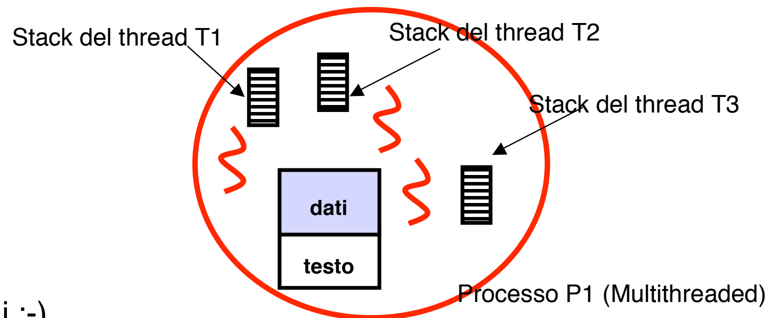
Generalita' sui Thread

Un thread (o processo leggero) è una attività, descritta da una sequenza di istruzioni, che esegue all'interno del contesto di esecuzione di un programma. Un thread procede nella sua esecuzione per portare a termine un certo obiettivo ***parallelamente*** agli altri thread presenti.

I thread ***interagiscono*** tra loro poiché possono scambiarsi dei messaggi e poiché condividono risorse di sistema (come i file) e gli oggetti dello stesso programma. I thread competono nell'uso della CPU e di altre risorse condivise (file o oggetti).

I thread di uno stesso processo ***condividono*** dati e codice del processo, ma lo stack di esecuzione ed il program counter sono privati.



Vantaggi :-)

- Il context-switch fra thread, e quindi l'attivazione di un thread, è meno oneroso rispetto all'equivalente per i processi.
- Lo scambio di dati fra thread di un processo è molto semplice, poiché condividono lo stesso spazio di indirizzamento.

I problemi di sincronizzazione sono più frequenti e rilevanti :-(

Modello a Thread

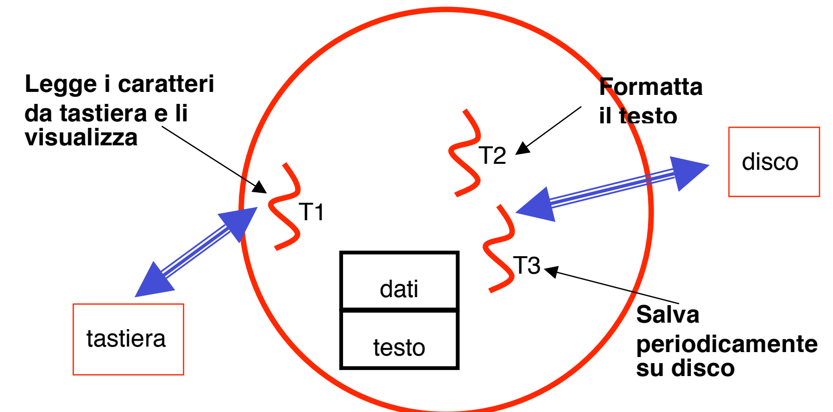
Per un processo con più thread di controllo, lo stato di avanzamento della computazione di ogni thread è dato da:

- valore del PC (prossima istruzione da eseguire)
- valore di SP/PSW e dei registri generali
- contenuto dello Stack privato di quel thread (ovvero le variabili locali ad un metodo)
- stato del thread: *pronto, in esecuzione, bloccato*

Sono invece comuni a tutti i thread:

- stato dell'area testo e dati globali (gli attributi di una classe)
- stato dei file aperti e delle strutture di IPC utilizzate

Esempio: un word processor multi-thread



Ciclo di vita dei Thread

- un thread viene creato, riceve un identificatore (tid), ed entra nello stato **created**;
- dopo l'avvio (tramite `start()`) il suo stato è **ready**;
- il thread sarà schedulato per l'esecuzione dal SO (o dal sistema di supporto run-time), quando arriva il suo turno, inizia l'esecuzione passando allo stato di **running**;
- il thread può trovarsi in stato di **blocked**, quando:
  - ha invocato `sleep(m)`
  - ha invocato `wait()`
  - è in attesa che una operazione di i/o si completi
  - ha chiamato un metodo `synchronized` su un oggetto il cui lock non è disponibile
- il thread termina la sua esecuzione e passa nello stato di **stopped.**

I thread possono essere realizzati da librerie che eseguono in modalità user

- il SO e lo scheduler non conoscono l'esistenza dei thread e gestiscono solo il processo
- lo scheduling viene effettuato dal supporto a run-time della libreria

Oppure, i thread possono essere realizzati all'interno del kernel

Thread in Java

Ogni esecuzione di una JVM dà origine ad un processo. Ogni programma in Java consiste di almeno un thread, quello che esegue il metodo `main()` della classe fornita alla JVM in fase di start up.

La JVM ha libertà sulla mappatura dei thread Java su quelli del SO. Può sfruttare il supporto multi-threading del SO sottostante (Windows) o prendersi carico della gestione dei thread interamente. In questo ultimo caso il SO vede la JVM come un processo con un unico thread (Unix).

In Java i thread sono nativi, cioè supportati a livello di linguaggio.

I thread si possono implementare in due modi:

- creando una sottoclasse della classe `Thread`
- creando una classe che implementa l'interfaccia *`Runnable`*

La classe `Thread` è una classe <u>non</u> astratta che fornisce vari metodi (es. `start()`, `isAlive()`, `interrupt()`) che consentono di controllare l'esecuzione del thread.

Procedimento per creare un thread tramite sottoclasse di `Thread`:

1. La sottoclasse di `Thread` deve implementare il metodo `run()`
2. Bisogna creare una istanza della sottoclasse tramite `new`
3. Si esegue il thread chiamando il metodo `start()` che a sua volta richiama il metodo `run()`

Classe Thread

Esempio:

Creiamo una classe che eredita dalla classe `Thread` e che implementa il metodo `run()`

```
class MioThread extends Thread {
  int id;
  MioThread(int n) {
    System.out.println("Creato miothread");
    id = n;
  }
  public void run() {
    // esegue istruzioni
    System.out.println("MioThread running");
    for (int i = 0; i < 1000; i++)
      if ((i%30)== 0) System.out.print(id);
  }
}
```

Creazione di istanze della classe `MioThread` ed esecuzione:

```
// creazione primo thread
MioThread t1 = new MioThread(1);
// creazione secondo thread
MioThread t2 = new MioThread(2);
// esecuzione parallela del primo thread
t1.start();
// esecuzione parallela del secondo thread
t2.start();
```


Interfaccia *Runnable*

Un modo per creare un thread che non è sottoclasse di `Thread`:

1. Implementare in una classe `R` l'interfaccia *`Runnable`* (che implementa il metodo `run()`)
2. Creare una istanza della classe `R`
3. Creare una istanza `t` della classe `Thread` passando come parametro al costruttore l'istanza della classe `R`
4. Invocare il metodo `start()` sul thread `t`, questo eseguirà il metodo `run()` della classe `R`

```
class MioRun implements Runnable {
  MioRun() {
    System.out.println("Creato oggetto");
  }
  public void run() {
    // esegue istruzioni
  }
}
```

Per creare il thread ed eseguirlo:

```
Thread t = new Thread(new MioRun());
t.start();
```

Quest'ultima è una modalità di creazione dei thread leggermente più complessa, rispetto a quella che eredita da `Thread`, ma libera dal vincolo di avere una superclasse fissata. Si rivela utile, non disponendo dell'ereditarietà multipla in Java, quando vogliamo realizzare un thread per una classe che ha bisogno di ereditare da una classe dell'applicazione.

Metodi della classe `Thread` - 1

`start()`

chiama il metodo `run()` sull'oggetto/thread `t` della classe `MioThread` che abbiamo implementato. Il thread `t` termina quando il metodo `run()` ha finito l'esecuzione.

Un thread **non** può essere fatto ri-partire, cioè il metodo `start()` deve essere chiamato solo una volta, altrimenti viene generata una `InvalidThreadStateException`.

`boolean isAlive()`

permette di testare se il thread è stato avviato e non è ancora finito.

`setPriority(int p)`

permette di cambiare la priorità di esecuzione del thread.

`p` può variare tra `Thread.MIN_PRIORITY` e `Thread.MAX_PRIORITY` (`1` e `10`). La priorità di default di un thread è pari alla priorità del thread che lo ha creato. Per default, il `main()` ha priorità 5.

`int getPriority()`

ritorna la priorità del thread.

`yield()`

metodo statico della classe `Thread` che ferma momentaneamente l'esecuzione del thread corrente per permettere ad un altro thread di eseguire.

Metodi della classe `Thread` - 2

`join()`

permette di bloccare il chiamante fino a che il thread sul quale si chiama `join()` non termina la propria esecuzione.

Il metodo `join(long millis)` fa aspettare il chiamante per la terminazione del thread al massimo `millis` milli-secondi.

Lancia `InterruptedException` se `interrupt()` è chiamato sul thread.

`sleep(long millis)`

metodo statico, fa aspettare il thread chiamante per `millis` milli-secondi. **Nessun** lock viene rilasciato. Lancia `InterruptedException` se `interrupt()` è chiamato sul thread.

`stop()`

forza la terminazione del thread sul quale si invoca. Tutte le risorse usate dal thread vengono liberate (inclusi lock). E' deprecato, poiché **non** garantisce la consistenza dell'oggetto.

`suspend()`

blocca l'esecuzione del thread sul quale si invoca. Il thread rimane in attesa di una operazione `resume()`. **Non** libera le risorse impegnate. E' deprecato, poiché può determinare situazioni di blocco critico (deadlock).

Metodi della classe **Thread** -3

`interrupt()`

permette di interrompere l'esecuzione del thread sul quale si invoca, solo quando lo stato dell'oggetto lo consente, cioè quando non è in esecuzione, ma in attesa di un evento. Ciò consente (a differenza del metodo `stop()`) di mantenere lo stato dell'oggetto consistente.

`Thread currentThread()`

è un metodo **statico** della classe `Thread` che restituisce un identificativo del thread che sta correntemente eseguendo.

`toString()`

restituisce una rappresentazione del thread, che include nome, priorità e gruppo.

L'uso di `stop()` e `suspend()` è sconsigliata poiché bloccano bruscamente l'esecuzione di un thread. Possono quindi generare problemi allo stato dell'oggetto, poiché una azione atomica (indivisibile) viene interrotta. Inoltre, con `suspend()`, tutte le risorse acquisite non sono rilasciate quando il thread è bloccato e possono rimanere inutilizzabili indefinitamente.

E' meglio usare dei metodi per la sincronizzazione fra thread: `wait()`, `notify()` e `notifyAll()`

Sincronizzazione

Differenti thread della stessa applicazione condividono lo stesso spazio di memoria. E' quindi possibile che più thread accedano alla stessa sezione di codice o allo stesso dato.

La durata e l'ordine di esecuzione dei thread non è predicibile. Non possiamo stabilire quando lo scheduler del SO interromperà l'esecuzione di un thread per eseguirne un altro.

Quando più di una attività esegue, l'esecuzione è necessariamente non-deterministica e la comprensione del programma non è data dalla semplice lettura sequenziale del codice.

Per esempio, una variabile che è assegnata con un valore in una istruzione di programma, può avere un differente valore nel momento in cui la linea successiva è eseguita (a causa dell'esecuzione di attività concorrenti).

Esempio:

thread_1	***thread_2***	***num***
`num=0;`		0
`genera();`		0
`num++;`		1
	`consuma();`	1
	`num--;`	0
`if (num > 0) notifica();`		0

Costrutto synchronized - 1

Tale “interferenza” è eliminata con una progettazione che usa meccanismi di sincronizzazione, tipo **semafori**.

Il **lock** su un semaforo permette di evitare l'ingresso di più thread in una **regione critica** (parte di un programma che accede a memoria o file condivisi o svolge azioni che possono portare a **corse critiche**) e di ottenere mutua esclusione.

Usando i costrutti primitivi di Java (parola chiave **`synchronized`**) possiamo realizzare un lock su una sezione di codice o possiamo realizzare un semaforo.

`synchronized` può delimitare un frammento di codice o agire da modificatore di un metodo di una classe.

Usato su un frammento di codice, per consentire l’esecuzione del codice ad un solo thread alla volta, necessita di una variabile su cui sarà acquisito il lock (per es. sull'oggetto `this`).

```
1  synchronized(this) {
2    num = 0;
3    generate();
4    num++;
5    if (num > 0) notifica();
6  }
```

Il lock su `this` è acquisito automaticamente all'ingresso del codice (linea 1), e rilasciato automaticamente alla sua uscita (linea 6).

Costrutto synchronized - 2

Quando `synchronized` è usato come modificatore di un metodo, la sua esecuzione è subordinata all’acquisizione di un lock sull’oggetto su cui si invoca tale metodo.

In una classe dove tutti i metodi sono dichiarati `synchronized` **un solo thread** può eseguire al suo interno in un determinato momento. Si ha quindi l'associazione automatica di un lock ad un oggetto di questo tipo e l'accesso esclusivo al codice della classe.

Il costrutto `synchronized` permette di aggiornare una variabile **in modo atomico** e di creare una classe che fa attendere i thread la cui richiesta non può essere soddisfatta.

La seguente classe può essere utile per abilitare 10 esecuzioni esclusive su un oggetto.

```
public class EsecSingola {
   private int value;
   public EsecSingola() {
      value = 10;
   }
   synchronized public void reset() {
      if (value == 0) value = 10;
   }
   synchronized public void elabora() {
      if (getValue() > 0) {
         --value;
         // fai qualcosa di utile
      }
   }
   synchronized public int getValue() { return value; }
}
```

Costrutto synchronized -3

Quando un thread tenta di accedere ad un oggetto istanza di questa classe, acquisisce implicitamente il lock (se nessun thread sta eseguendo all'interno dello stesso oggetto).

Il thread che detiene il lock per un oggetto di questo tipo può eseguire liberamente (senza alcun test) tutti i metodi dell'oggetto.

I thread che dopo tentano di accedere allo stesso oggetto verranno sospesi, e risvegliati quando il thread che è all'interno finisce l'esecuzione del metodo. In pratica, il thread che era all'interno rilascia automaticamente il lock.

Un metodo `synchronized` non è interrotto, cioè viene eseguito in modo atomico (ok, il thread che lo esegue può essere interrotto).

Se sono presenti dei metodi non `synchronized` all'interno della classe, su questi non viene acquisito il lock all'ingresso.

L'uso di `synchronized` introduce un overhead: il tempo necessario per cominciare ad eseguire il metodo è maggiore di quello di un metodo non `synchronized` (per alcune implementazioni costa 4 volte in più).

Ogni volta che usiamo metodi `synchronized` riduciamo il parallelismo possibile all'interno del programma e potenzialmente costringiamo alcuni thread ad attendere.

L'uso di `synchronized` ci protegge da eventuali “interferenze” durante l'esecuzione del codice ed è quindi utile per garantire la correttezza, ma richiede una certa attenzione per prevenire ritardi e deadlock.

Metodi di sincronizzazione-1

In Java ogni oggetto è potenzialmente un monitor. La classe `Object` mette quindi a disposizione i metodi di sincronizzazione: `wait()`, `notify()` e `notifyAll()`. Esse possono essere invocate solo dopo che è stato acquisito un lock, cioè all'interno di un blocco o metodo `synchronized`.

`wait()`

blocca l'esecuzione del thread invocante fino a che un altro thread invoca una `notify()` sull'oggetto. Si fa sempre dopo aver testato una condizione (ed in un ciclo, per essere sicuri che al risveglio la condizione è verificata).

```
while (! condition) // se non può procedere
    this.wait();    // aspetta una notifica
```

- Il thread invocante viene bloccato, il lock sull'oggetto è rilasciato automaticamente.
- I lock su altri oggetti sono mantenuti (bisogna quindi fare attenzione a possibili deadlock).

Un oggetto con metodi `synchronized` gestisce di per sè 2 code:

- una coda di lock, per i thread a cui l'accesso è escluso,
- una coda di wait per le condizioni di attesa.

Un thread può essere in una sola delle due code.

La variante `wait(long timeout)` blocca il thread per al massimo `timeout` millisecondi (se `timeout` > 0)

Metodi di sincronizzazione-2

`notify()`

risveglia un solo thread tra quelli che aspettano sull'oggetto in questione. Se più thread sono in attesa, la scelta di quale svegliare viene fatta dalla JVM. Una volta risvegliato, il thread compete con ogni altro (non in wait) che vuole accedere ad una risorsa protetta.

