Threads

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2024/2025, Canale San Giovanni

Dai processi ai thread

- Sommario
  - Processi e thread
  - Thread al livello d'utente e al livello del nucleo
- Riferimenti
  - P. Ancilotti, M.Boari, A. Ciampolini, G. Lipari, "Sistemi Operativi", Mc-Graw-Hill (Cap. 2, par. 10)
  - www.ostep.org, Cap. 26
  - W. Stallings, "Operating Systems : Internals and Design Principles (5th Edition) ", Prentice Hall (Cap. 4)

Processi

- L'astrazione di "processo" incorpora due concetti importanti
  - **Esecuzione**
    - Ogni processo ha un **flusso di controllo** (esecuzione di una sequenza di istruzioni)
    - Ogni processo ha uno **contesto di esecuzione** (registri, stack, stato di scheduling, ...)
  - **Possesso di Risorse (e Protezione)**
    - Ogni processo definisce un proprio spazio di indirizzamento, che identifica tutti gli indirizzi di memoria a cui un processo può accedere
    - Ad un processo possono essere assegnate risorse di memoria e di I/O

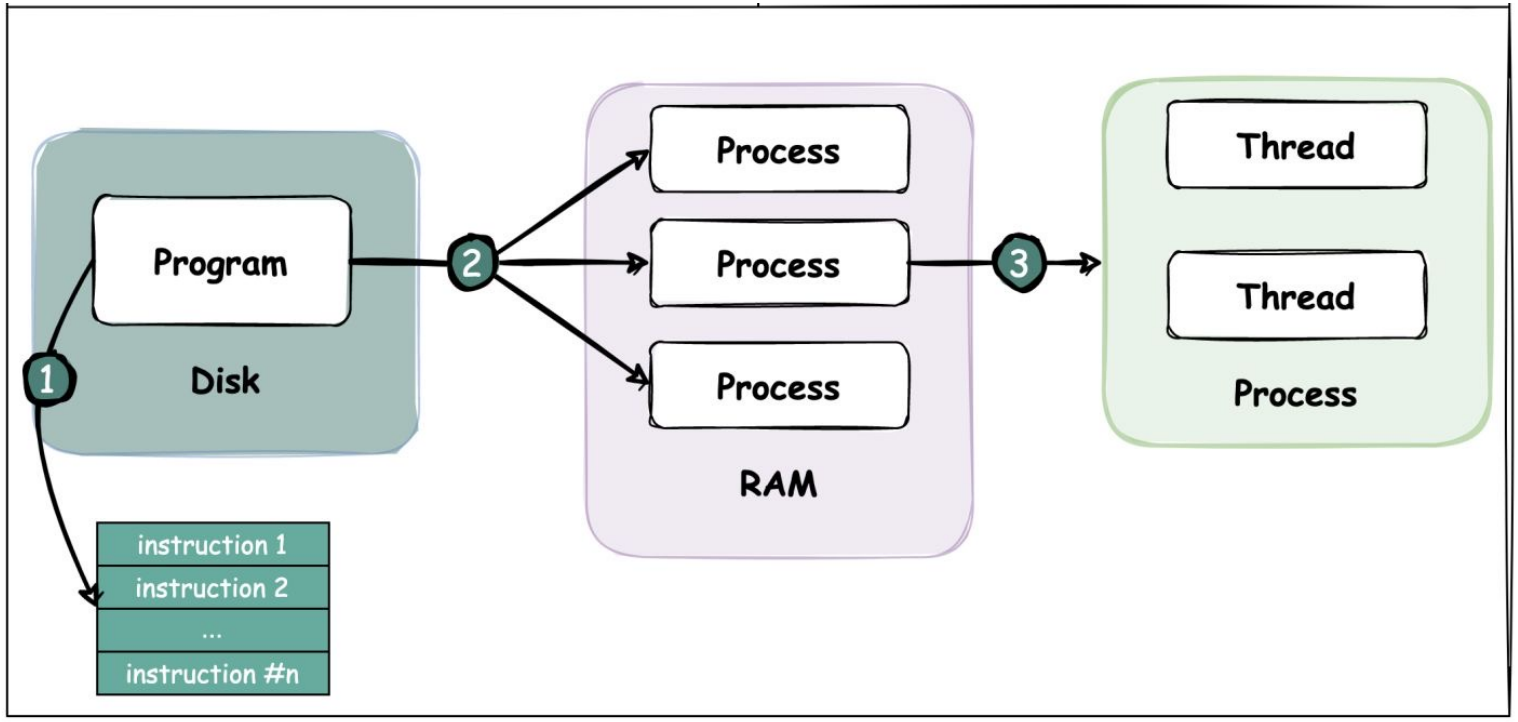
Processi e Thread

Concetto chiave:

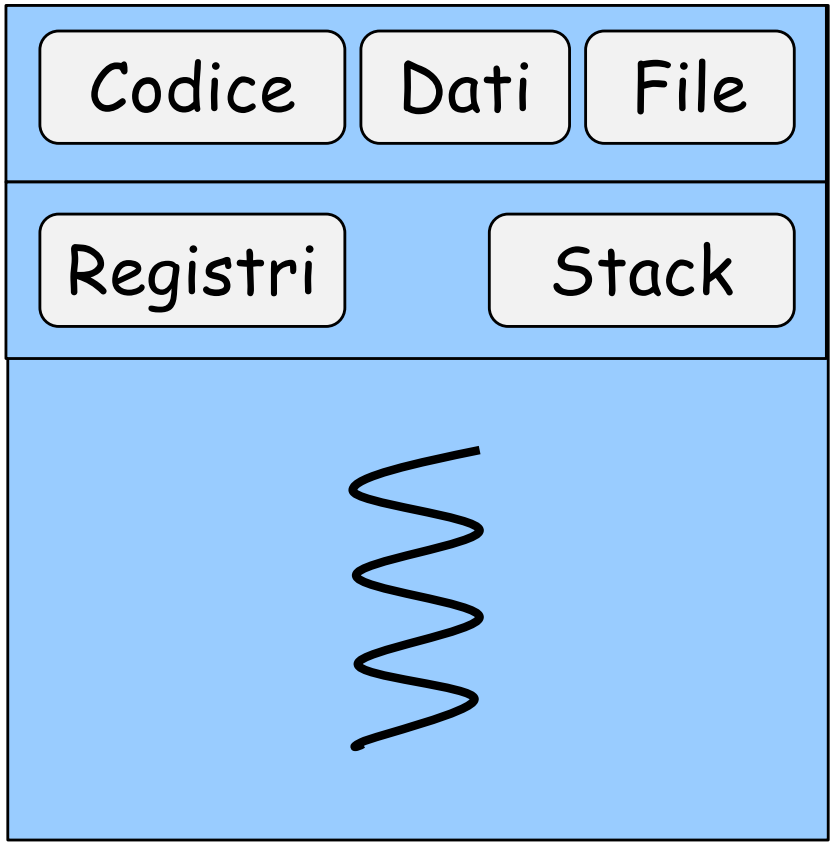
separazione tra esecuzione e possesso di risorse

