Azzolini Riccardo 2020-03

Thread

1 Modello di esecuzione semplificato

L'istanziazione ed esecuzione di un programma su un computer è un argomento complesso, ma il comportamento di base dei programmi concorrenti può essere spiegato usando un'architettura di computer semplificata. Nel seguito, verrà quindi presentato un modello di questo tipo.

2 Modello di memoria semplificato

Il **modello di memoria semplificato** (SMM, *Simplified Memory Model*), usato dalla **macchina virtuale semplificata** (SVM, *Simplified Virtual Machine*), comprende diversi tipi di memoria:

Heap: usato per memorizzare tutti gli oggetti e i loro dati.

Method area: contiene le definizioni delle classi e le istruzioni compilate.

Program context: contiene informazioni relative all'esecuzione del processo e dei thread:

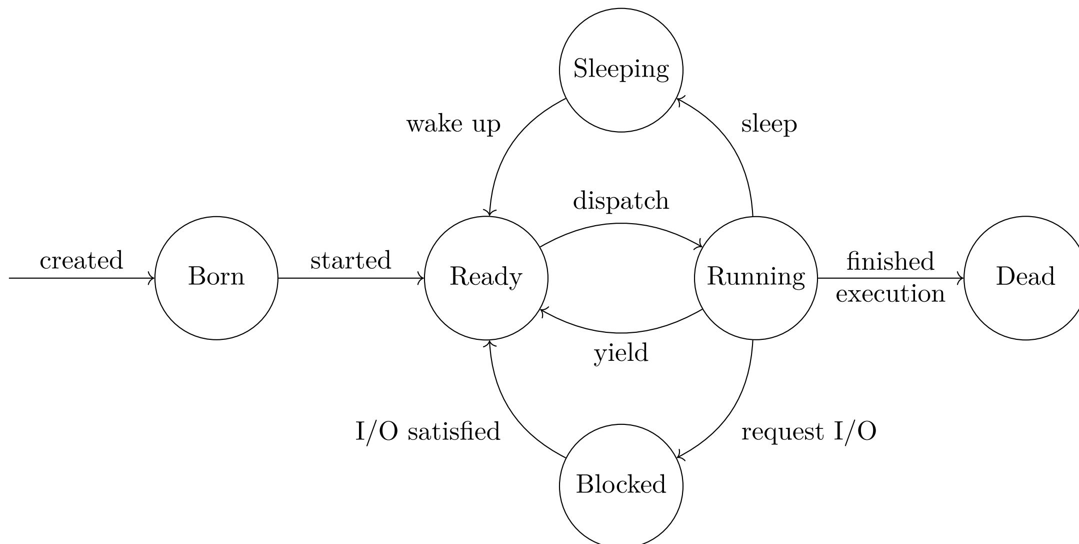
- per ogni thread, un **thread context**, che comprende lo stato del thread, il suo program counter (PC), il suo stack, ecc.;
- il program counter di processo, che indica l'istruzione del thread corrente da eseguire;
- le strutture dati necessarie per lo scheduling dei thread.

3 Stati sleeping e blocked

Oltre agli stati visti in precedenza (born, ready, running e dead), un thread può anche essere sospeso, diventando momentaneamente non schedulabile:

- una chiamata `Thread.sleep`, mediante la quale il thread rinuncia volontariamente all'esecuzione per un certo periodo di tempo, porta tale thread allo stato **sleeping**;

- se il thread richiede di eseguire un'operazione di I/O, ma il dispositivo che deve eseguirla non è pronto, tale thread va in stato **blocked** finché il dispositivo non diventa pronto.



Una delle principali differenze tra sleeping e blocked è la durata:

- per lo stato di sleeping, essa è prevedibile, in quanto specificata come argomento alla chiamata `Thread.sleep`;
- lo stato blocked ha una durata indeterminata, poiché l'uscita da esso dipende da un evento esterno (il fatto che un dispositivo di I/O diventi pronto), che potrebbe anche non verificarsi mai.

4 Altre osservazioni sui thread

- Una chiamata `t.start()` rende il thread `t` pronto all'esecuzione (stato ready), ma non ne provoca l'avvio immediato. Dopo tale chiamata, infatti, il controllo ritorna al chiamante, e sarà poi lo scheduler a invocare il metodo `run()` di `t`, quando lo riterrà opportuno. I due thread saranno quindi eseguiti in modo concorrente e indipendente.
- *Importante*: L'ordine in cui ogni singolo thread esegue le proprie istruzioni è noto, ma l'ordine in cui le istruzioni dei vari thread sono effettivamente eseguite è indeterminato (**nondeterminismo**).

- Quando un thread T_1 avvia un altro thread T_2, si dice che T_1 è **padre** di T_2, e, viceversa, T_2 è **figlio** di T_1.

