Licence SIL -ACPI

Programmation Concurrente en Java

Les processus (classiques ou légers)

Colette Johnen

johnen@labri.fr

www.labri.fr/~johnen

Définition

Un programmes est une entité statique (fichier situé sur un disque)

Un processus est une entité dynamique (un programme en cours d'exécution)

Un programme peut engendrer lors de son exécution plusieurs processus

Multi-Programmation/multi-Processeurs

La multi-programmation donne une illusion de parallélisme, comme le temps partagé donne à chaque utilisateur l'impression d'être le seul à utiliser la machine.

Les machines multi-processeurs sont appelées machines parallèles et présentent la véritable notion de processus exécutés en parallèle.

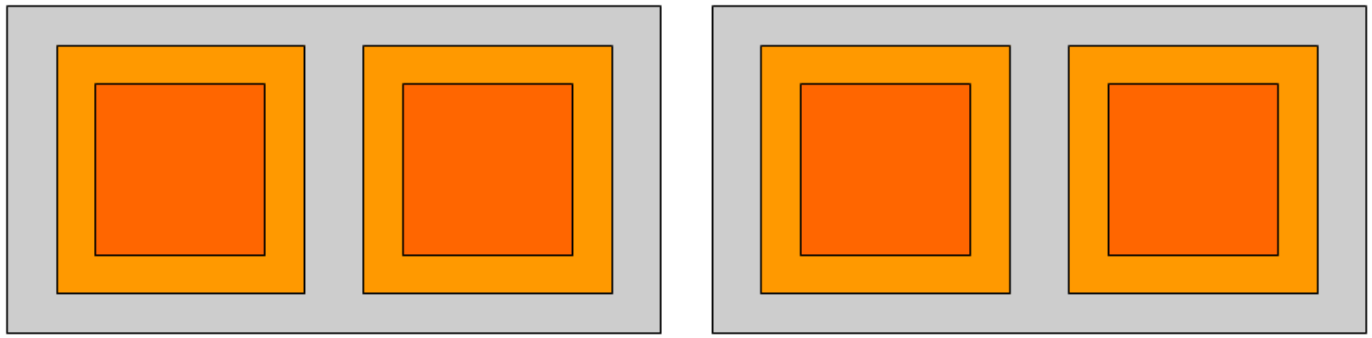
Dans une machine mono-processeur, il y a du vrai parallélisme: certaines entrées-sorties sont réalisées par des processeurs spécialisés pendant que le processeur central exécute une autre tâche.

