Laboratorio di Reti

Federico Matteoni

1 Thread

Processo Istanza di un programma in esecuzione

Thread **Flusso di esecuzione** all'interno di un processo ⇒ Ogni processo ha almeno un thread.
I **thread condividono le risorse di un processo**.
Possono essere eseguiti sia su single-core (es. interleaving, time-sharing...) che su multicore (più flussi di esecuzione in parallelo)

Multitasking Si può riferire a

Processi, controllato esclusivamente dal S.O.

Thread, controllato in parte dal programmatore

Contesto di un processo Insieme delle informazioni necessarie per ristabilire esattamente lo stato in cui si trova il S.O. nel momento in cui si interrompe l'esecuzione di un processo per passare ad un altro: registri del processore, memoria del processo...

Perché? Per gestire più funzionalità contemporaneamente, come gestire input, visualizzare a schermo, monitorare la rete ed eseguire calcoli.
Esempi noti: browser web, videogame multiplayer. Si creano **più componenti interagenti** in modo da:

Usare meglio le risorse

Migliorare le performance per applicazioni che richiedono grossi calcoli: si dividono i task per eseguirli in parallelo.

Anche problemi: difficile debugging e manutenzione, sincronizzazione, deadlocks...

In Java Il main thread, invocato dalla JVM all'esecuzione del programma, può attivare altri thread. La JVM attiva automaticamente altri thread come il garbage collector.
Un thread è un oggetto. Per creare un thread si definisce un task che implementi l'interfaccia `Runnable` e si crea un thread passandogli l'istanza del task creato. Altrimenti si può estendere la classe `java.lang.Thread`.

Runnable Appartiene a `java.language`, contiene solo la firma del metodo `void run()`. Un oggetto che la implementa è un frammento di codice che può essere eseguito in un thread.

Stati

Created/New: subito dopo l'istruzione `new`, variabili allocate ed inizializzate. Thread in attesa di passare in esecuzione

Runnable/Running: thread in esecuzione o in attesa per ottenere la CPU (Java non separa i due stati).

Not Runnable (**Blocked/Waiting**): thread non può essere messo in esecuzione, può accadere quando attende un'operazione I/O o ha invocato metodi come `sleep()` oppure `wait()`.

Dead: termine naturale o dopo l'invocazione di `stop()` da parte di altri thread (deprecato).

1.1 Threadpool

Perché? In caso di task leggeri molto frequenti risulta impraticabile attivare ulteriori thread. Diventa quindi utile definire un limite massimo di thread che possono essere attivati contemporaneamente, così da sfruttare meglio i processori, evitare troppi thread in competizione e diminuire i costi di attivazione/terminazione dei thread.

Threadpool Struttura dati la cui dimensione massima può essere prefissata, contenente riferimenti ad un insieme di thread. I thread possono essere riutilizzati: la sottomissione di un task al threadpool è **disaccoppiata** dall'esecuzione del thread. L'esecuzione può essere ritardata se non vi sono risorse disponibili.
La **politica di gestione dei thread** stabilisce quando i thread vengono attivati (al momento della creazione del pool, on demand, all'arrivo di un nuovo task...) e quando è opportuno terminare l'esecuzione di un thread.
Il threadpool, quindi, al momento della sottomissione di un task può:

Usare un thread attivato in precedenza e al momento inattivo

Creare un nuovo thread

Memorizzare il task in una coda, in attesa

Respingere la richiesta

Callable Classe per definire un task che può restituire un risultato e sollevare eccezioni

Future Rappresenta il risultato di una computazione asincrona. Definisce metodi per controllare se la computazione è terminata, attendere la terminazione oppure cancellarla. Viene implementata nella classe `FutureTask`.

2 Monitor

Lock Implicite Ogni oggetto ha associate una lock implicita ed una coda. La lock si acquisisce mediante metodi o blocchi di codice `synchronized`. Quando questo viene invocato:

Se nessun metodo `synchronized` della classe è in esecuzione, l'oggetto viene bloccato (la lock viene acquisita ed il metodo viene eseguito.

Se l'oggetto è già bloccato, il thread viene sospeso nella coda associata finché la lock non viene rilasciata.

Notare che la lock è **associata all'istanza della classe**, non alla classe. Metodi su istanze (oggetti) diverse della stessa classe possono essere eseguiti concorrentemente.
I costruttori non possono essere dichiarati `synchronized` (errore di compilazione), perché solo il thread che crea l'oggetto deve poterci accedere mentre l'oggetto viene creato.
Non ha senso specificare `synchronized` nelle interfacce poiché è riferito all'implementazione.
Inoltre il `synchronized` non viene ereditato.

Monitor Meccanismo linguistico ad alto livello per la sincronizzazione. classe di oggetti utilizzabili concorrentemente in modo safe. La risorsa è un oggetto passivo, le sue operazioni vengono invocate da entità attive (thread). La sincronizzazione sullo stato della risorsa è garantita esplicitamente: mutua esclusione sulla struttura garantita dalla lock implicita (un solo thread per volta è all'interno del monitor), meccanismi per sospensione/risveglio sullo stato dell'oggetto condiviso simili a variabili di condizione (`wait/notify`)
Il monitor è quindi un **oggetto** con un insieme di metodi `synchronized` che incapsula lo stato di una risorsa condivisa. Ha due code gestite implicitamente: entry set (thread in attesa di acquisire la lock) e wait set (thread che hanno eseguito una wait e attendono una notify)

Wait Sospende il thread in attesa che si verifichi una condizione (opzionalmente per un tempo massimo). Rilascia la lock (a differenza di `sleep` e `yield`).

Notify Sveglia ad un thread in attesa il verificarsi di una certa condizione (`notifyAll` sveglia tutti i thread in attesa)

Deadlock Due o più thread bloccati per sempre in attesa uno dell'altro

Starvation Thread ha difficoltà ad accedere ad una risorsa condivisa e quindi difficoltà a procedere. In generale task "greedy" che invocano spesso metodi lunghi obbligando gli altri ad aspettare

Livelock Programma che genera una sequenza ciclica di operazioni inutili ai fini dell'effettivo avanzamento della computazione.