Corso di Informatica

Modulo T5

B1-Programmazione *multithreading*

Prerequisiti

- Schedulazione
- Attesa indefinita
- Lo stallo
- Tecnica round-robin

Introduzione

La programmazione concorrente consente di chiedere l'esecuzione contemporanea di più programmi. I processi corrispondenti che si generano possono essere effettivamente eseguiti in parallelo, come nel caso di sistemi multiprocessor, oppure possono essere eseguiti con un parallelismo simulato dal nucleo del sistema operativo, tramite un algoritmo detto schedulatore.

Questi concetti si applicano anche ai *thread*, come vediamo in questa Unità.

Il ciclo di vita di un *thread*

Un *thread*, analogamente ai processi, durante la sua evoluzione, passa attraverso diversi **stati** (creazione, avanzamento, sospensione, riattivazione e terminazione).

Questo percorso prende il nome di **ciclo di vita del *thread*** ed è rappresentabile con il **diagramma degli stati** mostrato a fianco.

Gli stati di un *thread*

Dal grafico osserviamo che un *thread* può essere:

- **creato**: l'operazione **New Thread()** crea un nuovo *thread*;
- **avviato**: l'operazione **start()** fa partire l'esecuzione del *thread* creato;
- **sospeso**: il *thread* viene messo temporaneamente in stato di **wait** (detto **Not Runnable**), che pone il *thread*, nell'attesa di un evento;
- **riattivato**, al *thread* viene comunicato che può riprendere l'esecuzione;
- **terminato**: l'operazione **stop()** termina l'esecuzione del *thread*.

Il diagramma degli stati è semplificato: in pratica, esistono altre operazioni che consentono la sincronizzazione fra due o più *thread*. Queste operazioni vengono descritte nello studio dei relativi linguaggi.

Le transizioni

Il passaggio da uno stato al successivo prende il nome di **transizione**. Possiamo avere:

- **Pronto → Esecuzione** (**running**): il processo ha il controllo di un processore;
- **Esecuzione → Pronto** (**ready**): il processo è pronto ad essere eseguito, ed è in attesa di essere eseguito (compito dello *scheduler*, v. seguito);
- **Esecuzione → Attesa** o *sospeso* o *bloccato* (**suspended** o **blocked**): il processo ha eseguito una chiamata di sistema ed è fermo in attesa di un evento (ad es. lettura dati da disco);
- **Attesa → Pronto** il *thread* riceve il permesso di riprendere l'avanzamento

Schedulazione

Analogamente a quanto si fa per i processi, l'esecuzione, in un certo ordine, di più *thread* su un sistema avente una sola CPU si dice **shedulazione** e l'algoritmo del sistema operativo che stabilisce l'ordine di esecuzione si dice **schedulatore**.

Programmazione *multithreading*
- ambiente multicore

Nel caso di Java, la JVM è di per sé in grado di gestire processori multipli

Se:

- la macchina dispone di più di un processore
- il sistema operativo gestisce processori multipli

i programmi *multithreaded* non dovranno essere riscritti poiché sfrutteranno automaticamente i processori in parallelo.

Detto in modo molto semplificato, un sistema si dice **multiprocessore** quando contiene fisicamente più di una CPU. Si dice invece **multicore** quando contiene una sola CPU, ma nella quale sono alloggiati più nuclei (core).
Il costo di una sola CPU dualcore è inferiore a quello di due CPU.

Programmazione *multithreading*
- ambiente multicore

Ovviamente, in ambienti multiprocessore (o in quelli *multi-core*), il parallelismo è reale poiché ogni *thread* effettivamente esegue su un processore diverso.

In questi casi (specialmente per programmi di solo calcolo matematico) il *multithreading* è particolarmente indicato.

Programmazione *multithreading*
- ambiente *uniprocessor*

In ambienti uniprocessor, l'esecuzione parallela di *thread* è simulata:

i *thread* vengono eseguiti generalmente in partizione di tempo (*time sharing*), per cui un solo *thread* alla volta può avanzare.

Il processore commuta frequentemente da un *thread* all'altro (*context switch*) e l'utente ha l'impressione di un avanzamento parallelo: tutti i *thread* sembrano avanzare contemporaneamente (**multithreading**).

