UNIVERSITÀ DI PISA

Programmazione di Reti
Corso B

22 Febbraio 2016

Lezione 1(a)

Contatti

- Alina Sirbu
  - alina.sirbu@unipi.it
  - Stanza 331 DO – Lunedì 11:00 - 13:00 su appuntamento
- Alessandro Lulli
  - lulli@di.unipi.it
- Lezioni
  - Martedì – ‘teoria’ – si spiegano i concetti
  - Lunedì – pratica – in laboratorio – si lavora individualmente ai compiti

Corso

- elearning.di.unipi.it - corso “Laboratorio di reti B”
- iscrizione obbligatoria - chiave “LabRetiB”
- slide
- compiti
  - 1 per settimana – sottomissione fino a 2 settimane dopo (Domenica sera)
  - valutati – unico voto alla fine
  - contano come bonus al voto finale

Contenuti

- 3 settimane – Programmazione multithreaded
- 3 settimane – Programmazione di rete
- Approfondimento
- 4 settimane – Programmazione di rete ad alto livello (RMI)
- Esempi e approfondimenti

Bibliografia

- Goetz et al, “Java Concurrency in Practice”, Addison-Wesley
- Lewis & Berg, “Multithreaded programming with java technology”, Sun Microsystems
- Oaks & Wong, “Java Threads”, O’Reilly
- Harold, “Java Network Programming”, O’Reilly

- Java concurrency tutorial: https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- Java networking tutorial: https://docs.oracle.com/javase/tutorial/networking/index.html

- Sito del corso A - su Moodle - compiti e slide

Valutazione

- Un voto unico annuale per il corso “Reti di calcolatori e laboratorio”
- Il voto di laboratorio
  - compiti - bonus
  - progetto
    - 7 giorni prima della prova scritta – obbligatorio
    - si accede al scritto solo se il progetto e sufficiente
  - orale (solo se passata la prova scritta)
    - discussione progetto e altri argomenti del corso

Esempi ed esercizi

- Eclipse IDE (ambiente di sviluppo) o editore di testo + *command line*
- Codice in inglese
  - nomi dei variabili
  - nomi delle classe e metodi
  - commenti
- Obbligatorio fare delle domande

