Laboratorio di "Sistemi Distribuiti"
A.A. 2024-2025

Concorrenza (parte 1)

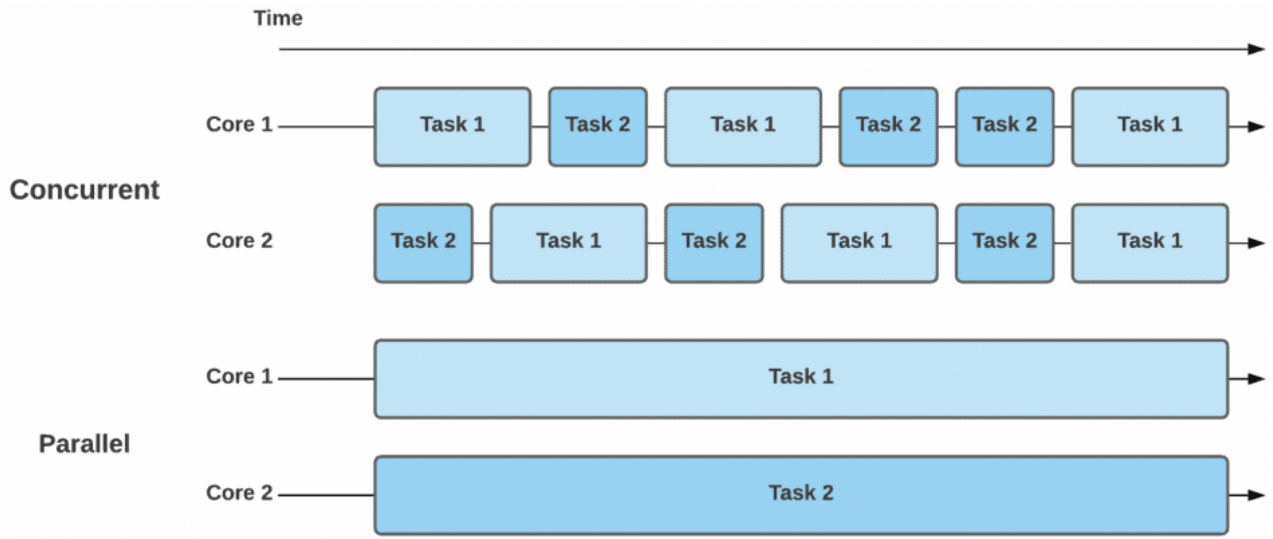
Emanuele Petriglia

Slide aggiornate al 3 Aprile 2025

Indice dei contenuti

1. Ripasso sulla concorrenza

Concorrenza vs Parallelismo



Fonte

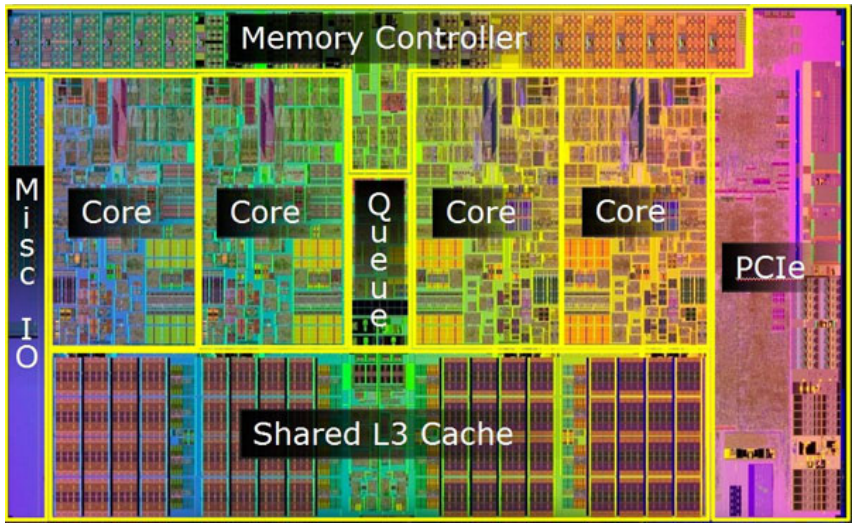
La concorrenza riguarda la struttura, mentre il parallelismo riguarda la velocità di esecuzione.