  Nel momento in cui un thread padre passa allo stato dead, esso non può sempre terminare immediatamente: deve continuare a esistere finché ci sono suoi figli non daemon che sono ancora alive.

5 Sleep

Il metodo statico `Thread.sleep`,

```
public static void sleep(long millis) throws InterruptedException;
```

permette di mettere in pausa il thread corrente, per un numero di millisecondi specificato come argomento. Ciò può essere utile, ad esempio, per fare sì che l'esecuzione di un nuovo thread inizi prima di proseguire con il thread corrente:

```
public class ThreadExample extends Thread {
    public void run() {
        // ...
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadExample t = new ThreadExample();
        t.start();
        Thread.sleep(1);
        // ...
    }
}
```

Siccome `sleep` è, appunto, un metodo statico, un thread può mettere in attesa solo sé stesso, e non un altro thread.

Un thread in sleep può essere interrotto da un altro thread (mediante una chiamata `interrupt()`). In tal caso, nel thread interrotto si ha un'`InterruptedException`. Essendo un'eccezione controllata, è necessario gestirla (mediante `try/catch` o `throws`) in ogni metodo che esegue `sleep()`. Comunque, quando un thread deve essere interrotto, è preferibile usare `wait`, piuttosto che `sleep`.

5.1 Confronto con un busy loop

Mentre è in attesa, il thread non utilizza cicli del processore, a differenza di quanto avverrebbe con un *busy loop*:

```java
private void busyLoop(long millis) {
    long startTime = System.currentTimeMillis();
    long stopTime = startTime + millis;
    while (System.currentTimeMillis() < stopTime) {}
}
```

Inoltre, con questa soluzione, soprattutto se ci sono molti thread, può facilmente succedere che, al momento in cui l'attesa dovrebbe terminare, il thread non sia in esecuzione. Se si fosse usata una chiamata `sleep`, lo scheduler "saprebbe" che sia terminata l'attesa, e potrebbe quindi scegliere di rimettere subito in esecuzione il thread. Invece, con un busy loop, il thread ripartirà solo quando verrà selezionato in base ai normali criteri di scheduling, cioè probabilmente in ritardo rispetto al momento desiderato.

6 Metodi `isAlive()` e `join()`

Il metodo `isAlive()` di un oggetto thread può essere utilizzato per testare se tale thread è alive ("vivo"). Un thread si considera alive da quando viene chiamato il suo metodo `start`, e fin quando il suo metodo `run()` non ritorna.

`isAlive()` permette di attendere la terminazione di un thread, ma *in modo inefficiente*, testando periodicamente se tale thread sia ancora vivo:

```java
while (t1.isAlive()) {
    Thread.sleep(100);
}
```

Per attendere in modo efficiente la terminazione di un thread, è invece disponibile il metodo `join()`, che blocca il thread corrente finché, appunto, non termina il thread sul quale `join()` è richiamato. Analogamente a `sleep()`, anche `join()` può lanciare una `InterruptedException`.

Esistono anche versioni di `join()` con uno e due argomenti, che permettono di specificare un *timeout*, cioè un tempo massimo di attesa (in millisecondi, o millisecondi e nanosecondi):

```java
public final void join(long millis)
    throws InterruptedException;
public final void join(long millis, int nanos)
    throws InterruptedException;
```

In seguito all'invocazione di `join` con timeout, si può usare il metodo `isAlive()` per determinare se l'attesa è finita a causa della terminazione del thread, oppure per la scadenza del timeout.

6.1 Esempio di attesa della terminazione

Si consideri, ad esempio, la seguente implementazione di un thread:

```
public class ExampleThread extends Thread {
    public ExampleThread(String name) {
        super(name);
    }

    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(getName() + ": in esecuzione");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {}
        }
        System.out.println(getName() + ": finito");
    }
}
```

Se, nel main, vengono avviate alcune istanze di questo thread, e poi si usa `System.exit` per terminare l'intera applicazione, l'uscita avviene immediatamente, senza lasciare ai thread il tempo di completare la loro esecuzione:

```
public class ExitImmediately {
    public static void main(String[] args) {
        System.out.println("I thread stanno per partire");
        Thread t1 = new ExampleThread("t1");
        t1.start();
        Thread t2 = new ExampleThread("t2");
        t2.start();
        Thread t3 = new ExampleThread("t3");
        t3.start();
        System.out.println("I thread sono partiti");

        System.out.println("Chiusura dell'applicazione");
        System.exit(0);
    }
}
```

Addirittura, potrebbe accadere che i thread non vengano neanche schedulati prima che l'applicazione termini. In tal caso, si avrebbe in output solo:

```
I thread stanno per partire
I thread sono partiti
Chiusura dell'applicazione
```