Multi-Coeurs

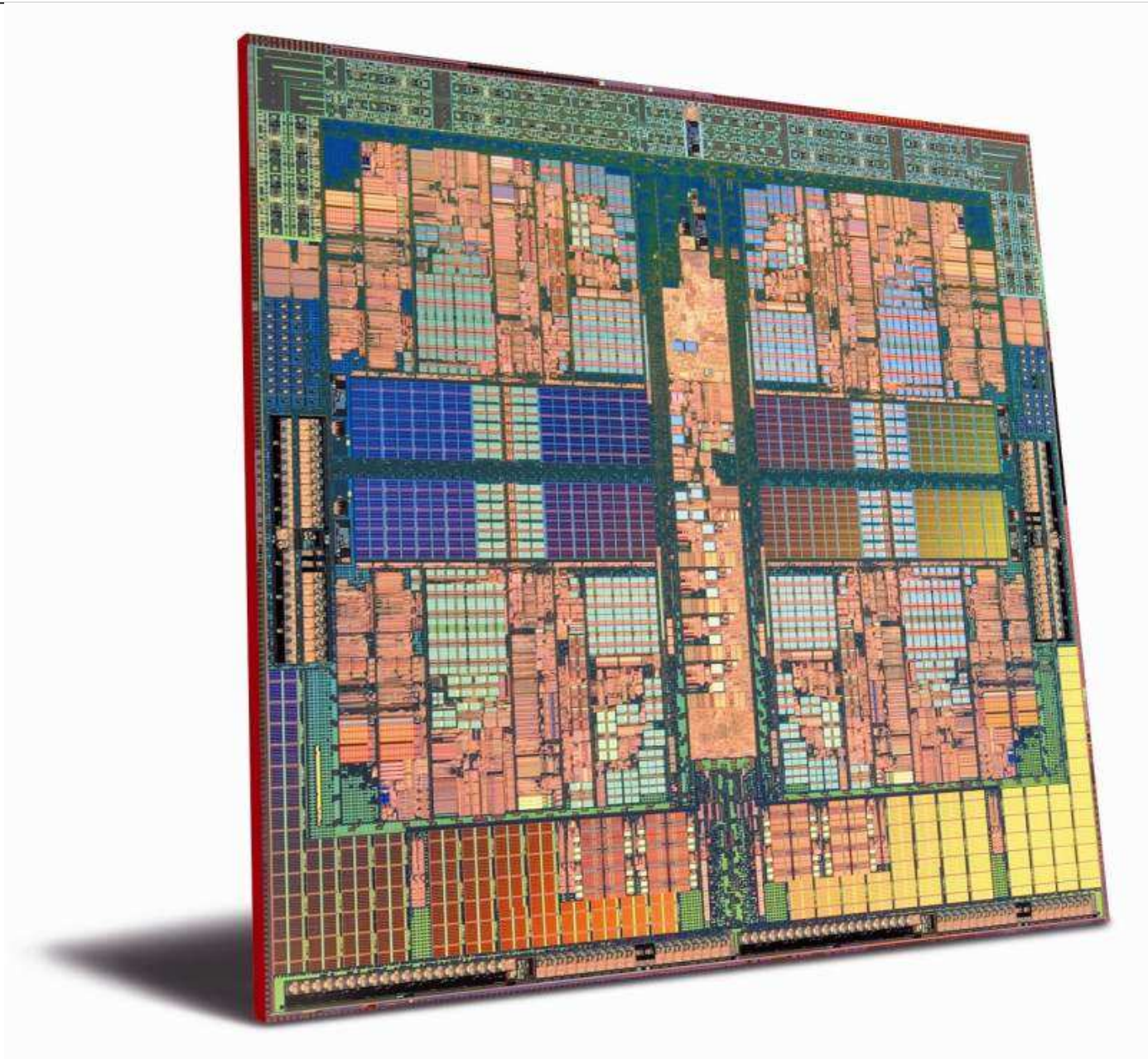
Multi-Coeurs : plusieurs cœurs (unité de calcul) côte-à-côte sur la même puce (processeur).

Le support (la connectique qui relie le processeur à la carte électronique) lui ne change pas.

Certains éléments, comme par la RAM, peuvent être mis en commun.
Ex: deux processeurs chacun étant un bi-coeurs



Multi-Coeurs



Machine la plus puissante du monde (juin 2012 : www.top500.org)

For the first time since November 2009, a United States supercomputer sits atop the TOP500 list

System: an IBM BlueGene/Q system
Location : at the Department of Energy's Lawrence Livermore National Laboratory (LLNL), achieved an impressive

16.32 petaflop/s on the Linpack benchmark using 1,572,864 cores.

Note : Mega :10^6, Giga :10^9, Teta: 10^{12} , Peta : 10^{15},
Exa :10^{18}, Zetta: 10^{21}, Yotta …

Sequoia : Machine la plus puissante du monde (juin 2012 : www.top500.org)

96 racks covering an area of about 280 m^2.

- A rack has 32 compute drawers
- A compute drawer has 32 compute cards
- A compute cards is an 18/16 core chip

= 1,572,864 cores.

