

Processi e Thread

Programmi e processi

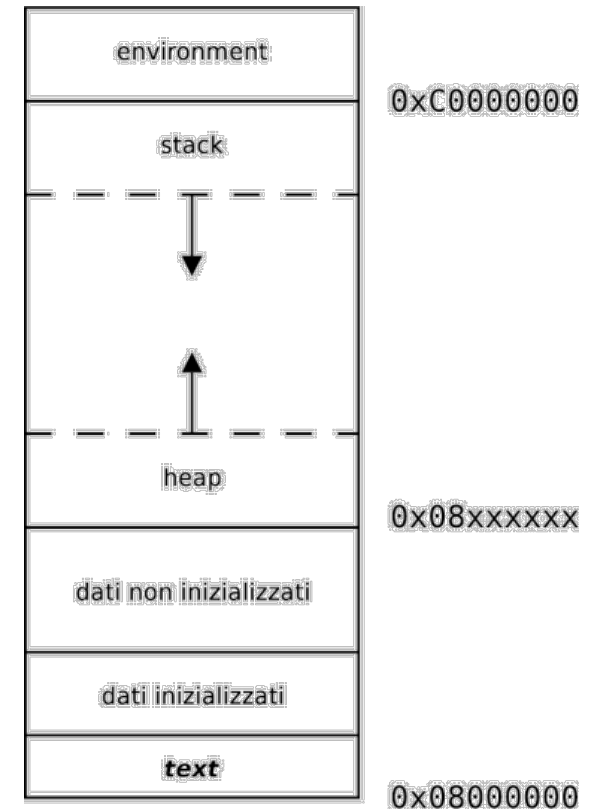
- Un **programma** è un file eseguibile residente sul disco
- Il **processo** è una istanza di un programma in esecuzione.
 - È l'unità di esecuzione all'interno del S.O.
 - L'esecuzione di un processo è solitamente sequenziale
 - le istruzioni vengono eseguite in sequenza, secondo l'ordine specificato nel testo del programma
 - Un S.O. Multiprogrammato consente l'esecuzione concorrente di più processi.

Programma = entità passiva

Processo = entità attiva

Caratteristiche di un processo

- **un corpo**
 - il codice che viene eseguito
- **uno spazio di memoria privato**
 - Composto a sua volta da :
 - **una zona dati**
 - dati non inizializzati, o BSS, e dati inizializzati.
 - **uno stack**
 - **uno heap**
- **Process ID - un descrittore di processo**
 - numero intero non negativo
 - PID
- **risorse**
 - file aperti
 - connessioni di rete
 - accesso a dispositivi
- **stato**
 - init, ready, running, sleeping



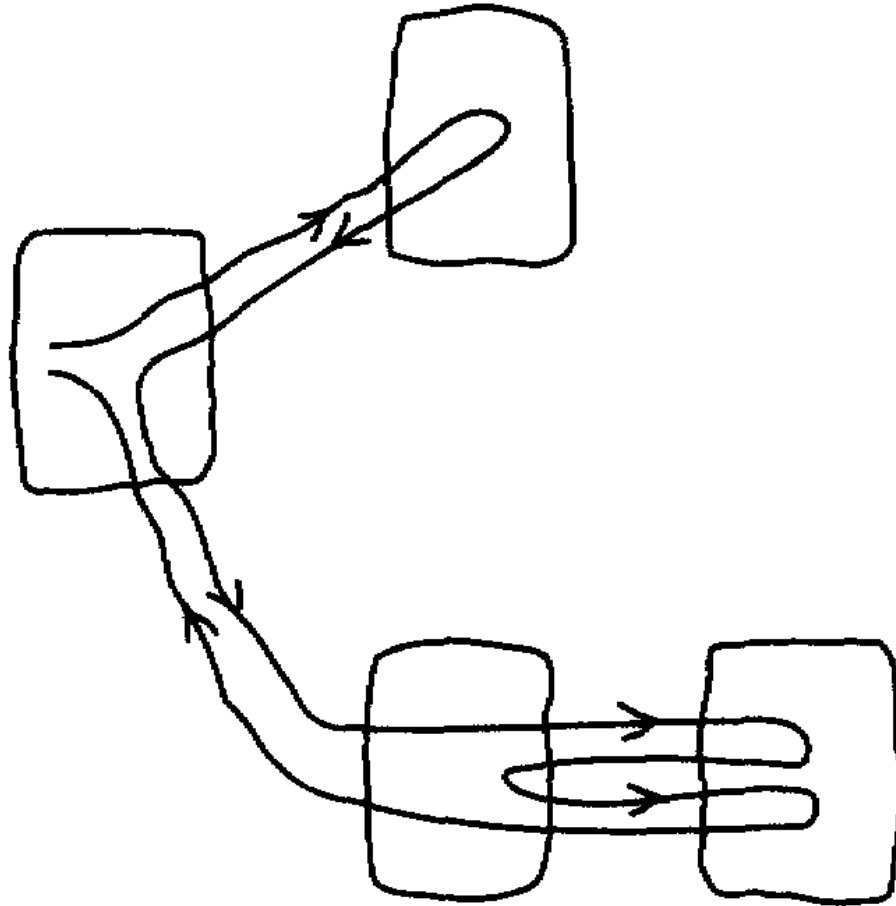
Processi nel dispositivo

- adb shell
 - in “platform-tools”
- ps
 - mostra i processi
- kill 321
 - elimina il processo 321

