



Corso di Laurea Triennale in Informatica  
Università Degli Studi della Basilicata

# Sistemi Operativi

Docente:  
[ugo.erra@unibas.it](mailto:ugo.erra@unibas.it)

4° Lezione

Thread

# Sommario della lezione



- Introduzione ai thread
- Vantaggi della programmazione multithread
- Modelli di programmazione multithread
- Considerazioni sulla gestione dei thread

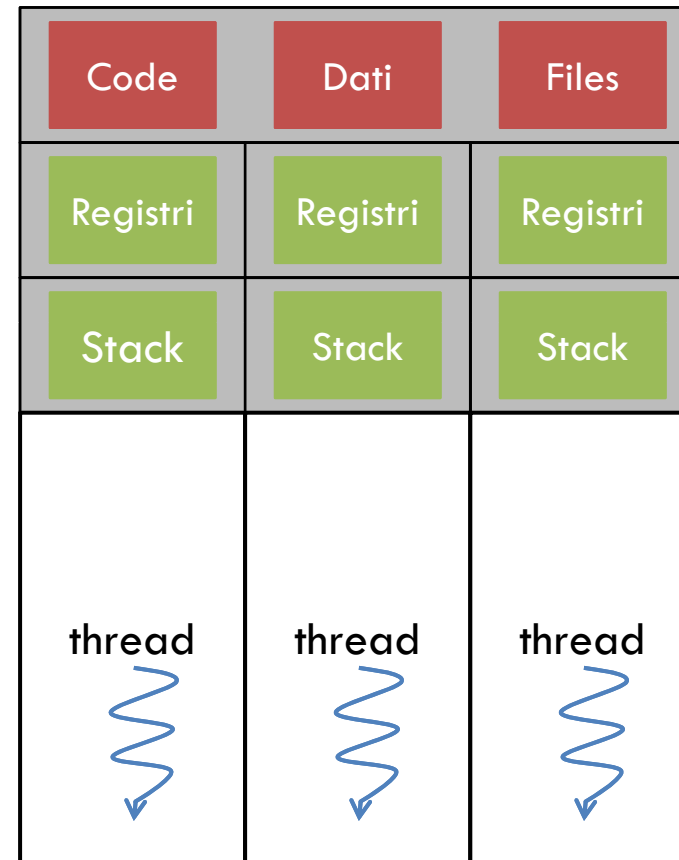
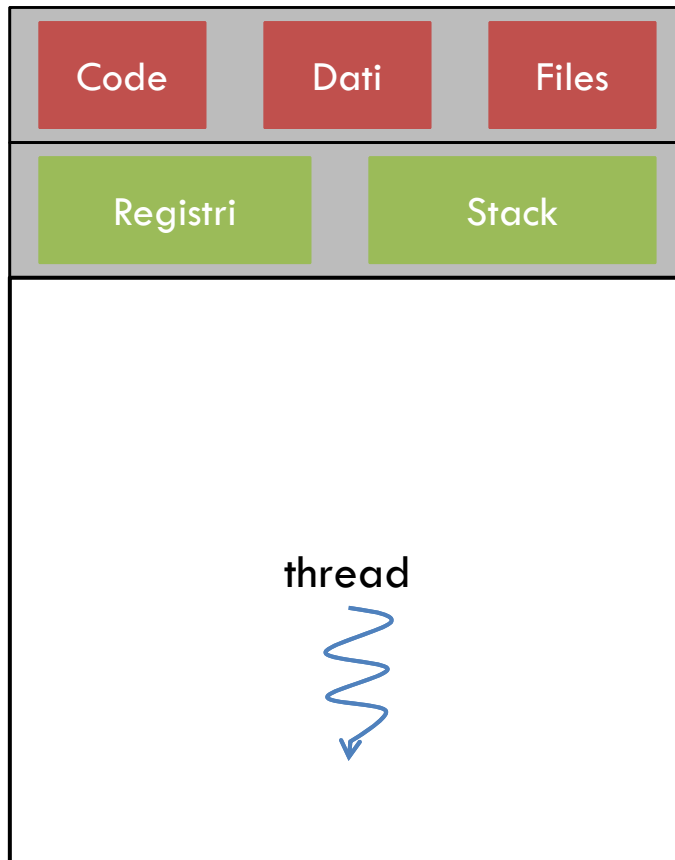
# Introduzione ai thread - 1

- In un Sistema Operativo multi-programmato il contex switch da un processo ad un altro è una operazione costosa
- Se in un Sistema Operativo girano molti processi l'overhead dovuto al contex switch può influire notevolmente
- L'idea della programmazione multithread è di sostituire tanti processi normali con un solo processo in grado di lanciare diversi flussi di esecuzione (thread)

