Thread e concorrenza

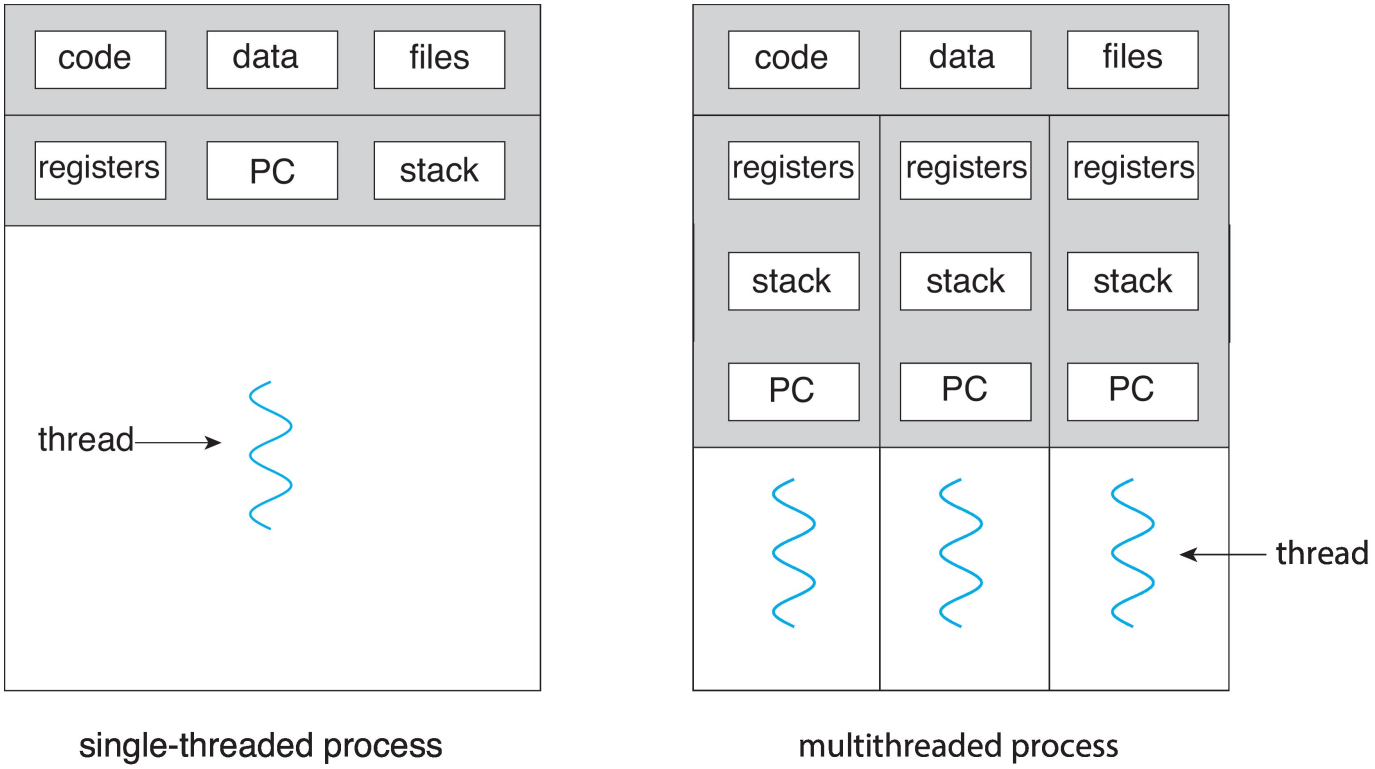
Pietro Braione

Reti e Sistemi Operativi – Anno accademico 2021-2022

Obiettivi

- Capire le differenze tra thread e processi
- Capire benefici e problematiche delle applicazioni multithreaded
- Descrivere come i sistemi operativi Windows e Linux rappresentano i thread

Processi single- e multithreaded



single-threaded process

multithreaded process

Motivazioni

- Parallelizzazione operazioni all’interno di un’applicazione
- Gestione di molti compiti simili tra loro
- Sfruttamento core multipli
- Può semplificare il codice ed aumentare l’efficienza
- I kernels in genere sono multithreaded

