Introduzione ai thread

Processi

- **Immagine di un processo** (codice, variabili locali e globali, stack, descrittore)
- **Risorse possedute:** (file aperti, processi figli, dispositivi di I/O..),
- L'immagine di un processo e le risorse da esso possedute costituiscono il suo **spazio di indirizzamento.**
- L'allocazione dello spazio di indirizzamento dipende dalla tecnica di **gestione della memoria** adottata. Potrà essere contenuto in tutto o solo in parte nella memoria principale (registri base ed indice, impaginazione, segmentazione).

Processi (*pesanti*)

- **Proprietà dei *processi***: *spazi di indirizzamento distinti*, cioe` non condividono memoria (es.**Unix**)
- **Complessità delle operazioni di *cambio di contesto*** tra due processi: comportano il salvataggio ed il ripristino dello spazio di indirizzamento **(overhead)**. Analogamente per le operazioni di creazione e terminazione di un processo.
- Utilizzo del modello a **scambio di messaggi** (es.Unix)

Processi & thread

Il concetto di processo e` basato su **due aspetti indipendenti:**

- **Possesso delle risorse** contenute nel suo spazio di indirizzamento.
- **Esecuzione**. Flusso di esecuzione, associato a un programma, che puo` condividere la CPU con altri flussi, possiede uno stato e viene messo in esecuzione sulla base della politica di scheduling.

- I due aspetti sono indipendenti e possono essere gestiti separatamente dal S.O.:
  - processo **leggero** (***thread***): elemento cui viene assegnata la CPU
  - processo **pesante** (***processo o task***): elemento che possiede le risorse

Un *thread* rappresenta un ***flusso di esecuzione*** all'interno di un *processo pesante*.

- **Multithreading**: molteplicità di flussi di esecuzione all'interno di un processo pesante.

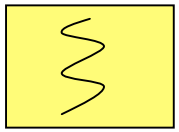
- Tutti i thread definiti in un processo ***condividono*** le risorse del processo, risiedono nello **stesso spazio di indirizzamento** ed hanno **accesso a dati comuni**.

Ogni thread ha:

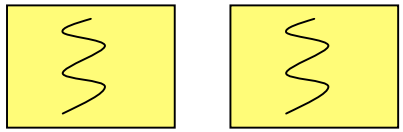
- uno **stato** di esecuzione (running, ready, blocked)
- un **contesto** che è salvato quando il thread non è in esecuzione
- uno **stack** di esecuzione
- uno spazio di memoria privato per le **variabili locali**
- accesso alla memoria e alle risorse del task **condiviso** con gli altri thread.

Vantaggi

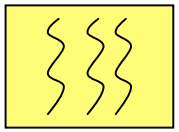
- maggiore efficienza: le operazioni di context switch, creazione etc., sono più economiche rispetto ai processi.
- maggiori possibilità di utilizzo di architetture multiprocessore.



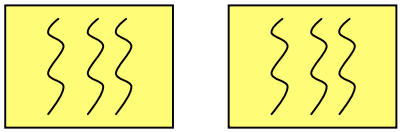
un thread per processo



Processi multipli: un thread per processo



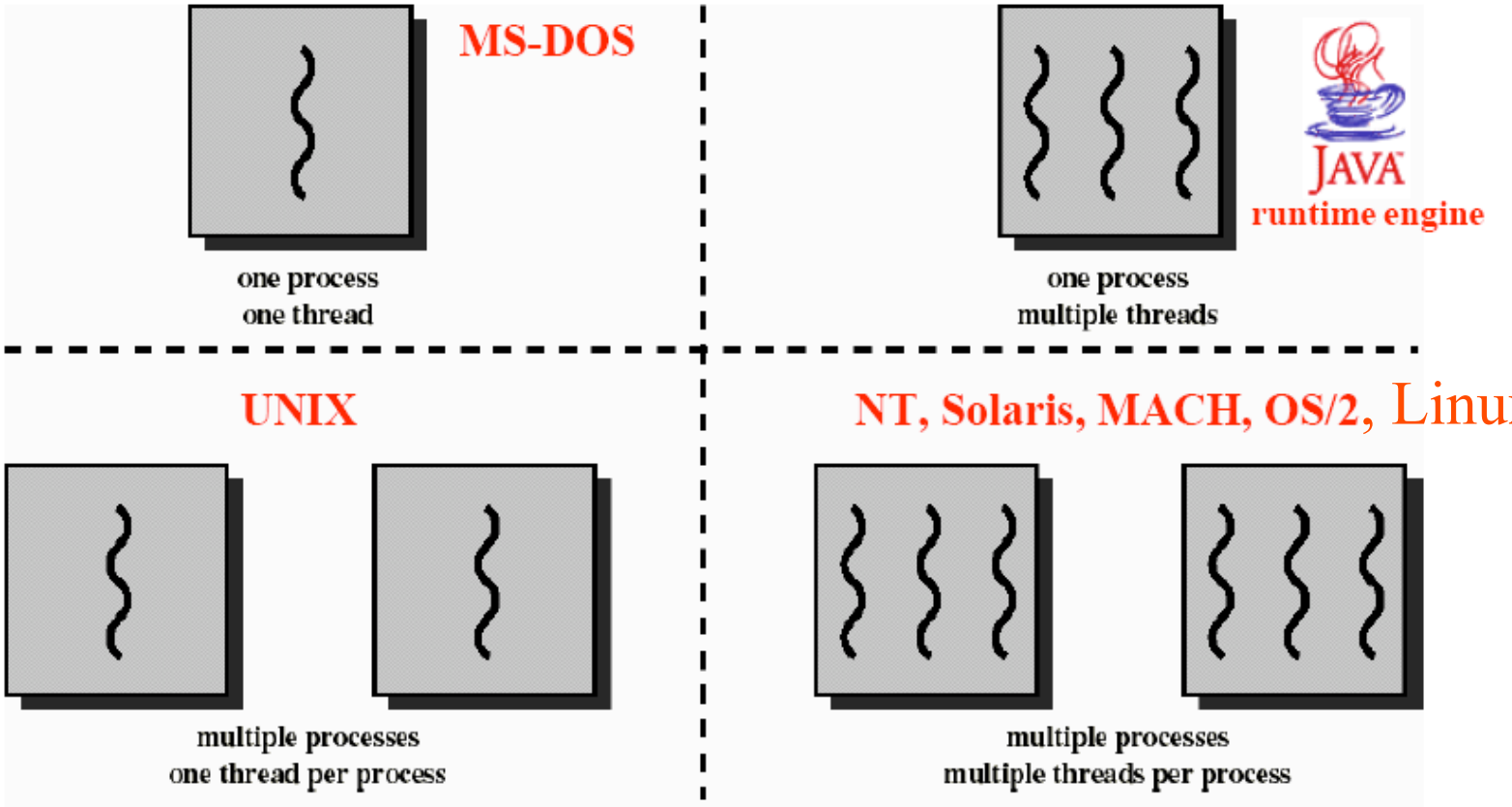
più thread per processo



Processi multipli: più thread per processo

Soluzioni adottate

- **MS-DOS:** un solo processo utente ed un solo thread.
- **UNIX**: più processi utente ciascuno con un solo thread.
- **Supporto run time di Java**: un solo processo, più thread.
- **Linux, Windows NT, Solaris**: più processi utente ciascuno con più thread.


MS-DOS
one process
one thread
JAVA
runtime engine
one process
multiple threads
UNIX
multiple processes
one thread per process
NT, Solaris, MACH, OS/2, Linux
multiple processes
multiple threads per process

- I thread possono eseguire ***parti diverse*** di una stessa applicazione.

Esempio: **web browser**

  - *thread* che scrive il testo sul video
  - *thread* che ricerca dati sulla rete

Esempio: **word processing**

  - thread che mostra i grafici sul video
  - thread che legge i comandi inviati dall'utente
  - thread che evidenzia gli errori di scrittura

- I thread possono eseguire le ***stesse funzioni*** (o funzioni simili) di un'applicazione.

Esempio: **web server**

  - *thread* che accetta le richieste e crea altri *thread* per servirle.
  - *thread* che servono la richiesta (possono essere uguali o diversi a seconda del tipo di richiesta).

Realizzazione dei thread

A livello utente (es. ***Java***)

- Libreria di funzioni (***thread package***) che opera a livello utente e fornisce il supporto per la creazione, terminazione, sincronizzazione dei thread e per la scelta di quale thread mettere in esecuzione (*scheduling*).
- Il sistema operativo ***ignora la presenza dei thread*** continuando a gestire solo i processi.
- Ogni processo parte con un solo thread che può creare nuovi thread chiamando una apposita funzione di libreria.

- Gerarchia di thread o thread tutti allo stesso livello.
- La soluzione è *efficiente* (tempo di switch tra thread), *flessibile* (possibilità di modificare l'algoritmo di scheduling), *scalabile* (modifica semplice del numero di thread).
- Se un thread *si blocca* in seguito ad una **chiamata ad una *funzione del package*** (es. wait), va in esecuzione un altro thread dello stesso processo.
- I thread possono **chiamare delle *system call*** (es. I/O): intervento del sistema operativo che ***blocca il processo*** e conseguentemente l'esecuzione di *tutti i suoi thread*.
- Il S.O. interviene anche nel caso allo ***scadere del quanto di tempo*** assegnato ad un processo (sistemi time sharing).
- Non è possibile sfruttare il parallelismo proprio di *architetture multiprocessore*: un processo (con tutti i suoi thread) è assegnato ad uno dei processori.

Realizzazione di thread

***A livello di nucleo* (es. *NT, Linux*):**

- Il S.O. si fa carico di tutte le **funzioni per la gestione dei thread**. Mantiene tutti i descrittori dei thread (oltre a quelli dei processi).
- A ciascuna funzione corrisponde una ***system call***.
- Quando un thread si *blocca* il S.O. può mettere in esecuzione un altro thread dello *stesso processo*.
- Soluzione ***meno efficiente*** della precedente.
- Possibilità di eseguire thread diversi appartenenti allo stesso processo su unità di elaborazione differenti (*architettura multiprocessore).*

Realizzazione di thread

Soluzione mista (es. Solaris):

- Creazione di thread, politiche di assegnazione della CPU e sincronizzazione a *livello utente.*
- I thread a livello utente sono mappati in un numero (minore o uguale) di thread a livello nucleo.

Vantaggi:

- Thread della stessa applicazione possono essere eseguiti in parallelo su processori diversi.
- Una chiamata di sistema bloccante non blocca necessariamente lo stesso processo

L'implementazione della libreria pthread nel sistema operativo LINUX: *Linuxthreads*

Uso dei thread in Linux: system call native vs. pthread

- Processi leggeri **realizzati a livello kernel**
- System call **clone**:

```
int  clone(int (*fn) (void *arg), void *child_stack, int
     flags, void *arg)
```

➡ **E` specifica di Linux: scarsa portabilita`!**

- Libreria pthread (*LinuxThreads*): funzioni di gestione dei threads, in conformita` con lo standard POSIX 1003.1c (*pthreads*):
  - *Creazione/terminazione threads*
  - *Sincronizzazione threads:lock, [semafori], variabili condizione*
  - *Etc.*

➢ Portabilita`

LinuxThreads

Caratteristiche thread:

- Il thread e` realizzato **a livello kernel** (e` l'unita` di schedulazione
- l'esecuzione di un programma determina la creazione di un thread iniziale che esegue il codice del main (a differenza di POSIX: non c'è *task*).
- Il thread iniziale può creare altri thread, ognuno dedicato all'esecuzione di una funzione data;
- ogni thread ne può creare altri: si crea una gerarchia di thread che condividono uno spazio di indirizzi.
- I thread vengono creati all'interno di un processo per eseguire una funzione
- ogni thread ha il suo PID (distinzione tra *task* e *threads*)
- Gestione dei segnali non conforme a POSIX (Linuxthread):
  - Non c'e` la possibilita` di inviare un segnale a un task.
  - SIGUSR1 e SIGUSR2 vengono usati per l'implementazione dei threads quindi non sono piu` disponibili.

Rappresentazione dei threads

Il thread e` l'unita` di scheduling, ed e` univocamente individuato da un indentificatore (intero):

```
pthread_t tid;
```

Il tipo **pthread_t** e` dichiarato nell'**header file**:

```
<pthread.h>
```

Creazione di thread: pthread_create

Creazione di thread:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,pthread_attr_t *attr,
                   void *(*start_routine)(void *), void * arg);
```

Dove:

- **thread**: e` il puntatore alla variabile che raccogliera` il thread_ID (PID)
- **start_routine**: e` il puntatore alla funzione che contiene il codice del nuovo thread
- **arg**: e` il puntatore all'eventuale vettore contenente i parametri della funzione da eseguire
- **attr**: puo` essere usato per specificare eventuali attributi da associare al thread (di solito: NULL):
  - ad esempio parametri di scheduling: priorita` etc.(solo per superuser!)
  - Legame con gli altri threads (ad esempio: *detached* o no)

Restituisce : 0 in caso di successo, altrimenti un codice di errore (!=0)

LinuxThreads: creazione di threads

Ad esempio:

```
int A, B; /* variabili comuni ai thread che verranno creati*/
void * codice(void *){ /*definizione del codice del thread */ …}
main()
{pthread_t t1, t2;
..
pthread_create(&t1,NULL, codice, NULL);
pthread_create(&t2,NULL, codice, NULL);
..
}
```

➔ Vengono creati due thread (di tid t1 e t2) che eseguono le istruzioni contenute nella funzione codice:

- I due thread appartengono allo stesso *task* (processo) e condividono le variabili globali del programma che li ha generati (ad esempio A e B).

Terminazione di thread: `pthread_exit`

Terminazione di thread:

```
void pthread_exit(void *retval);
```

Dove **`retval`** e` il puntatore alla variabile che contiene il valore di ritorno (puo` essere raccolto da altri threads, v. **`pthread_join`**).

E` una chiamata senza ritorno.

Alternativa: **`return();`**

pthread_join

Un thread puo` sospendersi in attesa della terminazione di un altro thread con:

```
int pthread_join(pthread_t th, void **thread_return);
```

Dove:

- **th**: e` il pid del particolare thread da attendere
- **thread_return**: e` il puntatore alla variabile dove verra` memorizzato il valore di ritorno del thread (v. **pthread_exit**)

Il valore restituto dalla pthread_join indica l'esito della chiamata: se diverso da zero significa che la pthread_join e` fallita (ad es. non vi sono thread figli).

Esempio: creazione di thread

```
/*Linuxthreads: esempio 1.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *my_thread_process (void * arg)
{ /*codice dei thread creati: */
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
}
```

```
main ()
{
  pthread_t th1, th2;
  int retcode;
  if (pthread_create(&th1,NULL,my_thread_process,"1") < 0)
  { fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
  if (pthread_create(&th2,NULL,my_thread_process,"2") < 0)
  { fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }
  retcode = pthread_join(th1, NULL);
  if (retcode != 0)
    fprintf (stderr, "join fallito %d\n", retcode);
  else printf("terminato il thread 1\n);

  retcode = pthread_join(th2, NULL);
  if (retcode != 0)
    fprintf (stderr, "join fallito %d\n", retcode);
  else printf("terminato il thread 2\n);
  return 0;
}
```

Compilazione

Per compilare un programma che usa i linuxthreads:

```
gcc -D_REENTRANT -o prog prog.c  -lpthread
```

```
[aciampolini@ccib48 threads]$ prog
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
terminato il thread 1
terminato il thread 2
[aciampolini@ccib48 threads]$
```

I Thread in Java

I threads in Java

- Ogni programma Java contiene almeno un *singolo thread*, corrispondente all'esecuzione del metodo *main()* sulla *JVM*.
- E' possibile creare dinamicamente nuovi thread *attivando concorrentemente* le loro esecuzioni all'interno del programma.

Due possibilita` di creazione:

1. Thread come oggetti di sottoclassi della classe **`Thread`**
2. Thread come oggetti di classi che implementano l'interfaccia **`runnable`**

Thread come oggetti di sottoclassi della classe **Thread**

- I threads sono oggetti che derivano dalla **classe Thread** (fornita dal package **java.lang).**

- Il metodo **run** della classe di libreria **Thread** definisce *l'insieme di statement Java* che ogni thread (oggetto della classe) eseguirà *concorrentemente* con gli altri thread.

- Nella classe **Thread** l'implementazione del suo metodo **run** è vuota.

- In ogni sottoclasse derivata da **Thread** deve essere ridefinito (*override*) il metodo **run** in modo da fargli eseguire ciò che è richiesto dal programma.

Possibile schema

```
class AltriThreads extends Thread {
  public void run() {
    <corpo del programma eseguito>
    <da ogni thread di questa classe>
    }
}

public class EsempioConDueThreads
{     public static void main (string[] args)
      {  AltriThreads t1=new AltriThreads();
         t1.start();//attivazione del thread t1
         <resto del programma eseguito
            dal thread main>
      }
}
```

- La classe **AltriThread** (estensione di Thread) implenta i nuovi thread *ridefinendo il metodo* **run**.
- La classe EsempioConDueThreads fornisce il **main** nel quale viene creato il thread **t1** come oggetto derivato dalla classe **Thread**.
- Per **attivare** il thread deve essere chiamato il metodo **start()** che invoca il metodo **run()** (il metodo **run()** non può essere chiamato direttamente, ma solo attraverso **start()**).

➔ In questo modo abbiamo creato *due thread concorrenti*:

  - il thread principale, associato al main;
  - il thread t1 creato dinamicamente dal precedente, con l'esecuzione dello statement t1.start() che lancia in concorrenza l'esecuzione del metodo run() del nuovo thread.

E se occorre definire thread che non siano necessariamente sottoclassi di Thread?

Thread come classe che implementa l'interfaccia **runnable**

Interfaccia Runnable: maggiore **flessibilità** → thread come sottoclasse di qualsiasi altra classe

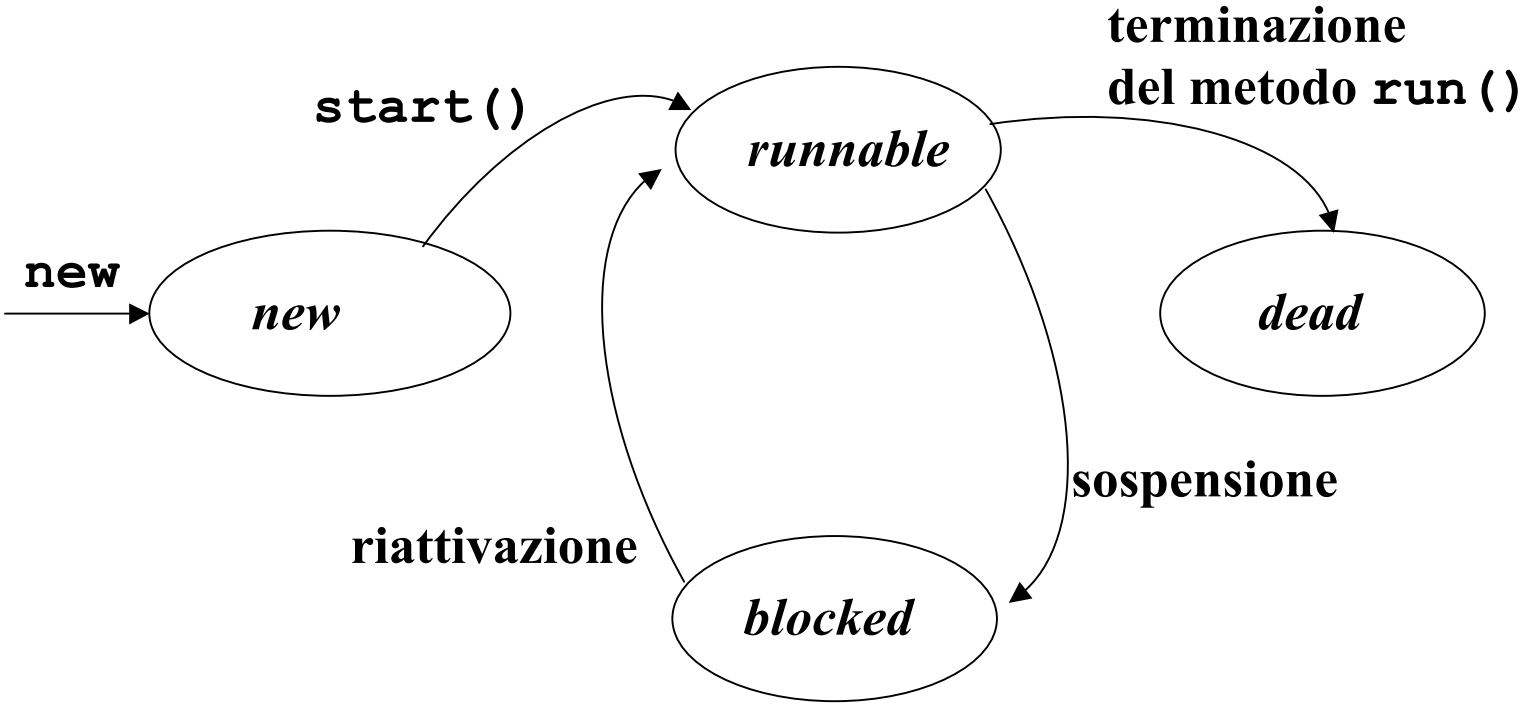
- implementare il metodo **`run()`** nella classe che implementa l'interfaccia **`Runnable`**
- creare un'istanza della classe tramite **`new`**
- creare un'istanza della classe **`Thread`** con un'altra **`new`**, passando come parametro l'istanza della classe che si è creata
- invocare il metodo **`start()`** sul thread creato, producendo la chiamata al suo metodo **`run()`**

Esempio di classe **EsempioRunnable** che *implementa l'interfaccia Runnable* ed è sottoclasse di **MiaClasse**:

```
class EsempioRunnable extends MiaClasse implements Runnable
{
   // non e' sottoclasse di Thread
   public void run()
   {
        for (int i=1; i<=10; i++)
             System.out.println(i + " " + i*i);
   }
}

public class Esempio
{
   public static void main(String args[])
   {
        EsempioRunnable e = new EsempioRunnable();
        Thread t = new Thread(e);
        t.start();
   }
}
```

Grafo di stato di un thread



Priorità e scheduling

- **Preemptive** priority scheduling con priorità fisse (crescenti verso l'alto).
- MIN-PRIORITY, MAX-PRIORITY: costanti definite nella classe thread.
- Ogni thread eredita, all'atto della sua creazione, la priorità del processo padre.
- Metodo **`set-priority`** per modificare il valore della priorità

JVM esegue l'algoritmo di scheduling:

- quando il thread correntemente in esecuzione esce dallo stato *runnable* (sospensione o terminazione);
- quando diventa *runnable* un thread a priorità più alta (preemption).

Metodi per il controllo di thread

- **`start()`** fa ***partire*** l'esecuzione di un thread. La macchina virtuale Java invoca il metodo **`run()`** del thread appena creato
- **`stop()`** ***forza*** la ***terminazione*** dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente ***liberate*** (lock inclusi), come effetto della propagazione dell'eccezione **`ThreadDeath`**
- **`suspend()`** ***blocca*** l'esecuzione di un thread in attesa di una successiva operazione di **`resume()`**. Non libera le risorse impegnate dal thread (lock inclusi)
- **`resume()`** ***riprende*** l'esecuzione di un thread precedentemente ***sospeso***. Se il thread riattivato ha una priorità maggiore di quello correntemente in esecuzione, avrà subito accesso alla CPU, altrimenti andrà in coda d'attesa

- **`sleep(long t)`** blocca per un ***tempo specificato*** (`time`) l'esecuzione di un thread. Nessun *lock* in possesso del thread viene rilasciato.
- **`join()`** ***blocca*** il thread chiamante in attesa della ***terminazione*** del thread di cui si invoca il metodo. Anche con ***timeout***
- **`yield()`** ***sospende*** l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread in ***coda d'attesa***

Il problema di `stop()` e `suspend()`

`stop()` e `suspend()` rappresentano azioni "brutali" sul ciclo di vita di un thread → rischio di determinare situazioni inconsistenti o di blocco critico (***deadlock***)

- se il ***thread sospeso*** aveva acquisito una ***risorsa*** in maniera ***esclusiva***, tale risorsa rimane ***bloccata*** e non è utilizzabile da altri, perché il thread sospeso non ha avuto modo di rilasciare il *lock* su di essa
- se il ***thread interrotto*** stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera ***atomica***, l'interruzione può condurre ad uno ***stato inconsistente*** del sistema

- JDK 1.5, pur supportandoli ancora per ragioni di *back-compatibility*, ***sconsiglia*** l'utilizzo dei metodi **`stop()`**, **`suspend()`** e **`resume()`** (***metodi deprecated***)

Si consiglia invece di realizzare tutte le azioni di ***controllo*** e ***sincronizzazione*** fra thread tramite gli stumenti specifici per la sincronizzazione (object locks, **`wait()`**, **`notify()`**, **`notifyAll()`** e variabili condizione)