# Introduzione ai thread - 2

- Un thread è simile ad un processo ma richiede meno risorse da parte del Sistema Operativo
- Per ogni thread abbiamo:
  - ▣ Thread ID (identificativo)
  - ▣ Contatore di programma
  - ▣ Registri
  - ▣ Stack
- Un insieme di thread lanciati dallo stesso processo condividono con il processo che li ha generati:
  - ▣ La stessa area di dati
  - ▣ Le stesse risorse
- Essendo il thread più leggero la fase di context switch sarà più rapida

# Processo a singolo thread e multithread



# Vantaggi

- Tempo di risposta
  - ▣ Un'applicazione multithread permette all'utente di interagire sempre con essa anche se una parte è bloccata (ad esempio in attesa di input)
- Condivisione delle risorse
  - ▣ I thread condividono la memoria e le risorse del processo di appartenenza. Poiché una applicazione può avere più thread il risparmio è notevole.
- Economia
  - ▣ Normalmente creare un processo è costoso, poiché un thread condivide parte delle risorse di un processo la sua creazione è più veloce
- Uso di sistemi multiprocessori
  - ▣ Nei sistemi multiprocessori i thread possono essere eseguiti in parallelo aumentando il grado di parallelismo

# Svantaggi



- La programmazione in ambienti multithread è più difficile rispetto ai singoli processi
  - ▣ Poiché i thread di un processo condividono le stesse risorse è necessario un accesso con mutua esclusione alle risorse condivise
  - ▣ Debugging dei programmi molto più difficile

# Uso dei thread

- I thread possono essere adoperati per eseguire funzioni diverse all'interno della stessa applicazione, ad esempio:
  - ▣ Web browser
    - Un thread si occupa di visualizzare le immagini ed il testo
    - Un thread si occupa di ricevere i dati da visualizzare
  - ▣ Word processing
    - Un thread mostra testo/immagini a video
    - Un thread si occupa dell'interfaccia utente
    - Un thread controlla gli errori ortografici
- I thread possono anche essere adoperati per eseguire le stesse funzioni all'interno di una applicazione, ad esempio:
  - ▣ Web server
    - Thread che accettano le richieste e creano altri thread per servirle
    - Thread che servono le richieste di connessione

# Modelli di programmazione multithread



- I thread possono essere definiti come:
  - ▣ **Thread a livello utente**
  - ▣ **Thread a livello kernel**

# Thread a livello utente

- Nell'implementazione dei thread a livello utente il sistema operativo ignora la presenza dei thread
- I thread sono gestiti da un processo a run-time in modalità utente
  - ▣ La creazione dei thread e lo scheduling avvengono a livello applicativo senza entrare nella modalità kernel
- Il sistema a run-time nel momento in cui sarà eseguito si occuperà di assegnare un frazione del tempo che gli è stato concesso ad uno dei suoi thread
- Lo svantaggio è che se un thread effettua una system call bloccate blocca il processo e quindi tutti i thread si bloccano

# Thread a livello kernel

- ❑ Le funzioni di creazione, scheduling e gestione sono a carico del Sistema Operativo e quindi implementate direttamente nel kernel
- ❑ A ciascuna funzione corrisponde una system call
- ❑ Quando un thread si blocca il Sistema Operativo può mettere in esecuzione un altro thread dello stesso processo
- ❑ Soluzione meno efficiente della precedente
- ❑ Possibilità di eseguire thread diversi appartenenti allo stesso processo su unità di elaborazione differenti

# Esempi di thread



- Thread a livello utente
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Thread a livello kernel
  - Windows XP
  - Linux
  - Mac OS X
  - Solaris
  - True64 UNIX

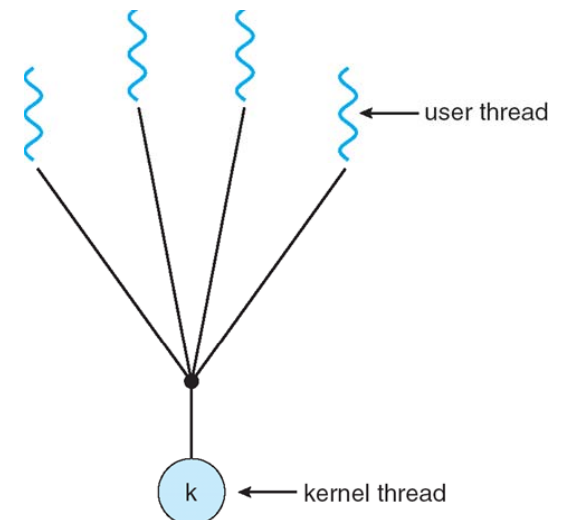
# Relazioni tra thread



- Le relazioni che esistono tra thread a livello utente e thread a livello kernel possono essere
  - ▣ Modello da molti a uno
  - ▣ Modello da uno a uno
  - ▣ Modello da molti a molti

