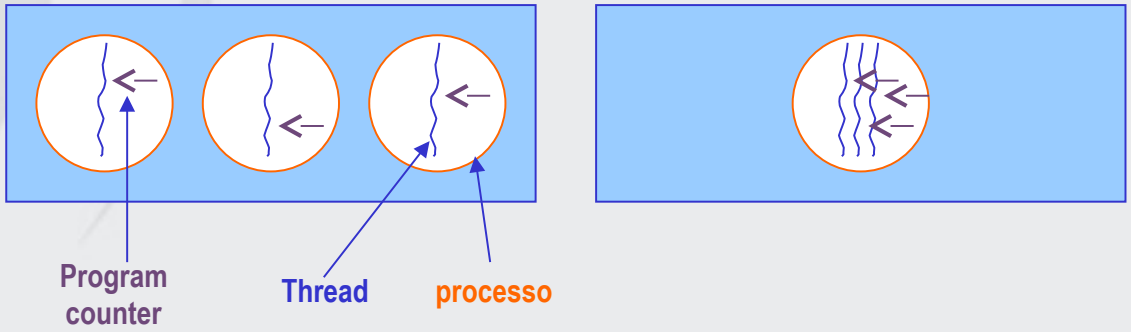
Thread

Introduzione ai thread
Modelli di programmazione
I Thread di Java

Thread

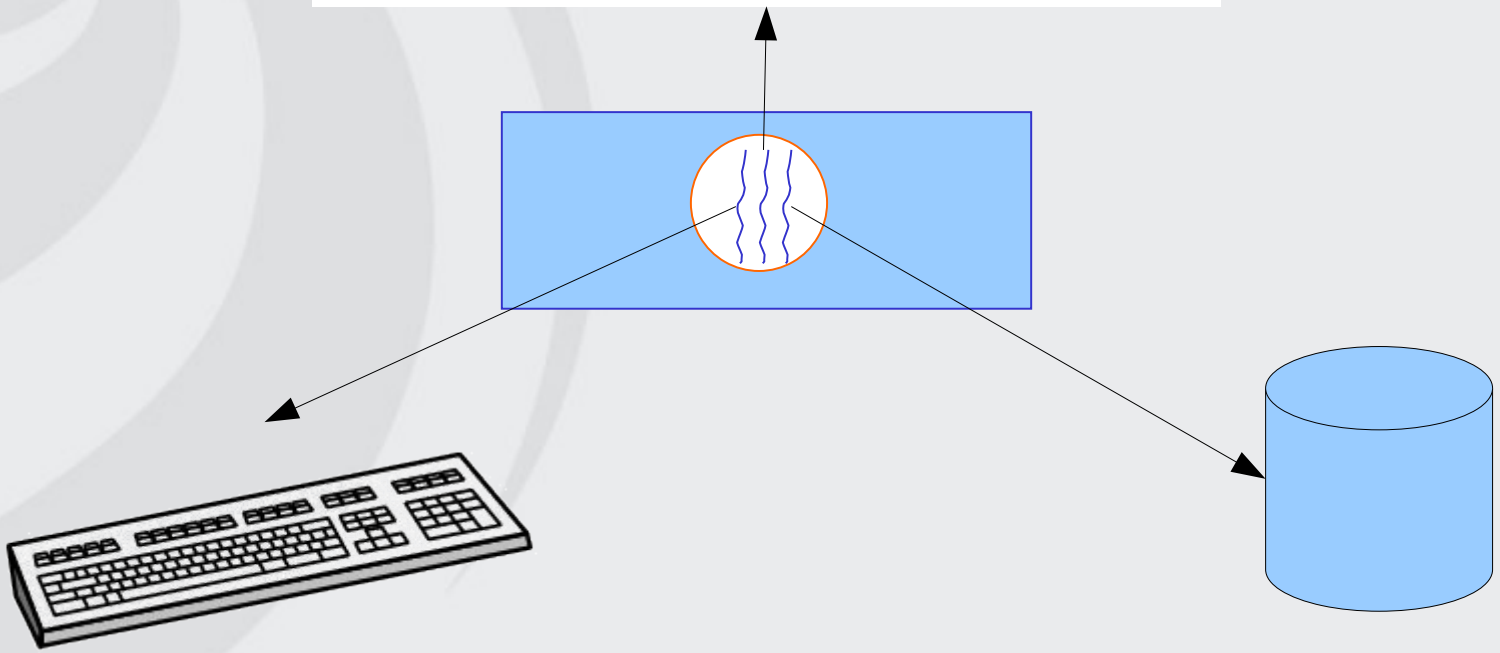
- Molti sistemi operativi forniscono il supporto per definire sequenze di controllo multiple all'interno di un singolo processo
- Queste sequenze di controllo sono solitamente chiamate **thread** o **processi leggeri**