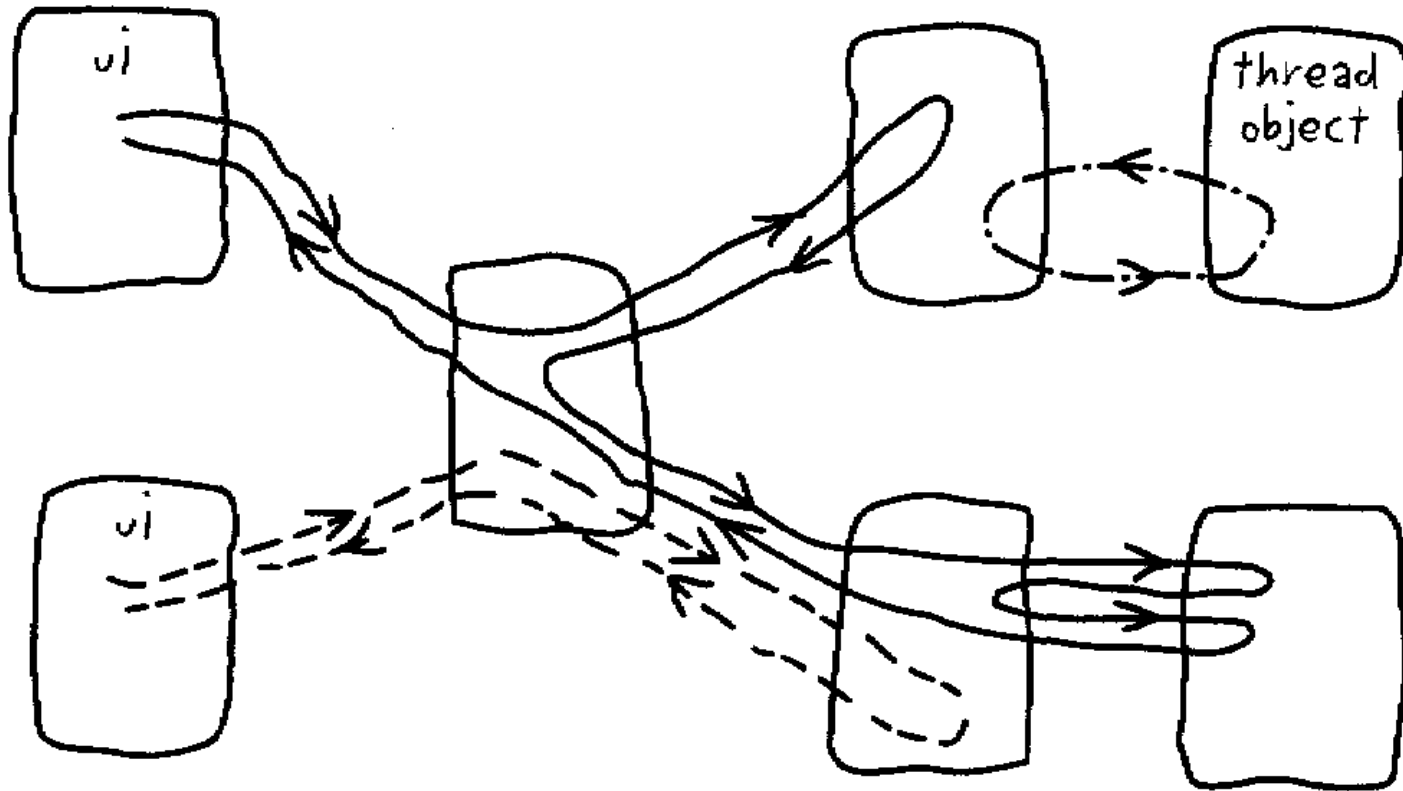
```
root    24      2      0      0      c004b2c4 00000000 S hid_compat
root    25      2      0      0      c004b2c4 00000000 S rpciod/0
root    26      1     740    248    c0158eb0 afd0d8ac S /system/bin/sh
system  27      1     812    284    c01a94a4 afd0db4c S /system/bin/servicemanager
root    28      1    3736   416    ffffffff afd0e1bc S /system/bin/vold
root    29      1    3716   432    ffffffff afd0e1bc S /system/bin/netd
root    30      1     668    248    c01b52b4 afd0e4dc S /system/bin/debuggerd
radio   31      1    5392   664    ffffffff afd0e1bc S /system/bin/rild
root    32      1   102456 25716  c009b74c afd0dc74 S zygote
media   33      1   22764  3388    ffffffff afd0db4c S /system/bin/mediaserver
root    34      1     812    308    c02181f4 afd0d8ac S /system/bin/install-d
keystore 35      1    1616    320    c01b52b4 afd0e4dc S /system/bin/keystore
root    36      1     740    248    c003da38 afd0e7bc S /system/bin/sh
root    37      1     852    352    c00b8fec afd0e90c S /system/bin/qemud
root    39      1    3380    172    ffffffff 0000ecc4 S /sbin/adbd
root    50     36     796    308    c02181f4 afd0d8ac S /system/bin/qemu-props
system  58     32   178504 31012  ffffffff afd0db4c S system_server
app_23  106     32   139364 20432  ffffffff afd0eb08 S jp.co.omronsoft.openwnn
radio   111     32   148008 22868  ffffffff afd0eb08 S com.android.phone
app_25  123     32   146400 24156  ffffffff afd0eb08 S com.android.launcher
system  132     32   137460 19336  ffffffff afd0eb08 S com.android.settings
app_0   150     32   148848 25888  ffffffff afd0eb08 S android.process.acore
app_9   157     32   132052 19124  ffffffff afd0eb08 S com.android.alarmclock
app_39  171     32   131832 17904  ffffffff afd0eb08 S ppl.test.appwidgettest
app_22  178     32   132412 18612  ffffffff afd0eb08 S com.android.music
app_12  186     32   134032 19060  ffffffff afd0eb08 S com.android.quicksearchbox
app_7   194     32   131244 18080  ffffffff afd0eb08 S com.android.protips
app_2   200     32   133820 19424  ffffffff afd0eb08 S android.process.media
app_15  212     32   144984 19932  ffffffff afd0eb08 S com.android.mms
app_30  226     32   135512 20380  ffffffff afd0eb08 S com.android.email
```