Programmazione *multithreading*
- ambiente *uniprocessor*

Non sempre in ambiente monoprocessore conviene il *multithreading*.

Solitamente, l'essere *multithreading*, è una caratteristica dei sistemi operativi (per esempio **Unix**), piuttosto che dei linguaggi di programmazione. La tecnologia **Java**, tramite la *Java Virtual Machine*, ci offre uno strato d'astrazione per poter gestire il *multithreading* direttamente dal linguaggio, anche su macchine uniprocessor.

Svantaggi del *multithreading*

- Ogni *thread* usa ulteriori risorse di memoria
- *Overhead* (superlavoro) dovuto allo scheduling dei *thread*
- *Context switch* (sospensione di un *thread* in favore di un altro) frequenti occupano molto tempo.
- Tempi per la creazione, avviamento, deallocazione risorse e terminazione di un *thread*

Ad esempio, invece di creare un *thread* ogni 5 minuti per controllare la posta, è meglio crearlo una volta e metterlo in pausa per 5 minuti fra un controllo e l'altro

Priorità

In tutti i sistemi (come nella **JVM**) l'esecuzione dei *thread* avviene in base alla **priorità** di ciascuno e si sfrutta la tecnica di **schedulazione a priorità fissa**.

La CPU esegue sempre il *thread* avente priorità più alta, fino a quando non si verifichi una delle seguenti condizioni:

- il *thread* viene sospeso (ad es. un *thread* a priorità più alta diventa **Runnable**)
- il *thread* termina (cessa definitivamene lo stato di running)
- il *thread* passa in stato di **Not Runnable** (ad es. su un sistema in partizione di tempo, termina il quanto di tempo assegnato al *thread*).

Tecnica round-robin

Quando un *thread* viene interrotto, sospeso o termina, la CPU passa ad eseguire il *thread* con priorità di valore immediatamente minore.

Se due *thread* hanno la stessa priorità, la CPU opera secondo la tecnica **round-robin**.

Assunzione di progresso finito

Da quanto detto, si deduce che qualunque sistema operativo deve garantire che tutti i *thread* prima o poi vengano serviti.

Questa garanzia, che prende il nome di **assunzione di progresso finito**, a volte, può essere messa a rischio, in particolare nel caso si verifichi:

- attesa indefinita
- stallo

Attesa indefinita

Generalmente, in ogni istante è in esecuzione il *thread* con priorità massima, ma questa non può essere la regola generale:

> Un processo A, in attesa di una risorsa R, si vede privare l'assegnazione di R da parte di un processo B, con priorità maggiore di A, che interviene non appena R diventa disponibile.

Se lo schedulatore sceglie questa politica si va incontro al fenomeno detto *starvation*.

Attesa indefinita

La *starvation* (letteralmente, "*morte per fame*") consiste nel fenomeno per cui il processore, quando si libera, viene subito assegnato a *thread* a priorità alta, non diventerà mai disponibile per un *thread* a priorità più bassa, il quale quindi non avanzerà mai.

La situazione di *starvation* può essere evitata in generale utilizzando una strategia **FIFO** nell'assegnazione delle risorse.

Stallo

A volte può capitare che due o più *thread* restino in attesa di un dato evento che non si verificherà mai. Questo fenomeno, ovviamente da evitare, prende il nome di **stallo**.

Ad esempio, un *thread* A detiene la risorsa R, mentre un *thread* B detiene la risorsa S. Per poter avanzare, A richiede S, mentre B richiede R

Anche lo stallo può essere prevenuto attraverso apposite tecniche, illustrate nei relativi corsi di sistemi operativi.

Argomenti

- Il ciclo di vita di un *thread*
- Gli stati di un *thread*
- Le transizioni
- Schedulazione
- Programmazione multithreading
  - ambiente *multicore*
  - ambiente *uniprocessor*
- Svantaggi del *multithreading*
- Priorità
- Tecnica round-robin
- Assunzione di progresso finito
- Attesa indefinita
- Stallo

Altre fonti di informazione

- P.Camagni, R.Nicolassy – Java: interfacce grafiche e programmazione concorrente, ed. Hoepli Informatica
- **http://java**.sun.com/docs/books/tutorial/essential/threads/index.html