5° lezione S.O

Due Date	@March 20, 2025
Multi-select	Thread
Status	Done

Scheduling dei thread

Il sistema di gestione dei thread dovrà scegliere a quale processo assegnare la CPU.

Si hanno tanti modi di fare questa scelta

1. prima viene scelto il processo da eseguire poi si scopre che vi sono tot thread all' interno e viene scelto il thread da eseguire

Qualunque SO moderno gestisce i thread, l'algoritmo di scheduling ragiona in termini di thread

Il costo del context switch va tenuto conto quando il SO fa la scelta per lo scheduling della CPU

Cambio di contesto (Context Switch)

Il **context switch** è l'operazione con cui il sistema operativo assegna l'unica CPU fisica a un altro processo. Come discusso nelle lezioni precedenti, questo meccanismo può essere attivato per vari motivi.

Prima del Context Switch, bisognerà salvare il set di registri della CPU e dovrà essere effettuata una riprogrammazione della MMU; ciò garantisce che ogni processo possa accedere solo al proprio spazio di indirizzamento, senza poter interferire con la memoria dedicata ad altri processi.

È fondamentale considerare il **context switch** per il processo uscente, poiché, come detto, occorre riprogrammare l'MMU e salvare i dati del processo che sta uscendo. Allo stesso tempo, bisogna salvare anche i dati necessari per l'esecuzione del processo che sta entrando.

Il Context Switch più veloce si riferisce al passaggio del contesto tra thread (cioè thread di uno stesso processo)

- **Cambio di contesto tra processi** (”classico”)→ salvataggio dei processi, riprogrammazione della MMU
- **Cambio di contesto tra thread** → più veloce e non bisogna riprogrammare l’MMU in quanto i thread fratelli lavorano sullo stesso spazio di indirizzamento: dobbiamo soltanto effettuare il cambio dei valori dei registri

Quest’ultimo cambio sarà più veloce rispetto al cambio processo - processo ma non è legato a questo bensì ai processi.

Il costo minore nel context-switching è rilevante in quanto quest’operazione di cambio avviene molto frequentemente.

Operazione sui thread

Task che si riscontrano su ogni sistema che gestisce thread:

- `thread_create`: un thread che ne crea un altro
- `thread_exit`: la virtual CPU si spegne ma gli altri thread dello stesso processo vanno avanti. È differente da Win perché fa terminare tutto il processo.
- `thread_join`: è come la `wait` consente ad un thread di addormentarsi fino alla fine di un altro thread; al termine del thread a cui si è fatta la join vi sarà una return al thread “principale”; è una **chiamata bloccante**
- `thread_yield`: un thread cede il controllo della CPU ad un thread fratello; implementa un modello collaborativo tra thread.

La `thread_join` è particolarmente utile per gli algoritmi di ordinamenti come QuickSort e MergeSort in cui è convenite che il main si addormenti fino a che i thread possano risolvere i sotto problemi e, successivamente, tornano al main.

Un thread viene sempre creato da un’altro thread

La differenza tra **syscall exit** e **thread_exit** riguarda il contesto in cui vengono utilizzate e l'ambito in cui terminano i processi o i thread.

- `syscall exit`: Questa chiamata di sistema termina l'intero processo, compreso tutti i thread che sono stati creati all'interno di quel processo. Quando viene invocata, il processo viene terminato e il sistema operativo rilascia le risorse ad esso associate. L'uscita di un processo comporta la fine di tutti i thread e la pulizia delle risorse a livello di processo.

- `thread_exit` : Questa funzione termina solo il thread che la invoca, senza influire sugli altri thread che potrebbero essere ancora attivi all'interno dello stesso processo.

Programmazione multicore

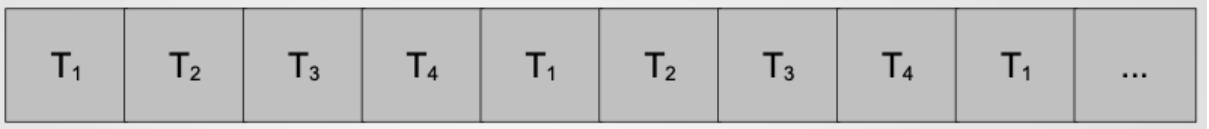
L'uso dei **thread** consente di sfruttare al massimo le capacità computazionali dei moderni calcolatori, sia in termini di **parallelismo** che di **multithreading** e **hyperthreading**.