Perché

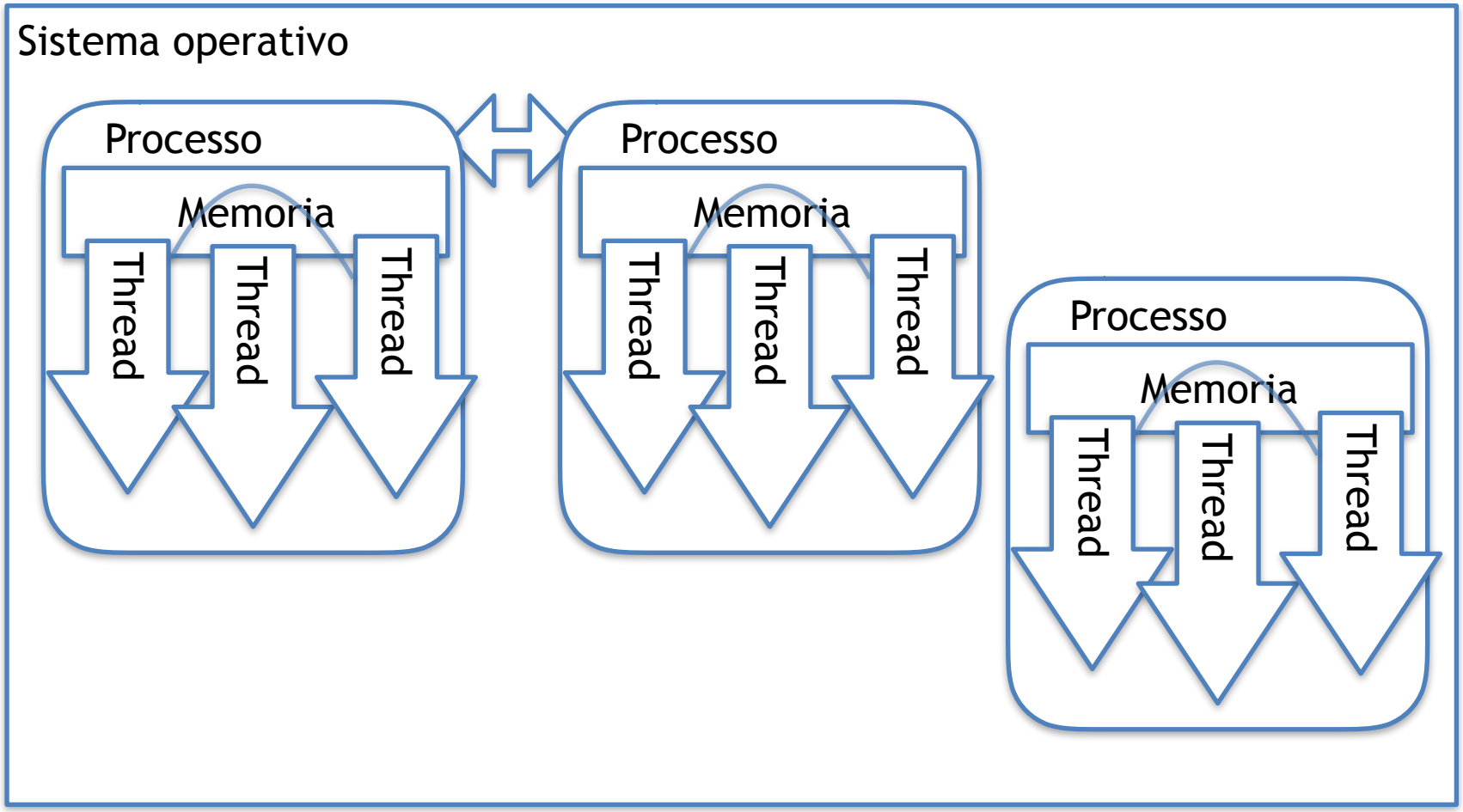
- **Tutte** le applicazioni moderne usano multithreading
  - Basi di dati
  - *Software design*
  - *Interface design*