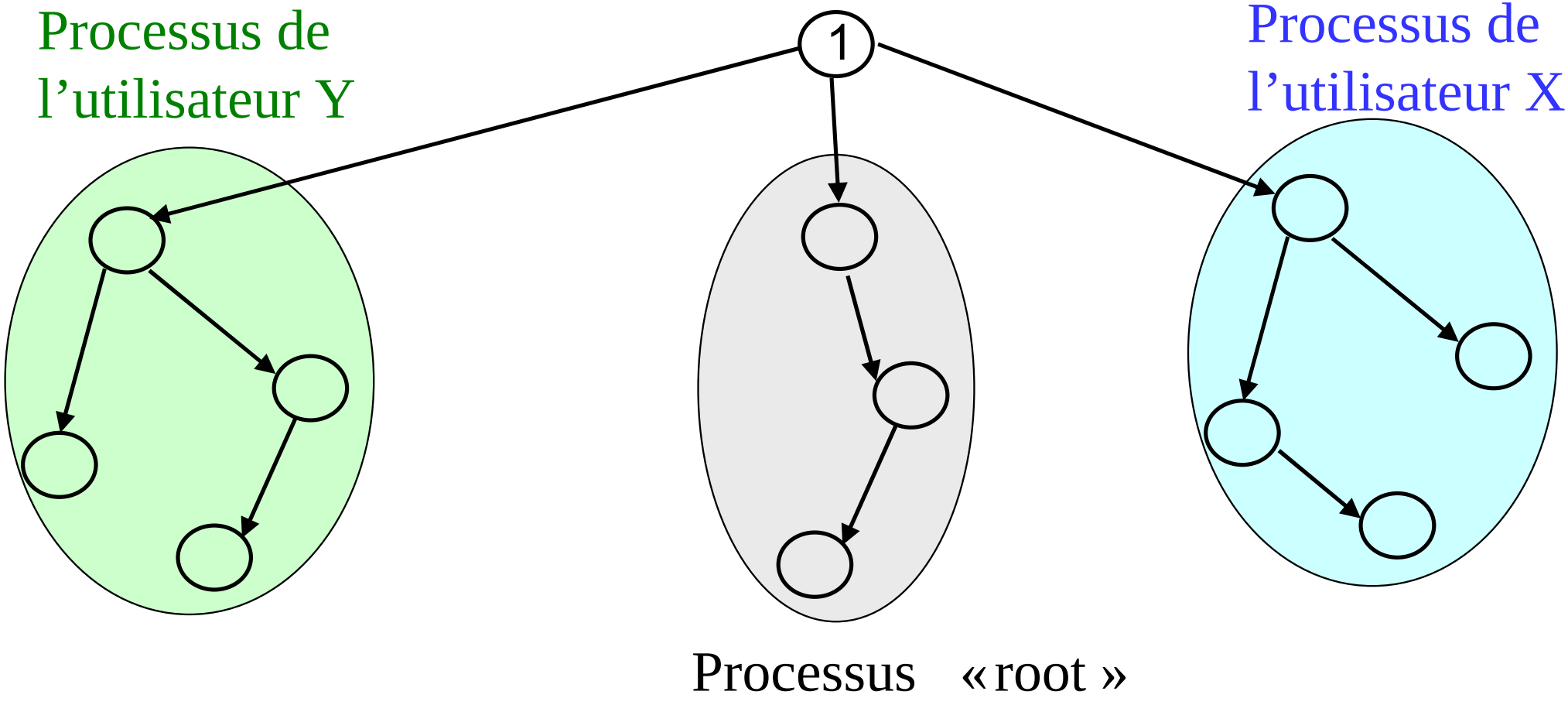
Linpack benchmark :16.32 petaflop/s
Peak : 20,1 petaflop/s