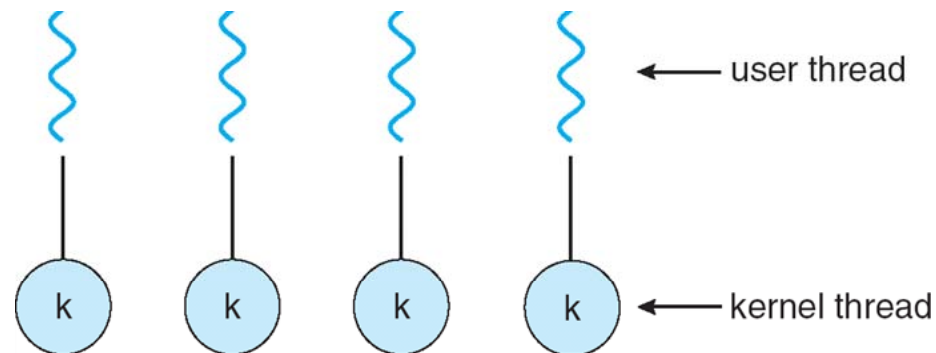
# Modello da molti ad uno

- Diversi thread a livello utente corrispondono ad un singolo thread a livello kernel
  - ▣ Impiegato quando il kernel non gestisce i thread
  - ▣ La gestione dei thread è efficiente poiché avviene a livello utente
  - ▣ Se un thread effettua una system call bloccante l'intero insieme dei thread si blocca
  - ▣ Un solo thread alla volta può accedere al kernel ed è quindi impossibile eseguire più thread in parallelo
  - ▣ Un esempio di questo modello è la libreria green threads per Solaris



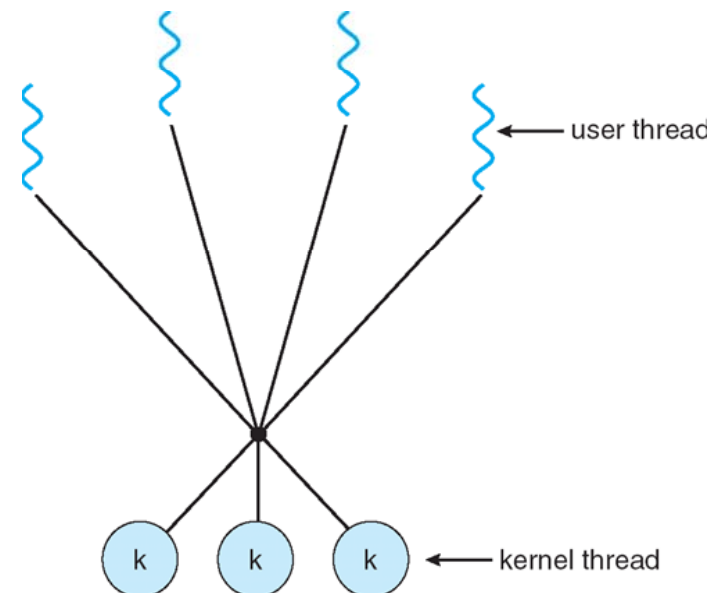
# Modello da uno a uno

- Ogni thread a livello utente corrisponde ad un unico thread a livello kernel
  - ▣ Maggiore livello di concorrenza poiché se un thread effettua una chiamata bloccante gli altri thread non si bloccano
  - ▣ Più thread possono accedere al kernel quindi è possibile eseguire più thread in parallelo
  - ▣ La gestione non è molto efficiente poiché per ogni thread utente è necessario creare il corrispondente thread kernel
  - ▣ Windows XP, 95, 98, 2000, NT e Linux utilizzando questo modello



# Modello da molti a molti

- Diversi thread a livello utente corrispondono ad un numero uguale o minore di thread del kernel
  - ▣ Offre il meglio dei modelli precedenti
    - Offre maggiore concorrenza rispetto al modello molti ad uno
    - Se il numero dei thread aumenta la gestione risulta sempre efficiente rispetto al modello uno ad uno
- Solaris2, IRIX, Linux (NPTL) utilizzando questo modello