- Un **thread** (o processo leggero) è un flusso di controllo sequenziale in un processo
- Un **processo** (o processo pesante) definisce lo spazio di indirizzamento e le risorse che possono essere condivise da **più threads**

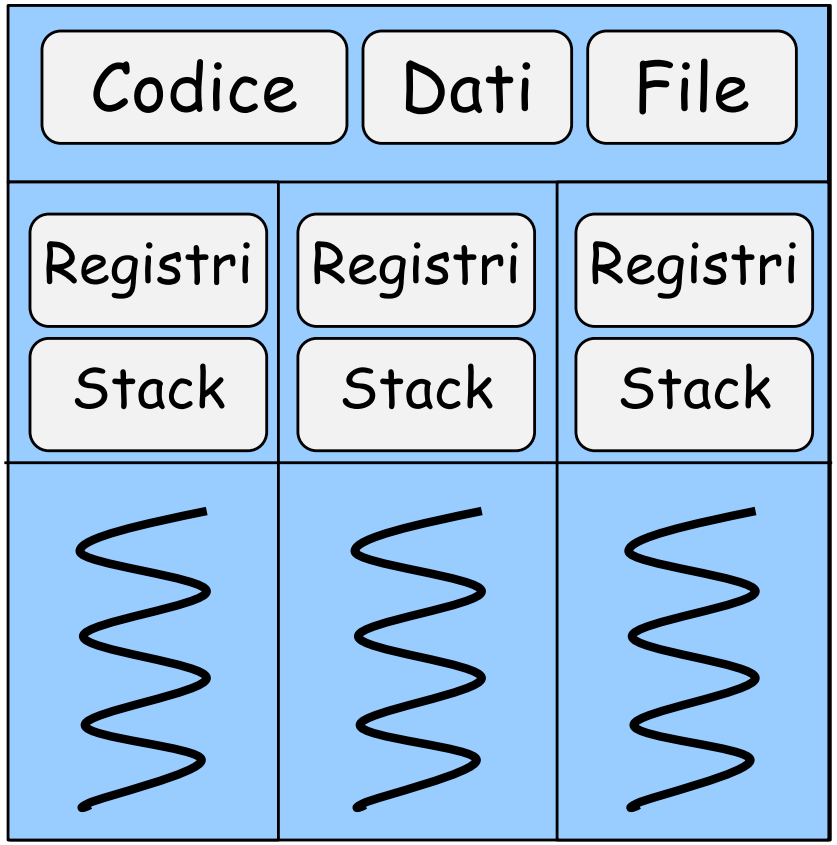
Processi e Thread



Processi e Thread



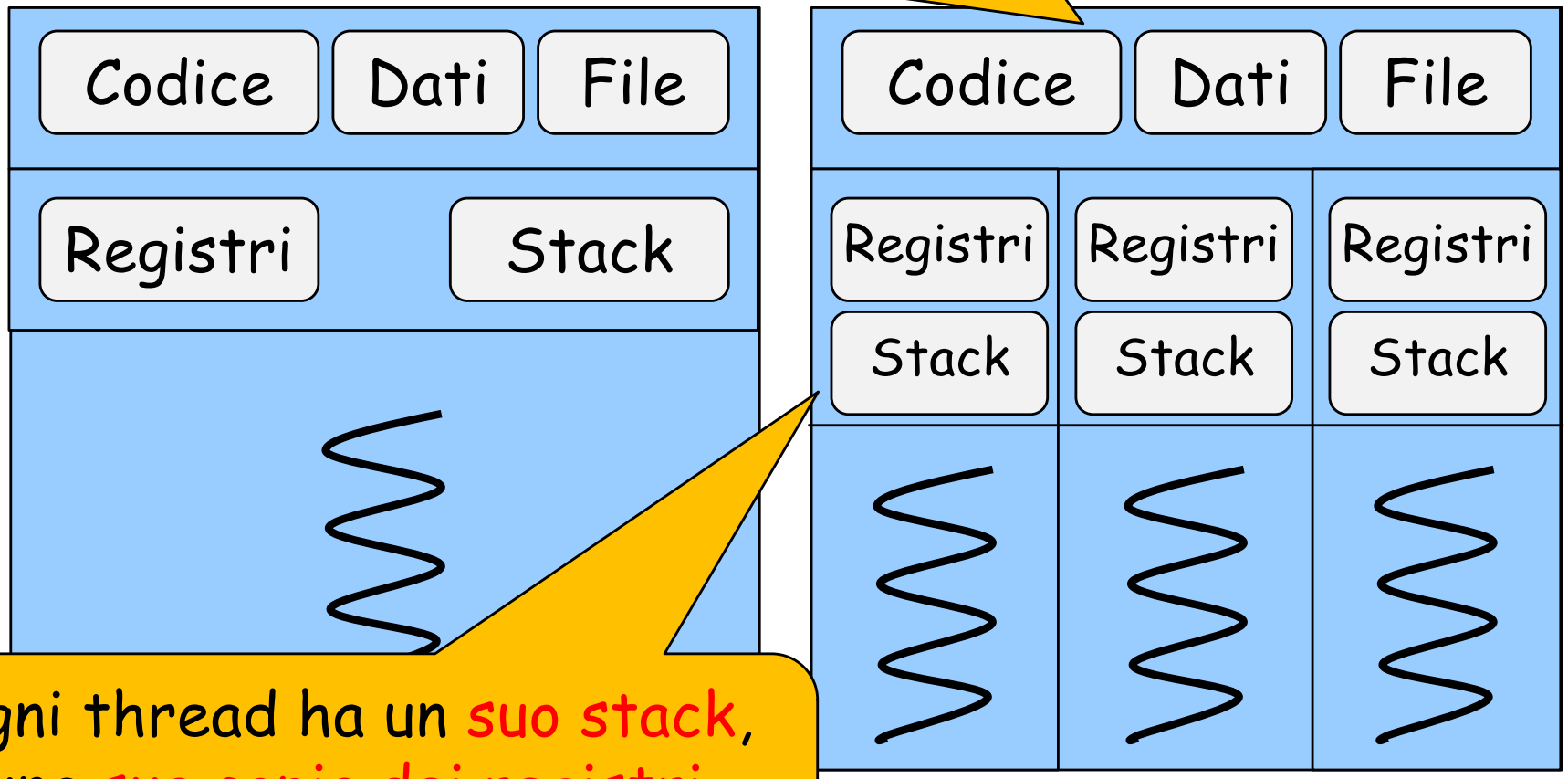
Processo
con singolo thread



Processo
multi-thread

Processi e Thre

Le altre risorse non sono esclusive di uno specifico thread, ma sono condivise



Ogni thread ha un suo stack, e una sua copia dei registri (es. program counter)