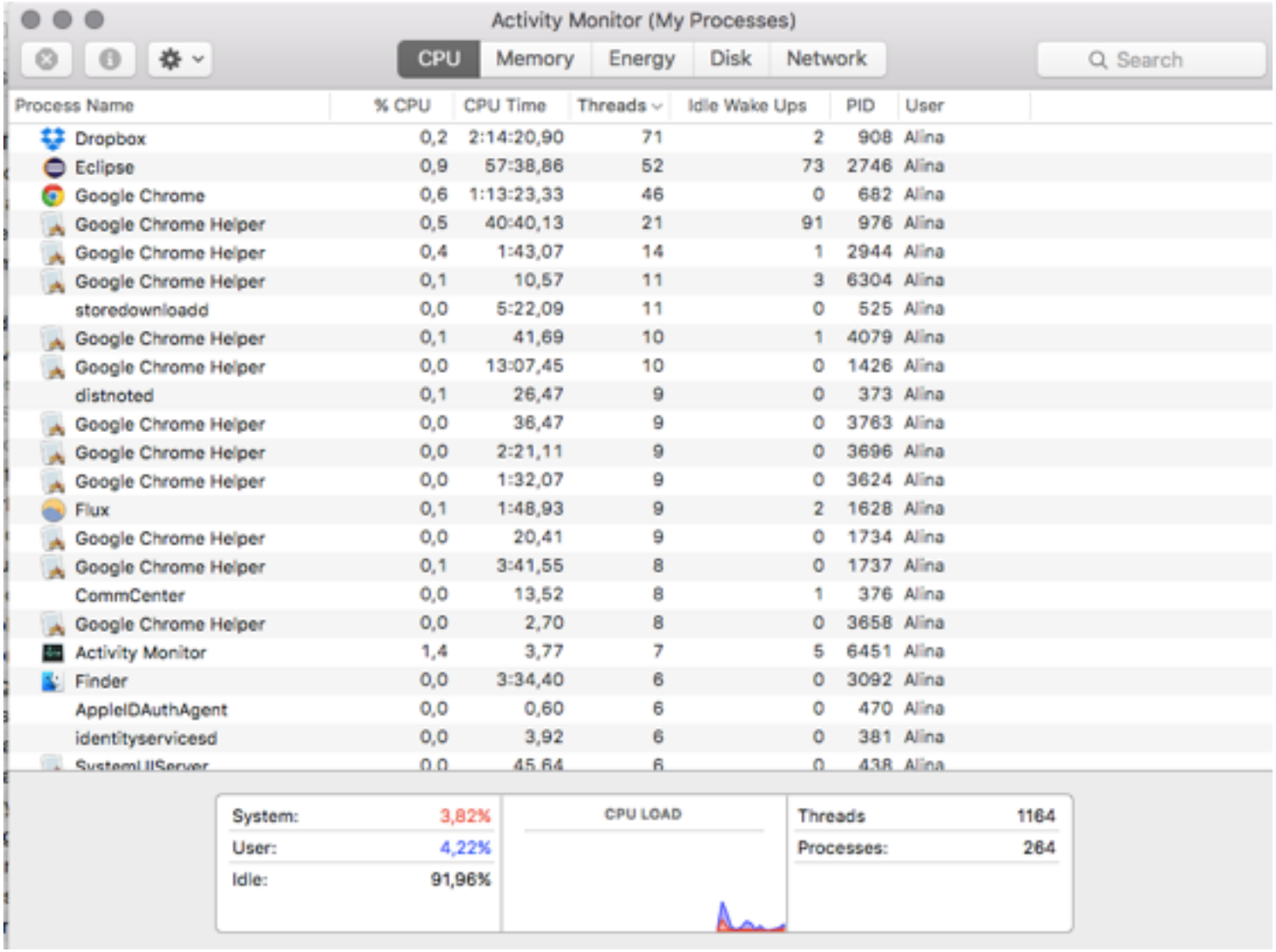
- **Molte** applicazioni moderne usano programmazione di reti - cliente e server
  - *Gaming industry*
  - *Chat* e *social network*
  - *Cloud computing*

Multithreading

- **processo**
  - programma (insieme di istruzioni) eseguito indipendentemente nel sistema operativa, con un spazio di memoria privata, file handler privati, e impostazioni di sicurezza
  - comunicano tramite *socket*, *pipe* o *file* - anche a distanza
- **thread**
  - insieme di istruzioni consecutive eseguite dentro un processo - filo di esecuzione, processo *lightweight*
  - condivide lo spazio di memoria e *file handler* con gli altri *thread* dello stesso processo
  - comunicano facilmente usando *shared memory*
- nel sistema operativo: ogni applicazione ha almeno un processo
- ogni processo ha almeno un *thread* - inizia e finisce con la funzione `main()`