I Thread

- Un thread rappresenta un flusso di esecuzione all'interno di un processo



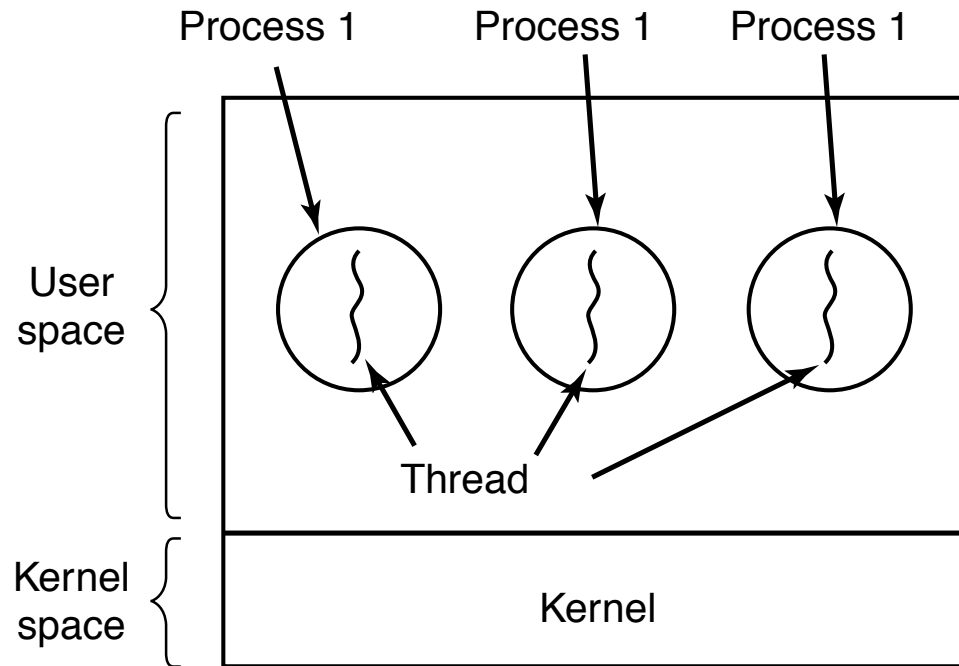
Diversi Thread



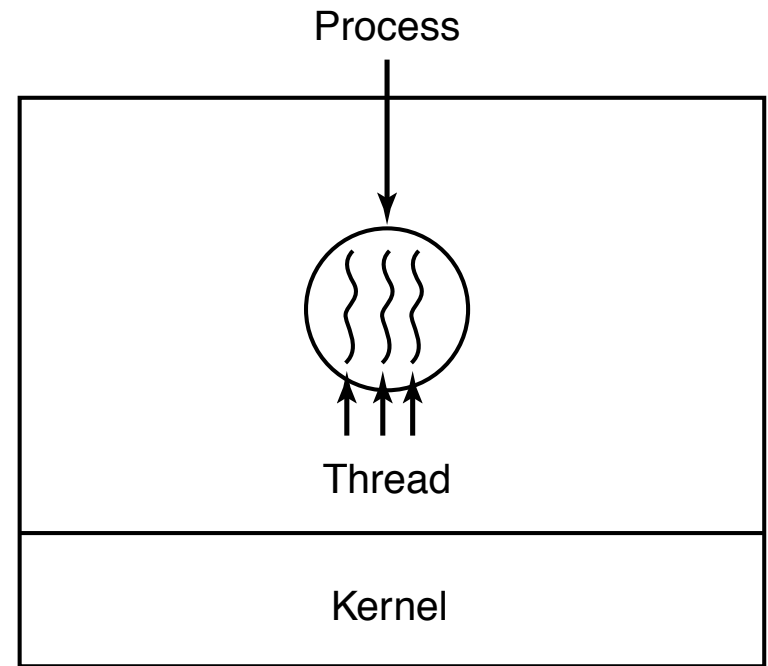
Caratteristiche

- Tutti i thread definiti in un processo condividono le risorse del processo, risiedono nello stesso spazio di indirizzamento ed hanno accesso agli stessi dati.
- 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
 - spazio di memoria statico per le variabili locali
 - accesso alla memoria e alle risorse del processo condiviso con gli altri thread.