# Considerazioni sulla gestione dei thread

- Chiamate a `fork()` ed `exec()`
- Cancellazione
- Gestione dei segnali
- Gruppi di thread
- Dati specifici dei thread

# Chiamate a `fork()` ed `exec()`

- La `fork()` è la system call di UNIX utilizzata per creare un nuovo processo
- Nel caso della creazione di un processo mediante la `fork()` il sistema operativo dovrà clonare l'intero processo compresi tutti i suoi thread?
  - ▣ Alcuni sistemi UNIX includono anche questa possibilità
- Se la system call `exec()` segue immediatamente la creazione di un processo mediante la `fork()` non è necessario duplicare anche tutti i thread
  - ▣ La `exec()` sostituisce il processo appena creato con una immagine del programma specificato nei parametri

# Cancellazione

- Un thread da cancellare è chiamato **thread bersaglio** (*target thread*)
- Un thread è normalmente cancellato quando il suo compito non è più necessario
  - Un thread che carica una pagina web può essere interrotto dall'utente e quindi può essere cancellato
- La cancellazione può avvenire in due modi:
  - **Cancellazione asincrona:** il thread viene terminato immediatamente
  - **Cancellazione differita:** il thread periodicamente controlla se deve terminare in modo da farlo opportunamente

# Gestione dei segnali - 1

- I **segnali** sono utilizzati in UNIX per comunicare ai processi particolari eventi
- I segnali possono essere inviati in modo
  - ▣ Sincrono: generati dallo stesso processo in esecuzione
    - Ad esempio: Accesso illegale alla memoria, divisione per zero
  - ▣ Asincrono: generati da altri processi ed inviati ad altri processi
    - Ad esempio: Combinazione di tasti `control+c`, scadenza di un timer
- La gestione dei segnali prevede tre fasi:
  1. Un segnale viene generato in funzione di un particolare evento
  2. Il segnale viene inviato al processo
  3. Il processo riceve il segnale e lo gestisce

# Gestione dei segnali - 2

- I segnali possono essere gestiti in due diversi modi
  - ▣ Un **gestore predefinito di segnali**
  - ▣ Un **gestore di segnali definito dall'utente**
- Nei processi con singolo thread è semplice la gestione dei segnali.
  - ▣ Ad esempio il `control+c` viene inviato al solo processo
- Nei processi multithread esiste più di una possibilità
  - ▣ Inviare il segnale al thread cui si riferisce il segnale
  - ▣ Inviare il segnale a ogni thread del processo
  - ▣ Inviare il segnale a specifici thread del processo
  - ▣ Definire un thread specifico per ricevere tutti i segnali

# Gruppi di thread

- Alcuni processi richiedono la creazione di thread distinti durante l'esecuzione
  - ▣ Ad esempio un Web Server richiede la creazione di un thread ad ogni richiesta di connessione
- Gli svantaggi di questo approccio sono:
  - ▣ Creare un thread ha un costo in termini di tempo
  - ▣ Al crescere del numero dei thread le risorse potrebbero esaurirsi
- L'impiego di **gruppi di thread** creati in anticipo ed in attesa di essere utilizzati all'occorrenza offre due vantaggi:
  - ▣ Un thread esistente permette di rispondere più velocemente ad una richiesta rispetto alla creazione di nuovo thread
  - ▣ Decidere a priori il numero di thread per una applicazione permette di pianificare meglio le risorse da utilizzare all'interno del sistema

# Dati specifici dei thread



- Il vantaggio della programmazione multithread è che gruppi di thread condividono i dati
- In particolari circostanze, ciascun thread può necessitare di una copia privata di alcuni dati, detti **dati specifici di thread**
- La maggior parte delle librerie forniscono il supporto per i dati specifici

# Esempio: Thread di Linux

- Linux prevede la system call `fork()` per duplicare un processo e prevede la chiamate `clone()` per generare nuovi thread
- In effetti non esiste una distinzione tra processi e thread ed al loro posto viene adoperato il termine task
- La system call `clone()` permette di specificare le risorse da condividere con il processo padre
  - ▣ `CLONE_FS` (Condivisione file system)
  - ▣ `CLONE_VM` (Condivisione spazio di memoria)
  - ▣ `CLONE_SIGHAND` (Condivisione gestore dei segnali)
  - ▣ `CLONE_FILES` (Condivisione file aperti)
- La chiamate `clone()` equivale a creare un thread in quanto di volta specifichiamo quale risorse condividere con il processo padre