Energy : 7890 KW/s

Note : Peta : 10^{15} ref : wikipedia

Hiérarchie des processus

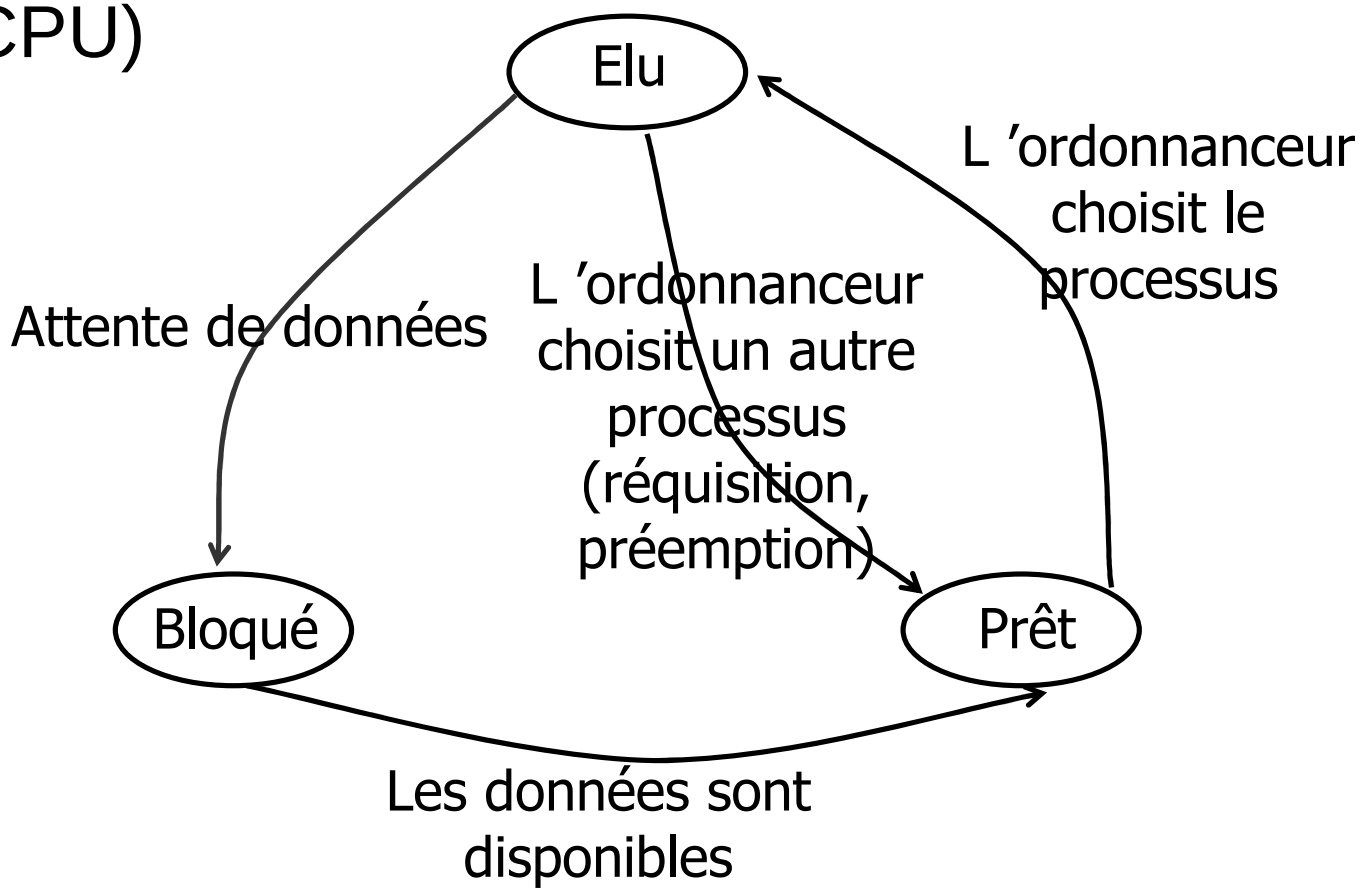
Les processus forment une arborescence via les liens « a créé ». Racine de l'arborescence processus de pid "1", le processus "init".



Etats des processus

Un processus peut ne pas être actif pour deux raisons différentes:

- attente de données (exemple prog1 | prog2)
- attente "d'unité centrale" (un seul processus est actif, par CPU)



Ordonnancement

Ordonnancement : choisir le processus à exécuter à chaque instant et la durée d'exécution par « Unité de calcul ».