Thread e processi

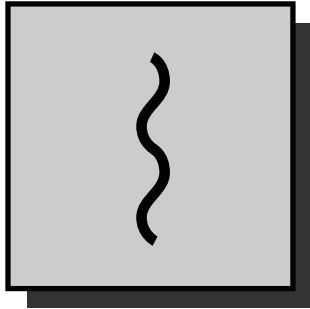


(a)

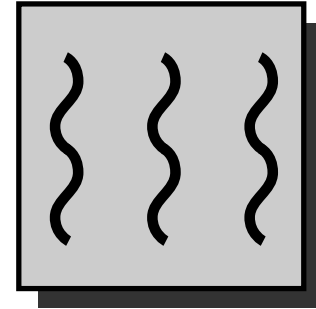


(b)

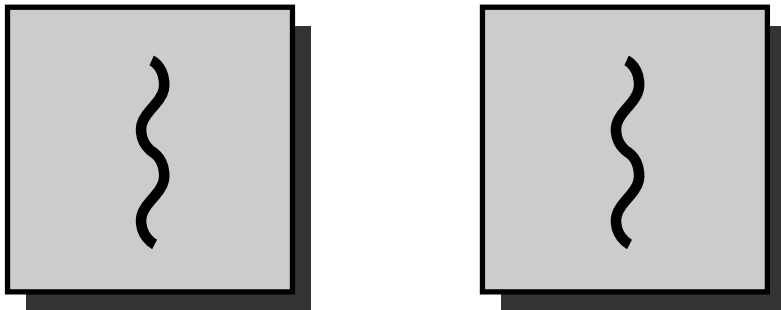
Quattro possibilità



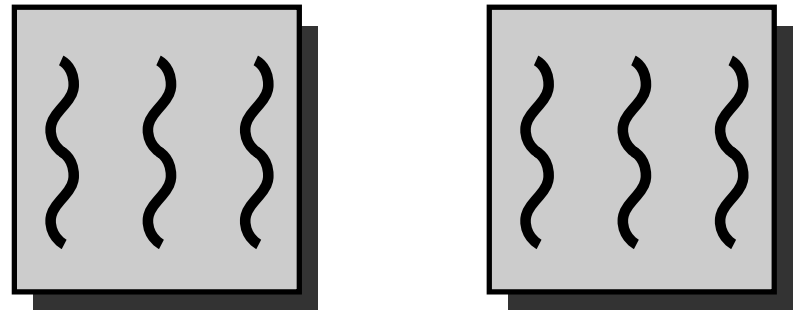
**one process
one thread**



**one process
multiple threads**



**multiple processes
one thread per process**



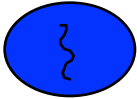
**multiple processes
multiple threads per process**

Condivisione delle risorse e della memoria

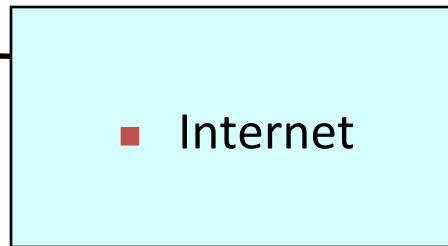
- Vantaggi:
 - maggiore efficienza
 - Creare e cancellare thread è più veloce (anche di 1000 volte): meno informazione da duplicare/creare/cancellare
 - Lo scheduling tra thread dello stesso processo è molto più veloce che tra processi
 - Cooperazione di più thread nello stesso task porta maggiore throughput e performance (es: in un server multithread, mentre un thread è bloccato in attesa di I/O, un secondo thread può essere in esecuzione e servire un altro client)
- Svantaggi:
 - Maggiore complessità di progettazione e programmazione
 - i processi devono essere “pensati” paralleli
 - sincronizzazione tra i thread
 - gestione dello scheduling tra i thread può essere demandato all'utente