Processi e *thread*

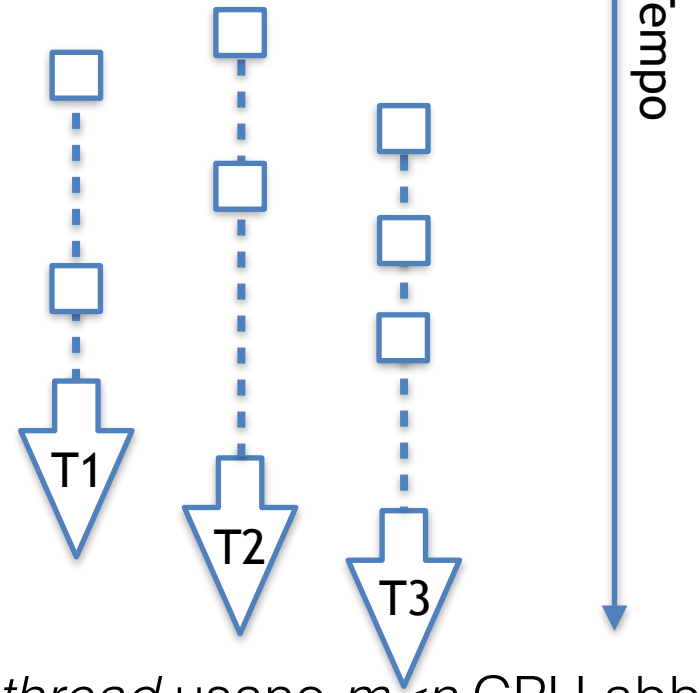




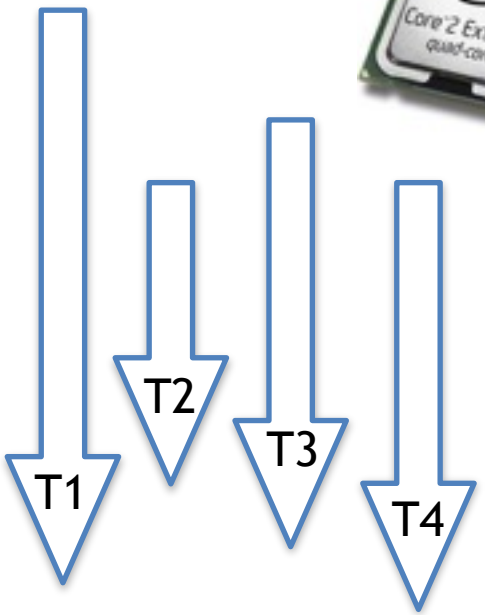
Quasi tutti i processi usano più di un *thread* (un'eccezione e la *command line - bash)*

Motivazione

- **Concorrenza**
  - n *thread* condividono lo stesso CPU



- **Parallelismo**
  - *n thread* usano *n* CPU (core)



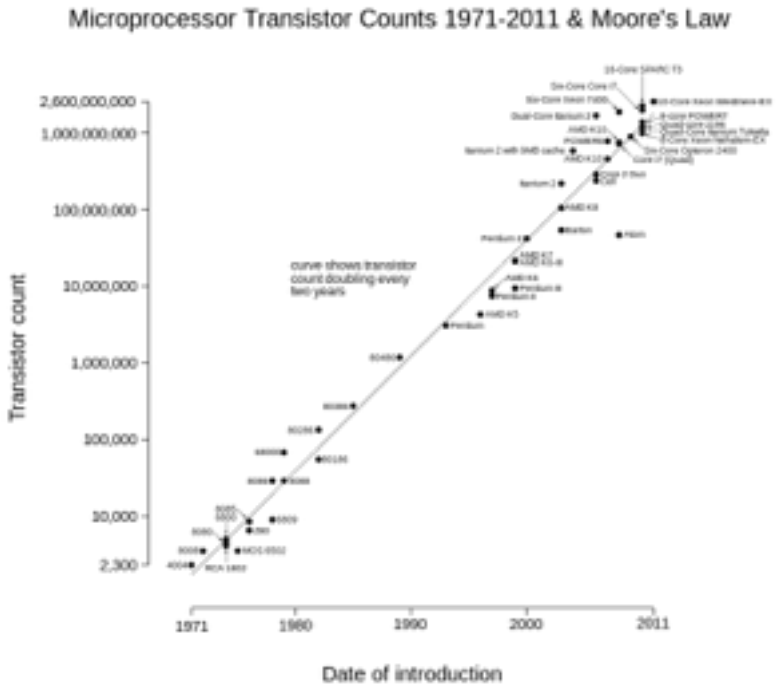
- Se *n thread* usano *m*<*n* CPU abbiamo concorrenza **e** parallelismo allo stesso tempo
- per il programmatore il numero di CPU disponibili non fa differenza

Motivazione

- **Concorrenza**
  - **Reattività** migliore - GUI
  - Uso migliore delle risorse - quando un *thread* si blocca gli altri continuano
  - *Fairness* - tutti gli utenti/ *thread* hanno accesso alle risorse senza aspettare (concorrentemente)
  - Semplifica programmi complessi - ogni *thread* responsabile di un *task* semplice
- **Parallelismo**
  - Uso migliore delle risorse - *core*
  - Computazioni più veloci
  - Possibile solo per piattaforme *multiprocessor* (virtualmente ogni processore moderno)

