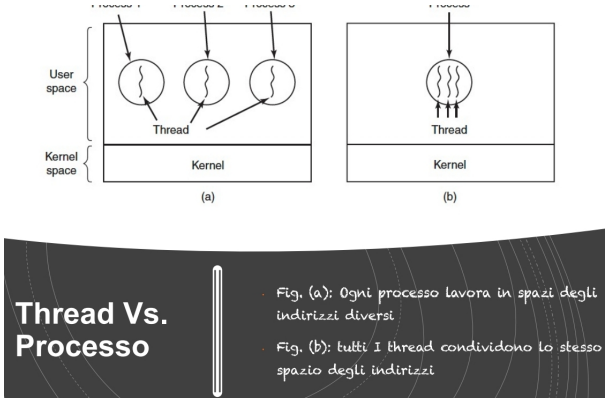
- Si devono usare i thread per assegnare a ognuon di essi un **compito specifico**: **ricevere** la richiesta (*dispatcher thread*), **cercare** la pagina nella cache (*worker thread*) , **caricare** dal disco fisso una pagina SE **non presente in cache**, visualizzare la pagina web
  - In questo *se un thread rimane bloccato* oppure rallenta l'esecuzione gli altri thread possono *continuare a lavorare* cooperando con gli altri per l'obiettivo finale **senza interferire uno sull'altro**

> *il **dispatcher** è un* ciclo infinito *che acquisisce richieste di lavoro e le passa ad un worker thread*

Modello Thread



- Nella *figura a sinistra* c'è un solo thread
- Nella *figura a destra* ci sono 3 thread (*filamenti*) e i dati relativi alla sua attività (`Program Counter` , `registri` , `Stack` , *attività specifica*) sono **PROPRI** del thread
  Si deduce il vantaggio (quando si **cambia un thread** con un altro thread): l'operazione è **più veloce** perchè si **devono scambiare solo le informazioni che riguardano il singolo thread** (che chiaramente sono minori rispetto alle informazioni memorizzate in un *INTERO PROCESSO* che si devono scambiare in caso di **scambi di processi**)



`Figura a:`

- **3 processi** con un solo thread
- **Il passaggio fra proessi** comporta il passaggio delle informazioni dal processo stesso con il successivo
- Questo richiede un notevole **consumo di tempo**

`Figura b:`

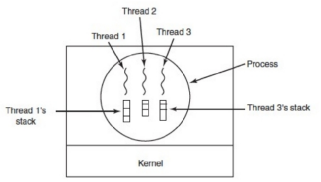
- **3 thread** e 1 processo
- lo **scambio** fra i thread è più **veloce**
- si completa l'operazione molto **più velocemente** rispetto alla `figura a`

Possibile problema (teorico) con i thread

- Thread dello stesso processo sono attività diverse fra loro ma **NON** sono del tutto indipendenti e questo è ai fini della condivisione di risorse e informazioni. (comunicazioni indirette)
- Questo vuol dire che condividono gli stessi dati e ciò potrebbe creare dei problemi perchè non c'è alcuna protezione nelle risorse condivise
    - *Teoricamente*, un thread può leggere, scrivere o cancellare lo stack di un altro thread
- *Praticamente*, però, le **informazioni condivise** sono usate **PER SCAMBIARE** informazioni ma visto che svolgono **attività diverse** allora non useranno mai la stessa parte di memoria e quindi ***NON SI SOVRAPPONGONO***.

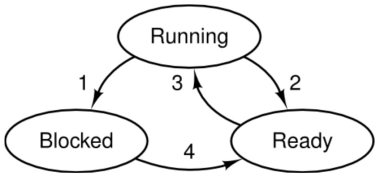
> *Se un thread apre un file, ciò è visibile ad ogni thread del processo. Sembra un problema ma la risorsa è gestita dal processo e non dal thread.*

Differenze fra processo e thread



Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

`A sinistra`: un processo con 3 thread e **ogni thread** ha un **proprio stack**
`A destra`: ci sono le **informazioni relative a un thread e ad un processo** e ogni thread **può assumere DIVERSI STATI** (*Blocked*, *Running*, *Ready*) proprio *come i processi*.