Nei sistemi attuali, caratterizzati dalla presenza di **più core fisici**, la gestione ottimale dei thread è fondamentale per migliorare le prestazioni e massimizzare l'uso delle risorse disponibili.

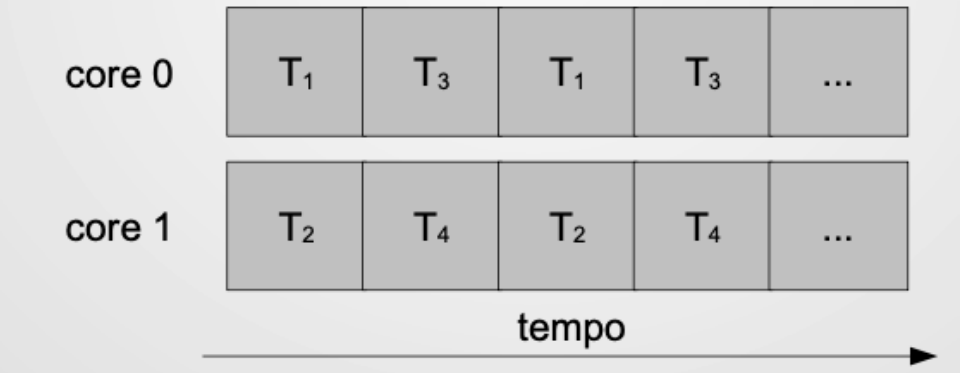
Prima dell'introduzione dei thread, per sfruttare appieno i **core** disponibili era necessario eseguire più **processi** indipendenti, spesso associati a più **utenti**, in modo da massimizzare il **throughput** complessivo del sistema.

Per risolvere questa esigenza, si possono adottare due strategie:

1. utilizzare un **sistema multiprogrammato e multiutente**, in cui più processi vengono eseguiti contemporaneamente su core diversi.



2. avere un **unico processo multithread**, capace di sfruttare tutti i core disponibili per eseguire più operazioni in parallelo.

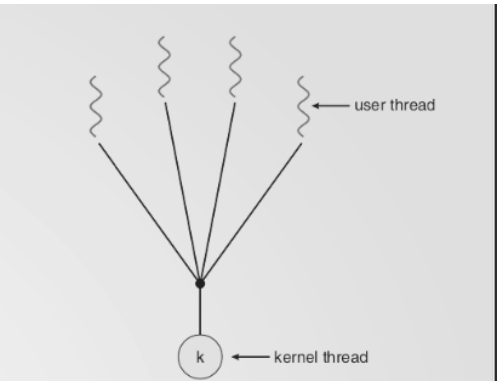


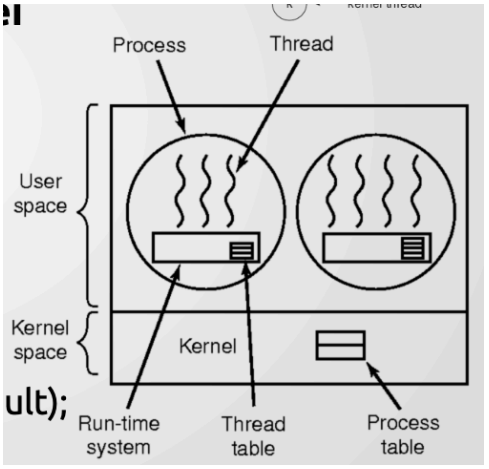
Programmi per sistemi multicore

Per scrivere deli programmi che sfruttino sistemi multicore vi è un lavoro di preparazione, bisognerà:

- **Individuazione dei task**: identificare le operazioni che possono essere eseguite in parallelo.
- **Bilanciamento del carico**: suddividere i task in modo equo, evitando che alcuni siano troppo brevi o troppo sbilanciati rispetto agli altri.
- **Suddivisione dei dati**: analizzare la condivisione e le dipendenze tra i thread, considerando se l'output di un thread è necessario per un altro o se vi sono strutture dati utilizzate da più thread.
- **Test e debugging**: la programmazione multithread è più complessa della programmazione sequenziale, poiché l'interlacciamento dei thread è fuori dal controllo diretto del programmatore. A seconda che si disponga di una o più CPU, il concetto di parallelismo può variare, e i thread potrebbero non essere eseguiti realmente in parallelo su CPU fisiche, ma essere gestiti dal sistema operativo. Un problema cruciale è la **concorrenza**, ovvero la gestione simultanea di thread che collaborano e accedono alla stessa struttura dati. Senza adeguati meccanismi di sincronizzazione, possono verificarsi condizioni di gara (race conditions), inconsistenze nei dati e comportamenti imprevedibili del programma. Questo rende il debugging ancora più difficile.

Thread a livello utente





L'implementazione dei thread può avvenire in diversi modi. Inizialmente, i thread furono teorizzati senza il supporto diretto del sistema operativo, ma era

comunque possibile implementarli tramite **thread a livello utente**.

Nel modello **uno-a-molti**, il sistema operativo non gestisce direttamente i thread, ma si limita a creare processi con un singolo flusso di esecuzione. Il nome **uno-a-molti** deriva dal fatto che il SO vede un solo thread per ogni processo (*uno*), mentre all'interno del processo possono esistere più thread gestiti autonomamente dall'applicazione (*molti*). Un'applicazione, quindi, può definire autonomamente più thread senza che il SO ne sia a conoscenza, permettendo di sfruttare il concetto di multithreading senza l'intervento diretto del sistema operativo.

Il sistema operativo offre un solo thread di base per ogni processo, noto come **thread kernel**, che viene utilizzato dall'applicazione per gestire i vari task interni. Periodicamente, il SO assegna la CPU ai processi e, una volta assegnata, il processo gestisce autonomamente l'esecuzione dei propri thread utente. L'interlacciamento tra questi thread avviene grazie a un **sistema runtime**, senza che il sistema operativo debba intervenire direttamente. In pratica, il processo dispone di una **virtual CPU**, attraverso la quale può portare avanti i suoi thread.

Lo scheduling dei thread interni può essere gestito dal processo stesso, ma questa operazione è estremamente complessa. Per facilitare questa gestione, si utilizzano delle **librerie utente**, che operano con gli stessi privilegi del codice applicativo e si occupano della gestione dei thread. Queste librerie implementano un **sistema di gestione dei thread utente**, fornendo funzioni simili a quelle dei thread gestiti direttamente dal sistema operativo.

All'interno del processo, la libreria deve mantenere una **tabella dei thread**, che contiene informazioni sullo stato e sull'esecuzione di ciascun thread, consentendo un controllo efficace delle risorse e delle operazioni in corso.

Context switch

Come fa la CPU a saltare da un thread all’altro? Questo avviene tramite la funzione **thread_yield**, con cui i thread cedono volontariamente la virtual CPU ad altri thread dello stesso processo. Il controllo passa alla libreria di gestione dei thread, che seleziona e avvia un altro thread utente all’interno del processo. In questo modo, il **context switch** viene implementato interamente in modalità utente, senza la necessità di utilizzare chiamate di sistema (*trap*).

Questa modalità di context switch si basa sulla **cooperazione** tra i thread, ovvero ogni thread deve volontariamente cedere il controllo.

Problemi

Uno dei principali problemi di questa gestione è legato alle **chiamate bloccanti**. Se un thread esegue una chiamata di sistema bloccante, il sistema operativo vede la richiesta come proveniente dall'intero processo e non dal singolo thread. Di conseguenza, **bloccherà l'intero processo**, impedendo l'esecuzione degli altri thread.