Vantaggi

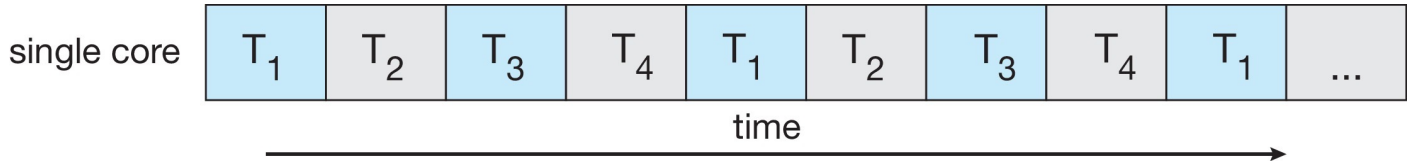
- Migliori tempi di risposta: l'applicazione può eseguire un'operazione lunga in un thread a parte e continuare a rispondere agli input utente
- Condivisione di codice e dati: i thread di un processo condividono lo spazio di memoria del processo
- Economia: creare un thread richiede meno risorse che creare un processo, ed il cambio di contesto è più rapido
- Scalabilità: il multithreading in una macchina con più processori aumenta il grado di parallelismo di un'applicazione

Concorrenza e parallelismo (1)

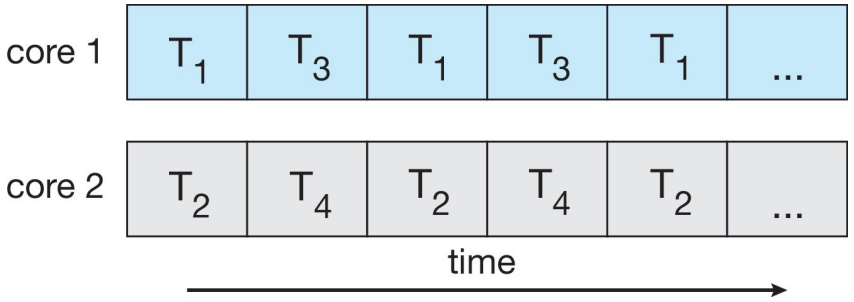
- **Concorrenza** = capacità di far progredire più di un'attività nel tempo
- **Parallelismo** = capacità di eseguire più di un'attività simultaneamente
- Single core + timesharing = concorrenza senza parallelismo
- Multicore = concorrenza attraverso il parallelismo

Concorrenza e parallelismo (2)

- Concorrenza senza parallelismo:



- Parallelismo:

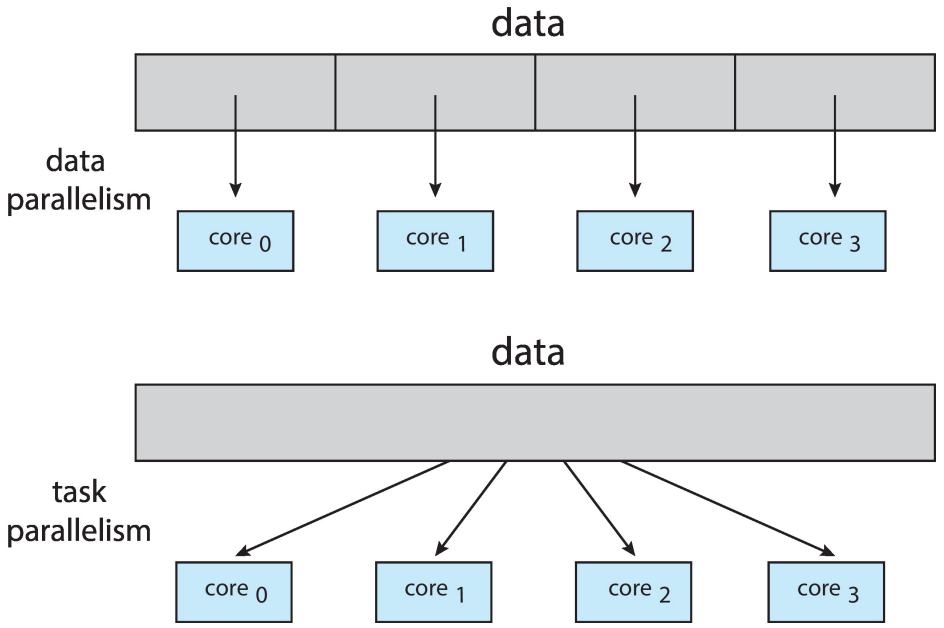


Le sfide della programmazione multicore

- Identificazione delle attività da eseguire su core separati
- Bilanciamento, in maniera che i compiti eseguiti dalle diverse attività abbiano dimensioni confrontabili
- Suddivisione dei dati per massimizzare l'accesso parallelo
- Dipendenze tra i dati prodotti/consumati dai task, per stabilire la rispettiva sincronizzazione
- Testing e debugging

Tipi di parallelismo

- **Parallelismo dei dati**: diversi core effettuano la stessa operazione, operando su diversi sottoinsiemi dei dati
- **Parallelismo delle attività**: diversi core effettuano diverse attività, operando su dati comuni
- (ibridi sono possibili)