La legge di Moore

- La velocità dei CPU si raddoppia ogni due anni
  - Non più valida - i CPU hanno raggiunto i loro limiti fisici
  - Adesso la potenza dei computer cresce istallando più CPU (core) in una sola machina
  - Parallelizzazione diventa onnipresente



La legge di Amdahl

- Un programma ha un tempo di esecuzione su 1 CPU uguale a T
- Quale sarà il tempo di esecuzione su N CPU
- F - frazione del programma che deve essere eseguito serialmente
- T'>Ts+Tp/N =TxF+ Tx(1-F)/N=

= Tx(F+(1-F)/N)

- Il limite sta nel nostro programma

Context switching

- Il scheduler (sistema operativo) decide quale *thread* mettere in esecuzione
- Strategia: *time slicing* - ogni *thread* ha una “fetta” di tempo a disposizione, poi viene sospeso
- *Context switching*: la sospensione di un thread e la ripresa di un altro *thread*
  - Un’operazione relativamente costosa

Synchronous vs asynchronous

- 2 modi di effettuare un'operazione
- **synchronous**
  - avvio e eseguo l'operazione fino ad arrivare al risultato
  - applicazioni con un solo thread
- **asynchronous**
  - avvio l'operazione da eseguire da un altro, e comincio a fare un'altra cosa
  - devo controllare più tardi il risultato
  - applicazioni con thread multipli

Blocking vs non-blocking

- Qualche operazioni possono aver bisogno di tempo per concludere
  - Open file gia aperto da un'altra applicazione
  - Leggere data dalla rete
  - etc.
- Due modi di eseguire queste operazioni
  - **blocking**: avvio, mi fermo e non faccio niente fino che l'operazione è avvenuta con successo.
  - **non-blocking:** provo di avviare, se funziona eseguo l'operazione, altrimenti faccio altro e ritorno più tardi

Interfaccia Java

- Classe: definizione di un tipo di oggetti: attributi (proprietà) e metodi (azioni disponibili)
- Oggetti comunicano tra di loro tramite i metodi - interfacciano
- Interfaccia*:* insieme di metodi che un oggetto deve avere per poter essere usato in un modo predefinito
- Interfaccia in Java - definizione dei metodi, senza l'implementazione
- Una Classe può implementare l'interfaccia per definire un tipo di oggetto

```
public interface GeometricShape {
    public double getSurface();
    public double getXCoordinate();
    public double getYCoordinate();

}

public class ShapeHandler {

    public void handle(GeometricShape s){
        System.out.format("I am working with "
                + "a shape located at (%f,%f), "
                + "with a surface of %f",
                s.getXCoordinate(),
                s.getYCoordinate(),
                s.getSurface());
    }
}
```

```
public class Circle implements GeometricShape{
    private double radius;
    private double x,y;
    public Circle(double radius, double x,double y){
        this.radius=radius;
        this.x=x;
        this.y=y;
    }
    @Override
    public double getSurface() {
        return this.radius*this.radius*Math.PI;
    }
    @Override
    public double getXCoordinate() {
        return this.x;
    }
    @Override
    public double getYCoordinate() {
        return this.y;
    }
}
```

```
public static void main(String[] args){
        GeometricShape shape= new Circle(13,0,0);
        ShapeHandler h = new ShapeHandler();
        h.handle(shape);
}
```

Creazione *thread* in Java

- Definizione del *task* in due modi:
  - Estendere la classe j`ava.lang.Thread`
  - Implementare l'interfaccia `java.lang.Runnable`
- Creazione oggetto `Thread`
- Richiamare metodo `start`()