Per mitigare questo problema, è possibile determinare in anticipo se una chiamata di sistema sarà bloccante. Ad esempio, nel caso dell'input da tastiera (`scanf` o `cin`), si può utilizzare la funzione `select` per verificare se l'operazione sarebbe bloccante. Se si prevede un blocco, il thread può chiamare `thread_yield` per cedere temporaneamente la CPU a un altro thread dello stesso processo. Questo approccio permette di gestire il blocco in modo più efficiente, evitando che l'intero processo venga sospeso inutilmente.

Passando l'esecuzione a un altro thread, possiamo continuare a lavorare in attesa che la periferica recuperi il dato richiesto. Quando la periferica sarà pronta, il thread che ha ceduto il controllo eseguirà la chiamata bloccante, ma essa tornerà subito, poiché la risorsa sarà ora disponibile.

Per molte funzioni, esistono versioni **non bloccanti** che permettono al thread di continuare a eseguire altre operazioni mentre attende una risposta o l'accesso a una risorsa.

Un altro concetto da considerare riguarda la **memoria virtuale**. In un sistema con memoria virtuale, un processo potrebbe avere una parte dei propri dati non caricata in **memoria centrale** (RAM), ma memorizzata su **disco**. Quando il processo richiede l'accesso a una porzione di dati che non è ancora in memoria, il sistema operativo interviene per caricare quella porzione di dati dal disco alla memoria. Questo processo è noto come **paging** e può causare un **page fault**, dove il sistema sospende temporaneamente l'esecuzione del processo per recuperare i dati dal disco.

Pro

Scheduling personalizzato: Quando si gestiscono più di due thread, è possibile implementare uno **scheduling personalizzato** che consenta una gestione più fine dei thread all'interno di un processo. Mentre lo scheduling del sistema operativo (SO) fa delle scelte di alto livello senza entrare nei dettagli delle operazioni svolte da ciascun thread all'interno di ogni processo, con la gestione

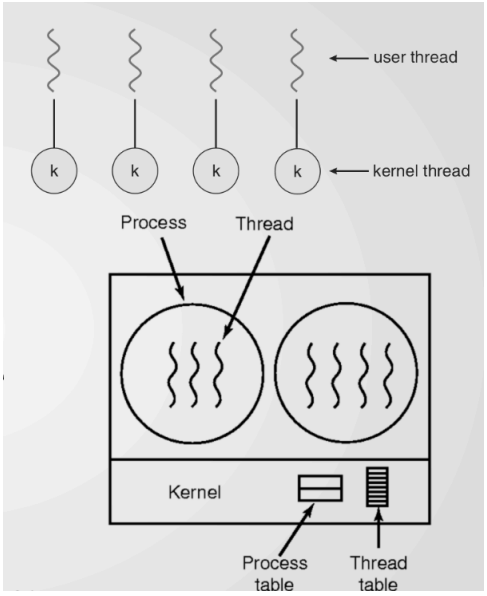
dei thread utente si acquisisce una maggiore conoscenza di come gestire i thread in base alle operazioni che stanno eseguendo.

Con la gestione dei thread utente, il sistema ha la possibilità di scegliere in modo più mirato quale thread deve essere eseguito, considerando fattori come la priorità, il tipo di operazione in corso e la dipendenza tra i vari thread. In pratica, il sistema utente può intervenire direttamente e in modo più ottimizzato, adattandosi meglio alle esigenze specifiche dell'applicazione.

L'**operazione di context switch** (o **dispatching**) avviene in modalità utente, il che significa che il passaggio da un thread all'altro avviene senza il coinvolgimento del kernel. Questo approccio è generalmente più efficiente rispetto al context switch a livello kernel, poiché comporta un **minor overhead**. Infatti, la gestione dei thread utente non richiede il passaggio al kernel, riducendo il tempo necessario per il cambio di contesto e migliorando le prestazioni complessive del sistema.

Questa parte non si sente benissimo

Thread a livello kernel



Nel sistema operativo, esistono due tabelle: una per i processi e una per i thread, entrambe gestite dal kernel.