Processo multi-thread

Utilità dei thread

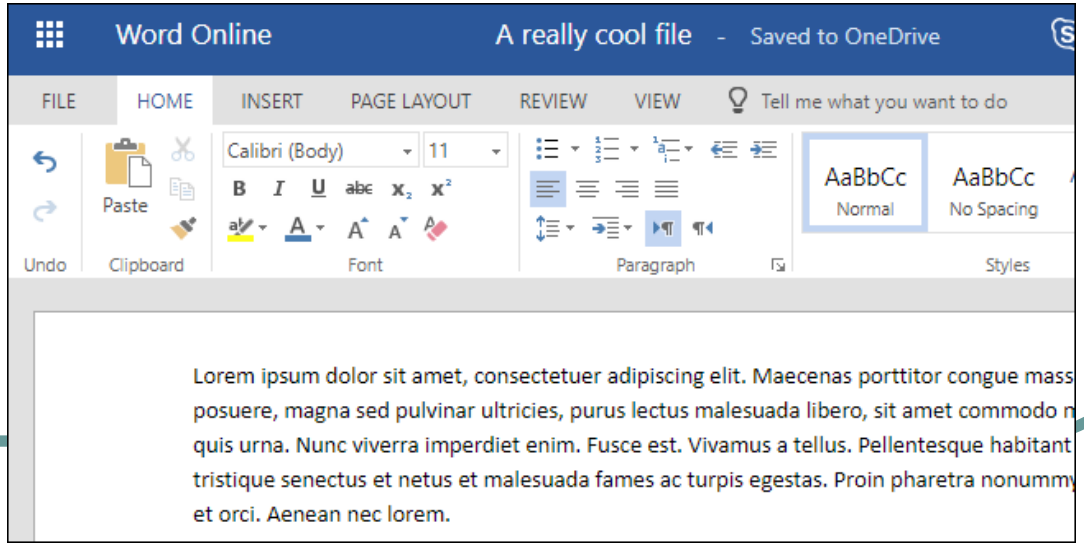
- I threads possono aumentare la **velocità di esecuzione**
  - Si **suddivide il lavoro in più parti**, ciascuna assegnata a un thread diverso
  - Ogni thread può essere eseguito su **CPU distinte**
- Esempi:
  - High-performance internet servers
  - Elaborazione di grossi vettori e matrici
  - ...

Utilità dei thread

- I thread possono conferire **maggiore modularità** ai programmi
- Esempio: separare le attività di
  - **foreground** (es., risposta immediata agli input degli utenti)
  - **background** (es., elaborazione di operazioni complesse)
- Esempio: implementare **funzioni asincrone** al normale flusso d'esecuzione del programma
  - gestione di eventi sporadici o periodici (es. backup)

Utilità dei thread

- Esempio: **elaborazioni di testi**
- Più attività concorrenti, che operano su **dati e risorse comuni** (in questo caso, il documento di testo)
  - Scrittura sul video
  - Lettura dei dati immessi
  - La correzione ortografica e grammaticale
  - Il salvataggio periodico su disco

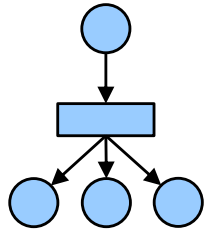


Threads vs. processi

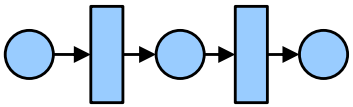
- In linea di principio, vantaggi simili possono essere ottenuti con applicazioni multi-processo
- I programmi multi-thread migliorano ulteriormente:
  - La **creazione/terminazione** di un thread è molto più efficiente della creazione di un processo
  - La **comunicazione** tra threads è molto più semplice ed efficiente di quelle tra processi, poiché non coinvolge il kernel
  - Il **context switching** tra threads ha un minor overhead di quello tra processi

Tipici modelli di programmazione multithread

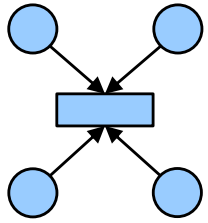
- **Manager/Workers**: un thread, il *manager*, riceve in input i comandi e assegna i lavori ad altri thread, i *workers*.



- **Pipeline**: un task è suddiviso in una serie di operazioni più semplici, che possono essere eseguite in serie, e concorrentemente, da diversi thread.



- **Peer**: simile al modello Manager/Workers, ma una volta che il thread principale assegna il lavoro agli altri thread, partecipa attivamente anch'esso nel lavoro.



Multithreading

- Multi-threading:
  La capacità di un sistema operativo di consentire l'esecuzione di più thread all'interno di un singolo processo

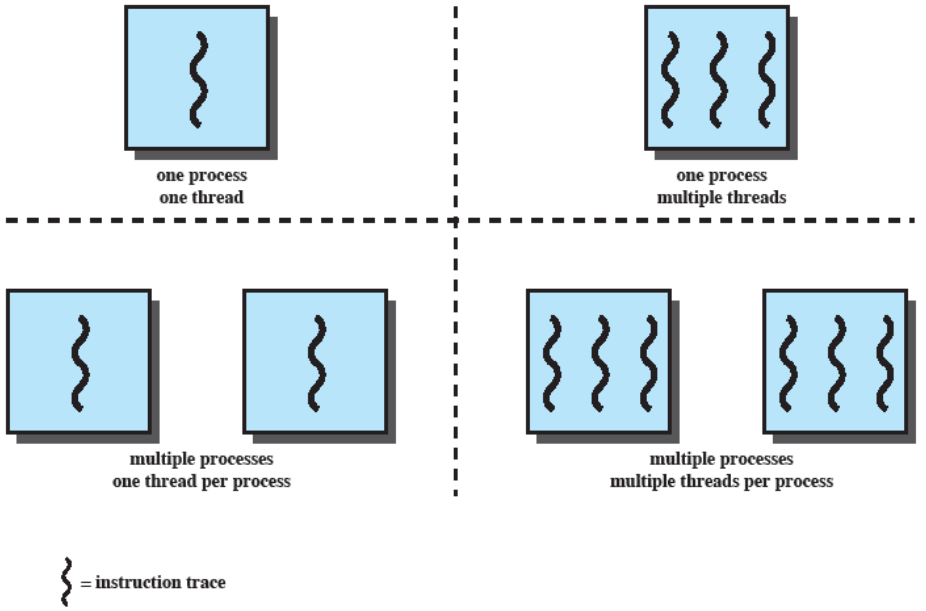


Figure 4.1 Threads and Processes [ANDE97]

Approcci Single Thread

- I vecchi sistemi mono-programmati (es. MS-DOS) supportano un unico processo (quindi un unico thread)
- I SO UNIX (multiprogrammati) consentono l'esecuzione di più processi, ognuno con un singolo thread