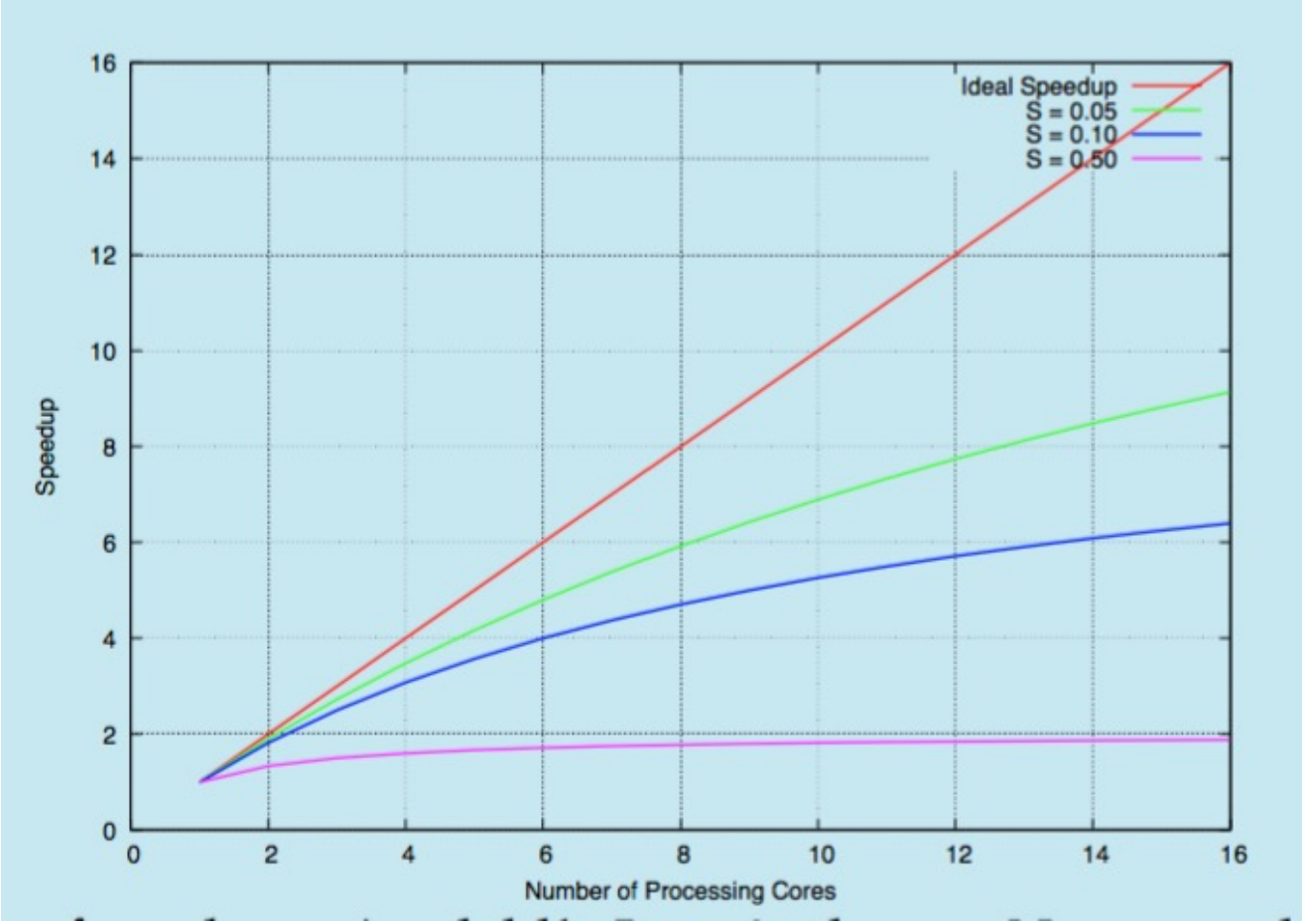
Legge di Amdahl

- Fornisce il guadagno in termini di performance derivante dall'aggiunta di core ad un'applicazione che ha componenti sia sequenziali che parallele:

$$incremento\ velocità \leq \frac{1}{S + \frac{1 - S}{N}}$$

- dove S è la porzione di applicazione che deve essere realizzata sequenzialmente, e N è il numero di core
- Esempio: se il 75% dell'applicazione è parallelizzabile (S = 25%):
  - N=2 core: fino a 1,6 volte più veloce di single-core
  - N=4 core: fino a 2,28 volte più veloce di single core

Legge di Amdahl

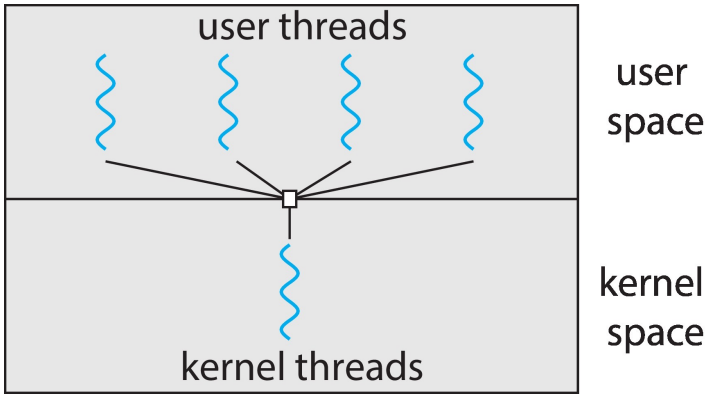


Modelli di supporto al threading

- **Thread a livello utente**: i thread disponibili nello spazio utente dei processi; di solito implementati da opportune librerie, il kernel non è consapevole della loro presenza
- **Thread a livello del kernel**: i thread implementati nativamente dal kernel; utilizzati per strutturare il kernel stesso in maniera concorrente
- I thread a livello utente vengono mappati sui thread a livello del kernel secondo qualche strategia:
  - **Molti-a-uno**
  - **Uno-a-uno**
  - **Molti-a-molti** e **a due livelli**

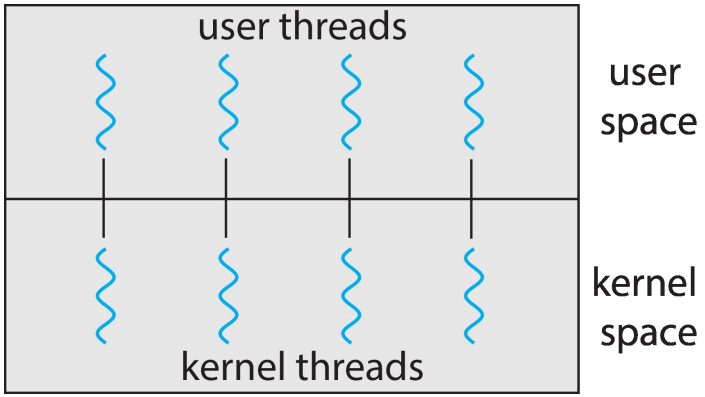
Molti-a-uno

- Molti thread utente mappati su un thread del kernel
- Unica soluzione se il sistema operativo non è multi-threaded
- Se un thread utente fa una chiamata di sistema bloccante causa il blocco degli altri thread utente dello stesso processo
- Non è in grado di sfruttare la presenza di più core
- Poco usato in pratica
- Esempi:
  - Solaris Green Threads
  - GNU Portable Threads