Servire client contemporaneamente

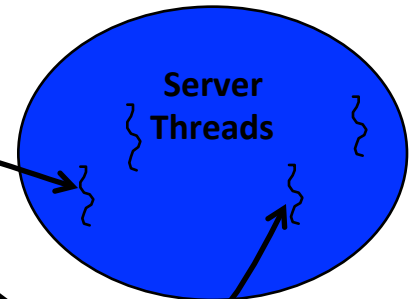
Client 1 Process



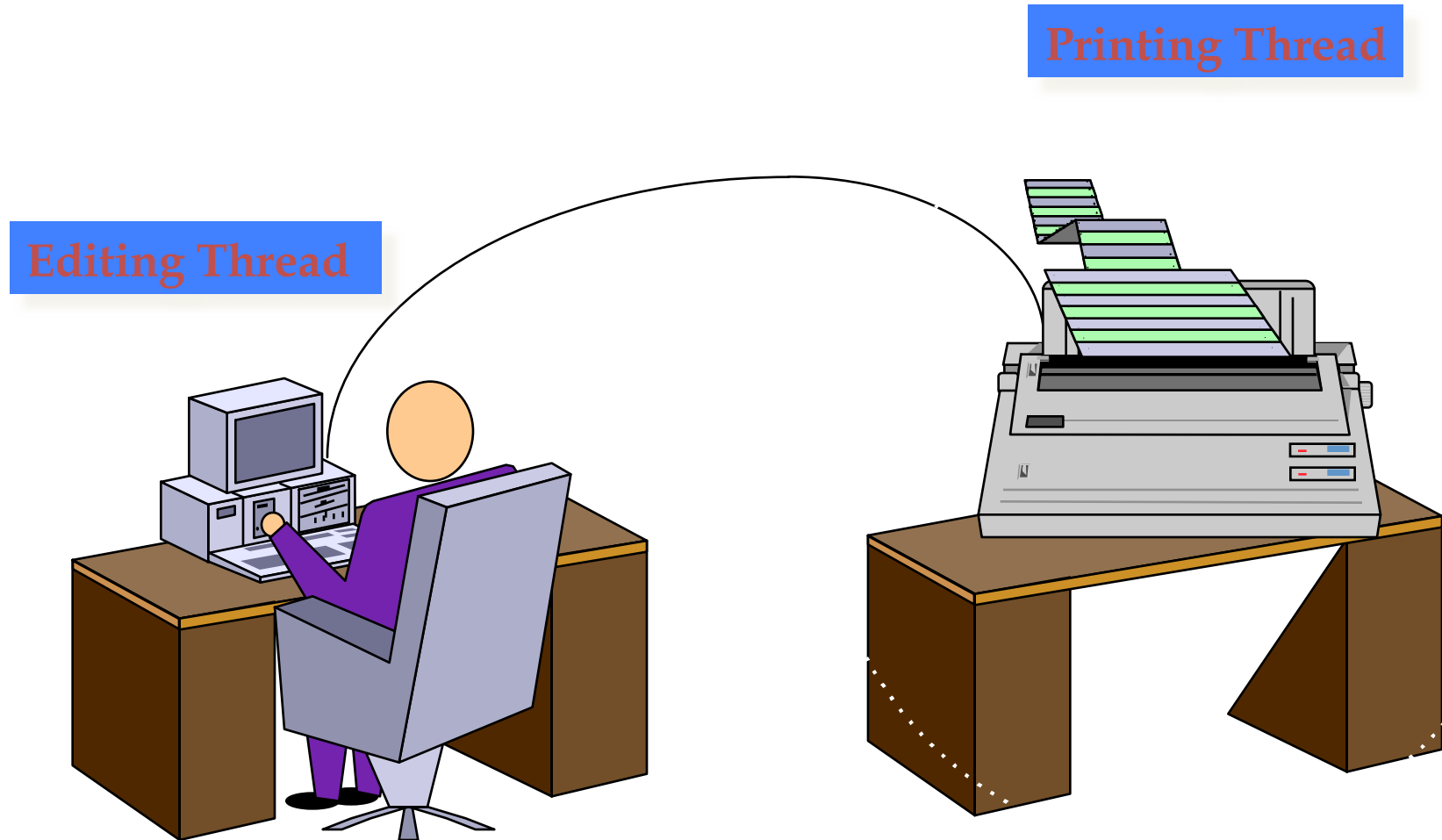
Client 2 Process



Server Process



Scrivere un documento e stampare documents in background.



Thread in JAVA

- Per la creazione di programmi multi-threaded è necessario individuare i metodi che possono essere eseguiti in modo “indipendente”
- In Java i thread sono classi che implementano il metodo **run()** che diventa la nuova “main”
 - il metodo non va chiamato direttamente, ma lo deve fare il SO
- Per scrivere il metodo run() ho due possibilità:
 - estendere la classe Thread
 - implementare l’interfaccia Runnable
- Finito run() il thread muore
 - Non è possibile “resuscitarlo”, è necessario crearne uno nuovo

Esempio

```
class MyThread extends Thread {          // il thread
    public void run() {
        System.out.println(" thread running ... ");
    }
}

class ThreadEx {
    public static void main(String [] args ) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Esempio

```
class MyThread implements Runnable{
    public void run()
    {
        System.out.println(" thread running ... ");
    }
}
```

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

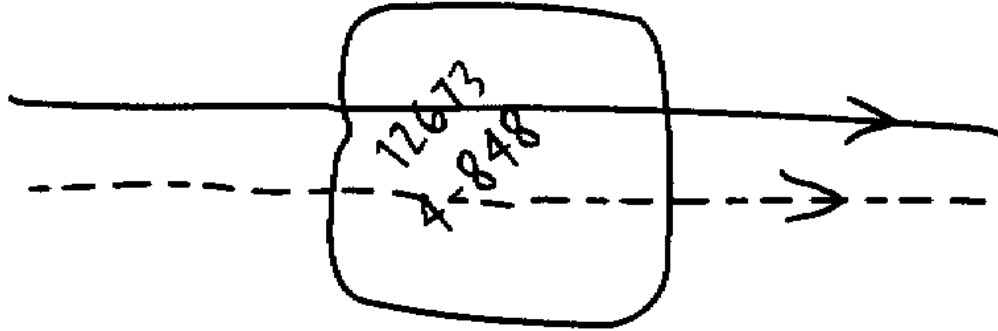
Ping PONG