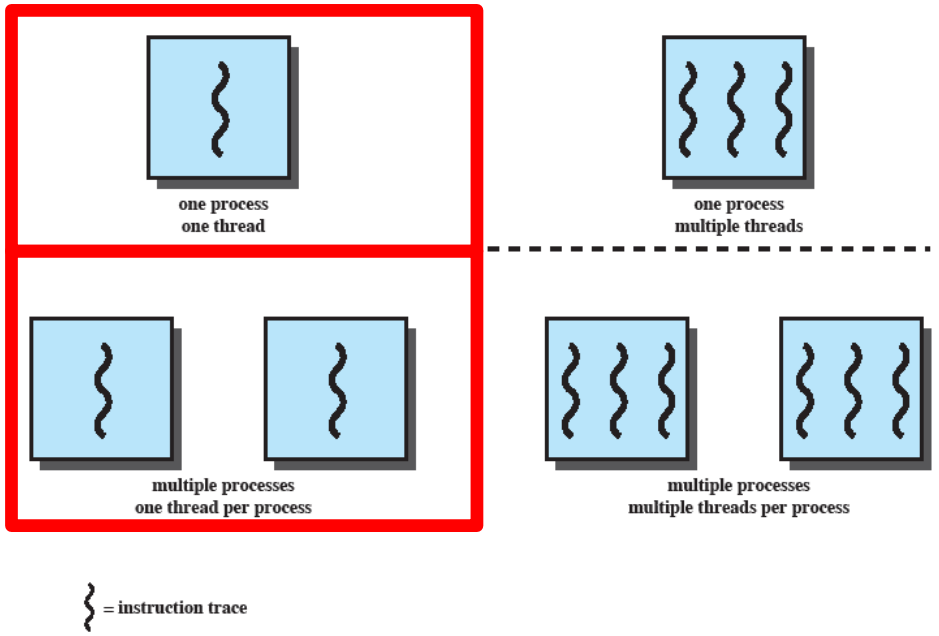


Figure 4.1 Threads and Processes [ANDE97]

Multithreading

- La macchina virtuale Java è un singolo processo con thread multipli
- Processi e thread multipli si trovano nei SO moderni quali Windows, Mac OS, e Linux

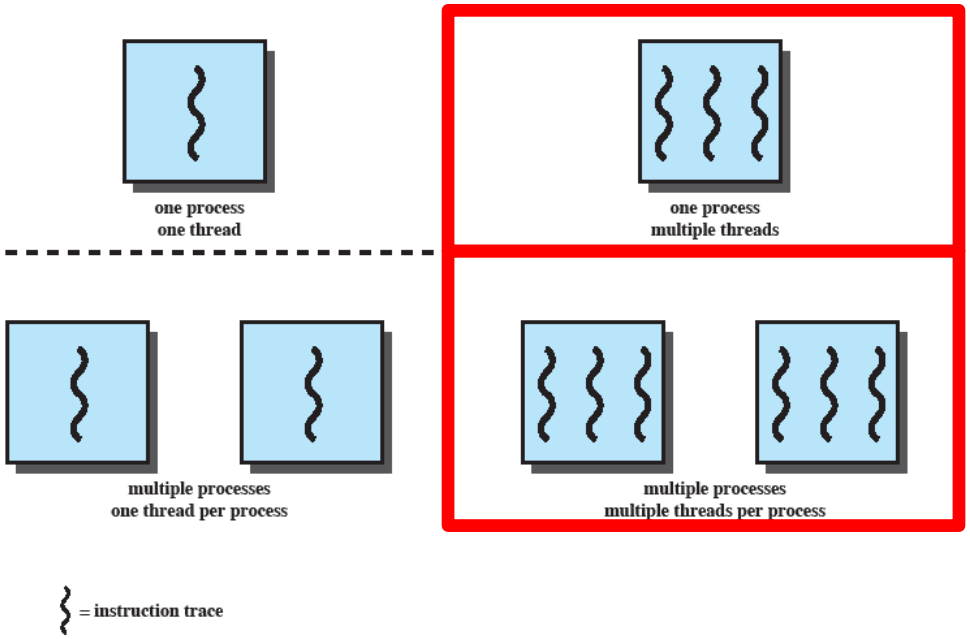
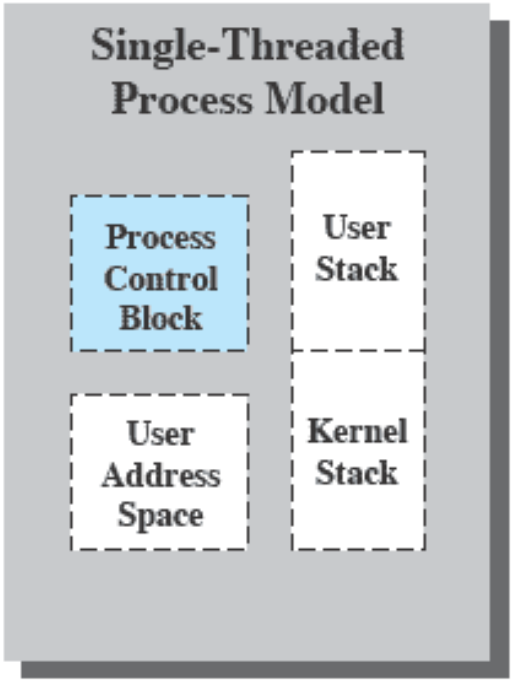
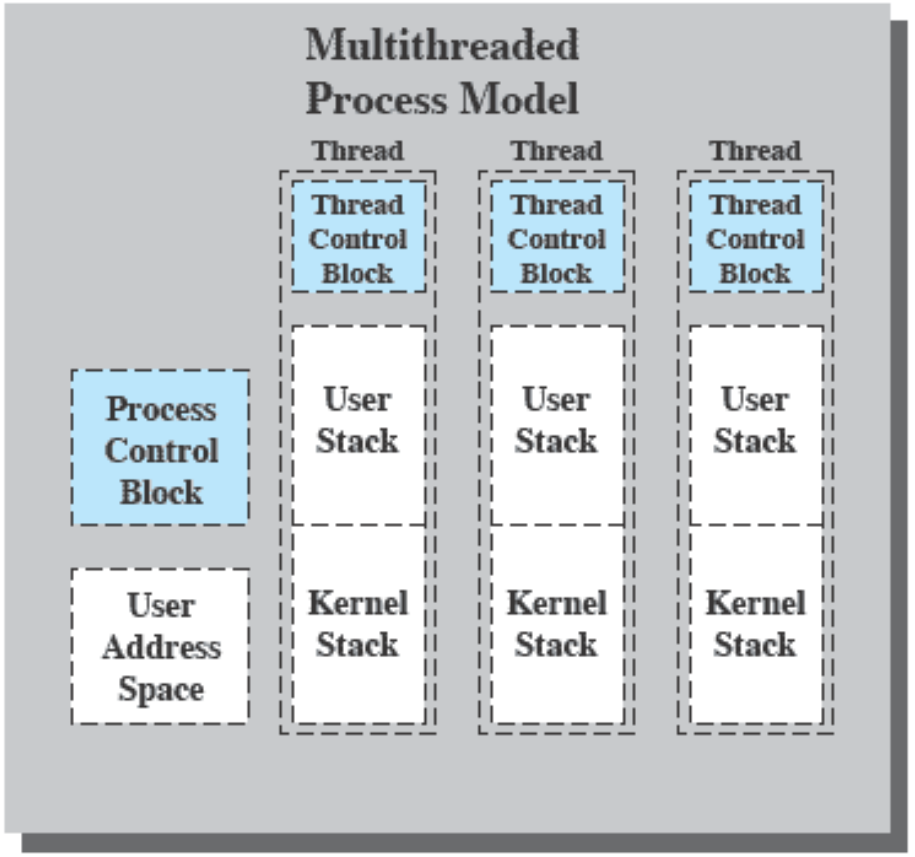


