06 - Thread Java POSIX

Thread e Multithreading

- Per risolvere i problemi di efficienza del modello a processi pesanti (modello ad ambiente locale) e' possibile far ricorso al modello a ambiente globale, a processi leggeri (o thread)
- **Thread**: singolo flusso sequenziale di esecuzione all'interno di un processo
- **Multithreading**: esecuzione concorrente (in parallelo o interleaved) di diversi thread nel contesto di un piccolo processo

Thread

Un thread e' un singolo flusso sequenziale di controllo all'interno di un processo. Un thread (o processo leggero) e' un'unita' di esecuzione che condivide codice e dati con altri thread ad esso associati

Un thread

- NON ha spazio di memoria riservato per dati e heap: tutti i thread sono appartenenti allo stesso processo condividono loro lo stesso spazio di indirizzamento
- ha stack e program counter privati

Modello ad ambiente globale

Caratteristiche del modello computazionale multithreaded (modello ad ambiente globale):

- I thread non hanno uno spazio di indirizzamento riservato: tutti i thread di un processo condividono lo stesso spazio di indirizzamento --> possibilita' di definire dati thread-local, sia in Java che in POSIX
- I thread hanno eecution stack e program counter privati
- La comunicazione fra thread puo' avvenire direttamente, tramite la condivisione di aree di memoria --> necessita' di meccanismi di sincronizzazione

Nel modello ad ambiente globale il termine "processo" non identifica piu' un singolo flusso di esecuzione di un programma ma invece il contesto di esecuzione di piu' thread, con tutti i dati che possono essere condivisi da questi ultimi.

Vantaggi e svantaggi del Multithreading

- Context switch e creazione piu' leggeri
- Minore occupazione di risorse
- Comunicazione tra thread piu' efficiente
- I thread permettono di sfruttare al massimo le architetture hardware con CPU multiple (prestazioni)

- Problemi di sincronizzazione piu' rilevati e frequenti
- Memoria virtuale condivisa limitata alla memoria riservata dal SO per un singolo processo
- Maggiore difficolta' nello sviluppo e nel debugging!!!

In presenza di applicazioni che richiedono la condivisione di informazioni si puo' ottenere un notevole incremento di efficienza e conseguente miglioramento delle prestazioni.

Multithreading: esempi di utilizzo

Applicazioni web

- Client-side: i browser web usano i thread per creare piu' connessioni simultanee verso server diversi, per scaricare diverse risorse in parallelo da uno stesso server, e per gestire il processo di visualizzazione di una pagina web. Tale concorrenza permette di ridurre il tempo di risposta alle richieste dell'utente e quindi aumentare la qualita' dell'esperienza percepita
- Server-side: i server web usano i thread per gestire le richieste dei client. I thread possono condividere in modo efficiente sia la cache in cui vengono memorizzate le risorse piu' utilizzate recentemente che le informazioni contestuali .

Altri esempi:

- Implementazione di algoritmi parallelizzabili, non-blocking I/O, timer multipli, task indipendenti, responsive User Interface (UI), ecc...

Multithreading in Java

Il linguaggio Java supporta nativamente il multithreading. Ogni esecuzione della JVM da origine a un unico processo, e tutto quello che viene mandato in esecuzione dalla macchina virtuale da origina a un thread. Un thread e' un oggetto particolare al quale si richiede un servizio (chiamato `start()`) corrispondente al lancio di un'attivita', di un thread, ma che non si aspetta che il servizio termini: esso procede in concorrenza a che lo ha lanciato.

A livello di linguaggio, i thread sono rappresentati da istanze della classe **Thread**.

Per creare un nuovo thread ci sono due metodi:

1. **Istanziare Thread** passando come parametro un oggetto ottenuto implementando **l'interfaccia Runnable**
2. Estendere direttamente la classe Thread

In entrambi i casi il programmatore deve implementare una classe che definisca il metodo `run()`, contenente il codice del thread da mandare in esecuzione.

La classe **Thread** e' una classe (non astratta) attraverso la quale si accede a tutte le principali funzionalita' per la gestione dei thread.

L'interfaccia **Runnable** definisce il solo metodo run(), identico a quello della classe Thread (che infatti implementa l'interfaccia Runnable). L'implementazione della interfaccia Runnable consente alle istanza di una classe non derivata da Thread di essere eseguite come un thread (purche' venga agganciata a un oggetto di tipo Thread)

Un programma Java termina quando termina l'ultimo dei thread in esecuzione (contrariamente a quanto accade in POSIX threads e nei thread Windows)

Multithreading in Java - Metodo 1 - Implementazione interfaccia Runnable

Procedimento;

1. Definire una classe che implementi l'interfaccia Runnable, quindi definendone il metodo run()
2. Creare un'istanza di tale classe
3. Creare un'istanza della classe Thread, passando al costruttore un reference all'oggetto Runnable creato precedentemente
4. Invocare il metodo start() sull'oggetto Thread appena creato. Cio' produrra' l'esecuzione, in un thread separato, del metodo run() dell'istanza di Runnable passata come argomento al costruttore di Thread.

Multithreading in Java - Metodo 2 - Sottoclasse di Thread

Procedimento:

1. definire una sottoclasse della classe Thread, facendo un opportuno override del metodo run()
2. creare un'istanza di tale sottoclasse
3. invocare il metodo start() su tale istanza, la quale, a sua volta, richiamera' il metodo run() in un thread separato

Multithreading in Java - confronto tra i due metodi

Metodo 1:

- (+) Maggiore flessibilita' derivante dal poter essere sottoclasse di qualsiasi altra classe (utoile per ovviare all'impossibilita' di avere ereditarieta' multipla in java).
- (-) Modalita' leggermente piu' macchinosa

Metodo 2:

- (+) Modalita' piu' immediata e semplice
- (-) Scarsa flessibilita' derivante dalla necessita' di ereditare dalla classe Thread, che impedisce l'ereditarieta' da altre classi

Java - ciclo di vita di un thread

Durante il suo ciclo di vita, un thread puo' essere in uno dei seguenti stati:

- **New Thread (creato)**: (subito dopo l'istruzione new) ole variabili sono state allocate e inizializzate. Il thread e' in attesa di passare allo stato Runnable, transizione che verra' effettuata in seguito a una chiamata al metodo start.
- **Runnable**: il thread e' in esecuzione, oppure pronto per l'esecuzione e in coda d'attesa per ottenere l'utilizzo della CPU
- **Not Runnable (Bloccato)**: il thread non puo' essere messo in esecuzione dallo scheduler della JVM. I thread entrano in questo stato quando sono in attesa di un'operazione di I/O o sospesi da una primitiva di sincronizzazione come sleep() o wait().
- **Dead**: il thread ha terminato la sua esecuzione in seguito alla terminazione del flusso di istruzioni del metodo run() o alla chiamata del metodo stop() da parte di un altro thread

Java - Metodi per il controllo di un thread

- `start()`: fa partire l'esecuzione di un thread. La macchina virtuale Java invoca il metodo run() del thread appena creato
- `stop()`: forza la terminazione dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente liberate (lock inclusi), come effetto della propagazione dell'eccezione ThreadDeath.
- `suspend()`: blocca l'esecuzione di un thread in attesa di una successiva operazione di resume. Non libera le risorse (neanche lock) impegnate dal thread (possibilita' di deadlock)
- `resume()`: riprende l'esecuzione di un thread precedentemente sospeso. Se il thread riattivato ha una priorita' maggiore di quello corrente in esecuzione, avra' subito accesso alla CPU, altrimenti andra' in coda l'attesa.
- `sleep(long t)`: blocca per un tempo specifico (t) l'esecuzione di un thread. Nessun lock in possesso del thread viene rilasciato
- `join()`: blocca il thread chiamante in attesa della terminazione del thread di cui si invoca il metodo. Anche con timeout
- `yield()`: sospende l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread in coda d'attesa

Java - il problema di `stop()` e `suspend()`

stop() e suspend() rappresentano azioni "brutali" sul ciclo di vita di un thread --> rischio di determinare situazioni di deadlock o di inconsistenze:

- se il thread sospeso aveva acquisito una risorsa in maniera esclusiva, tale risorsa rimane bloccata e non e' utilizzabile da altri perche' il thread sospeso non ha avuto modo di rilasciare il lock su di essa.

- se il thread interrotto stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera atomica, l'interruzione puo' condurre a uno stato inconsistente del Sistema

Java - scheduling dei thread

La macchina virtuale Java ha un algoritmo di scheduling basato su livelli di priorita' statici fissati a priori (Fixed Priority Scheduling):

- il thread da mandare in esecuzione viene scelto fra quelli nello stato runnable, tipicamente quello con priorita' piu' alta
- il livello di priorita' non viene mai cambiato dalla JVM (statico)
- le specifiche ufficiali della JVM non stabiliscono come venga scelto il thread da mandare in esecuzione tra quelli disponibili (es. caso di piu' thread con lo stesso livello di priorita'). La politica dipende dalla specifica implementazione della JVM (FIFO, Round-Robin, etc...)

Di conseguenza, non c'e' nessuna garanzia che si implementata una gestione in time-slicing (Round-Robin).

Il thread messo in esecuzione dallo scheduler viene interrotto soltanto se si verifica uno dei seguenti eventi:

- Il thread termina la sua esecuzione oppure chiama un metodo che lo fa uscire dallo stato runnable (ad esempio, Yield())
- Un thread con priorita' piu' alta diventa disponibile per l'esecuzione (ovverosia entra nello stato runnable)
- Il thread termina il proprio quanto di tempo (solo nel caso di scheduling Round-Robin)
- Il processore riceve un Interrupt Hardware

Java - priorita' di Scheduling

- Non si devono progettare applicazioni Java assumendo che il thread a priorita' piu' elevata sara' sempre quello in esecuzione, se nello stato runnable
- Il meccanismo di assegnazione di priorita' diverse ai bari thread e' stato pensato solo per consentire agli sviluppatori di dare suggerimenti alla JVM per migliorare l'efficienza di un programma
- Alla fine, e' sempre la JVM ad avere l'ultima parola nello scheduling. Diverse JVM possono avere comportamenti diversi
- L'assegnazione di diverse priorita' ai thread non e' un sostituto delle primitive di sincronizzazione

Sincronizzazione di thread

- Differenti thread condividono lo stesso spazio di memoria (heap)
    - e' possibile che piu' thread accedano contemporaneamente a uno stesso oggetto, invocando un metodo che modifica lo stato dell'oggetto

    - stato finale dell'oggetto sara' funzione dell'ordine con cui i thread accedono ai dati
- Servono meccanismi di sincronizzazione
- In Java, la sincronizzazione e' implementata a livello di linguaggio (modificatore synchronized) e attraverso primitive di sincronizzazione di basso (`wait()`, `notify()`, `notifyAll()` e di alto livello (concurrency utilities)
- Anche lo standard POSIX definisce funzioni e costrutti per la costruzione e la manipolazione di mutex e altri meccanismi di sincronizzazione

Accesso esclusivo

Per evitare che thread diversi interferiscano durante l'accesso ad oggetti condivisi si possono imporre accessi esclusivi in modo molto facile in Java. JVM supporta la definizione di lock sui singoli oggetti tramite la keywoord synchronized.

Synchronized puo' essere definita:

- **su metodo**
- su singolo blocco di codice

Synchronized

In pratica:

- a ogni oggetto Java e' automaticamente associato un unico lock
- quando un thread vuole accedere ad u metodo/blocco synchronized, si deve acquisire il lock dell'oggetto (impedendo cosi' l'accesso ad ogni altro thread)
- lock viene automaticamente rilasciato quando il thread esce dal metodo/blocco synchronized (o se viene interrotto da un'eccezione)
- thread che non riesce ad acquisire un lock rimane sospeso sulla richiesta della risorsa fino a che il lock non e' disponibile
- Ad ogni oggetto viene assegnato **un solo lock a livello di oggetto** (non di classe ne di metodo in Java)
    - due thread non possono accedere contemporaneamente a due metodi/blocchi synchronized diversi di uno stesso oggetto
- Tuttavia altri thread sono liberi di accedere a metodi/blocchi **non synchronized** associati allo stesso oggetto.

Esistono due situazioni in cui synchronized non e' sufficiente per impedire accessi concorrenti. Supponiamo che un metodo synchronized sia l'unico modo per variare lo stato di un oggetto. Che cosa accade se il thread che ha acquisito il lock si blocca all'interno del metodo stesso in attesa di un cambiamento di stato?

Soluzione tramite uso di wait()

Thread che invoca wait()

- si blocca in attesa che un altro thread invochi notify() o notifyAll() per quell'oggetto
- deve essere in possesso del lock sull'oggetto
- al momento della invocazione di wait() il thread rilascia il lock

notifyAll()

- notify() - il thread che la invoca risveglia uno dei thread in attesa, scelto arbitrariamente
- notifyAll() - il thread che la invoca
  - **risveglia tutti i thread in attesa**: essi competeranno per l'accesso all'oggetto

notifyAll() e' preferibile (puo' essere necessaria) se piu' thread possono essere in attesa

Alcune regole empiriche

1. Se due o piu' thread possono modificare lo stato di un oggetto, e' necessario dichiarare synchronized i metodi di accesso a tale stato
2. Se deve attendere la variazione dello stato di un oggetto, thread deve invocare wait()
3. Ogni volta che un metodo attua una variazione dello stato di un oggetto, esse deve invocare notifyAll()
4. E' necessario verificare che ad ogni chiamata a wait() corrisponda una chiamata a notifyAll()

Thread safety

- Qualsiasi programma in cui piu' thread accedono a una stessa risorsa (con almeno un'operazione di scrittura), senza un'adeguata sincronizzazione, e' **bacato** . Questo tipo di bug, secondo il quale l'output di una porzione di codice dipende dall'ordine in cui i thread accedono le risorse condivise, e' tipicamente molto difficile da individuare e correggere, in quanto potrebbe presentarsi solo in particolari condizioni.
- L'unico modo di risolvere il problema della thread safety e' quello di utilizzare opportunamente le primitive di sincronizzazione
- Una porzione di codice e' thread-safe se si comporta correttamente quando viene usata da piu' thread, indipendentemente da loro ordine di esecuzione, senza alcuna sincronizzazione o cordinazione da parte del codice chiamante.
- E' molto piu' facile scrivere codice thread safe, pianificando un adeguato uso delle primitive di sincronizzazione al momento del progetto, che modificarlo successivamente per renderlo tale.

Java - Daemon Thread

- I thread di Java possono essere di due tipi: **user thread** e **demo thread**
- L'unica differenza tra le due tipologie di thread sta nel fatto che la virtual machine di Java termina l'esecuzione di un daemon thread quando termina l'ultimo user thread.
- I daemon thread svolgono servizi per gli user thread, e spesso restano in esecuzione per tutta la durata di una sessione della virtual machine (ad esempio, il garbage collector).
- Di default, un thread assume lo stato del thread che lo crea. E' possibile verificare e modificare lo stato di un thread con metodi `isDaemon()` e `setDaemon()`, ma solo prima di mandarlo in esecuzione

Multithreading in POSIX threads

Per realizzare programmi multithreaded in UNIX, si deve fare riferimento alla API standard POSIX threads (o Pthreads):

- Va incluso il file header specifico:
  - ```
    #include <pthread.h>
    ```
- Primitiva di sistema per la creazione (spawning) di un nuovo thread:
  - ```
    int pthread_create (pthread_t * t,
                const pthread_attr_t * attr,
                void * (*Start_func)(void*),
                void *arg);
    ```
- Primitiva di sistema per effettuare il join con un thread:
  - ```
    int pthread_join (pthread_t t, void **retval);
    ```
- Primitiva di sistema per lo spawning di un nuovo thread:
  - ```
    int pthread_create (pthread_t * t,
                const pthread_attr_t * attr,
                void * (*Start_func)(void*),
                void *arg);
    ```
- Restituisce il valore 0 in caso di successo, o un valore positivo in caso di errore
- il thread t viene creato con gli attributi specifici in attr ed esegue la funzione start_func, con argomanto arg
- Attributi specificano diversi comportamenti: detached mode, stack size, scheduling priority, ecc.
- Primitiva di sistema per effettuare il join con un thread:
  - ```
    int pthread_join (pthread_t t, void **retval);
    ```
- Restituisce valore 0 in caso di successo, o un valore positivo in caso di errore
    ```

- il thread che lancia la chiamata di join si mette in attesa per la terminazione del thread
- Al ritorno della chiamata, retval conterra' il valore di ritorno della funzione eseguita dal thread t
    - retval puo' essere NULL
- Se un thread viene creato in modalita' noin-detached (default in POSIX), allora la join e' necessaria per evitare di lasciare zombie, e quindi un memory leak

Thread detached in POSIX

Anche Pthreads supporta la creazione di thread di tipo "demone", che pero' prendono il nome di thread detached. La differenza fra le due tipologie di thread sta nella gestione della memoria e nalla possibilita' di effettuare il join con un altro thread.
Un thread non-detached permette ad altri thred di sincronizzarsi con la sua terminazione, tramite la primitiva pthread_join(). E' necessario effettuare la join per liberare la memoria associata ai thread non-detached che abbiano finito la loro esecuzione (evitare di lasciare zombie).
Un thread detached non e' joinable, ma il sistema effettua automaticamente il release della memoria ad esso associata quando questo termina la sua esecuzione (non c'e' il problema di thread zombie)