**Parte II - Capitolo 4 - Thread**

Un ***thread*** è l'unità di base d'uso della CPU e comprende un identificatore di thread (ID), un contatore di programma, un insieme di registri e una pila (stack). Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse del sistema come i file aperti e i segnali. Un processo tradizionale chiamato anche *processo pesante* (*heavyweight process*), è composto da un solo thread. Un processo multithread è in grado di lavorare a più compiti in modo concorrente. La creazione dei processi è molto onerosa, sia a livello dei tempi che a livello di costi, quindi, se il nuovo processo si deve occupare degli stessi compiti del processo corrente, non c'è alcuna ragione di accettare l'intero carico che la sua creazione comporta. I vantaggi della programmazione multithread sono: tempo di risposta, cioè un'applicazione può permettere a un programma di continuare la sua esecuzione anche se parte di essa è bloccata o sta eseguendo un'operazione particolarmente lunga, condivisione delle risorse, cioè un'applicazione può avere molti thread di attività diverse nello stesso spazio di indirizzi, economia, cioè risulta molto più conveniente creare i thread che assegnare memoria e risorse per la creazione di nuovi processi, scalabilità, cioè nelle architetture multiprocessore è possibile eseguire i thread in parallelo. Una tendenza recente è quella di montare diverse unità di calcolo su un unico processore (processore multicore). La programmazione multithread offre un meccanismo per un utilizzo più efficiente di questi processori e aiuta a sfruttare al meglio la concorrenza. In particolare nella progettazione di sistemi multicore si deve porre l'attenzione su diversi obiettivi quali: separazione dei task (esaminare le applicazioni al fine di individuare aree separabili in task distinti e concorrenti), bilanciamento (i task che vengono eseguiti in parallelo devono eseguire compiti di mole e valore confrontabile), suddivisione dei dati (i dati che vengono utilizzati dai task devono essere suddivisi per essere utilizzati da unità di calcolo distinte), dipendenze dei dati (i dati a cui i task accedono devono essere esaminati per verificare le dipendenze tra due o più task) e test e debugging (effettuare queste operazioni è molto più complicato per programmi concorrenti).

I thread possono essere distinti in ***thread a livello utente***, che sono gestiti senza l'aiuto del kernel, e ***thread a livello kernel***, che sono gestiti direttamente dal sistema operativo. Analizziamo ora le relazioni tra i thread utente e i thread kernel. Il ***modello da molti a uno*** fa corrispondere molti thread a livello utente a un singolo thread a livello kernel. Poichè si svolge nello spazio utente, la gestione dei thread risulta efficiente, ma l'intero processo rimane bloccatose un thread invoca una chiamata di sistema di tipo bloccante. Inoltre, poichè un solo thread alla volta può accedere al kernel, è impossibile eseguire thread multipli in parallelo in sistemi multiprocessore. Questo modello è utilizzato nella libreria *green threads* disponibile per Solaris e per GNU *portable thread*. Il ***modello da uno a uno*** mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel. Questo modello offre un grado di concorrenza maggiore, poichè anche se il thread invoca una chiamata di sistema bloccante, è possibilie eseguire un altro thread. Il modello permette anche l'esecuzione di più thread in parallelo nei sistemi multiprocessore. Lo svantaggio di questo modello è che la creazione di ogni thread a livello utente, comporta la creazione di corrispondenti thread a livello kernel. Poichè la creazione di thread a livello kernel può compromettere le prestazioni di un'applicazione, la maggior parte delle realizzazioni di questo modello limita il numero di thread gestibili dal sistema. Il modello da uno a uno è utilizzato nei sistemi operativi Linux e Windows. Il ***modello da molti a molti*** mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel. Nonostante questo modello permetta di creare tanti thread a livello utente quanti se ne desiderano, non viene garantita la concorrenza reale, poichè il meccanismo di scheduling del kernel può scegliere un solo thread alla volta. Una variante del modello da molti a molti è il ***modello a due livelli*** che mantiene la corrispondenza fra più thread utente con un numero minore o uguale di thread del kernel, ma permette anche di vincolare un thread utente a un solo thread kernel.

Una ***libreria di thread*** fornisce al programmatore una API per la creazione e la gestione dei thread. I metodi con cui implementare una libreria dei thread sono essenzialmente due: o la libreria è collocata interamente a livello utente, o la libreria è implementata a livello kernel con l'ausilio diretto del sistema operativo. Attualmente le librerie di thread maggiormente in uso sono tre e vengono descritte in seguito. Con il termine ***Pthreads*** ci si riferisce allo standard POSIX che definisce la API per la creazione a la sincronizzazione dei thread. I sistemi che implementano le specifiche Pthreads sono Solaris, Linux, Mac OS X e Tru64 UNIX. Per vari sistemi Windows sono disponibili implementazioni *shareware* di dominio pubblico. La libreria definisce il comportamento dei thread, ma non li realizza. I progettisti dei sistemi operativi possono realizzare le API come meglio credono. I thread eseguiti da una funzione specifica condividono i valori globali. Inoltre, tutti i programmi che impiegano la libreria Pthreads devono includere il file d'intestazione pthread.h. La tecnica usata dalla libreria ***Win32*** per la creazione dei thread richiama per molti versi quella di Pthreads. Come per la libreria Pthreads è necessario includere il file d'intestazione windows.h e i dati condivisi da thread separati sono globali. I thread rappresentano il paradigma fondamentale per l'esecuzione di programmi in ambienti ***Java***; il linguaggio Java, con la propria API, è provvisto di una ricca gamma di caratteristiche per la generazione e la gestione dei threads. In un programma Java vi sono due tecniche per la generazione dei thread. La prima tecnica consiste nel creare una nuova classe derivata dalla classe Thread e sovrascrivere il suo metodo run(). L'alternativa consiste nel definire una classe che implementi l'interfaccia Runnable. Essendo un linguaggio orientato agli oggetti, Java non contempla la condivisione dei dati tra thread diversi come per Pthreads e Win32. La condivisione avviene tramite il passaggio di riferimenti a uno stesso oggetto.