Nei laboratori ci occuperemo solo di concorrenza.

Complessità della concorrenza

Perché esiste la concorrenza?

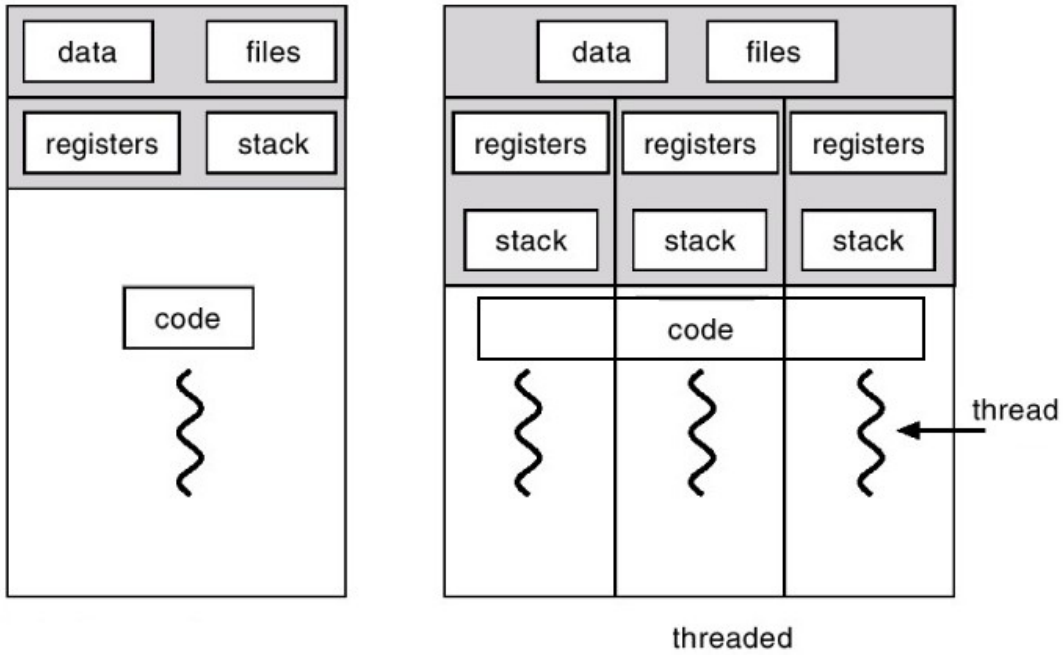
- **Per sfruttare processori multi-core e multi-socket.**
  - La VM ha una CPU AMD EPYC 7282 $\rightarrow$ 16 core / 32 thread fisici



Fonte

- **Per evitare blocchi su operazioni I/O**.
- **Per migliorare la reattività** (es. GUI).
- **Per scalabilità e modelli di programmazione più complessi** (es. simulazioni, web server, database...)

Processo vs Thread



Fonte

Noi ci concentriamo sui thread.

Punto di vista dell'OS

- Windows: processi e thread sono due entità diverse.
- Linux: processi e thread sono la stessa cosa.

Perché esistono i thread?

- **Condivisione codice e dati tra thread facilitata.**
  - Ma è più "pericoloso" (race condition...).
- **I thread sono più "leggeri" rispetto ai processi.**
  - Context switching e tempo di creazione ridotti.

Tipologie di thread:

- **Kernel thread**: gestiti dall'OS.
  - Sono i "platform thread" in Java. ← **laboratorio**
- **User thread**: gestiti dall'applicazione.
  - Sono i "virtual thread" in Java.

Attenzione!

La concorrenza (e thread) introducono problemi da affrontare, tra cui: *race condition*, *deadlock* e *starvation*.