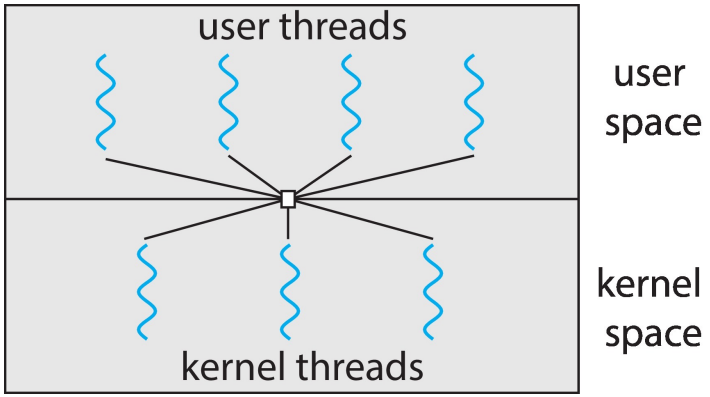
Uno-a-uno

- Ogni thread a livello utente è mappato su un thread a livello kernel
- Permette un maggior grado di concorrenza
- Permette di sfruttare il parallelismo nei sistemi multicore
- Minore performance del threading interamente a livello utente
- Stress del kernel
- Esempio: Linux, Windows



Molti-a-molti

- Permette di mappare un insieme di thread a livello utente su un numero minore o uguale di thread a livello kernel
- Cerca di combinare i vantaggi dei modelli molti-a-uno e uno-a-uno ma è complesso da implementare
- **Modello a due livelli**: permette anche di associare un thread utente ad un thread kernel
- Esempio: Solaris, AIX



Librerie di thread

- Sono le API fornite al programmatore per creare e gestire thread
- Librerie più in uso:
  - POSIX pthreads
  - Windows threads
  - Java threads

POSIX pthreads

- Non sono un'*implementazione*, ma una *specifica* (POSIX standard IEEE 1003.1c)
- Esistono implementazioni sia a livello utente che a livello kernel
- Comune nei sistemi Unix e Unix-like (BSD, Linux, macOS)

Windows threads

- Libreria di thread del sistema operativo Windows
- Implementata a livello kernel

Java threads

- Libreria di threads del linguaggio di programmazione (e ambiente di esecuzione) Java
- Implementata sopra un'altra libreria di threads (pthreads o Windows threads)
- Dalla versione 1.5 offre anche dei costrutti di livello più alto rispetto alle altre librerie di thread (executors, thread pools, fork-join…)

Problematiche di programmazione multithread

- Chiamate di sistema `fork()` ed `exec()`
- Cancellazione dei thread
- Dati locali dei thread
- Comunicazione tra libreria dei thread e scheduler dei thread kernel

Chiamate di sistema `fork()` ed `exec()`

- Una `fork()` dovrebbe duplicare solo il thread chiamante o tutti i thread? Alcuni sistemi operativi Unix-like hanno due diverse `fork()`
- `exec()` di solito sostituisce tutti i thread del processo in esecuzione

Cancellazione dei thread

- La cancellazione di un thread è un'operazione che determina la terminazione prematura del thread
- Due approcci:
  - Cancellazione asincrona: il thread è terminato immediatamente
  - Cancellazione differita: il thread controlla periodicamente se deve terminare, in modo da effettuare una terminazione ordinata

Cancellazione nei POSIX pthreads

- Si può attivare/disattivare la cancellazione, ed avere sia cancellazione differita (default) che asincrona
- Se la cancellazione è inattiva, le richieste di cancellazione rimangono in attesa fino a quando (se) è attivata
- In caso di cancellazione differita, questa avviene solo quando l'esecuzione del thread raggiunge un punto di cancellazione (di solito una chiamata di sistema bloccante)
- Il thread può aggiungere un punto di cancellazione controllando l'esistenza di richieste di cancellazione con la funzione `pthread_testcancel()`

Cancellazione nei Java threads

- È possibile impostare lo stato di cancellazione (differita) di un thread invocando il metodo `interrupt()` della classe `Thread`
- Un thread può verificare se ha una richiesta di cancellazione invocando il metodo `isInterrupted()`

Dati locali dei thread

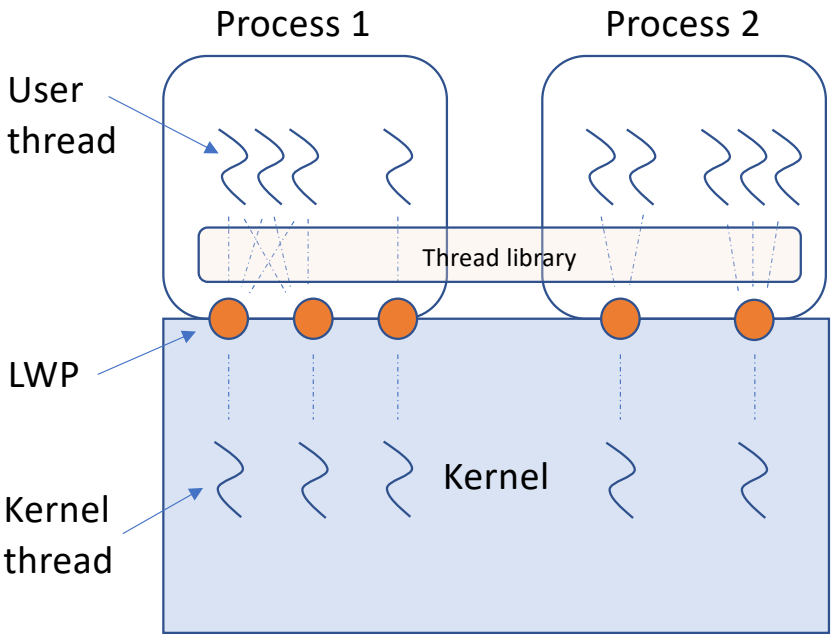
- In alcuni casi è utile assegnare ai thread dei dati privati (**thread local storage**, TLS)
- Utile quando non vi è controllo sul meccanismo di creazione dei thread (es. thread pools)
- Diversa dalle variabili locali (TLS è visibile a tutte le funzioni)
- Simile ai dati `static` del linguaggio C, ma sono unici per ogni thread

Comunicazione tra libreria dei thread e scheduler dei thread kernel

- Nel caso dei modelli di thread molti-a-molti e a due livelli, diventa complesso mantenere il mapping tra thread del kernel e thread utente
- Un problema in particolare è la comunicazione tra libreria dei thread a livello utente e scheduler dei thread kernel: entrambi devono prendere decisioni di scheduling, e quindi devono in qualche modo collaborare
- UNIX System V e derivati (Solaris, AIX, HP-UX) introducono queste componenti:
  - **Lightweight process** (LWP)
  - Attivazione dello scheduler

