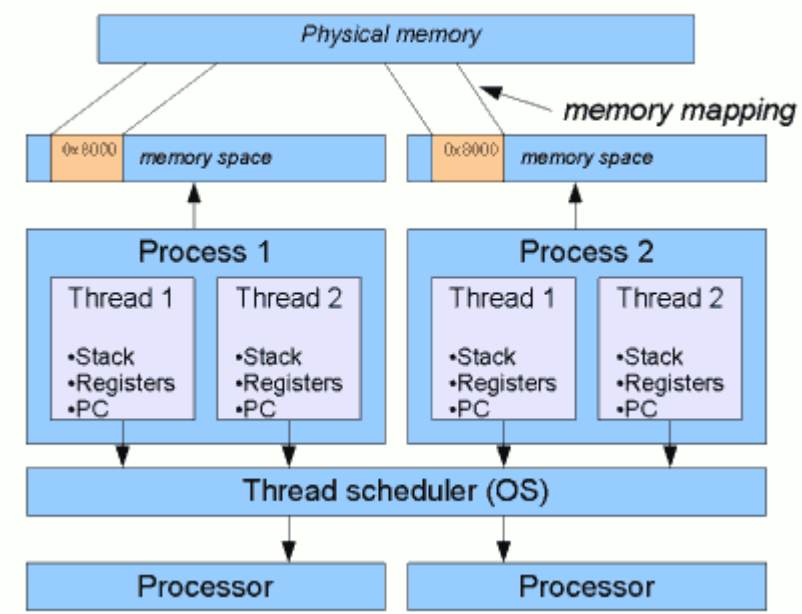
Ingineria Sistemelor de Programare

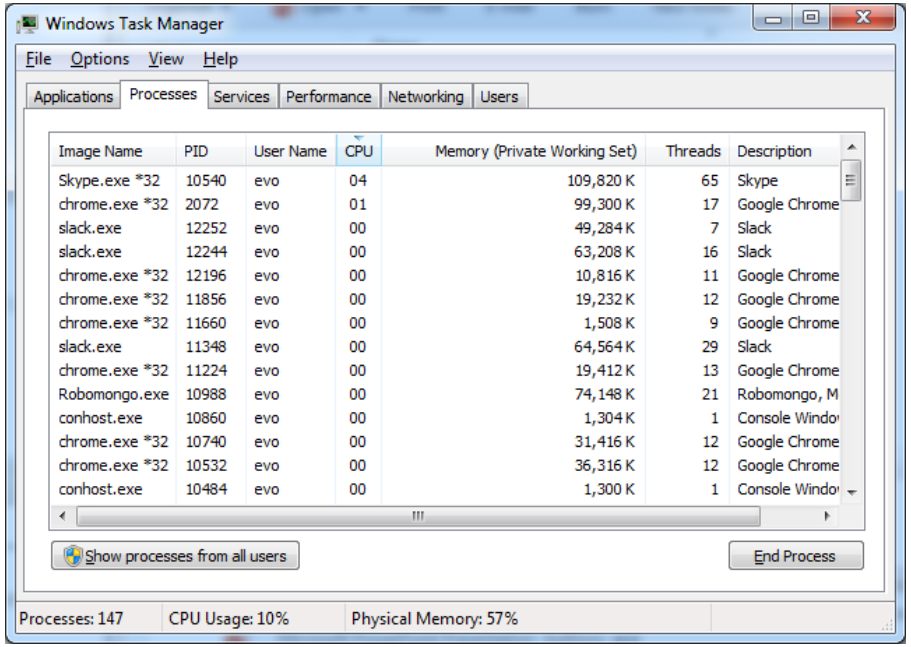
Cuprins

- Fire de executie
- Concurrency API (java.util.concurrent)