2. Concorrenza in Java

Thread in Java

Due classi principali:

- `Thread`: è una classe che rappresenta un thread.
  - Classe $\rightarrow$ Si deve estendere.
- `Runnable`: è un'interfaccia che rappresenta un'operazione che si può eseguire.
  - Interfaccia $\rightarrow$ Si deve implementare
  - **Attenzione**: richiede comunque un thread per eseguire l'operazione!

In generale: preferire `Runnable` rispetto a `Thread`.

Esempio Hello World Thread

```java
public class HelloWorldThread {
    public static void main(String[] args) {
        var thread = new ExampleThread();
        thread.start();

        var name = Thread.currentThread().getName();
        System.out.printf("Hello World from thread \"%s\"\n", name);
    }
}

class ExampleThread extends Thread {
    public void run() {
        try {
            Thread.sleep(1000); // 1 secondo.
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        var name = Thread.currentThread().getName();
        System.out.printf("Hello World from thread \"%s\"\n", name);
    }
}
```

Esempio Hello World Runnable

```
1   public class HelloWorldRunnable {
2       public static void main(String[] args) {
3           var thread = new Thread(new ExampleRunnable());
4           thread.start();
5
6           var name = Thread.currentThread().getName();
7           System.out.printf("Hello World from thread \"%s\"\n", name);
8       }
9   }
10
11  class ExampleRunnable implements Runnable {
12      public void run() {
13          try {
14              Thread.sleep(1000); // 1 secondo.
15          } catch (InterruptedException e) {
16              e.printStackTrace();
17          }
18
19          var name = Thread.currentThread().getName();
20          System.out.printf("Hello World from thread \"%s\"\n", name);
21      }
22  }
```

Pausa in un thread

- Metodo `Thread.sleep(long)`.
- La pausa può essere interrotta se qualcuno vuole interrompere (cioè terminare) il thread!
- Uso:

```
1  try {
2      Thread.sleep(2000); // 2 secondi (CIRCA).
3  } catch (InterruptedException e) {
4      e.printStackTrace();
5  }
```

- Quando si usa? Per simulare ritardi.

Esempio di interruzione

```
1   public class HelloWorldInterrupt {
2       public static void main(String[] args) {
3           var thread = new Thread(new ExampleTask());
4           thread.start();
5           try {
6               Thread.sleep(2000); // 2 secondi.
7           } catch (InterruptedException e) {
8               e.printStackTrace();
9           }
10          thread.interrupt();
11      }
12  }
13  class ExampleTask implements Runnable {
14      public void run() {
15          var name = Thread.currentThread().getName();
16          try {
17              Thread.sleep(5000); // 5 secondi.
18          } catch (InterruptedException e) {
19              System.out.printf("Thread \"%s\" interrotto durante il
                ↪  riposo!\n", name);
20          }
21          System.out.println("Thread in terminazione!");
22      }
23  }
```

Attesa di un thread

- Metodo `Thread.join()`.
- Pone in pausa il thread chiamante (non il thread invocato!) finché il thread invocato non termina.
- **Attenzione**: il thread chiamante può essere interrotto! (come per `sleep()`).
- Uso:

```
1  try {
2      threadToWait.join();
3  } catch (InterruptedException e) {
4      e.printStackTrace();
5  }
```

- Quando si usa? Per controllare l'esecuzione dei thread.

Esempio di attesa

```
1  public class HelloWorldJoin {
2      public static void main(String[] args) {
3          var first = new Thread(new ExampleTask());
4          var second = new Thread(new ExampleTask());
5          first.start();
6          try {
7              first.join();
8          } catch (InterruptedException e) { /* ... */ }
9          second.start();
10         try {
11             second.join();
12         } catch (InterruptedException e) { /* ... */ }
13     }
14 }
15
16 class ExampleTask implements Runnable {
17     public void run() {
18         try {
19             Thread.sleep(5000); // 5 secondi.
20         } catch (InterruptedException e) { /* ... */ }
21     }
22 }
```

3. Esercizio A

Consegna

Scrivere due classi che implementano `Runnable`:

- `PrintThreadName`: stampa su schermo il nome del thread e quante volte stampa il nome.
- `PrintThreadNumber`: stampa su schermo il nome del thread, quante volte stampa il nome e un numero intero casuale.