– préemption : arrêt du processus en cours d'exécution

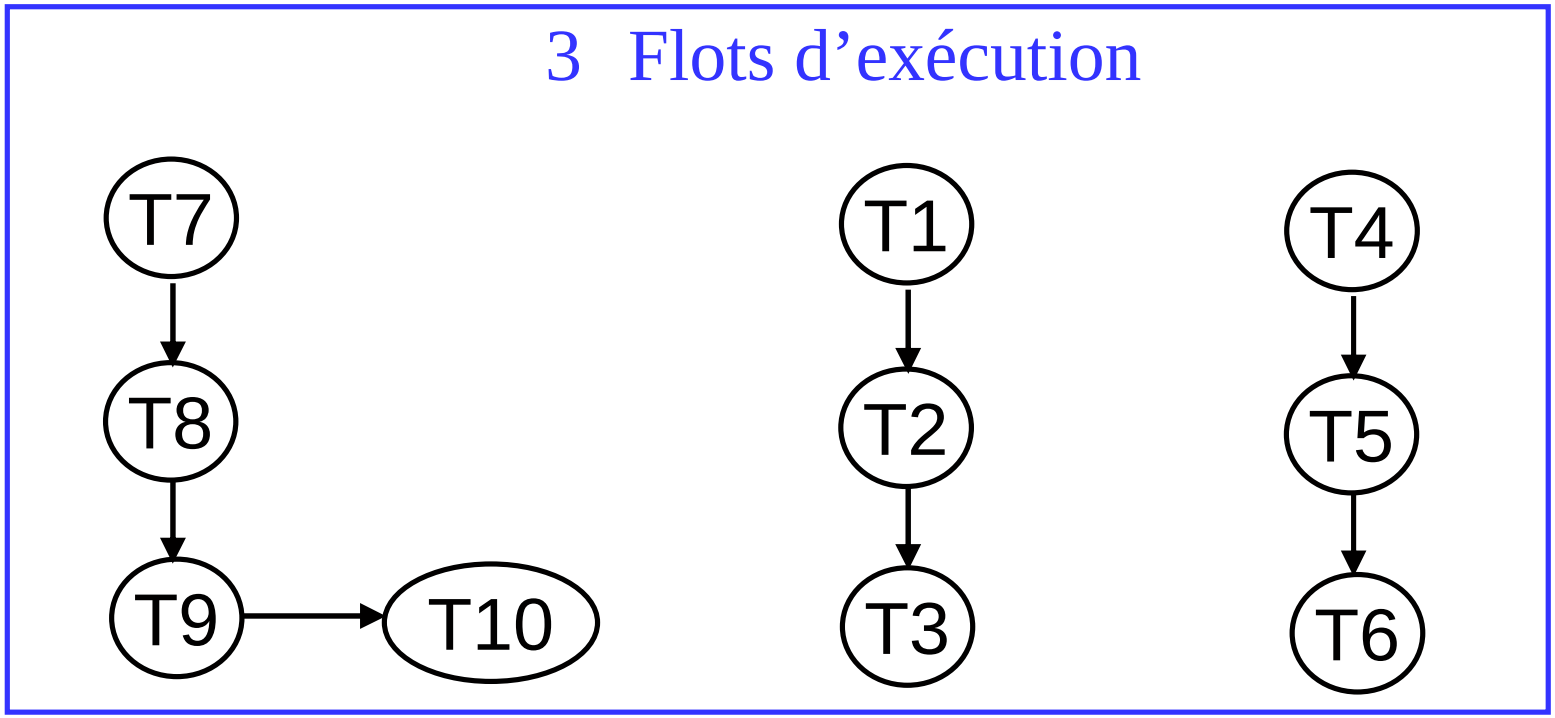
L'Ordonnanceur (le "scheduler") est chargé de ces choix.
Il utilise un algorithme d'ordonnancement

Sur Unix: le processus 0

Flots d'exécution

Flots d'exécution : suite de tâches s'exécutant de manière autonomes

- Processus au sein d'un ordinateur
- Processus légers (thread) au sein d'un programme

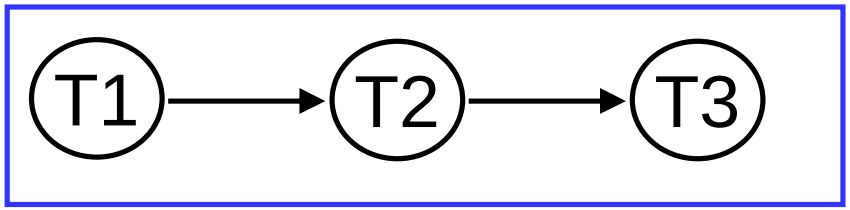


Programmation concurrente

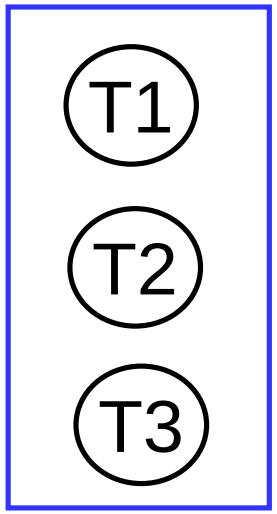
Programmation séquentiel : un seul flot d'exécution, les instructions s'exécutant d'une façon séquentielle

Programmation concurrente : Technique de Programmation où le programme consiste en plusieurs flots d'exécution parallèles