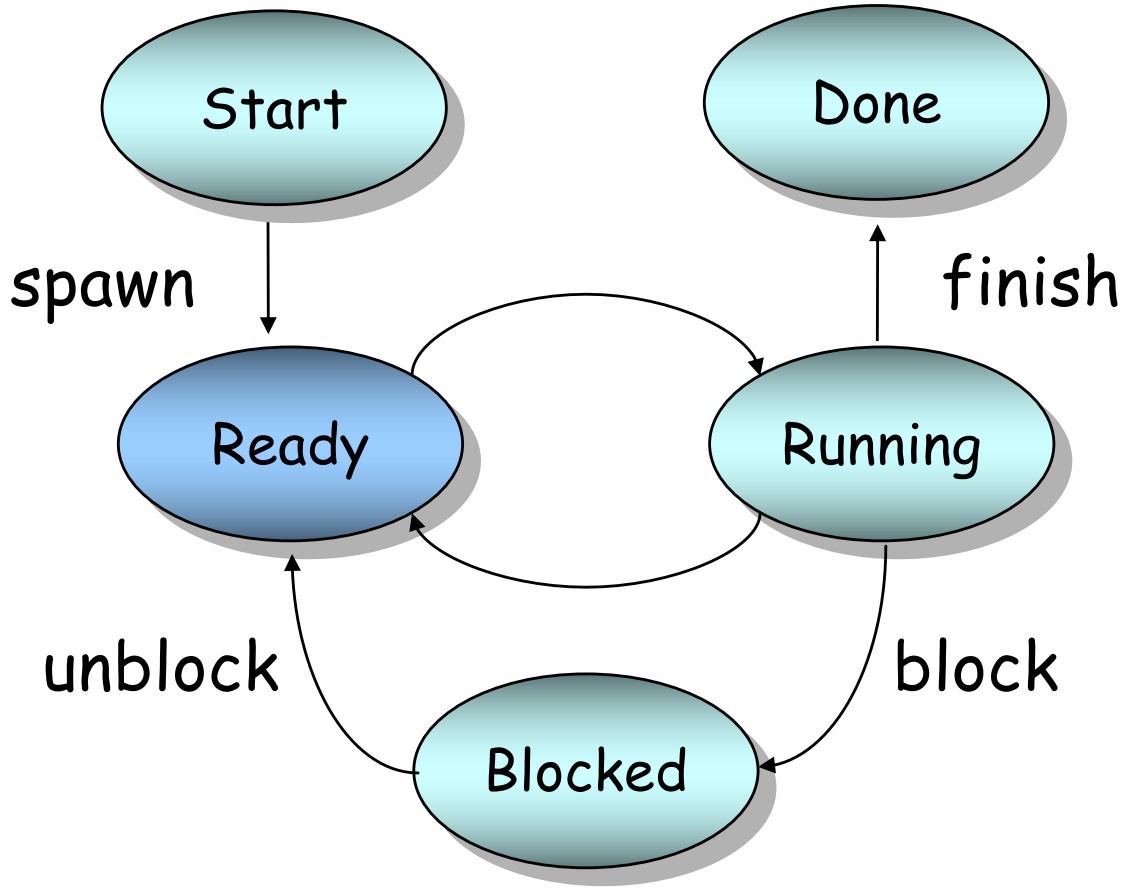
Figure 4.1 Threads and Processes [ANDE97]

Threads vs. processes





Thread: Stati



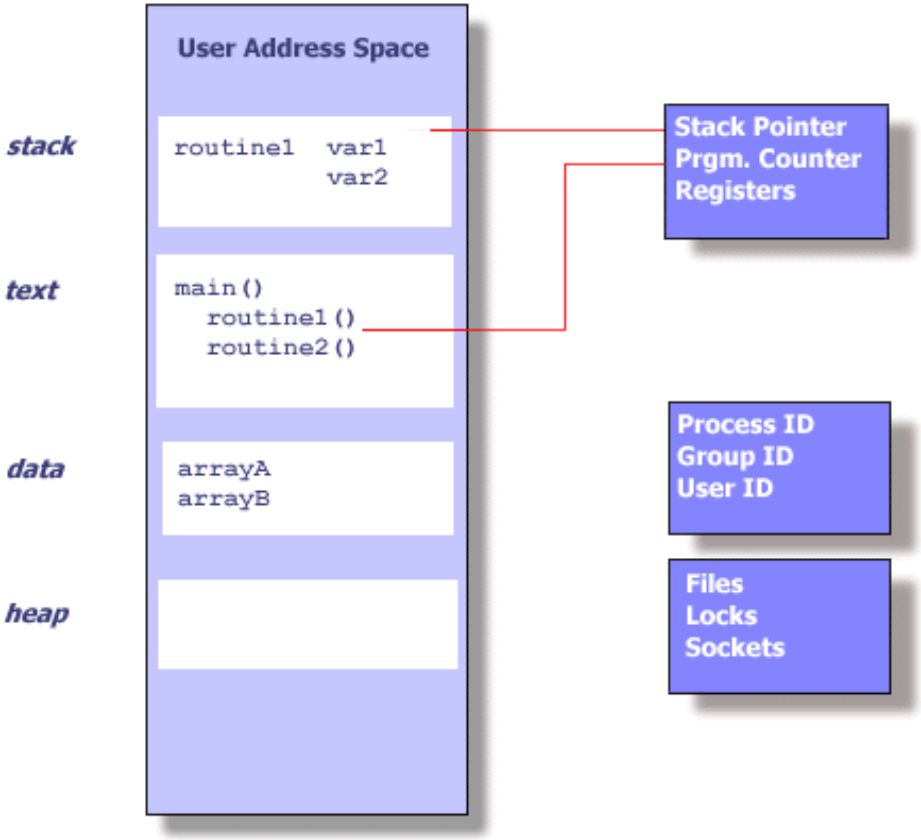
Thread e condivisione

- I thread che eseguono in uno stesso processo **condividono le risorse del processo**
- Esempi: tutti i thread hanno visibilità delle
  - modifiche alla memoria (es. variabili globali e dinamiche)
  - modifiche alle risorse condivise (es. un thread chiude un file)