Executia concurenta

- Concurenta prin intermediul proceselor si a firelor de executie





http://www.javamex.com/tutorials/threads/how_threads_work.shtml

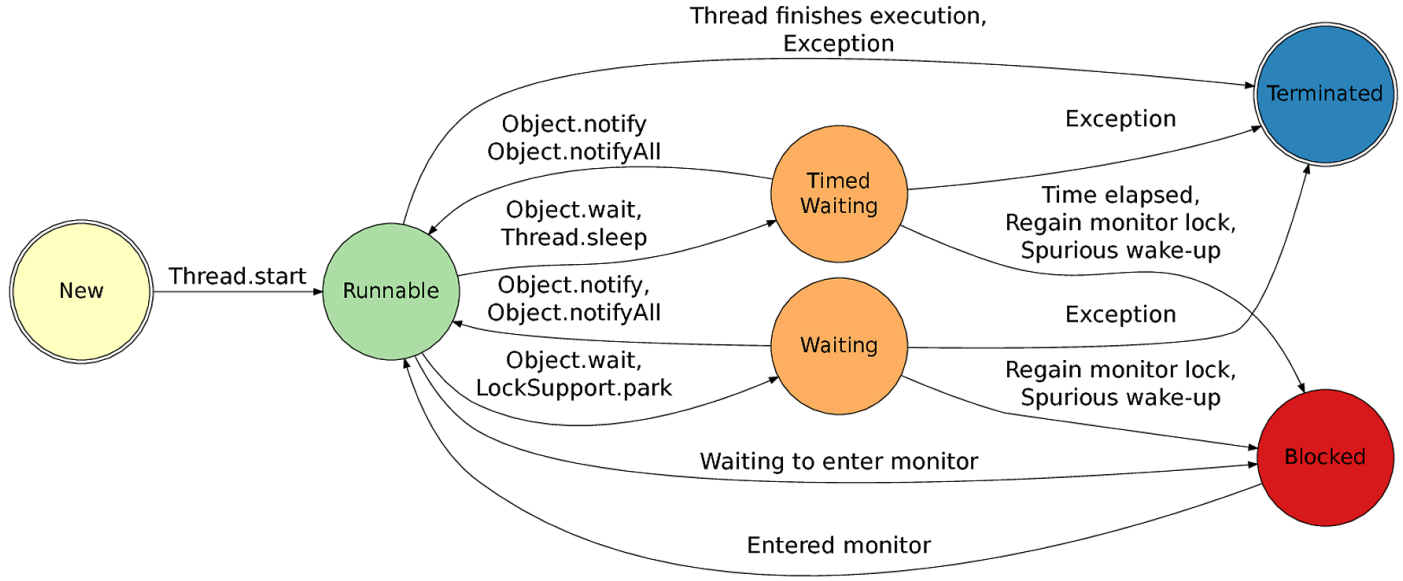
Executia concurenta

- Proces (eng. process) = Program aflat in executie in memoria calculatorului. Procesele sunt independete si nu isi pot accesa unele altora zonele de cod sau de date. Comunicarea intre procese se face prin mecanisme specifice puse la dispozitie de sistemul de operare.

- Fire de executie (eng. thread) = Secvente de instructiuni ce se ruleaza virtual in parallel in cadrul unui process. Firele de executie impart resursele unui proces. Comunicare intre firele de executie se face prin intermeidiul variabilelor si metodelor.

Construirea firelor de executie in Java

- Pasii necesari pentru contruirea unui fir de executie
  - Extinderea clasei *Thread* sau implementarea interfetei *Runnable*
  - Implementarea logicii firului in metoda *run()*
  - Lansarea in executie a firului prin apelarea metodei *start()*



Construirea firelor de executie

```
class FirstActivity implements Runnable{
    @Override
    public void run() {
        int k = 0;
        while(k++<10){
            System.out.println("Do first activity..."+k);
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
class SecondActivity extends Thread{
    @Override
    public void run() {
        int k = 0;
        while(k++<10){
            System.out.println("Do second activity..."+k);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        FirstActivity f1 = new FirstActivity();
        Thread t = new Thread(f1);
        t.start();

        SecondActivity f2 = new SecondActivity();
        f2.start();
    }
}
```

Lucrul cu fire de executie

- Un program isi incheie executia cand toate firele de executie (cu exceptia celor de tip *daemon* isi incheie executia)
- Metoda recomandata pentru terminarea executiei uni fir este prin verificarea unei conditii si parasirea metodei run(). Nu se recomanda utilizarea metodet *Thread.stop()*
- Numele firelor de executie (setName(), getName())
- Prioritatea firelor de executie (setPriority(), getPriority())
  - Programatorul nu trebuie sa isi bazeze logica programului pe prioritatea firelor
- Metoda Thread.sleep(milliseconds) pun un fir de executie in asteptare o perioadad definite de timp
- Metoda Thread.yeld() notifica planificatorul de fire ca firul current poate ceda procesorul
- Fire de executie de tip daemon (setDaemon())

Sincronizarea firelor utilizand metoda join()

- join() permite unui fir sa astepte terminarea unui alt fir
- Firul care apeleaza join() va fi pus in asteptare pana cand firul pentru care a fost apelat join() isi termina executia

```
class JoinTest extends Thread
{
    String n;
    Thread t;
    JoinTest(String n, Thread t){this.n = n;this.t=t;}

    public void run()
    {
        System.out.println("Firul "+n+" a intrat in metoda run()");
        try
        {
            if (t!=null) t.join();
            System.out.println("Firul "+n+" executa operatie.");
            Thread.sleep(3000);
            System.out.println("Firul "+n+" a terminat operatia.");
        }
        catch(Exception e){e.printStackTrace();}

    }

    public static void main(String[] args)
    {
        JoinTest w1 = new JoinTest("Proces 1",null);
        JoinTest w2 = new JoinTest("Proces 2",w1);
        w1.start();
        w2.start();
    }
}
```

Excluderea mutual

- Pentru implementarea excluderii mutual JVM asociaza cate un *lock* cu fiecare obiect si clasa
- Rechizitioarea *lock-ului* se face atunci cand un fir intra intr-un bloc sincronizat sau intr-o metoda sincronizata
- Eliberarea *lock-ului* se face la iesirea din zona sincronizata
- Un fir nu poate intra intr-o zona sincronizata daca un alt fir detine lock-ul corespunzator zonei sincronizate
- Daca un fir nu poate intra in zona sincronizata atunci acesta ese blocat (pus in asteptare) la intrarea in zona sincronizata
- La eliberarea *lock-ului* JVM va permite accesul (va debloca) in zona scrionizata a unui alt fir

Excluderea mutuala

Bloc sincronizat

```
synchronized(p){
a= p.getX();
b = p.getY();
}
```

La intrarea intr-un bloc sincronizat se rechizitioneaza lock-ul obiectului specificat in parantezele blocului.

Metoda sincronizata

```
synchronized void push(double d)
{
    content.add(new Double(d));
    notify();
}
```

La intrarea intr-o metoda sincronizata se rechizitioneaza lock-ul metodei din care face parte metoda.

http://control.aut.utcluj.ro/hmihai/doku.php?id=java1:fire:introducere#sincronizarea_firelor

Interblocaje

- Interblocajul (eng. deadlock) - folosirea blocurilor sincronizatea în mod greşit poate duce la situaţii de interblocaje între firele de execuţie. Astfel de situaţii apar atunci când două fire sunt blocate, fiecare aşteptând unul după celălalt eliberarea unui monitor.

http://control.aut.utcluj.ro/hmihai/doku.php?id=java1:fire:introducere#interblocaje_deadlocks

Metodele wait() si notify()

- Metodele wait() si notify() fac parte din clasa Object
- Sun folosite de firele de executie pentru
- Metoda wait()
  - Poate fi apelata doar dintr-o zona sincronizata
  - Poate fi apelata doar pentru obiectul pentru care se detine lock-ul
  - La apelare metoda va pune in astepatre (bloca) firul appellant
  - Dupa blocare firul appelant va elibera lock-ul pe care il detine!

- Metoda notify()
  - Poate fi apelata doar dintr-o zona sincronizata
  - Poate fi apelata doar pentru obiectul pentru care detine lock-ul
  - La apelare metoda va trezi un fir de executie blocat ca urmare a apelarii metodei wait()

http://control.aut.utcluj.ro/hmihai/doku.php?id=java1:fire:introducere#sincronizarea_firelor_utilizand_metodele_wait_notify_si_notifyall

Executors

- Concurrency API introduce conceptul *ExecutorService*
- - inlocuieste necesitatea de a lucra direct cu fire (sunt construite automat)
- - reutilizeaza firele de executie *(colectie de fire reutilizabile)*
- - permite executia asincrona de task-uri
- - ofera suport pentru lucrul cu task-uri de tip *Runnable* si *Callable*

Exempl Threda vs ExecutorService

```
class JobA implements Runnable{

    public void run(){
        Random r = new Random();
        int k = 1 + r.nextInt(10);
        while(k-->0) {
            System.out.println("Tick "+k+" by Thread "+Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {e.printStackTrace(); }
        }
    }
}
```

```
public class Main {

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

1 - Thread

```
ExecutorService executorService = Executors.newFixedThreadPool(2);
executorService.submit(new JobA());

executorService.shutdown();
```

2- Executor service

Pasi lucru cu executor service:

1. Construire executor service.
2. Trimtere task catre executor service
3. Inchide executor service.
    1. shutdown() vs shutdownNow()

Lucrul cu Callable \ Future

- Executarea de task-uri utilizand interfata Callable si obinerea rezultatelor executiei in mod asincron prin intermediul interfetei Future
- Future.get() blocheaza indefinite
- Future.get(1, TimeUnit.Seconds) asteapta un timp limitat returnarea rezultatului

```
class JobB implements Callable<Integer> {

   @Override
   public Integer call() throws Exception {
      Random r = new Random();
      int k = 1 + r.nextInt(10);
      while(k-->0) {
         System.out.println("Tick "+k+" by Thread
"+Thread.currentThread().getName());
         try {
            TimeUnit.SECONDS.sleep(1);
         } catch (InterruptedException e) {e.printStackTrace(); }
      }
      return k;
   }
}
```

```
Future<Integer> future = executorService.submit(new JobB());
Integer result = null;
try {
   result = future.get();
   System.out.println("Executed job with tick count = "+result);
} catch (InterruptedException | ExecutionException e) {
   e.printStackTrace();
}
```

Executia de task-uri periodice sau cu intarziere

- Delay – intarzierea initiala cu care se executa prima data task-ul
- Period – intervalul de timp dintre doua executii succesive ale task-ului

```
ScheduledExecutorService scheduledExecutor =
Executors.newScheduledThreadPool(1);

//periodic task
int delay = 3;
int period = 1;
scheduledExecutor.scheduleAtFixedRate(new JobC(), delay, period,
TimeUnit.SECONDS);


//one time delayed callable task
ScheduledFuture<Integer> future1 = scheduledExecutor.schedule(new JobB(), 3,
TimeUnit.SECONDS);
try {
        Integer result = future1.get();
        System.out.println("Delayed executed job result: "+result);
} catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
}
```