Un flot d'exécution



3 flots d'exécution

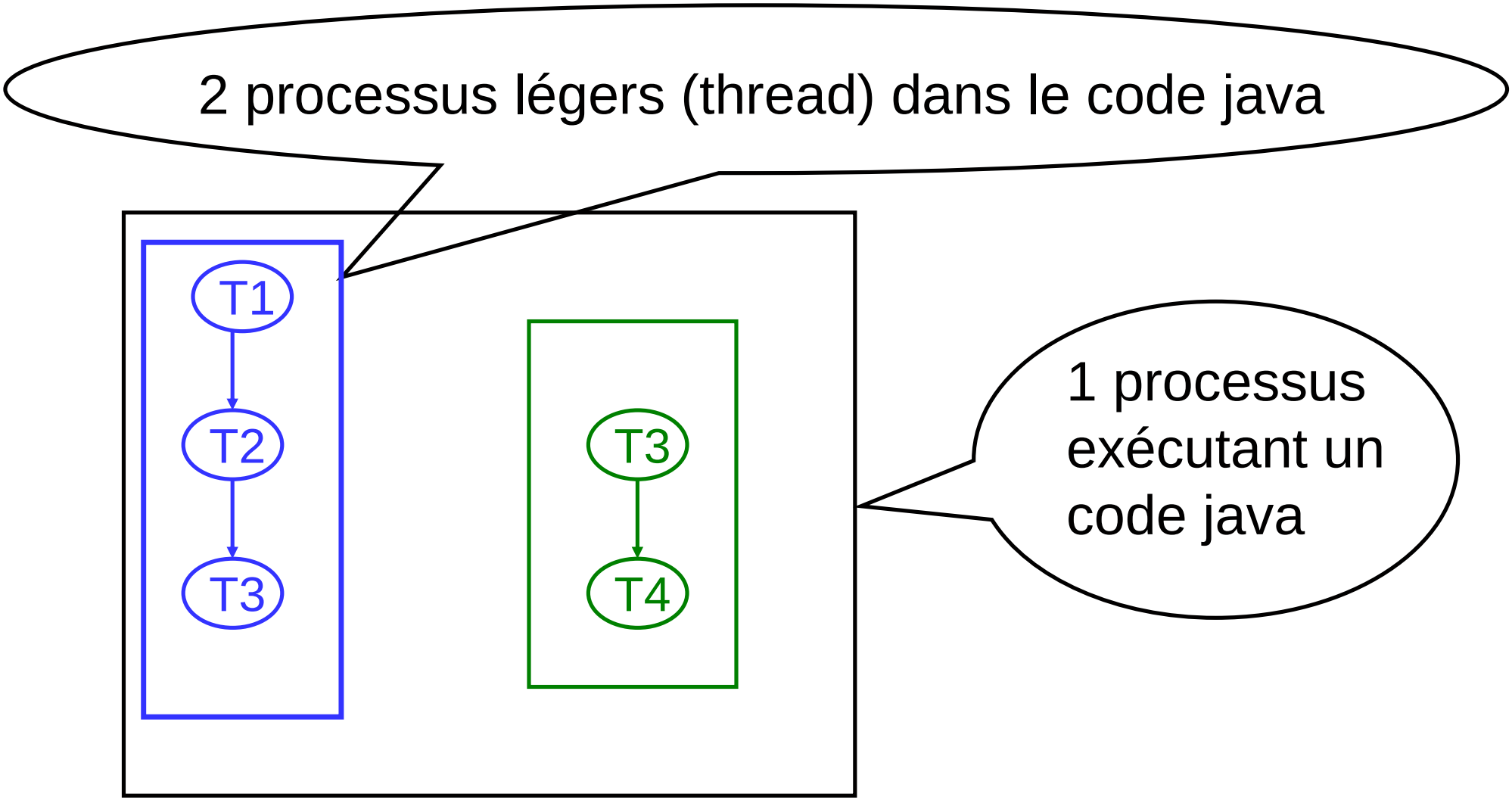


Problème lié à la Programmation concurrente

Dans un programme séquentiel l'ordre d'exécution des instructions élémentaires du programme est un unique
  – Il est identique d'une exécution à l'autre pour les mêmes paramètres en entrée – système déterminé

Dans un programme concurrent, l'ordre d'exécution des instructions forme un ordre partiel
  – Cet ordre varie d'une exécution à l'autre pour les mêmes paramètres en entrée – système indéterminé

2. Les Processus légers en Java



Exemple

```
// lancement de deux trains de vitesses différentes
class TrainDemarrage {  // classe de lancement
    public static void main ( String args [ ] )  {

 // création des trains
      Train tgv = new Train(10 , "TGV");
      Train teoz = new Train(20, "TEOZ");

// démarrage des trains (créations de 2 processus
// légers qui exécuteront la méthode run de Train)
      tgv.start();
      teoz.start();
      System.out.println("fin du main");
   }
}
```