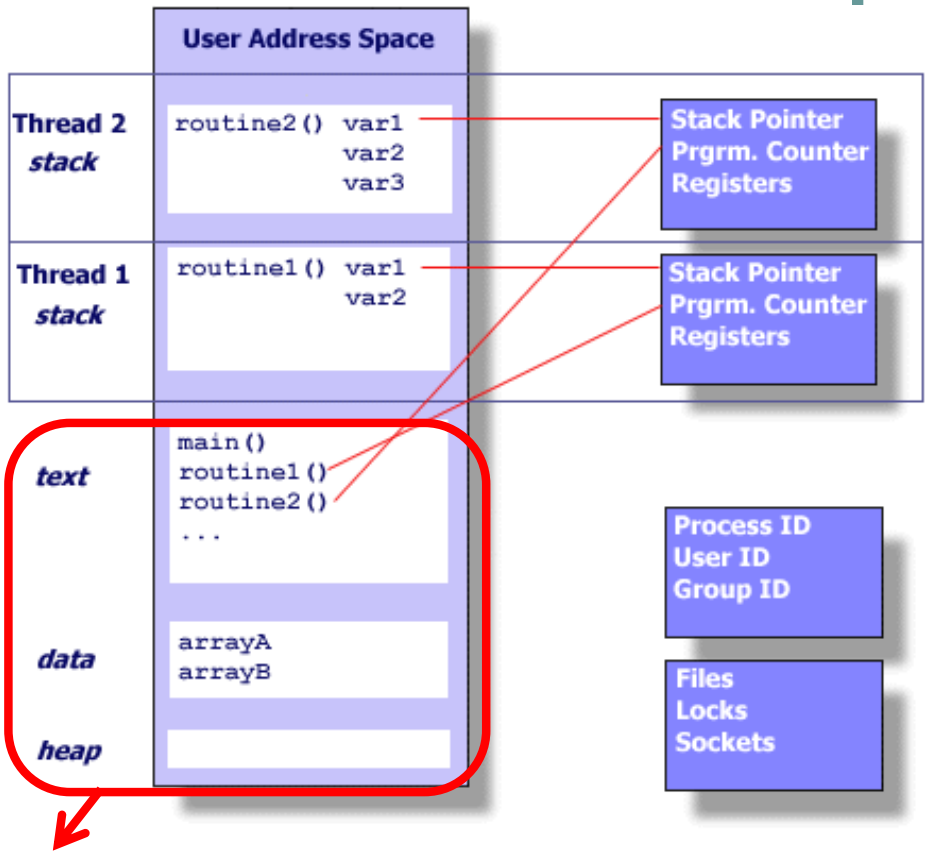
…per cui è richiesta una **sincronizzazione esplicita** tra i thread (attraverso semafori o monitor).

Thread e condivisione

Processo single-threaded



Processo multi-threaded



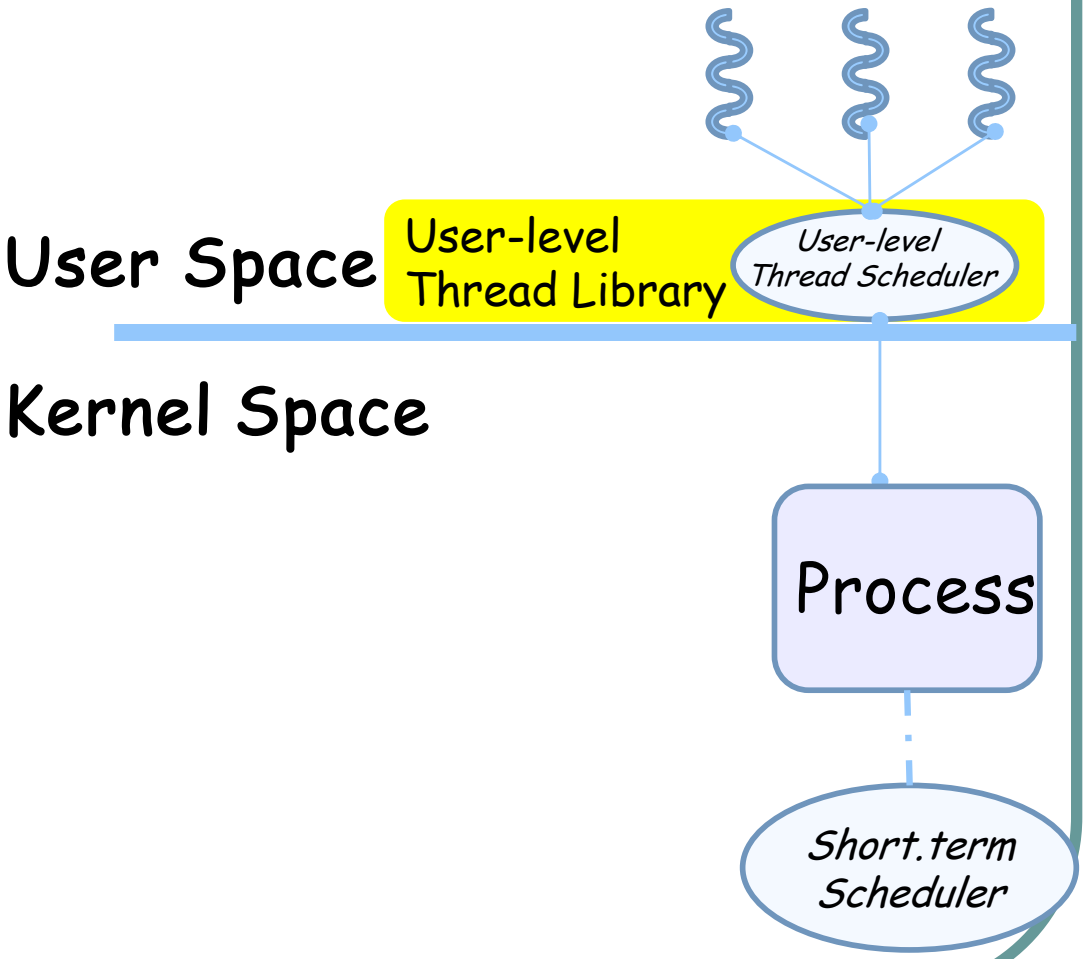
Le aree codice, heap, e dati globali sono condivisi tra i thread

Tipologie di impl. di thread

- **User-Level Thread** (ULT), anche chiamati:
  - green threads
  - co-routines
- **Kernel-Level Thread** (KLT), anche chiamati:
  - kernel-supported threads
  - lightweight processes

User-Level Threads

- La gestione dei thread è eseguita a livello applicativo
- Il kernel **non ha visibilità** dell'esistenza dei thread



User-Level Threads

- Con user-level thread, il programma ha un **unico thread a livello di SO**
- Il programma si avvale di una **thread library** per gestire il suo contesto di esecuzione
  - simula il context switch del SO

```
codice_thread () {
        ...scrive messaggio...
        yield_CPU()
        ...preleva risposta...
}
```

- Salva il contesto di esecuzione corrente (stack, registri)
- Attiva l'esecuzione di un altro thread (salta in un altro punto del programma)