Affrontiamo alcuni problemi riguardanti i programmi multithread. In un programma multithread la semantica delle chiamate di sistema fork() ed exec() cambia: se un thread in un programma invoca la chiamata di sistema fork(), il nuovo processo potrebbe, in generale, contenere un duplicato di tutti i thread oppure del solo thread invocante. Alcuni sistemi UNIX includono entrambe le versioni. La chiamata exec() di solito funziona come descritto per i processi: il programma specificato sostituisce l'intero processo, inclusi tutti i thread. La cancellazione dei thread è l'operazione che permette di terminare un thread prima che completi il suo compito. Un thread da cancellare è chiamato *thread bersaglio* (*target thread*) e la sua cancellazione può avvenire in due modi: ***cancellazione asincrona*** (un thread fa immediatamente terminare il thread bersaglio) o ***cancellazione differita*** (il thread bersaglio può controllare periodicamente se deve terminare, in modo da riuscirvi in modo opportuno). La cancellazione asincrona potrebbe non liberare una risorsa necessaria a tutto il sistema, ad esempio se si cancella un thread mentre sta aggiornando dei dati da condividere con altri thread. Il metodo della cancellazione differita permette di programmare la verifica in un punto dell'esecuzione in cui il thread sia cancellabile senza problemi. Nella libreria Pthreads questi punti sono detti ***punti di cancellazione*** (*cancellation point*). Nei sistemi UNIX vengono usati i ***segnali*** per comunicare ai processi il verificarsi di determinati eventi. Tutti i segnali seguono lo stesso schema: si generano all'occorrenza di un particolare evento, il segnale viene inviato a un processo, il segnale viene gestito una volta ricevuto. Ogni segnale si può gestire in due modi. Il segnale viene gestito dal kernel attraverso il ***gestore predefinito del segnale***, oppure attraverso una funzione di ***gestione del segnale definita dall'utente*** richiamata per gestire il segnale. In ogni caso i segnali possono essere ignorati o gestiti terminando l'esecuzione del programma. L'impiego di ***gruppi di thread*** (***thread pool***) è una possibile soluzione ai problemi dovuti al tempo richiesto per la creazione del thread prima di soddisfare una richiesta e al limite di thread attivi in modo concorrente nel sistema. L'idea è quella di generare un certo numero di thread alla creazione del processo e organizzarli in un gruppo in cui attendano il lavoro che gli sarà richiesto. I vantaggi offerti sono la maggiore rapidità poichè elimina l'attesa della creazione di un nuovo thread e la limitazione del numero di thread esistenti a un certo istante di tempo. Uno dei vantaggi principali della programmazione multithread è dato dal fatto che i thread appartenenti allo stesso processo ne condividono i dati. Tuttavia in particolari circostanze, ogni thread può necessitare di una copia privata di certi dati, chiamati ***dati specifici di thread***. Un'ultima questione da affrontare in merito ai programmi multithread riguarda la comunicazione tra la libreria del kernel e la libreria per i thread, che può rendersi necessaria nel modello a due livelli e in quello molti a molti. Proprio grazie a questa forma il numero dei thread è modificabile dinamicamente per avere prestazioni migliori. Molti sistemi che implementano questi modelli collocano una struttura intermedia tra i thread del kernel e dell'utente, nota come *LWP*(*Lightweight Process*): dal punto di vista del livello utente, essa si presenta come un processore virtuale a cui l'applicazione può richiedere lo scheduling di un thread a livello utente; ciascun LWP è associato a un thread del kernel che il sistema operativo mette in esecuzione sui processori fisici. Per un'efficiente esecuzione un'applicazione può avere bisogno di un numero imprecisato di LWP. Uno dei modelli di comunicazione tra la libreria a livello utente e il kernel è conosciuto come ***attivazione dello scheduler***: il kernel fornisce all'applicazione una serie di processi virtuali, mentre l'applicazione esegue lo scheduling ei thread dell'utente sui processori virtuali disponibili. Inoltre, il kernel deve informare l'applicazione se si verificano determinati eventi, seguendo una procedura nota come ***upcall*** che sono gestite dalla libreria dei thread mediante un apposito gestore eseguito su un processore virtuale.

- Esempi di sistemi operativi (pag 163)

Thread nel sistema Windows XP

Thread di Linux