Un modo per assicurarsi che l'esecuzione dei thread possa completarsi prima della terminazione dell'applicazione è implementare un'attesa mediante `isAlive()` e `sleep`:

```
public class IsAliveSleep {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("I thread stanno per partire");
        Thread t1 = new ExampleThread("t1");
        t1.start();
        Thread t2 = new ExampleThread("t2");
        t2.start();
        Thread t3 = new ExampleThread("t3");
        t3.start();
        System.out.println("I thread sono partiti");

        while (t1.isAlive() || t2.isAlive() || t3.isAlive()) {
            Thread.sleep(100);
        }

        System.out.println("Chiusura dell'applicazione");
        System.exit(0);
    }
}
```

In questo caso, come desiderato, si ha un output del tipo:

```
I thread stanno per partire
I thread sono partiti
t2: in execuzione
t1: in execuzione
t3: in execuzione
t2: in execuzione
t3: in execuzione
t1: in execuzione
t2: in execuzione
t3: in execuzione
t1: in execuzione
t2: finito
t1: finito
t3: finito
Chiusura dell'applicazione
```

La soluzione migliore, invece, consiste nell'uso del metodo `join()` per implementare l'attesa:

```java
public class Join {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("I thread stanno per partire");
        Thread t1 = new ExampleThread("t1");
        t1.start();
        Thread t2 = new ExampleThread("t2");
        t2.start();
        Thread t3 = new ExampleThread("t3");
        t3.start();
        System.out.println("I thread sono partiti");

        t1.join();
        t2.join();
        t3.join();

        System.out.println("Chiusura dell'applicazione");
        System.exit(0);
    }
}
```

Questo programma produce un output analogo a quello che utilizza `isAlive()`, ma è più efficiente, dato che il thread main non si deve svegliare ogni 100 millisecondi per controllare se gli altri thread sono ancora in esecuzione, e, al tempo stesso, si "accorge subito" quando i thread sono terminati (senza il ritardo che potrebbe essere causato dallo `sleep` tra un controllo di `isAlive()` e il successivo).

Osservazione: In questo caso, l'ordine delle chiamate `join()` non importa. Infatti, se, ad esempio, il thread `t2` dovesse terminare prima che il thread main raggiunga l'istruzione `t2.join()` (cioè mentre esso è ancora in attesa di `t1`), la successiva esecuzione di tale chiamata non bloccherebbe il main, evitando così di metterlo in attesa di un evento (la terminazione di `t2`) che è già avvenuto, e si passerebbe direttamente all'istruzione successiva, `t3.join()`.

7 Come fermare un thread

Nonostante Java sia stato concepito, fin dall'inizio, come un linguaggio multi-thread, i progettisti non sono riusciti a fornire un mezzo sicuro ed efficace per fermare un thread dopo il suo avvio.

In origine, la classe `Thread` includeva i metodi `stop()`, `stop(Throwable)`, `destroy()`, `suspend()` e `resume()`, che fornivano le funzionalità di base relative alla sospensione e

all'arresto di un thread. Tutti questi metodi si sono però rivelati problematici, e sono quindi stati deprecati.[1]

Con la versione 5.0 (o 1.5) di Java, è invece stato introdotto il metodo `interrupt()`, che consente non di arrestare direttamente un thread, bensì di richiedere a esso che la sua esecuzione termini. Infatti, `interrupt()` imposta semplicemente un flag di interruzione nel thread di destinazione, e ritorna subito al chiamante. Spetta poi al codice del thread che riceve l'interruzione controllare se il flag è stato settato, e, in tal caso, uscire dal metodo `run()` per terminare la propria esecuzione. Non si può quindi assumere che un thread sia stato interrotto quando il metodo `interrupt()` ritorna.

I metodi che mettono in pausa un thread (`sleep`, `join`, ecc.) controllano il flag di interruzione prima e durante lo stato di pausa: se esso risulta settato, lo resettano, e lanciano una `InterruptedException`. Il thread interrotto la intercetta, e "dovrebbe" terminare l'esecuzione, ma, in realtà, il fatto che termini o meno dipende interamente dal codice di gestione dell'eccezione.

7.1 Esempio

```java
public class ExampleThread extends Thread {
    public ExampleThread(String name) {
        super(name);
    }

    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(getName() + ": in esecuzione");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                System.out.println(getName() + ": interrotto");
                break; // Esci dal ciclo, e quindi termina
            }
        }
        System.out.println(getName() + ": finito");
    }
}
```

Invocando `interrupt()` su un'istanza di questo thread, viene sollevata (subito, se il thread si trova attualmente in stato sleeping, oppure appena chiama il metodo `sleep`) un'`InterruptedException`, e il blocco `catch` che la intercetta interrompe il ciclo. Allora,

[1] Inoltre, il metodo `destroy()` non è mai stato veramente implementato (ha sempre solo sollevato un'eccezione), ed è stato rimosso completamente in Java 11, insieme a `stop(Throwable)`, che già da Java 8 era stato reso non funzionale (facendo sì che anch'esso sollevasse sempre un'eccezione).

viene subito effettuata la stampa che indica la fine del thread, e poi l'esecuzione del thread termina.

Ad esempio, il programma

```
public class Interrupt {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("I thread stanno per partire");
        Thread t1 = new ExampleThread("t1");
        t1.start();
        Thread t2 = new ExampleThread("t2");
        t2.start();
        Thread t3 = new ExampleThread("t3");
        t3.start();
        System.out.println("I thread sono partiti");

        t1.interrupt();
        t2.interrupt();
        t3.interrupt();

        t1.join();
        t2.join();
        t3.join();

        System.out.println("Chiusura dell'applicazione");
        System.exit(0);
    }
}
```

produce un output del tipo:

```
I thread stanno per partire
I thread sono partiti
t2: in execuzione
t3: in execuzione
t2: interrotto
t1: in execuzione
t2: finito
t1: interrotto
t1: finito
t3: interrotto
t3: finito
Chiusura dell'applicazione
```

7.2 Controllo del flag di interruzione

Se il thread che si vuole interrompere non esegue mai metodi di attesa (ad esempio, se effettua solo calcoli in memoria), non verrà mai lanciata una `InterruptedException`. Allora, per poter essere interrotto, un thread di questo tipo deve cooperare, ad esempio controllando periodicamente il suo stato di interruzione. A tale scopo, esistono due appositi metodi della classe `Thread`:

- `isInterrupted()`, che controlla il flag di interruzione dell'istanza di `Thread` sulla quale si invoca il metodo;
- `Thread.interrupted()`, che controlla il flag di interruzione del thread corrente, e, se è settato, lo resetta.

Un esempio di uso di `Thread.interrupted()` è:

```
public void run() {
    while (condizione) {
        // operazioni complesse...

        if (Thread.interrupted()) {
            break;
        }
    }
}
```

8 Collaborazione: `Thread.yield()`

Una chiamata al metodo statico `Thread.yield()` suggerisce allo scheduler che il thread chiamante è disposto a lasciare volontariamente la CPU a un altro thread: con tale chiamata, si cede il controllo allo scheduler, che (probabilmente) manderà in esecuzione un altro thread (se esiste), al posto del thread chiamante.

`yield()` può essere utile per i thread che non eseguono spesso operazioni che li mettono in attesa:

- sui sistemi senza preemption, se non si usasse `yield()`, si potrebbe avere starvation, perché un thread che non si mette mai in attesa non lascerebbe mai la CPU;
- in alcuni casi molto particolari, l'uso di `yield()` su sistemi con preemption permette di ottimizzare la ripartizione della CPU tra i thread.

Ci sono però delle alternative che, a seconda della situazione, possono essere preferibili. Se, per esempio, si vuole "usare solo parte della CPU", è meglio stimare la quantità di CPU che il thread ha utilizzato nel suo ultimo blocco di elaborazione, e usare `sleep` per "compensare" con un adeguato periodo di inattività.

8.1 Esempio

```
public class CountDown implements Runnable {
    private final int id;
    private int countDown = 5;

    public CountDown(int id) {
        this.id = id;
    }

    private String status() {
        String msg = countDown > 0
            ? String.valueOf(countDown)
            : "Decollo!";
        return String.format("id=%d: %s", id, msg);
    }

    public void run() {
        while (countDown > 0) {
            countDown--;
            System.out.println(status());
            Thread.yield();
        }
    }
}

public class YieldExample {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            new Thread(new CountDown(i)).start();
        }
        System.out.println("In attesa del decollo");
    }
}
```

A causa della chiamata `yield()`, nell'output di questo programma si osserva che nessun thread scrive mai due (o più) volte di fila,

```
In attesa del decollo
id=1: 4
id=2: 4
id=0: 4
id=2: 3
id=1: 3
```

```
id=2: 2
id=0: 3
id=1: 2
id=2: 1
id=0: 2
id=1: 1
id=2: Decollo!
id=0: 1
id=1: Decollo!
id=0: Decollo!
```

a meno che non sia l'ultimo thread rimasto in esecuzione:

```
In attesa del decollo
id=1: 4
id=2: 4
id=0: 4
id=1: 3
id=2: 3
id=1: 2
id=2: 2
id=1: 1
id=2: 1
id=1: Decollo!
id=2: Decollo!
id=0: 3
id=0: 2
id=0: 1
id=0: Decollo!
```