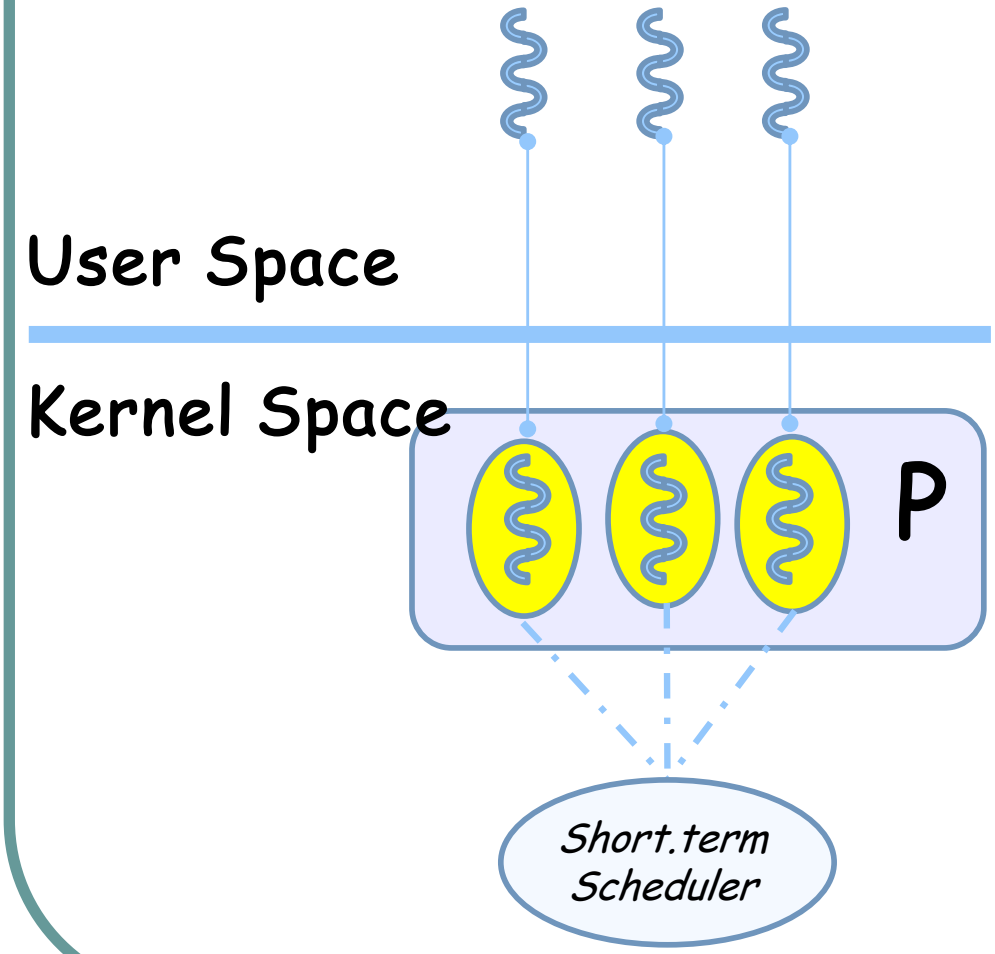
Questo approccio prende il nome di **scheduling cooperativo**: ci si avvale dell'aiuto del programma per preemption e context switch

Thread al livello utente: vantaggi e svantaggi

- **Minore overhead** per il context switch (che non richiede il passaggio in kernel mode)
- Lo **scheduler dei thread è indipendente** da quello dei processi: ogni applicazione può utilizzare uno scheduler personalizzato
- Le applicazioni sono **portabili**

- Quando un thread invoca una system call (bloccante), tutti i **thread di quel processo si bloccano**
- In generale, nelle **architetture multiprocessore**, non c'è vantaggio nell'utilizzo del multithreading: il kernel assegna un processo per ogni processore

Kernel-Level Threads



- Il Kernel gestisce le informazioni di contesto sia per il processo sia per i threads
- Lo scheduling è eseguito sui thread
- I moderni SO adottano questo approccio

Vantaggi dei thread a livello Kernel

- Il kernel è in grado di schedulare più thread dello stesso processo su più processori
- Se un thread di un processo è "blocked", il kernel potrà schedulare un altro thread dello stesso processo

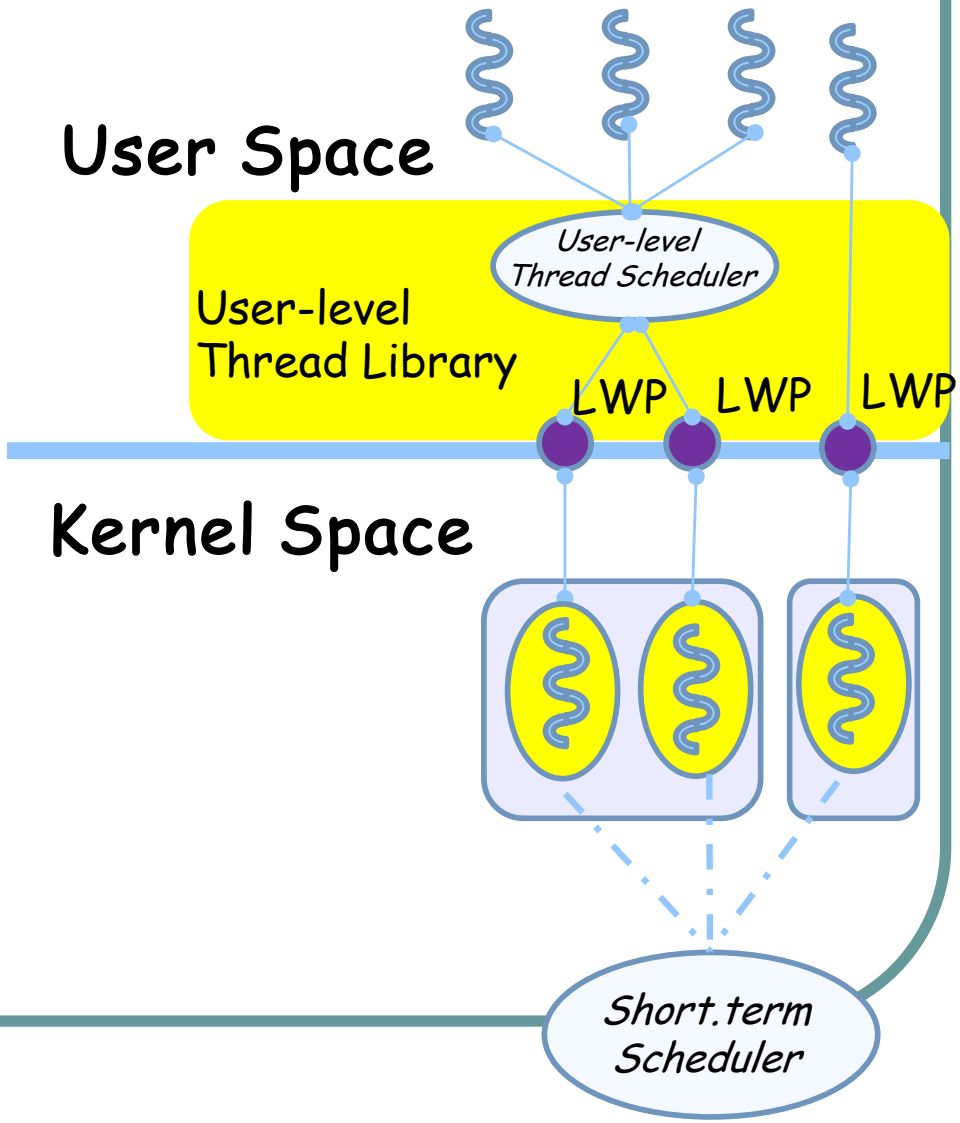
NOTA: anche il kernel stesso può essere sviluppato con un approccio multithreaded!