```
public class PingPong extends Thread {
    private String parola;
    private long ritardo;
    PingPong(String cosaDire, long attesa){
        parola = casaDire;
        ritardo = attesa;
    }
    public void run() {
        try {
            for (;;) {
                System.out.print(parola + " ");
                Thread.sleep(ritardo);
            }
        }catch (InterruptedException e) { return }
    }
    public static void main(String[] args) {
        new PingPong("ping", 333).start();
        new PingPong("PONG", 1 ).start();
    }
}
```

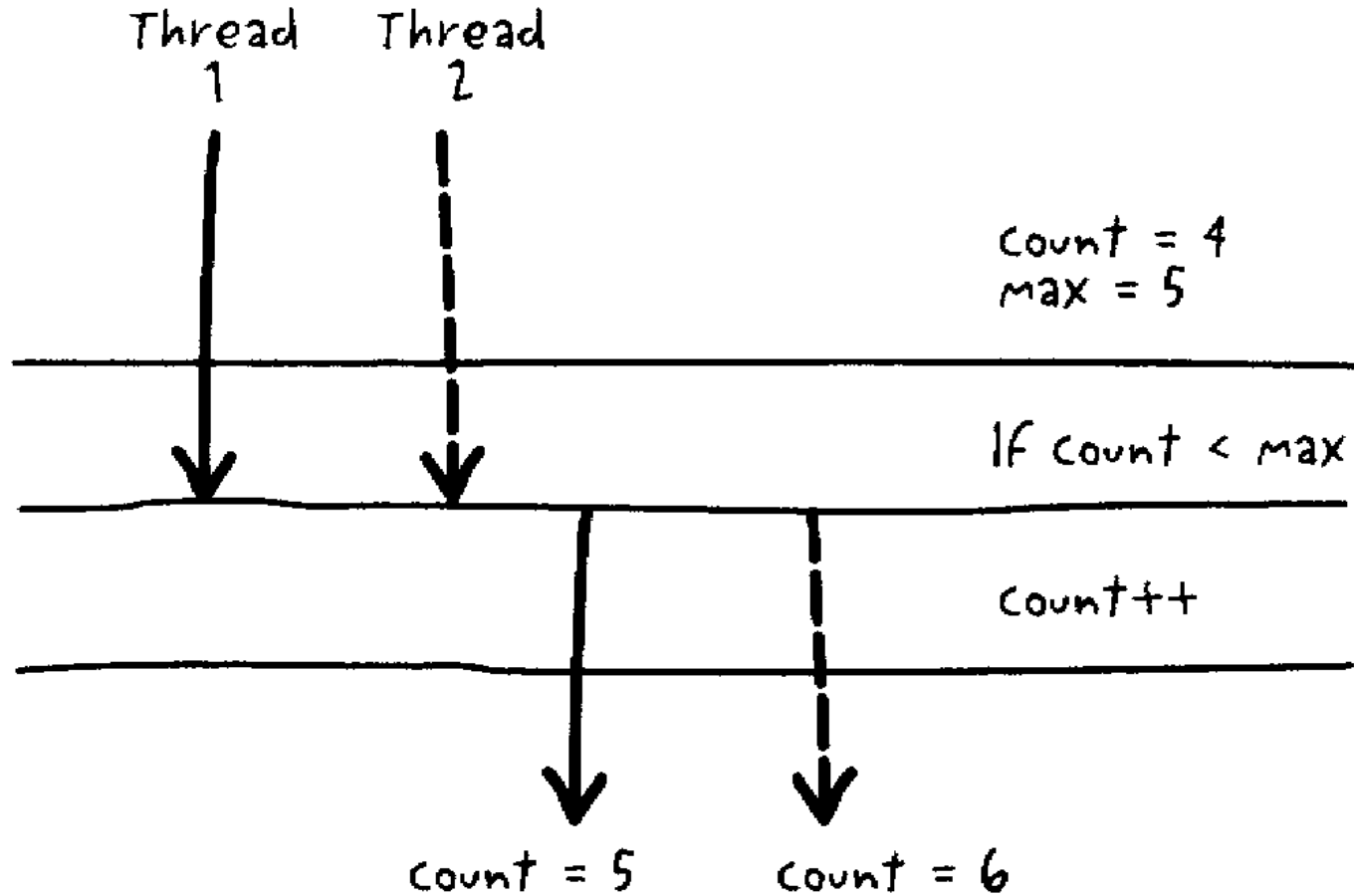

Riferimenti a Thread

- È possibile dare un nome ad un thread in due modi:
 - tramite un parametro di tipo String al costruttore
 - come parametro del metodo setName()
- Il nome di un thread è solo per comodità del programmatore
 - java non lo utilizza
- A volte è necessario poter intervenire su un thread in esecuzione senza però conoscere a priori quale esso sia
- In Java esiste il metodo statico **Thread.currentThread()** che ritorna un oggetto di tipo Thread
 - Un riferimento al thread correntemente in esecuzione

Thread ed valori condivisi



Thread ed valori condivisi



Sincronizzazione

- Esistono due motivazioni principali per l' utilizzo dei metodi di sincronizzazione:
 - la protezione delle risorse condivise
 - mantenere la coerenza dei dati
 - la notifica ad altri thread dell'avvenuta modifica di determinate condizioni
 - un' esecuzione condizionata da certi eventi per i quali non è noto a priori quando accadranno
- Nel primo caso per “risorse condivise” si intendono anche variabili e oggetti accessibili contemporaneamente da più thread

Metodi synchronized

- Risorse condivise:
 - Un Thread pone un blocco (lock) alla risorsa
 - Solo tale thread può accedere ad essa finché non rilascia il blocco

```
synchronized int some_method (int param)
```

- I metodi che inseriscono/rimuovono il lock sono individuati dal modificatore **synchronized**
 - un altro thread che invochi un metodo synchronized sullo stesso oggetto rimarrà bloccato in attesa in una coda (fino all'uscita del precedente)

Cosa sincronizzare

- Il costruttore in generale non necessita di essere sincronizzato perché viene eseguito al momento della creazione di un oggetto che può avvenire in un solo thread alla volta per ogni specifico oggetto
- I metodi di accesso a campi privati invece vanno sincronizzati per proteggerli da accessi multipli che possono corromperne il contenuto
 - Questo è un importante motivo per utilizzare metodi di accesso invece di campi public o protected
- Se il valore di un campo può essere modificato esso non deve mai essere letto mentre un altro thread lo scrive

Lock su oggetti

- È possibile inserire un lock ad un oggetto senza invocare un metodo synchronized su tale oggetto mediante l'istruzione
 - Synchronized (espressione) { istruzioni }
- L'espressione tra parentesi deve produrre l'oggetto da bloccare
- Quando è ottenuto il lock viene eseguito il blocco di istruzioni come se fosse un metodo synchronized su quell'oggetto

```
public static void abs(int[] valori) {  
    synchronized (valori) {  
        for (int i = 0; i < valori.length; i++) {  
            if (valori[i] < 0) valori[i] = -valori[i];  
        }  
    }  
}
```

Sezioni Critiche

- La sincronizzazione su valori fa sì che non possa essere modificato durante l'esecuzione di altro codice che sia sincronizzato allo stesso modo
- Non è indispensabile che l'oggetto nominato nell'espressione dell'istruzione `synchronized` venga utilizzato nel corpo delle istruzioni
 - può avere il ruolo di blocco per un'intera collezione di oggetti ed essere usato per serializzare un insieme di istruzioni che modificano uno o alcuni di tali oggetti
- Un blocco di codice serializzato si chiama sezione critica

Coordinare Thread

- Anche se la sincronizzazione realizza la modalità di accesso in mutua esclusione non ci sono garanzie sull'ordine delle operazioni
- Per stabilire un particolare ordine di esecuzione (o di accesso) dei thread è necessario implementare delle strategie che coordinano le attività mediante metodi dipendenti dall'applicazione
- L'uso dei livelli di priorità non è sufficiente da solo a stabilire un ordine di accesso e quindi diversi thread con la stessa priorità accedono in modo casuale

Comunicazione fra thread

- I metodi di sincronizzazione sono
 - `wait()`
 - `notify()`
 - `notifyAll()`
- Sono definiti nella classe `Object` e vengono ereditati da tutte le classi
- Come i lock essi si riferiscono a determinati oggetti
 - Quando `wait()` mette un thread in attesa di un evento si aspetta che un altro thread per notificare l'evento invochi `notify()` sullo stesso oggetto su cui si effettua l'attesa
- Più thread possono essere in attesa sullo stesso oggetto e un `notify` può risvegliarne uno qualsiasi, senza ordine per questo si usa in genere risvegliarli tutti con un `notifyAll`

Una coda

```
class Queue {  
    private Element head, tail;  
  
    public synchronized Object get() {  
        try {  
            while (head == null) wait();  
        } catch (InterruptedException e) {  
            return null;  
        }  
        Element p = head;  
        head = head.next;  
        if (head == null)  
            tail = null;  
        return p.item;  
    }  
}
```

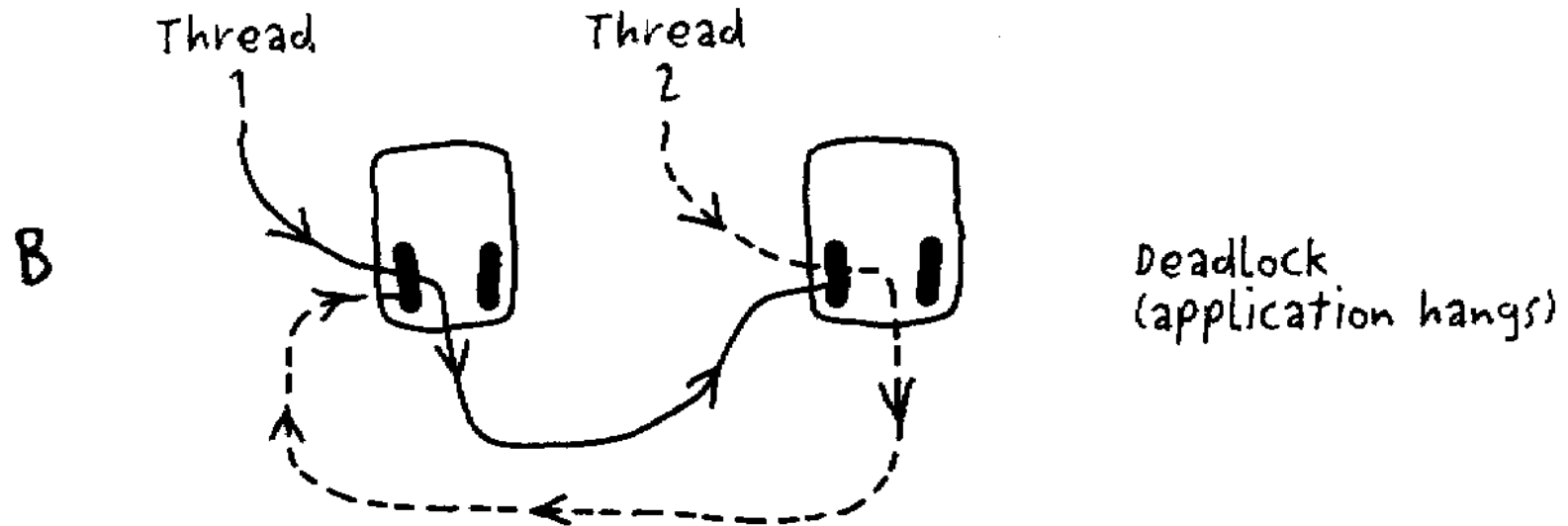
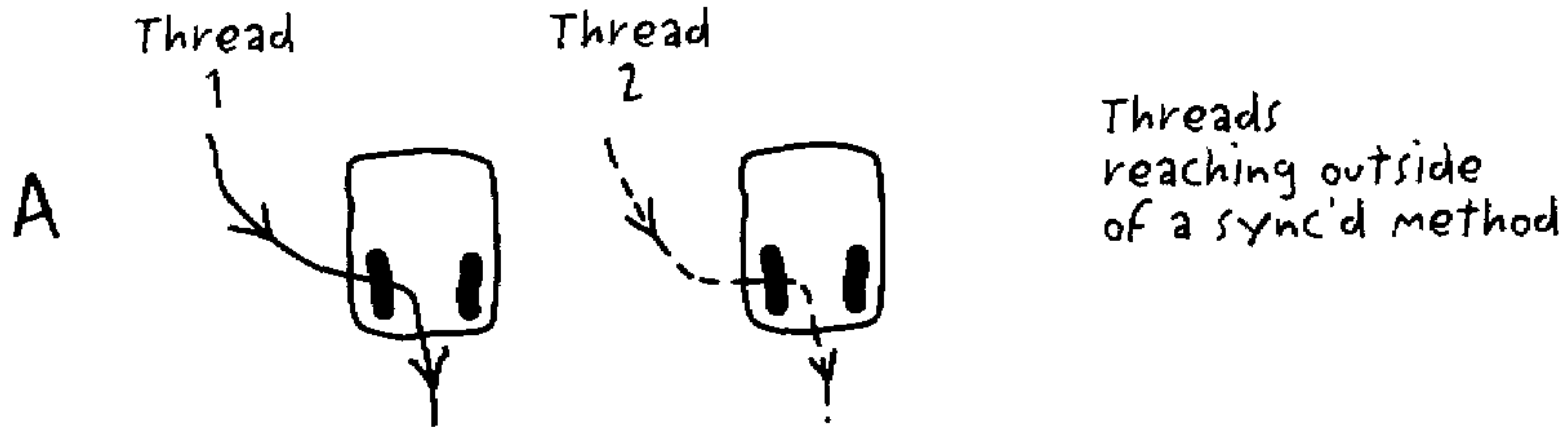
...

Una coda

...

```
public synchronized void append(Object obj) {  
    Element p = new Element(obj);  
    if (tail == null)  
        head = p;  
    else  
        tail.next = p;  
    p.next = null;  
    tail = p;  
    notifyAll();  
}  
}
```

DeadLock



Android e processi e thread

- Quando un componente di un app parte e **non** ci sono altri componenti in esecuzione, Android crea un nuovo processo Linux per l'app con un singolo thread di esecuzione
- Di default tutti i componenti di una applicazione sono eseguiti nello stesso processo e thread (main thread)
- Quando un componente di un app parte e ci sono altri componenti in esecuzione, allora il componente è eseguito nel processo già avviato ed usa lo stesso thread di esecuzione
 - è comunque possibile eseguire componenti diversi in processi separati ed è possibile creare nuovi thread per ogni processo

Attributo process

- [<activity>](#), [<service>](#), [<receiver>](#), e [<provider>](#) supportano l'attributo process che serve a specificare **il nome** del processo in cui il componente deve essere eseguito

```
<activity android:name="SecondActivity" android:process=":new_process" />
```

- Configurazioni
 - Un processo per i componenti di una app
 - Un processo per componente
 - Più app possono condividere lo stesso processo
 - se usano lo stesso user id e lo stesso certificato
- Per impostare un comportamento unico per tutti i componenti di una app si può applicare l'attributo android:process al tag [<application>](#)

Importanza dei processi

- I processi sono ordinati per importanza a seconda dei componenti che eseguono e dello stato di questi
- Livelli:
 - Foreground process
 - processo con cui l'utente sta interagendo (direttamente o indirettamente)
 - Visible process
 - è in vista ma non totalmente (e.g. c'è un dialog sopra)
 - Service process
 - Un servizio in esecuzione
 - Background process
 - Una activity nascosta
 - Empty process
 - un processo che non ha componenti in esecuzione
- Se più componenti sono in esecuzione il processo assume l'importanza di quello massimo
- Se un processo dipende da un altro questo può aumentare l'importanza