Classe java.lang.Thread

```
public class MyFirstThread extends Thread {

    public void threadPrint(String message){
        System.out.println("Thread “
          +Thread.currentThread().getId()+": “+message);
  }

    public void printInfo(){
        this.threadPrint("I am a new thread.");
        this.threadPrint(this.toString());
        this.threadPrint("My name is "+ this.getName());
        this.threadPrint("My priority is "+ this.getPriority());
    }

    public void run(){
        this.printInfo();
    }
}
```

```
public static void main(String[] args) {

    for (int i=0;i<10;i++){
        MyFirstThread thread=new MyFirstThread();
        thread.start(); //not thread.run()
    }
}
```

```
Thread 9: I am a new thread.
Thread 12: I am a new thread.
Thread 11: I am a new thread.
Thread 11: Thread[Thread-2,5,main]
Thread 10: I am a new thread.
Thread 13: I am a new thread.
Thread 11: My name is Thread-2
Thread 12: Thread[Thread-3,5,main]
Thread 12: My name is Thread-3
Thread 12: My priority is 5
Thread 9: Thread[Thread-0,5,main]
Thread 9: My name is Thread-0
Thread 9: My priority is 5
Thread 15: I am a new thread.
Thread 15: Thread[Thread-6,5,main]
Thread 15: My name is Thread-6
Thread 15: My priority is 5
Thread 14: I am a new thread.
Thread 17: I am a new thread.
Thread 17: Thread[Thread-8,5,main]
Thread 17: My name is Thread-8
Thread 11: My priority is 5
Thread 13: Thread[Thread-4,5,main]
Thread 13: My name is Thread-4
Thread 10: Thread[Thread-1,5,main]
Thread 13: My priority is 5
Thread 18: I am a new thread.
Thread 17: My priority is 5
Thread 14: Thread[Thread-5,5,main]
Thread 16: I am a new thread.
Thread 14: My name is Thread-5
Thread 14: My priority is 5
Thread 18: Thread[Thread-9,5,main]
Thread 18: My name is Thread-9
Thread 10: My name is Thread-1
Thread 18: My priority is 5
Thread 16: Thread[Thread-7,5,main]
Thread 10: My priority is 5
Thread 16: My name is Thread-7
Thread 16: My priority is 5
```

`start()` e `run()`

- `start()` crea un nuovo *thread (processo lightweight)* che esegue le istruzioni del metodo `run()`
- `run()` esegue le istruzioni del metodo `run()` nel thread già esistente - **NO MULTITHREADING**
- **mai usare** `run()` **direttamente**

```
Thread 1: I am a new thread.
Thread 1: Thread[Thread-0,5,main]
Thread 1: My name is Thread-0
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-1,5,main]
Thread 1: My name is Thread-1
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-2,5,main]
Thread 1: My name is Thread-2
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-3,5,main]
Thread 1: My name is Thread-3
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-4,5,main]
Thread 1: My name is Thread-4
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-5,5,main]
Thread 1: My name is Thread-5
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-6,5,main]
Thread 1: My name is Thread-6
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-7,5,main]
Thread 1: My name is Thread-7
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-8,5,main]
Thread 1: My name is Thread-8
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-9,5,main]
Thread 1: My name is Thread-9
Thread 1: My priority is 5
```

Interfaccia java.lang.Runnable

```
public class MyFirstRunnable implements Runnable{

    public void threadPrint(String message){
        System.out.println("Thread “+
            Thread.currentThread().getId()+": "+message);
    }

    public void printInfo(){
        this.threadPrint("I am a new thread.");
        this.threadPrint(Thread.currentThread().toString());
        this.threadPrint("My name is "+
            Thread.currentThread().getName());
        this.threadPrint("My priority is "+
            Thread.currentThread().getPriority());

    }
    public void run(){
        this.printInfo();
    }
}
```

```
public static void main(String[] args) {
    for (int i=0;i<10;i++){
        MyFirstRunnable runnable= new MyFirstRunnable();
        Thread thread=new Thread(runnable);
        thread.start();
    }
}
```

`Thread` e `Runnable`

- `Thread` più semplice per applicazioni piccole
- `Runnable` più flessibile ed elegante
  - si può estendere un'atra classe di base
  - la logica del *thread* (il *task*) è separata dall'oggetto `Thread`
  - usabile con meccanismi nuovi di multithreading (Java > 5)

Esercizio

- Consideriamo 2 file: matrix1.txt, matrix2.txt.
- Creiamo un thread che legge le due matrici, calcola la loro somma e la scrive sul schermo.
- Creiamo un secondo thread che fa la moltiplicazione.
- Avviamo i due thread concorrentemente.
- [Non è efficiente leggere i file 2 volte, questo e solo un'esercizio. Domani parliamo di memoria condivisa.]