Svantaggi dei thread a livello Kernel

- Il trasferimento del controllo da un thread ad un altro, pure se nell'ambito di uno stesso processo, richiede un context switch a livello kernel

Approcci combinati

- Più thread al livello d'utente corrispondono a ad un numero inferiore o uguale di thread al livello del nucleo
- La creazione del thread avviene nello spazio utente
- Il grosso dello scheduling e sincronizzazione è fatta nello spazio utente



Thread nel linguaggio Go

- Il linguaggio di programmazione **Go** utilizza l'approccio misto
- User-level thread, gestiti dalla libreria del linguaggio
- Sfrutta i threads forniti dal SO (kernel-level)
- Utilizzato per server di rete ad alte prestazioni

Thread nel sistema Linux

- In Linux, il **task** (un flusso di esecuzione) è l'unità fondamentale di scheduling
  - Un thread è **un task che condivide delle strutture con altri task** (codice, heap, etc.)
- Ogni task ha un **Process ID (PID)** univoco
- Un "processo multithreaded" (gruppo di task) è identificato da un **Thread Group ID (TGID)**

Thread nel sistema Linux

```
$ ps aux
USER    PID %CPU %MEM VSZ      RSS     ...  COMMAND
...
so     4050 35.9  6.9 3167244 280116 ... /usr/lib/firefox/firefox -new-window
...

$ ps -T -p 4050 |head
   PID    SPID TTY          TIME CMD
  4050    4050 ?        00:00:04 firefox
  4050    4058 ?        00:00:00 gmain
  4050    4059 ?        00:00:00 gdbus
  4050    4068 ?        00:00:00 IPC I/O Parent
  4050    4069 ?        00:00:00 Timer
  4050    4070 ?        00:00:00 Netlink Monitor
  4050    4071 ?        00:00:00 Socket Thread
  4050    4072 ?        00:00:00 Permission
  4050    4076 ?        00:00:00 JS Watchdog
   ...     ...
```

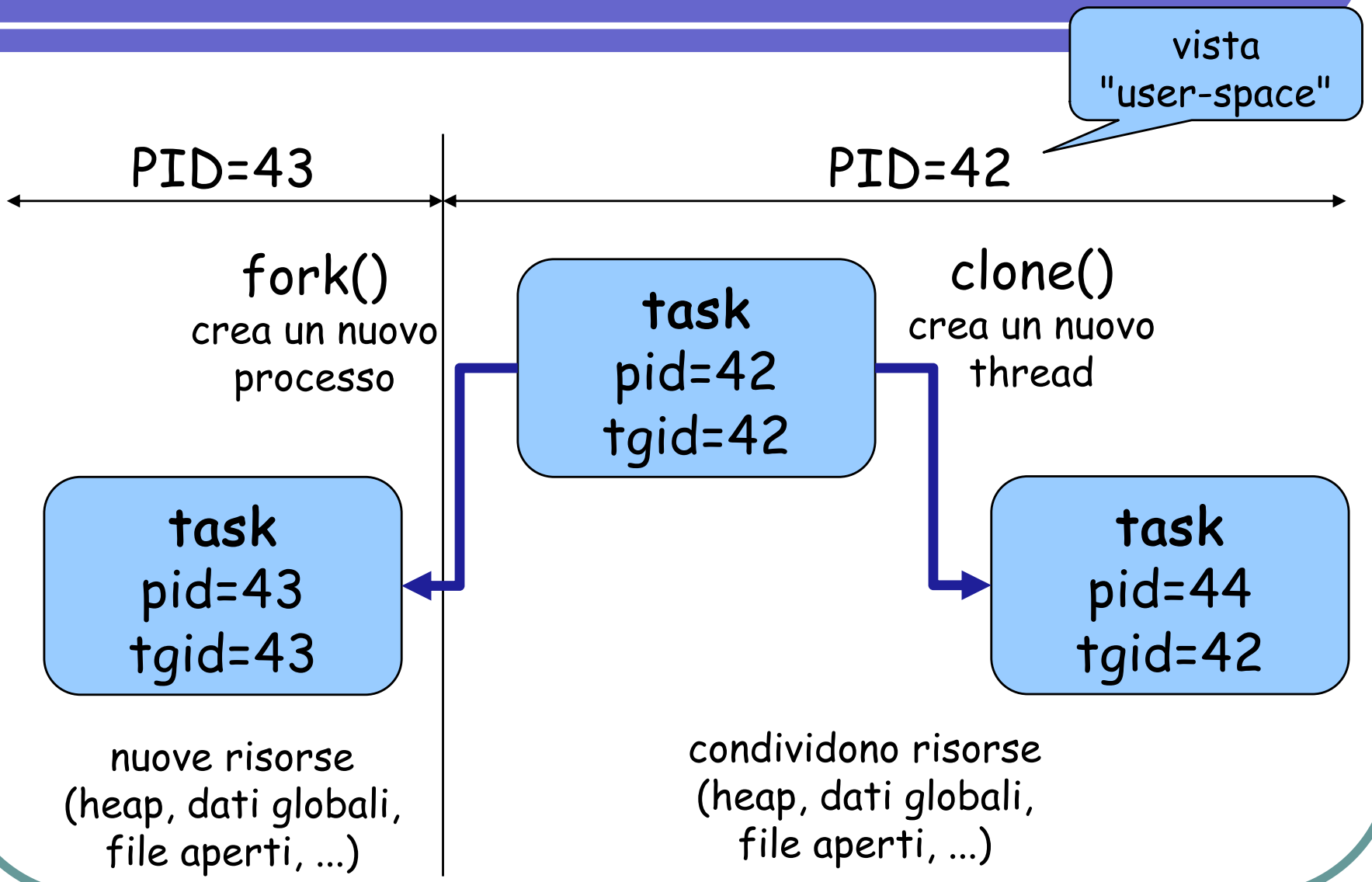
il TGID (era PID nello UNIX classico)

il "vero" PID (identificativo del task)

Thread nel sistema Linux

- La creazione di un thread avviene attraverso la chiamata del sistema **clone()**
  - **clone()** anziché creare una copia del task chiamante, crea un nuovo task che condivide lo spazio d'indirizzamento del task chiamante
- La **chiamata di sistema fork()** dello standard POSIX è implementata attraverso la clone()

Thread nel sistema Linux



"If threads share the same PID, how can they be identified?"
https://stackoverflow.com/questions/9305992/if-threads-share-the-same-pid-how-can-they-be-identified

Thread nel sistema Win 2000

- Mapping uno ad uno tra ULT e KLT
- Ciascun thread contiene:
  - un identificatore di thread (ID)
  - un insieme di registri
  - una pila d'utente e una pila del nucleo
  - un'area di memoria privata

Thread nel linguaggio Java

- I thread nel linguaggio Java possono essere creati
  - creando una nuova classe derivata dalla classe **Thread**
  - ridefinendo il metodo **run** di quella classe
- I thread nel linguaggio Java sono gestiti dalla macchina virtuale (JVM)

Quiz

1. Quali di queste affermazioni riguardo i thread sono vere? (selezionare più di una)

- [ ] I kernel-level thread permettono di sfruttare i sistemi con più processori
- [ ] Gli user-level thread possono effettuare chiamate di sistema di I/O bloccanti, senza bloccare gli altri thread
- [ ] I thread condividono l'area stack
- [ ] I thread condividono l'area heap
- [ ] I thread condividono l'area codice

https://forms.office.com/r/u1s9aLTkV5