Nel **modello 1-a-1**, ogni task definito nell'applicazione corrisponde direttamente a un thread del kernel. In altre parole, ogni thread creato a livello applicativo ha

un suo equivalente a livello kernel, gestito direttamente dal sistema operativo, che distingue chiaramente i thread dai processi.

Pro

Quelli che ci aspetteremo dal modello a thread:

- Le chiamati bloccanti non intralciano gli stessi thread

Contro

I principali svantaggi, di entità minore, sono i seguenti:

- **Cambio di contesto più lento**:
  - Il **context switch tra processi** è il più lento, poiché richiede il salvataggio completo dello stato del processo, la riprogrammazione dell'MMU e la gestione delle risorse.
  - Il **context switch tra thread del kernel e thread utente** è anch'esso piuttosto costoso, in quanto implica il salvataggio dei registri e la riprogrammazione dell'MMU.
    - Questo può avvenire tramite una **syscall trap**, che causa un'interruzione per passare dalla modalità utente a quella kernel.
    - Oppure tramite **prelazione**, che comporta lo stesso tipo di passaggio tra modalità utente e modalità kernel.
- **Cambio di contesto più veloce**:
  - Il **context switch tra thread del kernel** è più veloce, in quanto non richiede la riprogrammazione dell'MMU né la commutazione tra modalità utente e kernel.
  - Il **context switch tra thread utente imparentati** è il più veloce, poiché non comporta né la riprogrammazione dell'MMU né la commutazione tra modalità utente e kernel, essendo thread appartenenti allo stesso processo.
- **Creazione e distribuzione di thread più costosa per il SO**: La creazione e distribuzione di thread sono operazioni costose per il sistema operativo, poiché richiedono un maggiore overhead nella gestione delle risorse e della loro assegnazione rispetto alla gestione dei processi tradizionali.

Modello ibrido

Modello molti-a-molti: Nel modello **molti-a-molti**, un programmatore definisce un numero di task e decide come distribuirli sui thread a disposizione, con la possibilità di gestire più thread rispetto al numero di core fisici. In questo modello, un singolo thread del kernel può gestire molti thread utente, dando al sistema un controllo maggiore sulla gestione dei task.

Pro

- **Minore overhead**: Essendo la gestione dei thread più dinamica, si riduce l'overhead associato alla creazione, gestione e schedulazione dei thread.
- **Flessibilità**: Permette una gestione fine dei thread, poiché un singolo thread del kernel può gestire molti thread utente in base alla disponibilità di risorse.
- **Parallelismo**: Il modello molti-a-molti permette di sfruttare in modo più efficiente i processori multi-core, consentendo al sistema di eseguire più thread contemporaneamente, ottimizzando così le performance.
- **Scalabilità**: Il modello consente una gestione scalabile dei thread, poiché l'applicazione può crescere e utilizzare un numero maggiore di thread senza incorrere in problemi di performance.

Associamo ad alcuni thread kernel dei thread utente che svolgeranno dei certi task. Per fare ciò ci serviamo sia delle implementazione dei thread dek kernel del SO, sia di librerie a runtime per ottenere a tutti gli effetti un ibrido dei due modelli precedenti.

Thread nei nostri SO

Tutti i moderni sistemi operativi supportano i **thread kernel**, che sono gestiti direttamente dal sistema operativo stesso. Tuttavia, i **thread utente** esistono ancora e possono essere utilizzati tramite librerie specifiche, che sono dipendenti dal sistema operativo in uso.

Per risolvere il problema della dipendenza dal sistema operativo, poiché ogni sistema ha le proprie **syscall** per la gestione dei thread, si utilizzano i **Pthread di POSIX** (che useremo nel laboratorio). Questa libreria implementa i thread per tutti i sistemi che seguono lo standard POSIX, rendendo il codice più portabile.

Grazie a Pthread, è possibile scrivere codice che gestisce i thread senza preoccuparsi delle specifiche syscall del sistema operativo sottostante. Sarà la

libreria stessa a chiamare le opportune syscall in base al sistema su cui il codice è compilato, semplificando lo sviluppo e migliorando la portabilità.