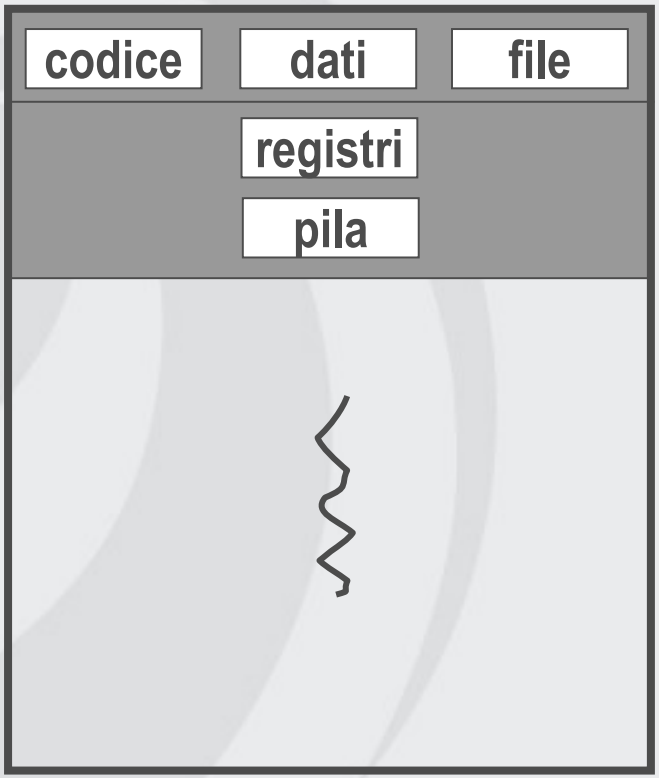
Thread

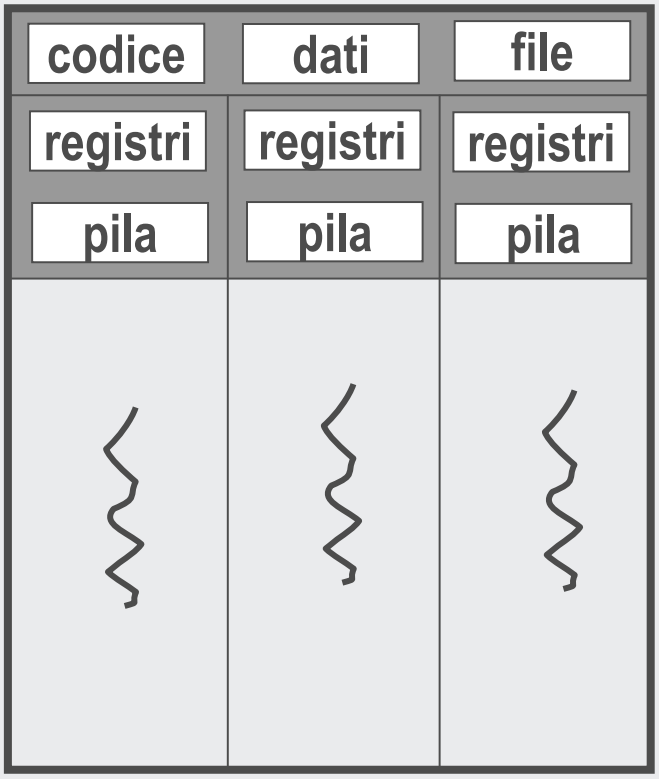
ort service is implemented by a **transport protocol** used between the two t
n some ways, transport protocols resemble the data link protocols we studied
oth have to deal with error control, sequencing, and flow control, among othe

significant differences between the two also exist. These differences are due t
ties between the environments in which the two protocols operate, as shown i
a link layer, two routers communicate directly via a physical channel, whereas
layer, this physical channel is replaced by the entire subnet. This difference ha
implications for the protocols, as we shall see in this chapter.



Processi a thread singoli e multipli





Vantaggi

- Tempo di risposta
- Condivisione delle risorse
- Economia
- Uso di più unità di elaborazione

Thread a livello utente

- La gestione dei thread è fatta a livello utente tramite librerie:
  - POSIX Pthreads
  - Java threads
  - C-threads (Mach)
  - API di win32

Pthread

```c
#include <stdio.h>
#include <pthread.h>

void* test_thread(void* pt)
{
   printf("thread: %s\n", (char *) pt);
   pthread_exit(pt);
}
int main(int argc, char *argv[])
{
   pthread_t thread;
   pthread_attr_t attr;
   void *result;
   pthread_attr_init(&attr);
   if(pthread_create(&thread, &attr, test_thread,  argv[1])) exit(1);
   pthread_join(thread, &result);
   if(result) printf("%s\n", (char*) result);
   return 0;
}
```

Thread a livello utente

- Vantaggi
  - La commutazione fra i thread non richiede l'intervento del nucleo
  - Lo scheduling dei thread è indipendente da quello del nucleo
  - Lo scheduling può essere ottimizzato per la specifica applicazione
  - Sono indipendenti dal sistema operativo in quanto implementati come libreria
- Problemi
  - Le chiamate di sistema sono bloccanti
  - Non si sfrutta un eventuale parallelismo hardware

Thread a livello del nucleo

- Sono gestiti direttamente dal sistema operativo
  - Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X
- Vantaggi
  - In un sistema multiprocessore il nucleo può assegnare thread dello stesso processo a CPU diverse
  - In caso di chiamate di sistema si blocca il singolo thread, non l'intero processo
- Problemi
  - La commutazione è costosa
    - Non vale più uno dei vantaggi dell'uso dei thread

Problemi di schedulazione

- **Processo A 1 thread**
- **Processo B 100 thread**

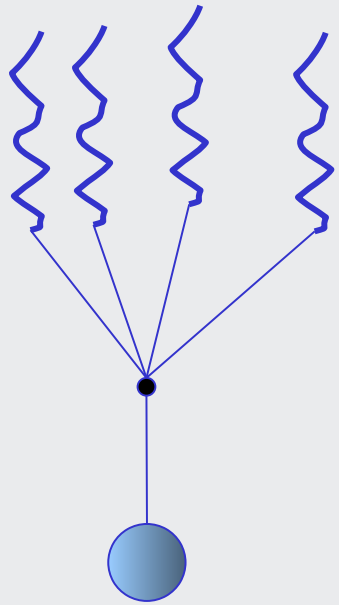
- **Thread utente**
  - **A e B ottengono lo stesso tempo macchina**
    - Ogni thread di B un centesimo dell'unico di A
- **Thread di sistema**
  - **Ogni thread ottiene lo stesso tempo**
    - A ottiene un centesimo del tempo di B

Modelli di programmazione multithread

- Molti a uno
- Uno a uno
- Molti a molti

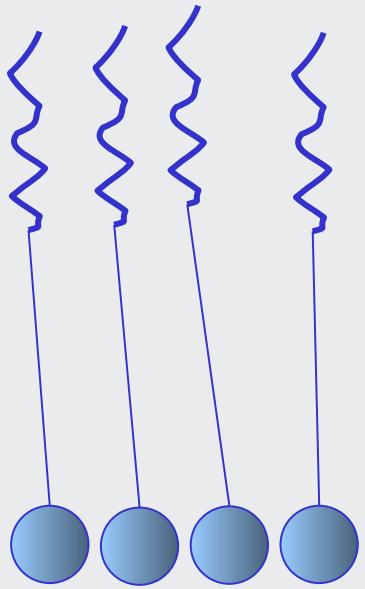
Molti a uno

- Molti thread di livello utente sono mappati in un singolo thread del nucleo
- Esempi
  - Solaris Green Thread
  - GNU Portable Thread



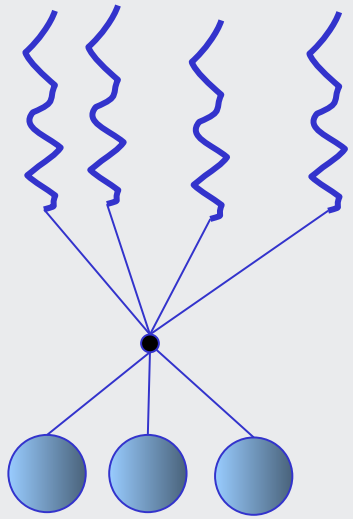
Uno a uno

- Ad ogni thread di livello utente corrisponde un singolo thread del nucleo
- Esempi
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 e successivi



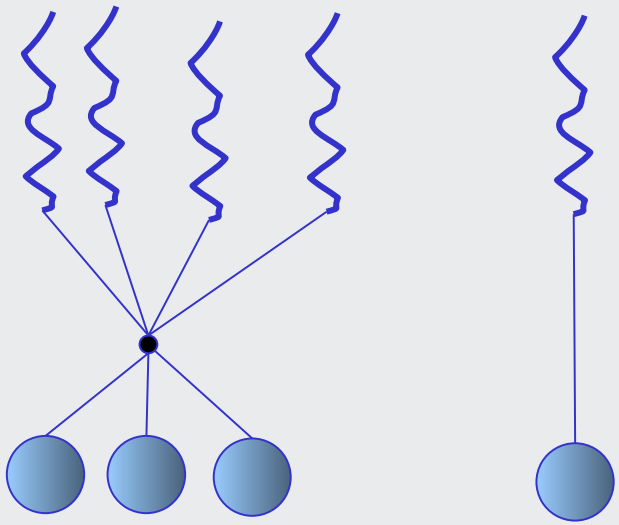
Molti a molti

- Consente a molti thread di livello utente di essere mappati in molti thread del nucleo
- Permette al sistema operativo di creare un numero sufficiente di thread a livello nucleo
- Solaris precedente la versione 9
- Windows NT/2000 con la libreria *ThreadFiber*



Modello a due livelli

- È simile al modello molti-a-molti, ma permette ad un thread utente di essere associato ad un thread del nucleo
- Esempi
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 e precedenti



Problemi di programmazione

- Semantica di `fork()` ed `exec()`
- Cancellazione di thread
- Trattamento di segnali
- Gruppi di Thread
- Dati specifici di Thread
- Schedulazione

Semantica di fork() ed exec()

- La fork() deve duplicare tutti i thread o solo quello chiamante?

Cancellazione

- Cancellazione di un thread prima della sua fine naturale
- Due diversi approcci:
  - Cancellazione **asincrona**: il thread interessato viene fermato immediatamente
  - Cancellazione **differita**: ogni thread controlla periodicamente se deve terminare

Trattamento di segnali

- I segnali sono usati in UNIX per comunicare ad un processo il verificarsi di un particolare evento
- Tutti i segnali seguono lo stesso schema
  - Il segnale è generato da un particolare evento
  - Il segnale è inviato ad un processo
  - Il segnale viene gestito
- Chi gestisce il segnale?
  - Il segnale viene inviato al thread a cui si riferisce
  - Ogni thread riceve il segnale
  - Solo alcuni thread ricevono il segnale
  - Esiste un thread particolare che gestisce tutti i segnali

Segnali

osboxes@osboxes: ~

```
$ kill -SIGINT 8008                    # man kill per informazioni

$ kill -SIGINT 8008
```

osboxes@osboxes: ~

```
$ trap -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5)
 6) SIGABRT      7) SIGEMT       8) SIGFPE       9) SIGKILL     10)
11) SIGSEGV     12) SIGSYS      13) SIGPIPE     14) SIGALRM     15)
16) SIGURG      17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20)
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU     25)
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGPWR      30)
31) SIGUSR2     32) SIGRTMAX

$ echo $$
8008
$ trap "echo ricevuto > /dev/stderr" SIGINT
$ ricevuto                     # man bash - sezione trap per informazioni

$ ricevuto
```

Funzione signal

```
#include <signal.h>

void gestore1(int s)
{
    printf("gestore1: %d\n", s);
}
void gestore2(int s)
{
    printf("gestore2: %d\n", s);
}
int main(int argc, char *argv[])
{
    printf("pid=%d\n", getpid());
    signal(1, gestore1);
    signal(30, gestore2);
    signal(31, gestore2);
    scanf("%*s");
    return 0;
}
```

Segnali

```
osboxes@osboxes: ~
$ kill -30 4788

$ kill -31 4788

$ kill -1 4788

$ kill -9 4788
```

```
osboxes@osboxes: ~
$ ./sig
pid=4788
gestore2: 30
gestore2: 31
gestore1: 1
Killed
$
```

Gruppi di Thread

- Si creano un certo numero di thread che attendono di lavorare
- Vantaggi:
  - È più veloce usare un thread esistente che crearne uno nuovo
  - Limita il numero di thread esistenti

Dati specifici dei thread

- **Permette ad ogni thread di avere dei propri dati**
  - **Si può per esempio associare un identificatore diverso ad ogni thread**
  - **Attenzione quando si usano gruppi di thread**
    - La reinizializzazione dei dati potrebbe non essere corretta

Windows XP

- Modello uno a uno
- Ogni thread contiene
  - Un thread ID
  - Un insieme di registri
  - Pile separate per il modo utente ed il modo nucleo
  - Dati privati
- Registri, pile, e memoria sono definiti come il **contesto** dei thread

Linux

- Nella terminologia di Linux si parla di *task* invece che di *thread*
- La creazione viene fatta attraverso la chiamata di sistema `clone()`
- `clone()` permette ad un task figlio di condividere lo spazio di indirizzi del task genitore (processo)
  - La fork diventa un caso particolare della clone

I thread di Java

- I thread di Java sono gestiti dalla JVM
- Possono essere creati in due modi:
  - Estendendo la classe Thread
  - Implementando l'interfaccia Runnable

La classe Thread

- **Costruttori:**
  - `Thread( threadName )`
    - Crea un thread con il nome specificato
  - `Thread()`
    - Crea un thread con un nome di formato predefinito: Thread-1, Thread-2, ...
- **Metodi:**
  - `run()`
    - “fa il lavoro effettivo” del thread
    - Deve essere sovrascritto nelle sottoclassi
  - `start()`
    - Lancia l’esecuzione del thread, permettendo quindi il proseguimento dell’esecuzione del chiamante
    - Richiama il metodo `run`
    - È un errore chiamare due volte `start` riguardo allo stesso thread

Estensione della classe Thread

```
class Worker1 extends Thread {
    public void run() {
        System.out.println("I Am a Worker Thread");
    }
}

public class FirstThreadTester {
    public static void main(String args[]) {
        Worker1 runner = new Worker1();
        runner.start();

        System.out.println("I Am The Main Thread");
    }
}
```

L'interfaccia Runnable

```
public interface Runnable {
    public abstract void run();
}
```

- Si noti che anche la classe Thread implementa l'interfaccia Runnable

Implementazione di Runnable

```
class Worker2 implements Runnable {
    public void run() {
        System.out.println("I Am a Worker Thread ");
    }
}

public class SecondThreadTester {
    public static void main(String args[]) {
        Runnable runner = new Worker2();
        Thread thread = new Thread(runner);
        thread.start();

        System.out.println("I Am The Main Thread");
    }
}
```

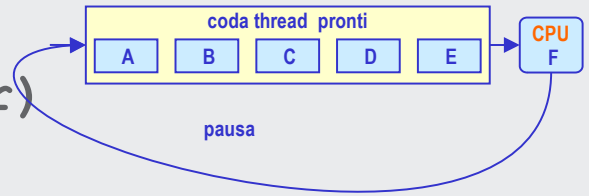
Creo un Thread a partire da un oggetto Runnable

Lambda Runnable (Java 8)

```
public class ThirdThreadTester {

    public static void main(String args[]) {
        Runnable runner = () -> {
            System.out.println("I Am a Worker Thread");
        };
        new Thread(runner).start();

        System.out.println("I Am The Main Thread");
    }

}
```

Priorità dei Thread

- **Ogni applicazione o applet Java è multithread**
  - **La priorità dei Thread va da 1 a 10**
    - Thread.MIN_PRIORITY (1)
    - Thread.NORM_PRIORITY (5 default)
    - Thread.MAX_PRIORITY (10)
    - Ogni nuovo thread eredita la priorità di chi lo ha creato
- **Timeslice**
  - **Ogni thread ottiene un quanto del tempo del processore per l'esecuzione**
    - Al termine il processore passa al prossimo thread di pari priorità (se esiste) schedulazione di tipo round-robin
- **Metodi per controllare la priorità:**
  - `setPriority(int priorityNumber)`
  - `getPriority()`
- **Metodi per il controllo dello scheduling**
  - `yield()` **viene rilasciata volontariamente la CPU**



Esempio priorità

```
public class TestPriority {

  public static void main(String[] args) {

    Thread[] threads = new Thread[9];
    for(int i=0; i<threads.length; i++)
      threads[i] = new Work(9-i);
      // il parametro è la priorità
    Thread th = Thread.currentThread();
    th.setPriority(10);
    for(int i=0; i<threads.length; i++)
      threads[i].start();
    System.out.println(th.getName());
  }
}
```

Esempio priorità – classe Work

```
class Work extends Thread {

  Work(int priority) {
    setPriority(priority);
  }

  public void run() {
    double s=0;
    for(int i=0; i<2000000; i++) {
      s += Math.sqrt(i);
      yield();
    }
    System.out.println(getName());
  }

}
```

```
$ java TestPriority
main
Thread-0
Thread-2
Thread-1
Thread-4
Thread-3
Thread-5
Thread-6
Thread-8
Thread-7
```

Yield

- Se l'ambiente non consente la prelazione si consiglia nei programmi l'uso di yield()

```
public void run() {
    while(true) {
        FaiQualcosa(2000); // vedi slide successiva
        yield();
    }
}
```

Sleep

```
public void FaiQualcosa(long time) {
        // aspetta per un certo tempo
        // time = (long) (Math.random()*time); tempo casuale
        try {
           Thread.sleep(time);
        } catch(InterruptedException e) {}
}
```

```
static void sleep( millisecondi )
```

- Il thread si sospende (non richiede l'uso del processore) per un certo numero di millisecondi
- Nel frattempo possono andare in esecuzione thread a bassa priorità

Altri metodi

`void interrupt()`

- il thread viene bloccato

`boolean isInterrupted()`

- Verifica se per il thread è stata richiesta una interruzione

`static Thread currentThread()`

- Restituisce il thread attualmente in esecuzione

`boolean isAlive()`

- Verifica se il thread è attivo

`void join()`

- Aspetta che il thread specificato termini prima di continuare l'esecuzione
- Se non vengono usati parametri o se il parametro è 0 l'attesa è indefinita altrimenti il parametro specifica il numero di millisecondi di attesa massima
- Può portare al deadlock o alla starvation

Uso di join

```
class JoinableWorker implements Runnable {
    public void run() {
        System.out.println("Worker working");
    }
}

public class JoinExample {
    public static void main(String[] args) {
        Thread task = new Thread(new JoinableWorker());

        task.start();

        try {
          task.join();
        }
        catch (InterruptedException ie) { }

        System.out.println("Worker done");
    }
}
```

Uso di join

```
public class TestJoin {

   public static void main(String[] args)
           throws InterruptedException {

       Thread[] threads = new Thread[8];
       for(int i=0; i<threads.length; i++) {
          threads[i] = new Thread(){ public void run()
              { System.out.println(this);}};
          threads[i].start();
       }
       if(args.length==0) {
          for(int i=0; i<threads.length; i++) {
              threads[i].join();
          }
       }
       System.out.println(Thread.currentThread());
    }

}
```

Test di Join

```
osboxes@osboxes: ~
$ java TestJoin
Thread[Thread-1,5,main]
Thread[Thread-6,5,main]
Thread[Thread-7,5,main]
Thread[Thread-5,5,main]
Thread[Thread-0,5,main]
Thread[Thread-4,5,main]
Thread[Thread-3,5,main]
Thread[Thread-2,5,main]
Thread[main,5,main]

$ java TestJoin 1          # join NON viene eseguito
Thread[Thread-2,5,main]
Thread[Thread-6,5,main]
Thread[Thread-1,5,main]
Thread[Thread-3,5,main]
Thread[main,5,main]
Thread[Thread-4,5,main]
Thread[Thread-5,5,main]
Thread[Thread-0,5,main]
Thread[Thread-7,5,main]
```

Cancellazione di Thread

```
Thread thread = new Thread (new InterruptibleThread());
thread.start();

. . .

thread.interrupt(); // now interrupt it

public class InterruptibleThread implements Runnable {

   public void run() {
      Thread currentThread = Thread.currentThread();
      while(! currentThread.isInterrup()) {
         FaiQualcosa(2000);
      }
      System.out.println("I'm interrupted!");
      // release resources
   }
}
```

Cancellazione di Thread

```
public class ThreadTester extends Thread {

  static public void main(String[] args)
       throws InterruptedException {
    ThreadTester thrd = new ThreadTester();
    thrd.interruptible = args.length>0;
    thrd.start();
    Thread.sleep(500);
    thrd.interrupt();
  }
  boolean interruptible;

  public void run() {
    for(int i=0; i<10000; i++) {
      System.out.println(i);
      if(interruptible && currentThread()
          .isInterrupted()) break;
    }
  }
}
```

Dati specifici dei Thread

```
class Service {
    private static ThreadLocal<Exception> errorCode =
         new ThreadLocal<Exception>();

    public static void transaction() {
        try {
            /* some operation where an error may occur */
        }
        catch (Exception e) {
            errorCode.set(e);
        }
    }
    /* get the error code for this transaction */
    public static Exception getErrorCode() {
        return errorCode.get();
    }
}
class Worker implements Runnable {
    public void run() {
        Service.transaction();
        System.out.println(Service.getErrorCode());
    }
}
```

Dati specifici dei Thread

```
public class MyObject implements Runnable { … }

MyObject obj = new MyObject();
for(int i=0; i<10; i++) {
    new Thread(obj).start();
}
// 10 thread, tutti condividono lo stesso oggetto



for(int i=0; i<10; i++) {
    MyObject obj = new MyObject();
    new Thread(obj).start();
}
// 10 thread, ognuno associato a un oggetto diverso
```

Cosa viene stampato?

```
int value = 0;
void *run(void *param) {
    value = 5;
    return 0;
}
int main(int argc, char *argv[]) {
    pthread_t tid;
    pthread_attr_t attr;
    pid_t pid = fork();
    if (pid == 0) {
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, run, NULL);
        pthread_join(tid, NULL);
        printf("CHILD: value = %d\n", value);
    } else if (pid > 0) {
        wait(NULL);
        printf("PARENT: value = %d\n", value);
    } else { exit(1); }
    return 0;
}
```

thread

Processo figlio

Processo padre

error

?