

PCAD

Programmazione Concorrente

Algoritmi Distribuiti

Arnaud Sangnier
arnaud.sangnier@unige.it

PROCESSI E THREAD

Processi

- Un processo è un **programma** (di natura statica) in corso di **esecuzione** (di natura dinamica). La sua esecuzione necessita un ambiente :
 - spazio di indirizzamento dedicato
 - input and output (per es. gli standard)
- Più processi possono eseguirsi sulla stessa macchina in un modo quasi-simultaneo
 - Se il sistema rispetta il **tempo condiviso** o è **multi-task**
 - Questo vale anche se c'è un solo processore
 - Il sistema operativo è in carica di allocare le risorse (memoria, tempo processore, input/output etc)
 - Un processo **non ha accesso** allo spazio di indirizzamento degli altri
 - Abbiamo l'**illusione del parallelismo**

Creare processi Unix

- In C, si può creare un nuovo processo grazie alla funzione fork:
 - **pid_t fork(void);**
- Questa funzione:
 - Crea un processo figlio
 - Ritorna un identificatore del processo PID
 - Uno può testare se è nel processo padre o figlio grazie al PID ritornato
 - fork ritorna 0 per il figlio
 - fork ritorna l'identificatore del figlio al padre
- Altre funzioni utili:
 - **pid_t getpid(void);** per avere il PID del processo corrente
 - **pid_t getppid(void);** per avere il PID del processo padre

Creare processi Unix (2)

- In pratica:
 - Tutto è nello stesso codice
 - Si testa il valore ritornato da fork per sapere se siamo nel processo figlio o padre
 - **if(fork()==0) {...}**
 - Questo permette di fare la distinzione le esecuzione del processo padre e del figlio
- Bisogna essere attento ai seguenti fatti
 - Le variabile non sono condivise
 - Alla creazione del figlio, lo spazio di indirizzamento del padre viene copiato

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main() {
    int id=getpid();
    printf("Sono il %d\n",id);
    printf("Je fais un fork\n");
    int idfork=fork();
    if(idfork==0){
        int idfiglio=getpid();
        int idpadre=getppid();
        printf("Sono %d figlio di %d\n",idfiglio,idpadre);
    } else {
        printf("Il mio figlio e %d\n",idfork);
        int stato;
        waitpid(idfork, &stato, 0);
    }
    return 0;
}
```

Le variabile non sono condivise

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int main(){
    int x=0;
    int pid = fork();
    if ( pid == 0 ) {
        x=25;
        printf("Valore di x per il figlio:%d\n",x);
        printf("Fin del processo figlio\n");
    }
    else {
        while(x!=25){
            sleep(2);
            printf("Valore di x per il padre %d\n",x);
        }
        printf("Fine del processo padre\n");
    }
    return 0;
}
```

Fare eseguire un command a un figlio

- Si può usare la funzione `execvp`:

```
int execvp(const char *file, char *const argv[]);
```

- L'array `argv` contiene
 - il nome della command nel primo campo
 - gli argomenti nei campi seguenti
 - **deve finire con NULL**
- non crea nessun nuovo processo: semplicemente lo spazio di indirizzamento viene rimpiazzato con quello del nuovo programma

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main() {
    int id=getpid();
    printf("Sono il processore %d\n",id);
    printf("Faccio un forks\n");
    int idfork=fork();
    if(idfork==0){
        int idfiglio=getpid();
        int idpadre=getppid();
        printf("Sono %d figlio di %d\n",idfiglio,idpadre);
        char *args[]={"ls","-l",NULL};
        execvp(args[0],args);
        printf("Fine exec\n");
    } else {
        printf("Il mio figlio e %d\n",idfork);
        int stato;
        waitpid(idfork, &stato, 0);
        printf("Fine del figlio\n");
    }
    return 0;
}
```


Comunicazione fra processi

- Per comunicare fra processi, si possono usare dei 'tubi' (pipe)
- I tubi sono dei canali unidirezionali considerati come dei descrittori di file
- Un processo può scrivere ad una estremità ed un altro può leggere all'altra estremità



- Come sono visti come descrittori di file, alla creazione di un figlio, il figlio recupera la tabella con i descrittori del padre ed ha accesso al tubo

Usare tubi

- La commanda

```
int pipe(int fds[2])
```

crea un tubo e mette le sue due estremità nel array fds

- Si scrive su fds[1] e si legge su fds[0]
- Dopo il fork, **entrambi** processi possono scrivere e leggere, ma perché funziona bene, bisogna chiudere il 'lato' non usato
- Ad esempio, se il padre scrive, chiude fds[0] prima di tutto e se il figlio legge allora chiude fds[1]
- Bisogna chiudere bene appena è finito l'uso

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main() {
    int fds[2];
    pipe(fds);
    int idfork=fork();
    if(idfork==0){
        close(fds[1]);
        char buffer[100];
        read(fds[0],buffer,sizeof(buffer));
        printf("Sono il figlio e ho ricevuto %s dal mio padre\n",buffer);
        close(fds[0]);
    } else {
        close(fds[0]);
        write(fds[1],"Luke, I am your father",22);
        int stato;
        waitpid(idfork, &stato, 0);
        close(fds[1]);
        return 0;
    }
}
```

Condividere delle variabile



- Useremmo dei processi leggeri (**thread**)

Threads vs Processi

- **Un thread** è un filo d'esecuzione dentro un programma
- Il thread è eseguito da un processo
- Un processo può gestire più thread
 - Si parla di processo **multi-thread**
 - Al minimo c'è uno thread
- Ogni filo d'esecuzione è diverso e ha come attribuiti:
 - Un **puntatore di esecuzione** o **PC (Program Counter)**
 - Una stack d'esecuzione (**stack**)
- È più difficile programmare con i thread in C MA :
 - l'implementazione è più efficace
 - Condividere dei dati è più facile (ma e anche per questo che bisogna essere attento)

L'API POSIX

- In C, esiste una libreria 'classica' per la creazione e la manipolazione di thread
 - la libreria **POSIX**
- Per usarla, bisogna includerla nel codice:
 - **#include <pthread.h>**
- Bisogna anche dire al momento della compilazione che si desidera usare questa libreria con l'opzione -pthread o -lpthread
- Ad esempio, per compilare un file test.c:
 - **gcc -pthread -Wall test.c -o test**

Creazione di thread

- Per creare uno thread, si usa la funzione:

```
int pthread_create(  
    pthread_t *thread, //Per ricordare i dati del nuovo thread  
    const pthread_attr_t *attr, // Attributi dello thread  
    void *(*start_routine) (void *), // Funzione da eseguire  
    void *arg); // Argomenti della funzione
```

- In pratica, per gli attributi, si può mettere **NULL** per avere gli attributi di default
- Questi attributi permettono ad esempio di definire una politica di ordinamento per lo scheduler
- Il codice eseguito dallo thread è dato dalla funzione **start_routine**
- Appena la funzione **pthread_create** è finita, lo thread è creato e può iniziare ad eseguirsi

Creazione di thread in pratica

- Si scrive una o più funzioni contendo il codice da eseguire da i thread. Il prototipo deve essere:

```
void *function(void *)
```

- Per dare argomenti alla funzione alla creazione dello thread, si danno come ultimo argomento della funzione `pthread_create`
- È avvolta necessario aspettare la fine dell'esecuzione degli thread creati
 - se il programma principale termina prima, allora gli thread vengono distrutti
 - per questo scopo si può usare la funzione

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- Il primo argomento corrisponde a quello creato da `pthread_create`
- `value_ptr` è il valore ritornato dallo thread chiamando la funzione

```
void pthread_exit(void *value_ptr);
```

- Warning: chiamare questa ultima funzione, risulta nella terminazione dello thread!!!

Esempio

- Bisogna a dare il codice degli thread

```
void *stampa(void *ptr){  
    char *s=(char *)ptr;  
    printf("Il mio messaggio e: %s",s);  
    return NULL;  
}
```

- Per creare uno thread che esegue questa funzione:

```
pthread_t th1;  
char *s1="Sono il thread 1\n";  
int r1=pthread_create(&th1, NULL, stampa, s1);
```

- Alla fine, si può usare join per aspettare la fine dello thread

```
pthread_join(th1, NULL);
```

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void *stampa(void *ptr);

int main(){
    pthread_t th1, th2;
    char *s1="Sono il thread 1\n";
    char *s2="Sono il thread 2\n";
    pthread_create(&th1, NULL, stampa, s1);
    pthread_create(&th2, NULL, stampa, s2);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return 0;
}

void *stampa(void *ptr){
    char *s=(char *)ptr;
    printf("Il mio messaggio e: %s",s);
    return NULL;
}
```

Esempio con valore di ritorno

```
int main(){
    pthread_t th1, th2;
    char *s1="Sono il thread 1\n";
    char *s2="Sono il thread 2\n";
    pthread_create(&th1,NULL,stampa,s1);
    pthread_create(&th2,NULL,stampa,s2);

    char *r1;
    char *r2;
    pthread_join(th1,(void **)&r1);
    pthread_join(th2,(void **)&r2);
    printf("%s",r1);
    printf("%s",r2);
    return 0;
}

void *stampa(void *ptr){
    char *s=(char *)ptr;
    printf("Il mio messaggio e: %s",s);
    char *mess=(char *)malloc(100*sizeof(char));
    strcat(mess,s);
    strcat(mess," Fini\n");
    return mess;
}
```

Esempio con pthread_exit

```
int main(){
    pthread_t th1, th2;
    char *s1="Sono il thread 1\n";
    char *s2="Sono il thread 2\n";
    pthread_create(&th1,NULL,stampa,s1);
    pthread_create(&th2,NULL,stampa,s2);

    char *r1;
    char *r2;
    pthread_join(th1,(void **)&r1);
    pthread_join(th2,(void **)&r2);
    printf("%s",r1);
    printf("%s",r2);
    return 0;
}

void *stampa(void *ptr){
    char *s=(char *)ptr;
    printf("Il mio messaggio e: %s",s);
    char *mess=(char *)malloc(100*sizeof(char));
    strcat(mess,s);
    strcat(mess," Fini\n");
    pthread_exit(mess);
}
```

Condividere delle variabile



- O si usa delle variabile globale
- Oppure si può dare il loro indirizzo come argomento di `pthread_create`
- Le variabile devono essere dichiarate come `volatile`

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

void *inc(void *ptr);

volatile int a=0;

int main(){
    pthread_t th1, th2;
    pthread_create(&th1, NULL, inc, NULL);
    pthread_create(&th2, NULL, inc, NULL);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("a vale %d\n", a);
    return 0;
}

void *inc(void *ptr){
    a=a+1;
    return NULL;
}
```

Attenzione con i dati condivisi

- Gli thread si eseguono in parallelo
- Non potete fare ipotesi sullo scheduler (è lui chi interrompe gli thread e che decide quale thread deve eseguirsi ad ogni momento)
- Se uno non è attento alla manipolazione delle variabile condivise, si può osservare dei comportamenti strani
- Ad esempio:
 - Uno thread testa si una variabile intera è positiva prima di ridurla
 - È interrotto fra il test e l'update
 - Un altro thread mette il valore della variabile à 0
 - Il primo thread mette la variabile a -1... mentre forse non era voluto!

Esempio di dati condivisi

```
volatile int a=0;

int main(){
    pthread_t th1, th2, th3;
    pthread_create(&th1, NULL, inc, NULL);
    pthread_create(&th2, NULL, dec, NULL);
    pthread_create(&th3, NULL, dec, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    printf("a vale %d\n", a);
    return 0;
}

void *inc(void *ptr){
    a=a+1;
    return NULL;
}

void *dec(void *ptr){
    if(a>0){
        a=a-1;
    }
    return NULL;
}
```

**Questo codice
potrebbe
stampare -1 !!!**

Come proteggere i dati

- L'accesso ai dati condivisi deve essere protetto
- In C, si possono usare dei lock (**mutex**)
- Ecco quale il principio di funzionamento
 - Uno thread che desidera accedere a un dato condiviso richiede il lock
 - Se il lock è libero, prosegue
 - Se il lock è preso, rimane bloccato finché il lock sia liberato
 - Quando ha finito di accedere ai dati condivisi, libera il lock
- Questi lock sono condivisi fra gli threads
- **Buona pratica:** è lo thread che ha preso il lock che lo deve liberare

I locks in C

- La libreria POSIX ha dei lock di tipo:

```
pthread_mutex_t lock;
```

- La prima cosa da fare è di inizializzare il lock
- Il più semplice consiste in fare:

```
lock=PTHREAD_MUTEX_INITIALIZER ;
```

- Per prendere il lock:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- E per liberarlo

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Queste due funzione ritornano 0 se tutto è andato a buon fine

Esempio di utilizzazione di lock

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
void *inc(void *ptr){  
    int r=pthread_mutex_lock(&lock);  
    if(r!=0){  
        printf("Problem with lock\n");  
        return NULL;  
    }  
    a=a+1;  
    r=pthread_mutex_unlock(&lock);  
    if(r!=0){  
        printf("Problem with unlock\n");  
        return NULL;  
    }  
    return NULL;  
}
```

```
void *dec(void *ptr){  
    int r=pthread_mutex_lock(&lock);  
    if(r!=0){  
        printf("Problem with lock\n");  
        return NULL;  
    }  
    if(a>0){  
        a=a-1;  
    }  
    r=pthread_mutex_unlock(&lock);  
    if(r!=0){  
        printf("Problem with unlock\n");  
        return NULL;  
    }  
    return NULL;  
}
```

Attenzione ai deadlock!!!

```
void *foo(void *ptr);
void *goo(void *ptr);

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
volatile int a=0;

int main(){
    pthread_t th1, th2;
    pthread_create(&th1, NULL, foo, NULL);
    pthread_create(&th2, NULL, goo, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("a vale %d\n", a);
    return 0;
}
```

Attenzione ai deadlock!!! (2)

```
void *foo(void *ptr){
    pthread_mutex_lock(&lock);
    printf("foo prende lock1\n");
    pthread_mutex_lock(&lock2);
    printf("foo prende lock2\n");
    a=10;
    pthread_mutex_unlock(&lock2);
    printf("foo rende lock2\n");
    pthread_mutex_unlock(&lock);
    printf("foo rende lock\n");
    return NULL;
}

void *goo(void *ptr){
    pthread_mutex_lock(&lock2);
    printf("goo prende lock2\n");
    //swhile(a!=10){};
    pthread_mutex_lock(&lock);
    printf("goo prende lock\n");
    a=20;
    pthread_mutex_unlock(&lock);
    printf("goo rende lock\n");
    pthread_mutex_unlock(&lock2);
    printf("goo rende lock2\n");
    return NULL;
}
```

Gli threads in Java

- Gli thread sono gestiti dalla JVM (Java Virtual Machine)
- Per usarla, ci appoggiamo su due meccanismi
 - L'interface `java.lang.Runnable`
 - Contiene solo una funzione da implementare
 - `void run()`
 - Questa funzione contiene il codice da eseguire da lo thread
 - La classe `java.lang.Thread`
 - Ci permette di manipolare gli threads
- In breve, nella funzione `run()`, si mette il codice dello thread (la parte statica) e si usa l'oggetto Thread per gestire il filo d'esecuzione (la parte dinamica)

Legamento tra Thread e Runnable

- Ad ogni thread, si lega un oggetto implementando **Runnable**
 - Ad esempio uno degli costruttori in `java.lang.Thread`
`public Thread(Runnable target)`
- Lo stesso oggetto implementando **Runnable** può essere associato a più thread
 - In questo case, ogni thread esegue in un modo concorrente la funzione `run()` del oggetto dato al costruttore

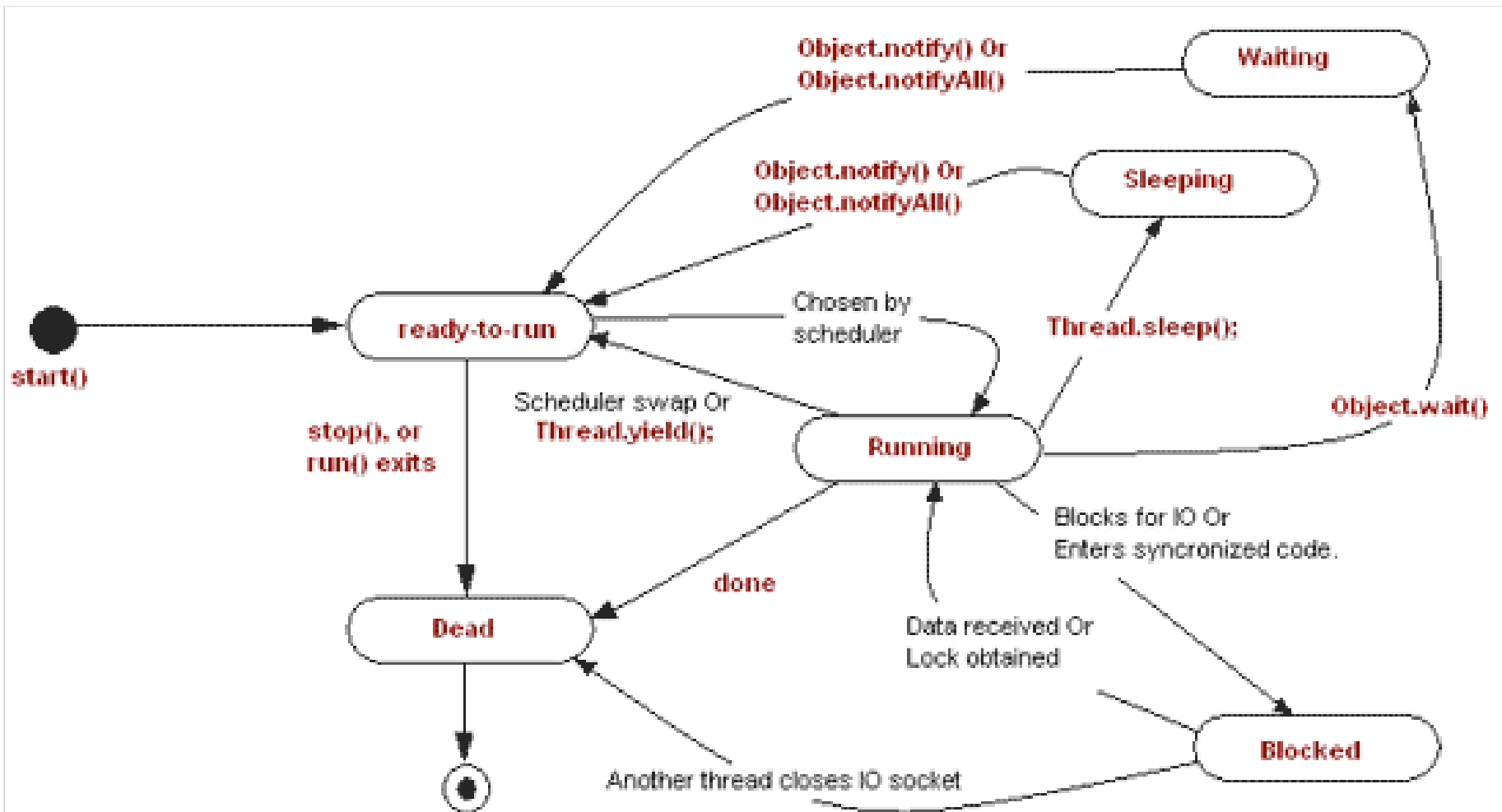
L'interface `java.lang.Runnable`

- `java.lang.Runnable` è dunque una interface contenendo una unica funzione da implementare
 - `public void run()`
- Quando uno thread inizia la sua esecuzione
 - Chiama la funzione `run()` del `Runnable` che lui è associato
 - Termine la sua esecuzione quando `run()` termina
- Attenzione: per avviare uno thread, non si chiama direttamente `run()` ma si usa una altra funzione `start()`

La classe `java.lang.Thread`

- Gli thread in Java hanno tanti attributi:
 - **String name** : il nome dello thread
 - **long id** : l'identità dello thread
 - **int priority** : la sua priorità (gli thread sono ordinati secondo questa priorità)
 - **boolean daemon** : il suo modo di esecuzione (demon o no, vedremo dopo cosa significa)
 - **Thread.state state** : il suo stato
 - **NEW, RUNNABLE, BLOCKED, WAITING, TERMINATED, ...**
 - La sua stack che può solo essere osservata
 - Il suo gruppo di thread
 - etc (cf la documentazione)
- In **java.lang.Thread**, si trovano i getter/setter corrispondenti

Gli stati di uno thread



Terminazione della JVM

- Si dice spesso che un programma finisce quando si esce dal main
 - Un programma non finisce, è il processo che si ferma
 - Bisogna anche ad aspettare che TUTTI gli thread che non sono degli daemon abbiano finito la loro esecuzione
 - Esiste al meno uno thread daemon
 - **Il garbage collector**
- Spesso ne esiste un altro
 - Il thread che gestisce l'interfaccia graficas

Creare e controllare degli thread

- Per creare uno thread esistono più modi
 - `Thread(Runnable target)`, `Thread(Runnable target, String name)`
- Esiste più metodi per controllare gli thread
 - `void start()`
 - Permette di avviare lo thread
 - Il thread chiamerà poi la funzione `run()` del **Runnable**
 - `void join()`
 - Permette di aspettare la fine di un thread
 - `void interrupt()`
 - Cambia lo stato dello thread a interrotto
 - non ha nessun effetto immediato (permette allo thread di sapere che un altro thread lo vuole interrompere)
 - **IMPORTANTE:** non esistono tecniche per fermare un thread, bisogna aspettare che finisce di eseguire il suo `run`

Metodi static di Thread

- **Thread currentThread()**
 - Permette di recuperare lo thread corrente (che esegue questa funzione)
 - Utile per esempio per recuperare il nome
- **boolean isInterrupted()**
 - Per testare se lo thread ha ricevuto una richiesta di interruzione
- **void sleep(long ms)**
 - Per fare dormire il thread un certo tempo espresso in milisecondi (tempso minimo)
- **void yield()**
 - Lo thread si mette in pausa temporaneamente e sarà riavviato dallo scheduler

Exemple Runnable

```
import java.lang.*;
import java.io.*;
public class ServiceTest implements Runnable {
    public void run(){
        try{
            while(true){
                Thread.sleep(1000);
                System.out.println("Hello"+Thread.currentThread().getName());
            }
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Example

```
import java.lang.*;
import java.io.*;

public class TestThread {
    public static void main(String[] args){
        try{
            Thread t1=new Thread(new ServiceTest(),"Bob");
            Thread t2=new Thread(new ServiceTest(),"Alice");
            //t.setDaemon(true);
            t1.start();
            t2.start();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```