Terminazione di un Thread

Nella scorsa lezione abbiamo visto come terminare un thread: il metodo duale di start() è il metodo **stop()**, che ci permette di terminare l'esecuzione di un thread, anche se non è consigliato; questo perchè il thread potrebbe essere interrotto mentre sta lavorando su un oggetto, senza terminare le operazioni che stava eseguendo, lasciando l'oggetto in uno stato inconsistente.

Un altro modo è portare il thread a terminazione, e questo viene ottenuto facendo terminare le operazioni contenute all'interno del metodo run(). Questo metodo è assolutamente da preferire rispetto all'operazione di stop().

All'interno del corpo di un thread non dovrebbe **mai essere presente un ciclo infinito**, questo perchè in questo caso saremmo costretti a terminare forzatamente il thread. L'alternativa al ciclo infinito è usare un ciclo condizionato da un flag che possiamo controllare, in modo da portare il thread in uno stato di terminazione:

```
while (!halt){
    ...
}
```

Non è necessario prevedere questo flag esplicitamente perchè la classe **Thread** ne prevede già uno! Questo flag presente all'interno delle classi (ogni oggetto che istanziamo) è controllato in scrittura e lettura, ed è gestito attraverso i metodi:

- **interrupt():** impostiamo il flag a **true** , in modo da interrompere il processo.
- **isInterrupted():** quando viene invocato possiamo verificare se lo stato è interrotto oppure no.

Usiamo quindi il flag di default presente nella classe Thread che possiamo gestire grazie ai metodi appena visit; scriveremo il codice del thread nel seguente modo:

- **Thread 1:** `thread2.interrupt();` // interrompiamo il thread -> flag=true
- **Thread 2:** `while (!isInterrupted()){ ... }` // ciclo sul flag -> esco dal ciclo quando flag == true
- **Thread 2:** `try{ sleep(); }catch(InterruptedException e){ ... }` //

Nel terzo esempio, il thread 2 sta eseguendo un'operazione (sleep()) che è un metodo sospensivo; se il thread è sospeso ed arriva un segnale di interruzione come quello prodotto da t2.interrupt() il thread si risveglia (in modo anomalo, perchè svegliato da un segnale di interruzione) ed in questo caso invece di eseguire le istruzioni che eseguirebbe normalmente, esegue un'eccezione, di tipo **InterruptedExceptio**.

Capiamo quindi che questo meccanismo ci permette di interrompere anche dei thread in sleep.

Deamon Thread

In java i thread possono essere **Thread Utente** o **Thread Daemon**; I thread Daemon sono thread **a servizio** degli altri thread, e solitamente presentano **cicli infiniti**. Un thread può essere caratterizzato come Daemon invocando, all'atto della sua creazione, il metodo **setDaemon(true)**. Un thread Daemon che crea altri thread, li crea automaticamente Daemon; quindi il thread figlio eredita dal padre il tipo di thread.

I thread Daemon sono importanti per un aspetto che riguarda la **terminazione delle applicazioni.:** in Java un'applicazione termina quando non ci sono più **thread utente in esecuzione.** Nel caso di programmi sequenziali (programmi scritti finora) che presentano un main(), vengono eseguiti **da un unico thread**, contenuto all'interno del processo; questo unico thread ha come **corpo, il main()**.

In questo caso nel momento in cui viene completata l'esecuzione del main il processo termina. Nel caso in cui abbiamo un programma che a tempo di esecuzione da vita a più thread, il processo che esegue questi thread, terminerà solo quando **tutti i thread che esegue saranno terminati.** Se però rimane in vita un **thread Daemon e tutti gli altri sono terminati**, il processo termina comunque; i thread daemon non devono terminare affinchè il processo termini.

Questo perchè i thread daemon sono di supporto, quindi i thread daemon possono rimanere in vita.

Scheduling

Questi appunti verranno presi in modo vago, proprio perchè questi argomenti sono già stati abbondantemente analzzati nella parte di Architettura dei calcolatori; link

Nella scorsa lezione abbiamo visto che la CPU viene assegnata alle attività da un'entita, chiamata scheduler, con una logica ben precisa, usando i meccanismi della Preemption per sottrarre la CPU a dei processi in modo che possa essere assegnata ad altri processi.

Scheduling basato sulla priorità

La priorità è un intero compreso tra **MIN_PRIORITY e MAX_PRIORITY** e può essere assegnata ad un thread invocando il metodo **setPriority(int)**; se non specifichiamo la priorità del thread, questo la eredita dal thread padre. Il thread associato al main è di "media priorità"; solitamente le priorità vanno da 1 a 10, quindi il main è a priorità 5.

Scheduling a priorità fissa

Il meccanismo realmente usato in Java è un meccanismo basato sulla priorità in modo da preferire dei thread piuttosto che altri, quindi un thread a priorità più alta viene selezionato maggiormente. Viene usato il concetto di **probabilità**, quindi non è certo che un thread a piorità più alta venga schedulato piu spesso.

Per garantire che anche i thread a più bassa priorità vengano eseguiti utilizziamo un **meccanismo probabilistico**.

Per quanto riguarda i thread alla stessa priorità viene usato un meccanismo basato sul **Round Robin**; se è stato selezionato un thread con priorità x, successivamente verrà selezionato un altro thread a priorità x.

Metodo Yield

Questo metodo può essere usato nel corpo di un thread e quando il thread lo usa è perchè vuole cedere volontariamente il controllo ad un altro thread. Lo si usa spesso nel momento in cui lo scheduler del processore **non prevede preemption**, e quindi è il thread a cedere il controllo **volontariamente**.

```
public class SelfishRunner extends Thread{
    ...
    public void run(){
        while(tick < 4000){
```

```
            tick ++;
            if((tick % 500) == 0){
                Sout.println("Thread #" + num + ", tick = " + tick);
            }

            yield();
            ...
        }
    }
}
```

Thread Groups

Un altro strumento utile nella programmazione dei thread java è il meccanismo dei gruppi: i thread possono essere **raggruppati ed identificati da un gruppo**; operare su un gruppo è vantaggioso quando vogliamo che un'operazione venga eseguita su un insieme di thread, e non su un singolo thread.

Possiamo ad esempio interrompere l'esecuzione di tanti thread inviando un segnale di interruzione attraverso il metodo **interrupt()** su un gruppo, invece che su un singolo thread.

Come vengono gestiti i gruppi?

Un gruppo è all'atto pratico un oggetto, e possiamo crearlo a partire dalla classe **ThreadGroup**: `ThreadGroup aTG = new  ThreadGroup("esempio");` dopo aver creato il gruppo assegnamo il thread al gruppo nel momento della sua creazione: `Thread aT = new Thread(aTG, aRunnable, "thread_name");`

Ci basta quindi creare prima il gruppo e poi creare il thread; nel momento in cui creiamo il thread passiamo come parametro il gruppo.

Possiamo inoltre creare gruppi in modo gerarchico:

E' sufficiente utilizzare opportunamente il costruttore di threadGroup: `ThreadGroup tg = new ThreadGroup(ThreadGroupParent, String name);` se ho già creato un gruppo, in fase di creazione del nuovo gruppo posso passare il riferimento al gruppo precedente, in modo da creare una gerarchia.

La gerarchia serve per poter operare in maniera differenziata: se invio un segnale di interrupt al gruppo main, questo segnale viene invato a **tutti i gruppi**, mentre se ad esempio lo invio al gruppo 2 (vedi foto) il segnale viene inviato solo al gruppo 2 e al gruppo 3.

Progettare classi per la concorrenza - problemi

E' molto semplice scrivere un programma java orientato alla concorrenza commettendo molti errori, quindi è meglio seguire delle linee guida:

Proprietà principali di un programma concorrente

- **Sicurezza** - **Safety** : integrità dei dati che caratterizzano lo stato delle nostre applicazioni. E' possibile che attività in esecuzione concorrente abbiano uno stato inconsistente (vedi interrupt()).
- **Liveness:** Dobbiamo fare in modo che la nostra applicazione concorrente sfrutti adeguatamente la concorrenza, e che i thread abbiano chanche di eseguire (dobbiamo evitare di sospenderli il più possibile);
- **Efficienz:** per operare a regime di concorrenza abbiamo necessità di usare dei meccanismi che spesso introducono dell'inefficienza; dobbiamo quindi fare attenzione a come utilizziamo questi meccanismi.

Esempio di programma non-safe

🏁 00:50 2020-10-22

```
class EvenNumbers{
  private int n = 0;
  public int next(){
    ++n;
    ++n;
    return n;
  }
}

class MyThread extends Thread{
  private EvenNumbers en;
  public MyThread(EvenNumbers e){
    this.en = e;
  }

  public void run(){
    Sout.println(en.nex());
  }
}
```

In questo esempio consideriamo una classe caratterizzata da un variabile di istanza e da un metodo next(); questa classe è pensata per mettere a disposizione un metodo, che ogni volta che viene invocato restituisce un numero pari.

Se pensiamo ad un contesto sequenziale, questa classe funziona normalmente; ciò non avviene in un contesto concorrente.

Ci aspettiamo che ogni volta che invochiamo next() sia restituito un metodo pari; partiamo da 0 ed abbiamo un incremento ad 1 e poi 2, infine viene ritornato n (2, poi 4, poi 6 ...).

Il problema

Immaginiamo che questa classe venga usata da **più thread**; cosa accade nel momento in cui più istanze della classe MyThread sono mandate in esecuzione, e queste istanze condividono lo stesso oggetto di tipo EvenNumbers?

O meglio, la domanda giusta è: **il comportamento che ci aspettiamo dal metodo next(), sarà soddisfatto?**

Quello che accade nel momento in cui mandiamo in esecuzione il programma è:

Quello che potrebbe accadere, è che il primo thread viene schedulato, esegue la sua prima istruzione (++n) ma subito dopo viene schedulato anche il secondo thread, che anch'egli esegue la sua istruzione (++n); a questo punto il primo thread aggiorna nuovamente n facendo ++n, e ritorna il valore.

Il problema è che adesso il valore ritornato non è pari, ma è dispari (3)!

Dovuta precisazione

Non dobbiamo pensare al fatto che il codice del programma sia scritto male (perchè per risolvere il problema ci basterebbe fare n+=2 invece di due operazioni), ma dobbiamo pensare a come risolvere il problema generale.

Dobbiamo quindi evitare che mentre il thread a sta eseguendo le sue istruzioni, un altro thread possa fare altrettanto. (problema della sezione critica)

Progettare le classi per la concorrenza - soluzioni

Per risolvere questi problemi useremo due Patterns.

- **Immutabilità** (pattern): vogliamo evitare violazioni di integrità dei dati, evitando che i dati possano essere modificati
- **Locking** (pattern) Vogliamo evitare violazioni di integrità garantendo un accesso esclusivo
- **State Dependency** : coordinamento dei thread in funzione allo stato che gli oggetti condivisi assumono nel tempo.

Immutabilità

E' abbastanza intuitivo pensare che per risolvere il problema una soluzione sia evitare la possibilità di modificare la variabile di istanza di una classe; questo non sempre è possibile.

Ci sono dei casi in cui le classi possonoe ssere progettate in modo da evitare il problema visto prima, quindi progettiamo una classe **senza prevedere delle variabili di istanza**, o progettiamo delle classi che presentano delle variabili di istanza ma queste non possono essere modificate (static final).

Stateless Objects

```
class StatelessAdder{
  int addOne(int i){return i+1};
  int addTwo(int i){return i+2};
}
```

I metodi della classe operano sui parametri e non sulle variabili di istanza; non abbiamo nessun problema in questo caso, questo perchè non ci sono oggetti condivisi tra thread diversi.

Oggetti con stato immutabile - Pattern

```
class ImmutableAdder{
  private final int offset;

  ImmutableAdder(int x){
    offset = x
  }

  int add(int i){
    return i + offset;
  }
}
```

Con una variabile dichiarata **final**, possiamo assegnare un valore in fase di costruzione, ma questo non può più essere modificato.

Una classe così progettata, non è una classe di cui dobbiamo preoccuparci, nel momento in cui è usata in un contesto concorrente.

In java esistono diverse classi scritte in questo modo:

- **java.lang.String**
- **java.lang.Integer**
- **java.lang.Long**

Infatti, quando lavoriamo su un oggetto String in java, e lo modifichiamo, non stiamo effettivamente modificando l'oggetto, ma ne stiamo creando uno nuovo. Lo stesso succede con le classi wrapper (e.g. Integer).

Immutabilità parziale

Ovvero un oggetto avente una parte dello stato mutabile ed una parte non mutabile:

```
class FixedList{
  private final FixedList next;

  FixedList(FixedList next){
    this.next = next;
  }
  FixedList Successor(){
    return next;
  }

  private Object elem = null;
  synchronized Object get(){
    return elem;
  }
  synchronized void set(Object x){
    this.elem = x;
  }
}
```

In questo caso abbiamo un nodo di una lista concatenata: prevediamo la definizione di una classe che rappresenta il nodo che avrà come successivo un nodo che indichiamo con next.

elem viene dichiarato normalmente (senza final), mentre next viene dichiarato **final**; il fatto che questi due oggetti siano dichiarati in questo modo, abbiamo una **parte immutabile (next) ed una parte mutabile (elem)**.

Locking ed esclusione - Pattern

Questa soluzione mira a garantire l'esecuzione atomica, o in mutua esclusione, delle classi di oggetti che i thread potrebbero eseguire.

Il locking è un **meccanismo di accettazione delle richieste di invocazione**; associare un **lock** associato ad un oggetto consente di abilitare o inibire l'accesso a quell'oggetto; in particolare in Java ogni ogetto possiede un Lock, e quando questo lock viene acquisito da un thread, il thread ha l'esclusiva su quell'oggetto. Questo vuol dire che quando un thread ha l'esclusiva su un oggetto, può operare senza interferenze.

Ovviamente, se altri thread vogliono acquisire il lock sull'oggetto non potranno fare finchè il lock è posseduto da un altro thread.

Operare sui lock con Java

In java per operare sul lock, quindi gestire l'acquisizione del lock ed il suo rilascio, viene usato un meccanismo **dichiarativo**, basato sull'impiego della parola riservata **synchronized**.

Synchronized è usata per dichiarare un metodo, e serve a dire che quando quel metodo sarà invocato da un thread, il thread che lo invoca tenterà di acquisire il lock dell'oggetto; se il lock è libero viene acquisito dal thread, e potrà eseguire le istruzioni del metodo.

Quando l'esecuzione del metodo termina, il lock precedentemente acquisito dal thread, viene **rilasciato** (tutto automaticamente).

Quando un thread invoca un metodo synchronized di un oggetto, non solo non è possibile invocare lo stesso metodo da parte di altro thread, ma non è possibile neanche invocare **altri metodi synchronized della stessa classe**; quindi un thread che ha avuto accesso ad un oggetto di tipo FixedList (esempio prec) impedisce ad altri thread non solo di invocare get(), ma anche di invocare set() !

Ad ogni modo, però, quando un thread ha un lock su un oggetto, non impedisce ad altri thread di invocare metodi **non synchronized** ! Per questo motivo è importante avere la parte non synchronized **immutabile** (proprio come abbiamo visto prima), ovvero dichiarata final.

Un design corretto per la classe EvenNumbers

```
class EvenNumbers{
  private int n = 0;
  public synchronized int next(){
    ++n;
    ++n;

    return n;
  }
}
```

Come possiamo ben vedere non abbiamo modificato il comportamenteo della classe, infatti abbiamo l'incremento ancora una volta eseguito in due istruzioni;

quello che cambia è il fatto che il metodo **next()** viene dichiarato come **synchronized**, quindi un solo thread alla volta avrà accesso al metodo, in modo da non creare interferenze.

Blocchi sincronizzati

Possiamo usare questo meccanismo anche per dichiarare dei blocchi:

Sintassi: `synchronized (adObject) {<blocco>}`

Queste istruzioni non potranno essere eseguite da altri thread, perchè quando si tenterà l'esecuzione si tenterà di acquisire il lock dell'oggetto passato come parametro.

In particolare: `synchronized void method(){...}`

E' equivalente a: `void method(){ synchronized (this) {...}}`

In questo modo andiamo a proteggere tutte le istruzioni presenti nel metodo chiudendole in questo blocco.

Il vantaggio

Il vantaggio di usare un blocco synchronized rispetto ad usarlo nella dichiarazione del metodo, è che possiamo prevedere una **parte delle istruzioni eseguite in mutua esclusione**, mentre **lasciare un'altra parte di istruzioni non eseguite in mutex.**

In questo modo non dobbiamo proteggere l'intero metodo, ma solo il blocco che potrebbe crearci problemi, in modo da velocizzare il tutto.

Coordinazione dei thread

Il coordinamento tra i thread è molto importante, perchè non sempre un thread può eseguire, ed in alcuni casi deve aspettare, e quindi **sincronizzarsi** con altri thread.

Questo accade perchè lo stato degli oggetti su cui il thread vuole operare non è tale per cui il thread possa andare avanti; è necessario che altri thread apportino modifiche di stato utili, in modo che un altro thread possa proseguire.

In questo schema abbiamo un produttore che scrive continuamente dati in un buffer, ed un consumatore legge dati dal buffer; la classe che possiamo scrivere per implementare il buffer è la seguente:

```
public class SimpleBuffer{
  private int buf;

  public int get(){
    return buf;
  }

  public void put(int val){
    buf = val;
  }
}
```

La classe prevede una variabile intera che permette di ospitare un valore, e due metodi get() e put() che consentono di settare ed estrarre l'elemento dal buffer.

Produttore

```
public class Producer extends Thread{
  private SimpleBuffer sbuf;
  private int id;

  public Producer(SimpleBuffer sb, int id){
    sbuf = sb;
    this.id = id;
  }
```

```
  public void run(){
    for(int i = 0; i < 10; i++){
      sbuf.put(i);
      Sout.println("Producer #" + this.id + "put: " + i);
      try{
        sleep((int) (Math.random() * 100));
      }catch(InterruptedException e){}
    }
  }
}
```

Questa classe produttore è un Thread e prevede una variabile di istanza di tipo SimpleBuffer, quindi il produttore è in grado di operare su un buffer condiviso con un consumatore.

Successivamente abbiamo una variabile id che identifica il produttore.

Il metodo run() prevede un ciclo con una variabile che va da 0 a 9, e per ogni iterazione il produttore inserirà un intero all'interno del buffer condiviso, e stamperà un messaggio.

Successivamente si sospende per un intervallo di tempo random.

Consumatore

```
public class Consumer extends Thread{
  private SimpleBuffer sbuf;
  private int id;

  public Consumer(SimpleBuffer sv, int id){
    sbuf = sb;
    this.id = id;
  }

  public void run(){
```

```
    int value = 0;
    for(int i = 0; i < 10; i++){
      value = sbuf.get();
      Sout.println("consumer #" + this.id + "got: " + value);

      try{
        sleep((int) (Math.random() * 100));
      }catch(InterruptedException e){}
    }
  }
}
```

Anche in questo caso abbiamo un ciclo e ad ogni iterazione viene estratto l'intero dal buffer e viene stampato; alla fine di ogni iterazione il processo viene sospeso.

Possibile applicazione

```
public class ProducerConsumerTest{
  public static void main(String[] args){
    SimpleBuffer sb = new SimpleBuffer();
    Producer p1 = new Producer(sb, 1);
    Consumer c1 = new Consumer(sb, 1);

    p1.start();
    c1.start();
  }
}
```

Il produttore ed il consumatore condivideranno lo stesso buffer;

Avviando produttore e consumatore potremmo ottenere diversi risultati:

Caso 1: produttore è più veloce del consumatore

```
Consumer #1 got: 3
Producer #1 put: 4
Producer #1 put: 5
Consumer #1 got: 5
```

Il consumatore ha letto prima 3 e poi 5, quindi "si è perso" un valore, non è infatti riuscito a leggere 4.

Caso 2: il consumatore è più veloce del produttore

```
Producer #1 put: 4
Consumer #1 got: 4
Consumer #1 got: 4
Producer #1 put: 5
```

Il consumatore ha letto due volte lo stesso valore.

Il problema

Il problema che si verifica in questa implementazione è che uno dei due tra produttore e consumatore esegue più velocemente dell'altro; di conseguenza non si ha il tempo per leggere/scrivere.

La soluzione è quella di non consentire di eseguire get() se non è stato scritto un nuovo dato (put()). Allo stesso modo non dobbiamo di eseguire put() se il dato precedente non è stato letto.

Fine lezione 13