- Il **processo** è **gestore delle risorse**
- Il **thread** è il flusso di esecuzione del processo

Multithread: Ci sono **molteplici Thread** nello stesso processo

> *Lo scambio fra processi è circa 40 volte più lento rispetto a uno scambio fra thread*

Conoscenza del proprietario di un processo

Lo **SCHEDULING dei thread** determina l'**ordine di esecuzione** dei thread. Un processo è scomposto in thread perchè si **velocizzano le operazioni**. Se un thread si blocca, allora viene sostituito da un altro thread (dello stesso processo, chiaramente) seguendo le linee guida dello scheduling. Il SO, se conosce il **proprietario di un thread**, va a ricercare thread dello stesso processo dello stesso utente per utilizzare al massimo il vantaggio dell'uso dei thread.

28-03-2023

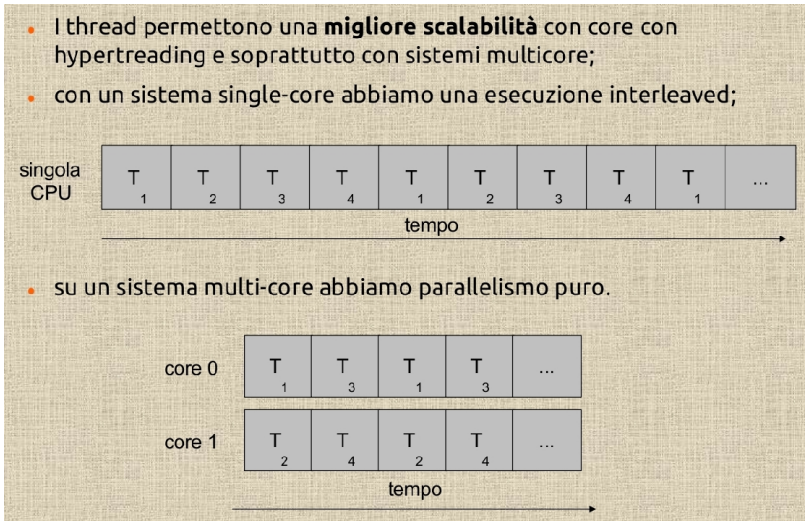
Operazioni sui Thread

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

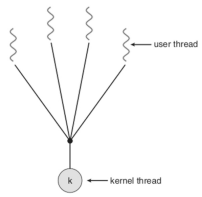
Gestione dei thread

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)

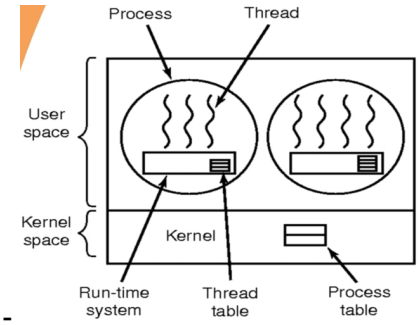
Si ha: *Un kernel -> molti thread*



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)**

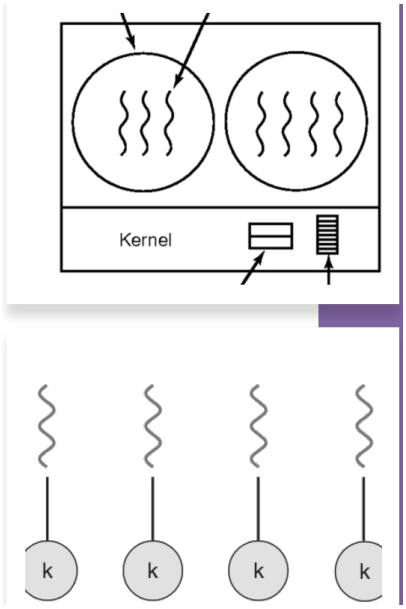
Svantaggi

- **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

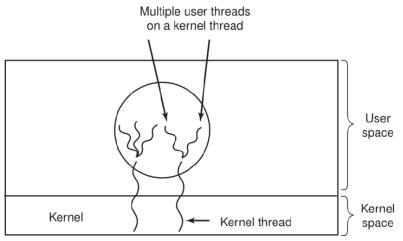
Svantaggi

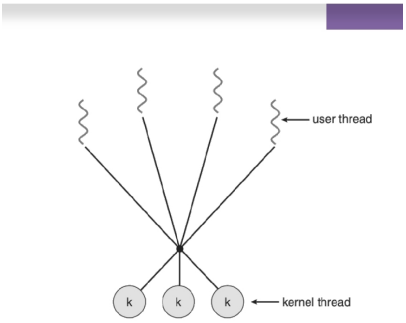
- tutti i thread seguono lo **STESSO SCHEDULING** perchè è *unico per tutti*
- cambio di contesto **più lento**
  - *Si hanno più chiamate di sistema **TRAP**(si ha un **rallentamento**)*

> *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*).