Per entrambe le classi la stampa viene fatta cinque volte a intervalli temporali casuali tra $[0, 3000)$ millisecondi.

Il thread principale:

1. Crea e avvia tre thread con `PrintThreadName`.
2. Aspetta la terminazione dei thread.
3. Crea e avvia tre thread con `PrintThreadNumber`.
4. Aspetta la terminazione dei thread e poi esce.

Suggerimenti

- Partire dallo scheletro su e-Learning!
- Usare `Random` per generare numeri casuali:

```
1  var rng = new Random();
2  var number = rng.nextInt(); // Nessun limite.
3  var number = rng.nextInt(5); // Limite [0, 5).
```

- **Attenzione**: le risorse a cui accede il thread devono essere private al thread!
- Esempio di esecuzione:

```
1  > java PrintRandom.java
2  Thread "main": creazione e avvio di PrintThreadName
3  Thread "Thread-1", opt. 0
4  Thread "Thread-2", opt. 0
5  // ...
6  Thread "Thread-0", opt. 4
7  Thread "main": creazione e avvio di PrintThreadNumber
8  Thread "Thread-5", opt 0: -1727931346
9  // ...
10 Thread "Thread-4", opt 4: -501528627
11 Thread "Thread-3", opt 4: -1099484989
12 Thread "main": in terminazione
```

4. Esercizio B

Consegna

Migliorare la soluzione testuale dell'esercizio B dello scorso laboratorio in modo che il server gestisca ogni connessione in un thread separato.

Promemoria

L'esercizio era il gioco "Guess The Number" con il protocollo testuale in TCP (derivato a sua volta da UDP).

La logica del gioco deve essere incorporata in una classe `Runnable`.

Non c'è bisogno che il thread principale aspetti i thread creati.

Provare a eseguire più client in contemporanea e vedere cosa succede.

Suggerimenti

- Partire dallo scheletro su e-Learning o dalla propria soluzione!
- Usare un costruttore per la classe che implementa `Runnable` che salvi il socket e il numero segreto da indovinare.
- **Attenzione**: le risorse a cui accede il thread devono essere private al thread (tranne socket che viene passato).
- Dei buoni log aiutano a capire cosa sta succedendo:

```
1  > java ServerText.java
2  Server in ascolto...
3  Client "50717" connesso!
4  Client "50717" -> Secret: 17    Guess: 55
5  Client "50721" connesso!
6  Client "50721" -> Secret: 84    Guess: 76
7  Client "50717" -> Secret: 17    Guess: 87
8  Client "50717" -> Secret: 17    Guess: 17
9  Client "50717" disconnesso (indovinato)!
```

5. Sincronizzazione in Java

A cosa serve la sincronizzazione?

- Cosa hanno in comune gli esercizi A e B? I singoli thread non comunicano tra di loro.
  - Non c'è bisogno della sincronizzazione.
- Quando serve la sincronizzazione tra thread? Quando più thread accedono a risorse condivise (oggetti in Java).

Perché serve la sincronizzazione?

1. Per prevenire le race condition.
   - Se due thread scrivono su un oggetto senza controllare l'accesso cosa viene scritto?
2. Per garantire la coerenza dei dati condivisi.
   - Se un thread scrive e un thread legge, quest'ultimo che valore legge?

Esempio Hello World Race Condition

```
1   public class HelloWorldRaceCondition {
2       public static void main(String[] args) {
3           var thread1 = new Thread(new CounterTask());
4           var thread2 = new Thread(new CounterTask());
5           thread1.start();
6           thread2.start();
7
8           try {
9               thread1.join();
10              thread2.join();
11          } catch (InterruptedException e) { /* ... */ }
12
13          // Dovrebbe uscire 2000...
14          System.out.printf("Valore finale del contetore: %d\n",
            ↪  CounterTask.counter);
15      }
16  }
17
18  class CounterTask implements Runnable {
19      public static int counter = 0;
20      public void run() {
21          for (var i = 0; i < 1000; i++) { counter++; }
22      }
23  }
```

