UD 2 - THREAD

Thread

Tradizionalmente, i processi fanno uso di un singolo flusso di controllo che svolge tutte le attività dell'applicazione (controllo del flusso delle operazioni, le operazioni di ingresso-uscita e l'elaborazione all'interno del processore).

Ci possono essere situazioni in cui una singola applicazione deve poter gestire molti compiti simili tra loro, o in cui si vuole continuare ricevere richieste nonostante altre stiano venendo servite. Oltre a maggiore flessibilità ed efficienza nella computazione.

Una prima soluzione potrebbe essere quella di creare, attivare tanti processi che erogano il servizio quante sono le richieste per lo stesso (ovviamente stiamo parlando di sistemi multiprogrammati e multitasking), ma questa soluzione, essendo fattibile, può portare a difficoltà di raggiungimento degli obiettivi di disponibilità e basso tempo di risposta, in quanto la creazione dei processi è molto onerosa sia a livello di tempi che di risorse.

Un'altra possibile soluzione è quella di introdurre il concetto di thread, ovvero un gruppo di flussi di esecuzione autonomi sullo stesso programma che accedono alla stessa porzione di memoria centrale. Il thread può essere anche considerato come l'unità base dell'utilizzo della CPU, e comprende un identificatore, un program counter, un set di registri e uno stack. Inoltre condivide con gli altri thread, che appartengono allo stesso processo, la sezione di codice, quella dei dati e altre risorse del sistema operativo.

Un processo mono-thread è caratterizzato da un solo flusso di esecuzione che fa tutto mentre un processo multithread è caratterizzato da più flussi di esecuzione di istruzioni in parallelo, operanti contemporaneamente e con parte delle informazioni condivise in memoria centrale.

In questo modo ogni thread svolgerà le proprie operazioni e potrà eventualmente trasmetterne i risultati a flussi diversi della computazione, proprio in virtù della condivisione dell'accesso allo stesso spazio di indirizzamento. Ovviamente andranno sincronizzati opportunamente, o altrimenti si potrebbero avere dati inconsistenti (perché ad esempio modificati da altri).

Pur avendo in comune lo stesso codice, due thread diversi hanno comunque contesti specifici su cui operare (quindi registri e stack separati), in modo da poter eseguire operazioni diverse prese da parti diverse del codice. Se infatti condividessero anche stack e registri, finirebbero inevitabilmente per eseguire le stesse operazioni nello stesso momento.

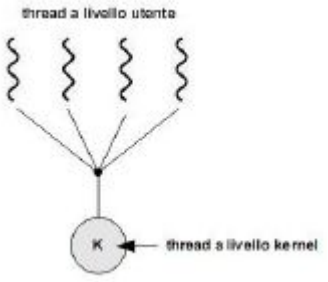
L'utilizzo dei thread porta a diversi benefici:

- Maggior disponibilità: è possibile gestire delle richieste mentre altri thread sono occupati;
- Condivisione delle risorse: tutti i thread condividono tra loro codice, dati e risorse, e ciò permette loro di accedervi semplicemente e anche di comunicare (sincronizzando però gli accessi per evitare problemi di consistenza);
- Economia: creare un nuovo thread richiede meno tempo e memoria rispetto a un nuovo processo, poiché non è necessario copiare nulla;
- Sfruttamento delle architetture multiprocessore: i thread possono essere eseguiti in parallelo su processori differenti, raggiungendo così una condivisione efficiente e diretta delle informazioni.

Il supporto ai thread può essere fornito nello spazio utente (thread livello utente) o nel kernel (thread livello kernel). Una libreria di thread fornisce al programmatore le funzioni per la creazione e la gestione dei thread. Per i thread livello utente, tutto il codice e le strutture dati della libreria si trovano nello spazio utente e le chiamate alle sue funzioni sono semplici chiamate a procedura. Per i thread livello kernel, invece, codice e dati si trovano nella memoria del SO, e le chiamate si traducono in chiamate di sistema.

Modelli multi-thread

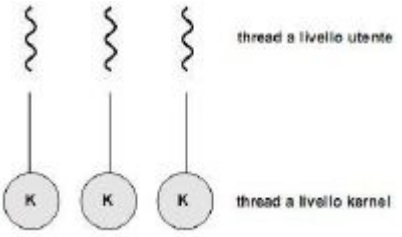
Modello molti-a-uno



Tutti i thread utente sono mappati su un solo thread kernel. La libreria livello utente ha quindi il compito di simulare un ambiente multithread schedulando i thread utente sul thread kernel.

Questo modello è semplice da realizzare ma presenta dei problemi: poiché vi è un solo thread kernel, non sfrutta le architetture multiprocessore (il SO vede solo un processo da schedulare) e inoltre se uno dei thread in esecuzione si blocca (ad esempio per una operazione I/O), verranno bloccati di conseguenza tutti gli altri thread che fanno parte del suo gruppo (bloccherà cioè l'unico thread kernel e quindi l'intero processo).

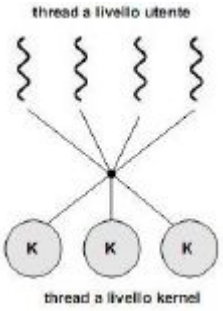
Modello uno-a-uno



E' il più utilizzato e mappa ogni thread livello utente su un thread a livello kernel. Permette il massimo sfruttamento dei sistemi multiprocessore (più thread vengono eseguiti in parallelo su diversi processori) e la massima concorrenza, permettendo ad un thread di essere eseguito nonostante un'eventuale chiamata bloccante da parte di un altro (se un thread si blocca, si blocca solo quello).

Lo svantaggio di tale modello è che la creazione di un thread utente richiede la creazione del corrispondente kernel thread, con un conseguente overhead di gestione che grava le prestazioni del sistema all'aumentare dei thread.

Modello molti-a-molti



Raggruppa in vario modo i thread a livello utente verso un numero inferiore (o equivalente) di kernel thread (n thread utente sono mappati su m thread kernel, quindi m<=n).

Con questo modello vengono superati entrambi i limiti dei due precedenti, perché non ho più i problemi di concorrenza del molti-a-uno né limiti nella creazione di thread utente come nell' uno-a-uno. Col molti-a-molti posso infatti creare quanti thread voglio, che potranno comunque essere eseguiti in parallelo.

Modello due livelli

Una variante comune è quella del modello a due livelli, che mappa molti thread di livello utente verso un numero più piccolo o equivalente di kernel thread, ma permette anche di associarne alcuni in modalità uno-a-uno.

Cooperazione tra thread

Per quanto riguarda la cooperazione tra più thread, questa può essere rappresentata secondo tre modelli:

- thread simmetrici: tutti i thread possono svolgere lo stesso insieme di attività. Quando arriva una richiesta, un qualsiasi thread disponibile può gestirla;
- thread gerarchici: un thread coordinatore riceve le richieste, e dà ordini a più thread lavoratori che le eseguono;
- thread in pipeline: come in una catena di montaggio, ogni thread è specializzato nell'eseguire una piccola operazione velocemente, passando poi il risultato parziale al thread successivo, fino ad ottenere il risultato completo. Permette un elevato throughput, dal momento che ogni thread torna in attesa di soddisfare nuove richieste dopo il tempo che impiega per svolgere la sua piccola sequenza di operazioni.

Gestione dei thread

Creazione

La sintassi della chiamata di sistema per creare un nuovo thread è uguale a quella dei processi, ovvero fork(). Però la semantica cambia. In un processo tradizionale, la chiamata fork crea una copia del processo, ma in un'applicazione multithread si può comportare in 2 modi: può duplicare tutti i thread del processo chiamante, oppure solo il thread chiamante. Dipende dal sistema operativo.

Se è possibile scegliere, si sceglie in base alle istruzioni successive: se subito dopo c'è una exec(), il nuovo processo sarà sovrascritto quindi basterà duplicare solo il thread chiamante; in caso contrario può essere meglio duplicarli tutti.

Cancellazione di un thread

Cancellare un thread significa terminarlo prima che abbia completato la sua esecuzione. Il thread che sta per essere cancellato viene spesso chiamato thread target e la sua cancellazione può avvenire in due modi:

- **Cancellazione asincrona**: il thread viene terminato immediatamente, ma potrebbe lasciare risorse in stato inconsistente
- **Cancellazione differita**: si imposta una variabile nel thread target che questi deve controllare periodicamente in punti sicuri (punti di cancellazione), in cui non si rischi di lasciare risorse in stato inconsistente. Questo però permette a un thread di ignorare la richiesta e attendere che finisca la sua computazione, ottenendo una terminazione ordinaria.

Comunicazione

Se ho due processi che lavorano insieme ed ho un thread che vuole comunicare con l'altro processo, la comunicazione può avvenire con tutti i thread del processo destinatario, con un loro sottoinsieme o con uno specifico. La soluzione da adottare dipende dall'applicazione. Tipicamente il SO consegna al primo thread che non lo blocca, così che sia gestito una sola volta.

Sincronizzazione e comunicazione

Nei sistemi che implementano i modelli molti a 1, molti a molti e a 2 livelli si introduce una struttura dati intermedia tra un thread livello kernel e i thread livello utente che devono essere mappati su esso: il **lightweight process** (LWP), che separa le due visioni e memorizza dei dati di gestione.

Dal punto di vista del kernel, un LWP è visto come un processo, e come tale viene schedulato, con la particolarità che tutti gli LWP appartenenti a un processo hanno lo spazio di indirizzamento condiviso.

Dal punto di vista della libreria utente, un LWP appare come un processore virtuale su cui schedulare l'esecuzione dei thread utente in base a qualche algoritmo. La libreria, inoltre, può variare il numero di thread a livello kernel (e quindi LWP) così da avere sempre un certo numero di thread in esecuzione.