`notifyAll()`

risveglia tutti i thread che aspettano sull'oggetto in questione.

In pratica i thread nella coda di wait vengono trasferiti nella coda di lock ed aspettano il loro turno per entrare.

`notifyAll()` è più sicura, poichè il thread scelto da `notify()` potrebbe non essere in grado di procedere e venire sospeso immediatamente, bloccando l'intero programma.

Esempio con classi Produttore - Consumatore

Una classe Produttore produce un item e lo inserisce in una classe Contenitore. Il Consumatore estrae l'item presente nel Contenitore, se esiste.

Produttore e Consumatore sono 2 thread.

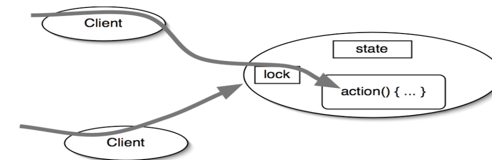
Contenitore è la risorsa condivisa.

Produttore non deve sovrascrivere l'item già presente su Contenitore, ma deve aspettare che qualcuno lo rimuova. Consumatore, una volta rimosso l'item di Contenitore, deve notificare i thread in attesa della risorsa.

Progettazione di sistemi paralleli - 1

Per proteggere l'accesso a risorse (oggetti) condivise possiamo usare:

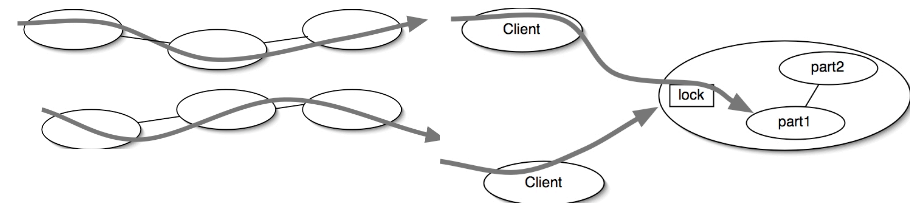
- Oggetti completamente sincronizzati: tutti i metodi sono dichiarati `synchronized` e le variabili sono `private`.



- Contenimento: uso di tecniche di incapsulamento per garantire che al massimo una attività avrà accesso ad un oggetto. Sono simili alle misure per garantire la sicurezza.

  Evito che il riferimento ad alcuni oggetti sia conosciuto al di fuori di un certo numero di oggetti/thread, imponendo un unico percorso per accedere a certe risorse.

  Posso realizzare il contenimento tramite incapsulamento di oggetti all'interno di altri.



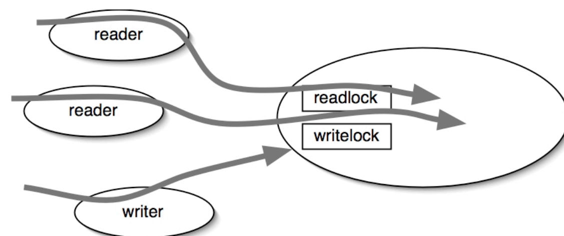
Progettazione di sistemi paralleli - 2

Per aumentare il parallelismo (eliminando qualche collo di bottiglia) possiamo realizzare:

- Divisione dei lock: anziché associare un lock ad un insieme di funzionalità di un oggetto, dispongo di diversi lock, ciascuno per una distinta funzionalità.

  In questo caso posso sempre avere un singola classe con tutte le funzionalità, ma più lock che regolano l'accesso dei thread. L'implementazione può essere ottenuta, per esempio, con l'uso di `synchronized` su un blocco di codice e non sui metodi della classe.

- Coppie di lock, per lettura e per scrittura: posso identificare quali operazioni sono in lettura (non modificano lo stato) e quali in scrittura e consentire più lettori contemporaneamente, ma un solo scrittore alla volta.



Gruppi di thread

Si possono raccogliere tanti thread all'interno di un gruppo e così facilitare le operazioni di sospensione o di ripartenza dell'insieme di thread con una sola invocazione.

La JVM associa un thread ad un gruppo al momento della creazione del thread. Tale associazione non può essere modificata a run-time.

```
ThreadGroup mytg = new ThreadGroup(“mio gruppo”);
Thread myt = new Thread(mytg, “mio t”);
```