Esempio Hello World Race Condition (FIX 1)

```
1   class Counter {
2       private int counter = 0;
3
4       public synchronized void increment() {
5           counter++;
6       }
7
8       public int getValue() {
9           return counter;
10      }
11  }
12
13  class CounterTask implements Runnable {
14      public static Counter counter = new Counter();
15
16      public void run() {
17          for (var i = 0; i < 1000; i++) {
18              counter.increment();
19          }
20      }
21  }
```

Esempio Hello World Race Condition (FIX 2)

```
1   class CounterTask implements Runnable {
2       public static int counter = 0;
3       public static Object lock = new Object();
4
5       public void run() {
6           for (var i = 0; i < 1000; i++) {
7               synchronized (lock) {
8                   counter++;
9               }
10          }
11      }
12  }
```

Sincronizzazione ad alto e basso livello

Java offre due livelli di sincronizzazione:

- **Alto livello**: costrutti forniti dal package `java.util.concurrent`. Più avanzati.
  - `Executor()`, `Lock()`, `Semaphore()` e molto altro.
  - Più sicuri contro i problemi della concorrenza.
  - In generale meglio usare questi costrutti.

- **Basso livello**: usa metodi offerti direttamente dalla JVM. Meccanismi più semplici ma meno flessibili.
  - Parola chiave `synchronized` su metodi e oggetti.
  - Monitor su ogni oggetto.
  - Metodi `wait()`, `notify()` e `notifyAll()`.
  - Possono causare problemi di starvation o deadlock.

Nel laboratorio vedremo i costrutti a basso livello.

Sincronizzazione a basso livello

- Ogni oggetto in Java è associato un `monitor`.
  - Il monitor è trasparente, è gestito dalla JVM.
- `synchronized`, usato come metodo o come blocco, acquisisce il monitor.
  - Un thread che chiama tale metodo (o blocco) sarà l'unico ad accedere alla sezione critica. Altri thread verranno bloccati.
  - Al termine del metodo (o blocco) il monitor verrà rilasciato.
- Meglio `synchronized` come metodo o come blocco?
  - Come metodo se l'intero metodo deve essere protetto.
  - Come blocco se solo alcune porzioni devono essere protette.

Attenzione!

Mai usare `synchronized` sui tipi primitivi (`int`, `double`...) e su oggetti immutabili (`String`).

6. Esercizio C

Consegna

Scrivere il server di un'applicazione di chat che usa un protocollo testuale tramite TCP.

Il client è fornito. Il client chiede all'utente un messaggio da inviare. Se nel frattempo riceve un messaggio, lo stampa su schermo.

Il server deve:

- Accettare un nuovo client (il suo nome è la porta).
- Ogni messaggio ricevuto lo invia a tutti i client connessi (tranne che al mittente).

Usare i thread e `synchronized` per gestire più client ed evitare le race condition.

Esempio di interazione

- Client 50743:

```
> java Client.java
Connesso al server! Nome del client: 50743
Ciao come stai? # INVIO
50744: Tutto molto bene
E io invece male # INVIO e poi CTRL+C
>
```

- Client 50744:

```
> java Client.java
Connesso al server! Nome del client: 50744
50743: Ciao come stai?
Tutto molto bene # INVIO
50743: E io invece male
```

- Server:

```
> java Server.java
Server avviato e in ascolto alla porta 8080
Client 50743 connesso!
Client 50744 connesso!
Client 50743 disconnesso dal server!
```

Suggerimenti (pt. 1)

- Partire dallo scheletro su e-Learning!
  - Dare un'occhiata al client per vedere come è strutturato con i thread.
- Il protocollo è molto semplice: ogni messaggio è terminato con un newline. Usare quindi `Scanner.nextLine()` sia per il socket sia per l'input utente.
- Attenzione a quali risorse sono condivise tra thread. Una volta identificate, proteggere l'accesso.

Suggerimenti (pt. 2)

Seguendo lo scheletro, ogni thread gestisce un client. A ogni messaggio ricevuto, il metodo `run()` si comporta come: