



Threads 1 (Fils)

MP09 UF2 A2.1



Processament

Les màquines executen instruccions de codi les unes rere les altres, aquestes instruccions fan ús dels recursos de la màquina, habitualment la memòria (registres, ...)

```
1  Load r1, X
2  Load r2, Y
3  Mult r3, r2, r1
4  Load r4, A
5  Mult r2, r4, r1
6  Add r5, r2, r4
7  Mult r1, r2, r5
8  Load r3, B
9  Add r7, r1, r3
```

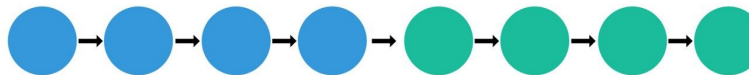


Processament seqüencial i concurrent

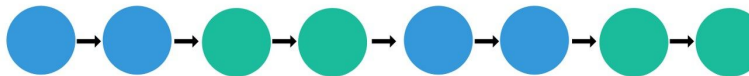
Tasques seqüencials són les que s'executen una rere l'altre, fins que no s'acaba una tasca no se'n executa una altra.

Tasques concurrents són les que s'executen de manera intercalada, compartint els recursos.

Sequential execution



Concurrent execution



● Task 1 ● Task 2

source: BetterDataScience.com



Monotasking i multitasking

Els sistemes **monotasking** o de un sol procés, són aquells que només tenen un fil d'execució, és a dir, que només poden executar un programa al mateix temps.

Actualment es poden executar diferents programes simultàniament, el què es coneix com a **multitasking**, i per tant, es poden fer programes que facin processament concurrent de dades i es poden executar diversos programes al mateix temps.



Multitasking

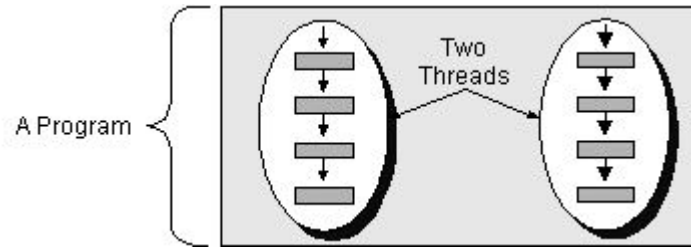
El multitasking es pot aconseguir de diverses maneres, per fer-ho cal la col·laboració del sistema operatiu:

- **Multiprogramming**, el propi sistema operatiu s'encarrega de decidir quin programa pot fer ús del processador, i en quina preferència
- **Multithreading**, són processadors per permeten diferents fils d'execució de manera simultània
- **Multiprocessor**, quan hi ha dos o més processadors disponibles



Threads

Els 'threads' o 'fils d'execució' són petits conjunts d'instruccions que es poden executar de manera independent del procés principal. Habitualment, de manera paral·lela al procés principal.





Threads en JAVA, dues maneres

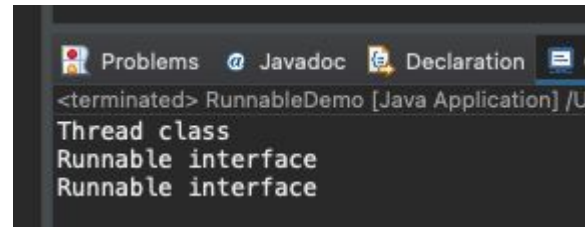
```
class Task implements Runnable {  
    @Override  
    public void run () {  
        System.out.println ("Runnable interface");  
    }  
}  
  
class ThreadDemo extends Thread {  
    @Override  
    public void run() {  
        System.out.println ("Thread class ");  
    }  
}  
  
class RunnableDemo {  
    public static void main (String ... args) {  
        new ThreadDemo().start();  
  
        new Thread (new Task(), "Thread 1").start();  
  
        new Thread (new Task(), "Thread 2").start();  
    }  
}
```

- **'extends Thread'** (no recomanat):

Especialitza la classe thread, ja no es pot estendre de cap altre classe

- **'implements Runnable'** (millor):

Crea una classe que pot ser digerida per un thread i per tant estendre's d'altres classes





Threads en JAVA, crida al SO des d'un fil

```
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;

public class ThreadsSO implements Runnable {

    public ThreadsSO() { }

    public void run () {
        try {

            String line;
            Process p = Runtime.getRuntime().exec("ls -ltr", new String[0], new File("./src/"));
            BufferedReader input = new BufferedReader(new InputStreamReader(p.getInputStream()));
            while ((line = input.readLine()) != null) {
                System.out.println(line);
            }
            input.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main (String[] args) {

        Thread ths = new Thread(new ThreadsSO());
        ths.setDaemon(false);
        ths.start();
        System.out.println("Programa acaba");
    }
}
```

setDaemon(false)
avisa que el
programa ha
d'esperar a que
tots els fils acabin
per tancar-se

El programa
principal acaba
abans que el fil que
fa la crida al SO

Problems Javadoc Declaration Console X

<terminated> Threads2 [Java Application] /Users/albertpalaciosjimenez/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64

Programa acaba

total 25888

-rw-r--r--	1	albertpalaciosjimenez	staff	652	Aug 27 10:24	Threads0.java
-rw-r--r--	1	albertpalaciosjimenez	staff	697	Aug 27 10:28	Threads1.java
-rw-r--r--	1	albertpalaciosjimenez	staff	6612013	Aug 27 10:55	llista.txt



Threads en JAVA, `.join()` espera a que acabi el fil

```
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;

public class ThreadsS0 implements Runnable {

    public ThreadsS0() { }

    public void run () {
        try {
            String line;
            Process p = Runtime.getRuntime().exec("ls -ltr", new String[0], new File("./src/"));
            BufferedReader input = new BufferedReader(new InputStreamReader(p.getInputStream()));
            while ((line = input.readLine()) != null) {
                System.out.println(line);
            }
            input.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main (String[] args) {

        Thread ths = new Thread(new ThreadsS0());
        ths.setDaemon(false);
        ths.start();
        try {
            ths.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Programa acabat");
    }
}
```

`.join()`

Espera a que el fil
hagi acabat per
continuar

Ara acaba primer el
thread i després el
programa principal

Nota: `.join(2500)`
espera 2.5 segons i
si no força
l'acabament



Threads en JAVA, `wait()` i `notify()`

```
public class Sender {  
    private String packet;  
    private boolean transfer = true;  
    public synchronized String receive() {  
        while (transfer) {  
            try {  
                wait();  
            } catch (InterruptedException e) {  
                Thread.currentThread().interrupt();  
                System.out.println("Receive Interrupted");  
            }  
        }  
        transfer = true;  
        String returnPacket = packet;  
        notifyAll();  
        return returnPacket;  
    }  
    public synchronized void send(String packet) {  
        while (!transfer) {  
            try {  
                wait();  
            } catch (InterruptedException e) {  
                Thread.currentThread().interrupt();  
                System.out.println("Send Interrupted");  
            }  
        }  
        transfer = false;  
        this.packet = packet;  
        notifyAll();  
    }  
}
```

`.wait()`

Deixa el fil en espera, i el sistema pot executar altres tasques

`.notify()`, `.notifyAll()`

Reactiven fils en espera

En aquest exemple, la variable '**transfer**' diu si s'ha d'esperar o no i es crida a `wait()` si cal esperar

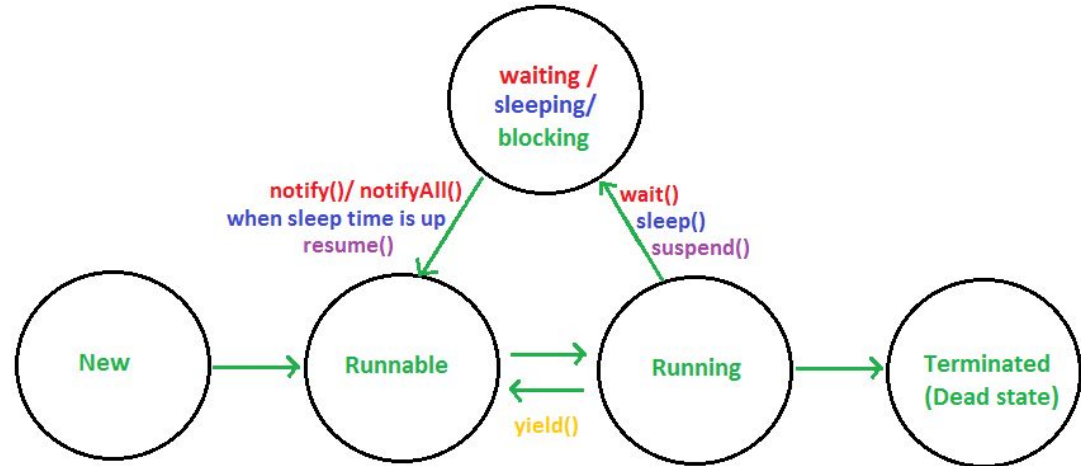


Threads, estats

Els fils d'execució no sempre s'estan executant:

- Els para el procés principal
- Esperen dades
- Estan bloquejats
- S'esperen
- ...

Per aquest motiu tenen diferents estats





Threads en JAVA, sincronització

Les aplicacions amb diversos fils presenten problemes quan diferents fils accedeixen a les mateixes dades:

- **Exclusió:** només un thread hauria d'executar seccions crítiques en un moment donat
- **Visibilitat:** Els canvis que un fil fa a les dades s'han de veure de manera consistent pels altres fils
- **Sincronia:** Els diferents fils s'han de posar d'acord per no trepitjar-se les modificacions a les dades entre ells

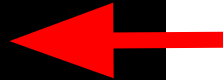
Nota: JAVA ens pot ajudar en aquests punts, però ha de ser el programador el que ha de fer el codi correctament per complir-los



Threads en JAVA, memoria compartida (static)

```
class Tmp2 {  
    static int compres = 0;  
    public static void afegirAlCarret (int  
quantitat) {  
        compres = compres + quantitat;  
    }  
}
```

```
Tmp2.compres = 5;      // Sense instància  
Tmp2.afegirAlCarret(3);  
  
Tmp2 tmp = new Tmp2(); // Amb instància  
tmp.compres = 6;  
tmp.afegirAlCarret(4);
```



Per diferents fil·ls d'una mateixa classe, es pot fer servir variables '**static**'.

Les variables i funcions '**static**' pertanyen a la classe i no a la instància de la classe i per tant tots els fil·ls d'aquella mateixa classe hi poden accedir.

Nota: Habitualment es fan servir per donar funcionalitat a les classes sense que calgui fer un nou objecte d'aquella classe (instància)



Threads en JAVA, memoria compartida (static)

Les variables 'static' es creen al posar en funcionament el programa i comparteixen espai de memòria per totes les instàncies de l'objecte.

Les altres variables es creen al fer cada instància de l'objecte (new ...) i tenen un espai de memòria propi per cada instància.



Threads en JAVA, memoria compartida

```
public class DadesCompartides {  
    public static int compartida = 0;  
}
```

```
public class FuncionalitatX extends Thread {  
  
    private DadesCompartides objCompartit;  
  
    public FuncionalitatX() {  
        objCompartit = new DadesCompartides();  
        objCompartit.compartida = 3;  
    }  
    ...  
}
```


Quan fils de diferents classes han d'accedir a un mateix conjunt de dades, podem encapsular les variables estàtiques compartides en un objecte que facin servir totes les classes que volen accedir a aquelles dades.

Nota: Ens hem d'assegurar que cada fil (de cada classe) entén que són dades compartides i només les escriu si pot



Threads en JAVA, memoria compartida (volatile)

```
public class DadesCompartides {  
    public static volatile int compartida = 0;  
}
```



Les variables '**volatile**' avisa'n que aquell valor es farà servir des de diferents fils i que les modificacions des de cada un s'han de fer en ordre d'execució.

El programador ha de sincronitzar igualment les lectures i escriptures, és un avís a nivell de màquina virtual. Té un cost en RAM i pot enlentir l'aplicació, però ajuda a evitar errors si s'executen diversos fils al mateix temps.

No té massa sentit si la sincronització només permet un fil simultàniament

```
public class FuncionalitatX extends Thread {  
  
    private DadesCompartides objCompartit;  
  
    public FuncionalitatX() {  
        objCompartit = new DadesCompartides();  
        objCompartit.compartida = 3  
    }  
  
    ...  
}
```




Threads en JAVA, sincronització de les dades

Tenir informació compartida entre diferents fils, implica que aquestes dades es poden sobreescriure o poden llegir en ordre incorrecte. Per evitar-ho hi ha diferents estratègies:

- Funcions '**synchronized**'
- Objecte '**ReentrantLock**', similar a l'anterior amb funcionalitats avançades
- Objecte '**Semaphore**', controlar l'accés a memòria des de múltiples fils

Nota: [Aquest post](#) explica molt bé l'ús de cada mètode

Nota: Els [semàfors](#) són el mètode clàssic i es fa servir a nivell de sistema operatiu i llenguatges de programació com C o C++



Threads en JAVA, synchronized

```
static Object mutex = new Object();  
static int compartida = 0;  
  
public synchronized void augmentaValor () {  
    synchronized(mutex) {  
        compartida = compartida + 1;  
    }  
}
```

Les funcions 'synchronized' poden assegurar que un fil de la mateixa classe executa un tros de codi en un moment donat.

Cal un objecte que permet aquesta exclusió. Habitualment s'anomena 'mutex', perquè fa referència a l'exclusió mútua



Threads en JAVA, ReentrantLock

```
static ReentrantLock mutex = new ReentrantLock();
static int compartida = 0;

static void augmentaValor () {
    try {
        mutex.lock();
        compartida = compartida + 1;
    } finally {
        mutex.unlock();
    }
}
```

‘**ReentrantLock**’ és una versió millorada de ‘**synchronized**’ que permet més configuracions i control sobre l’exclusió mutua de fils.

És l'opció més recomanable i senzilla per la majoria de tasques.



Threads en JAVA, Semàfors

```
static Semaphore mutex = new Semaphore(1);
static int compartida = 0;

static void augmentaValor () {
    try {
        try {
            mutex.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        compartida = compartida + 1;
    } finally {
        mutex.release();
    }
}
```

Els semàfors controlen l'accés a un recurs a través d'un comptador.

En aquest exemple només es permet 1 procés simultàniament, és un semàfor binari.

Nota: Si volem que 3 processos accedeixin a aquest tros de codi podem crear el semàfor amb:

'new Semaphore(3)'



Threads en JAVA, Semàfors vs ReentrantLock

- Els semàfors són un mètode de senyalització, ReentrantLock és un mètode de bloqueig
- Cap thread és propietari d'un semàfor, en canvi l'últim mètode que ha blocat un codi és el propietari del 'ReentrantLock'. Això té implicacions a la hora de programar amb 'ReentrantLock'
- Els semàfors asseguren que un mateix thread no aconseguirà el control un cop rere l'altre bloquejant els altres processos, evitant que el codi sigui fàcil de penjar
- Els semàfors permeten configuracions i codis més complexes amb operacions com 'wait'