Exemple (suite)

```
class Train extends Thread {
   int _vi ;  String _nom ;
   public Train(int v , String n)
                { _vi = v; _nom = n; }

   public void run () {
     System.out.println ("le train "+nom+" part" );
     try{
       Thread.sleep(vi*500) ; // sleep vi/2 secondes
       System.out.println("le train "+_nom+
                  " roule");
       Thread.sleep(vi*500);
       System.out.println("le train "+_nom+
                  " s'arrête");
     } catch (Exception e) { }
   }}
```

Exemple (résultat)

le train TGV part
le train TEOZ part
// Il y a 3 processus légers actifs
fin du main
// Il y a 2 processus légers actifs
le train TGV roule
le train TEOZ roule
le train TGV s'arrête
// le processus léger « TGV » a fin sa tâche
le train TEOZ s'arrête
// le programme java est fini, le dernier processus léger a terminé

La méthode start de Thread crée le processus léger
qui exécute la méthode run

Création des processus légers

La classe **`Train`** dérive de la classe Thread.

L'héritage multiple n'existant pas en java, il existe un deuxième mécanisme pour utiliser les Threads :

l'utilisation de l'interface **`Runnable`**

Utilisation de l'interface Runnable

```
class Train2Demarrage {  // classe de lancement
  public static void main ( String args [ ] )  {

// création des processus légers
   Thread tgv = new Thread(new Train2(10 ,"TGV")) ;
   Thread teoz = new Thread(new Train2(20,"TEOZ"));

// créations de 2 processus légers qui exécuteront la
// méthode run de Train2
 tgv.start () ; teoz.start ( ) ;
  System.out.println("fin du main") ;
   }}
```

Utilisation de l'interface Runnable (suite)

```
class Train2 implements Runnable {
  int _vi ;    String _nom ;

  public Train2(int v , String n) { … }

  public void run () {
    System.out.println("le train "+_nom +" part");
                …
    System.out.println("le train "+_nom+" s'arrête");
  }
}
```

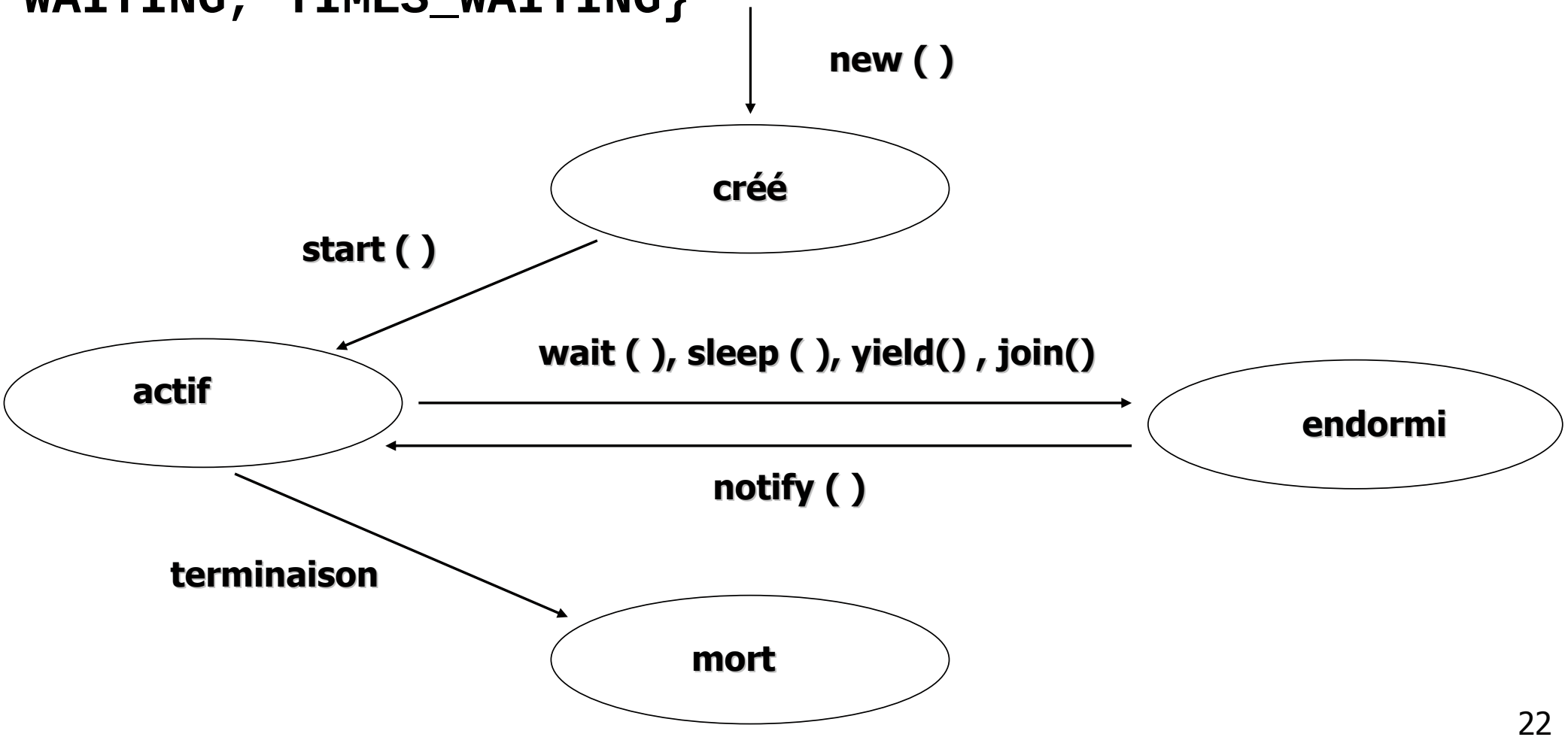
Utilisation de l'interface Runnable (résultat)

```
fin du main
                                        // processus léger « main » est fini
le train TGV    part
le train CORAIL part
                        // deux processus légers « train » s'exécutent
le train TGV    roule
le train CORAIL roule
le train TGV    s'arrête
                                              // fin du processus léger tgv
le train CORAIL s'arrête
                                   // plus de processus léger utilisateur,
                                              //le processus java est mort
```