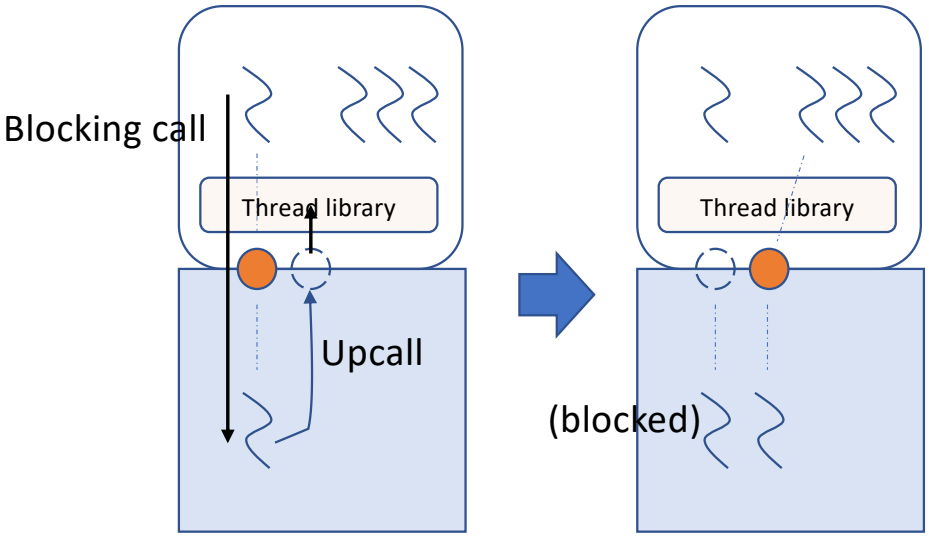
Lightweight process

- Un LWP è l'interfaccia che il kernel offre alle librerie di threading per accedere ai thread dello scheduler
- Per le librerie di threading un LWP è un processore virtuale, che la libreria può usare per schedulare un thread utente
- Ogni LWP è associato staticamente dal kernel ad esattamente un thread del kernel
- In più l'LWP mantiene (minime) informazioni relative al contesto utente



Attivazione dello scheduler

- Il kernel comunica alla libreria dei thread l'occorrenza di determinati eventi attraverso opportune **upcall** a funzioni della libreria
- Le upcall sono effettuate nel contesto di un LWP
- La libreria dei thread può prendere opportune decisioni di scheduling, ad esempio continuare ad eseguire il thread utente sul nuovo LWP che gli è stato assegnato

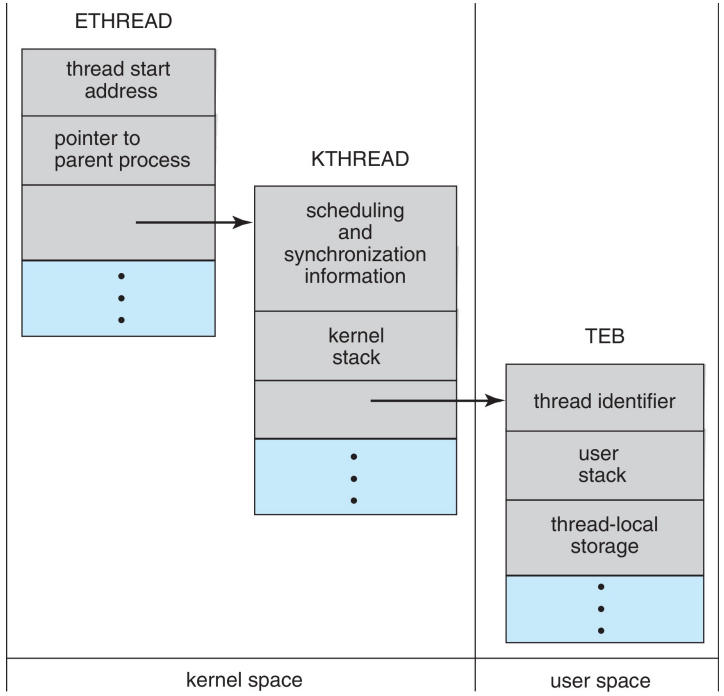


Implementazione threads in Windows (1)

- Modello uno-a-uno
- Ogni thread ha:
  - Un identificatore
  - Un insieme di registri, incluso il PC
  - Uno stack per l'esecuzione in modalità utente e uno per l'esecuzione in modalità kernel
  - Un'area di memoria privata usate da diverse librerie
- Registri, stack e memoria privata formano il **contesto** del thread

Implementazione threads in Windows (2)

- Le strutture dati principali sono:
  - ETHREAD (spazio kernel)
  - KTHREAD (spazio kernel)
  - TEB (spazio user)



Implementazione threads in Linux

- Linux non distingue tra processi e threads e utilizza il termine **task** per indicare un generico flusso di controllo nell'ambito di un programma
- La funzione di sistema `clone()` permette di creare un nuovo task
- Una serie di flags indica quali risorse il task figlio deve condividere con il padre
- Questo permette un'ampia granularità di comportamenti, dalla creazione di threads a funzionalità simili a quelle della `fork()`
- Tale funzionalità è alla base anche della creazione dei containers