Les états d'un processus léger

```
public Thread.State getState();
```

```
Thread.State = {NEW, TERMINATE, RUNNABLE, BLOCKED,
WAITING, TIMES_WAITING}
```



Les principales méthodes

static Thread currentThread ()

static int activeCount ()

static int enumerate (thread [] threadArray)

static void sleep () throws InterruptedException

void join () throws InterruptedException

static void yield ()

void setDaemond (boolean on)

boolean isDaemond ()

Lister les processus légers

```
Int nb = Thread.activeCount ( ) ;

Thread th[ ] = new Thread[nb] ;
nb = Thread.enumerate ( th ) ;

for ( i==0 ; i<nb ; i++ )
    System.out.println(
    «  nom du thread »+th[i].getName( ) ) ;
```

Les démons (daemon)

Il existe deux sortes de processus légers

les ordinaires (utilisateurs)

les démons (créés par la machine java)

Exemple de démon: le garbage collector

Deux méthodes

```
setDaemon (boolean b)
                    // à exécuter avant méthode start
boolean isDaemon( )
```

Un « processus » Java s'arrête lorsqu'il ne reste plus que des threads démons

L'ordonnancement (scheduling)

Question : Quel processus léger s'exécute à cet instant ?

Réponse : celui qui a la plus grande priorité parmi les threads actifs

Pour changer la priorité d'un thread :

- `int getPriority ( )`
- `void setPriority ( int newPrio )`

Thread.MIN_PRIORITY
Thread.MAX_PRIORITY
Thread.NORM_PRIORITY

Méthode **join**

```
…
    int at;
    at = Thread.activeCount();
    System.out.println("threads actifs  "+at);

// processus main attend la fin du processus TGV
    try { TGV.join(); } catch(Exception ex){};

    at = Thread.activeCount();
    System.out.println("threads actifs  "+at);
```

Arrêter un Thread (V1)

```
class Train extends Thread {
    private boolean arrete = false ;
    public void run () {
        while ( ! arrete ) {     ...  }
        System.out.println("le train s'arrête");
      }

    public void arreter( ) {
          arrete = true ;
      }
}
```

Arrêter un Thread (V2)

```
class Train extends Thread {
   private boolean arrete = false ;
   public void run () {
 while(!Thread.currentThread().isInterrupted())
         {  ...  }
      System.out.println("le train s'arrête");
   }

   public void arreter( ) {
         this.interrupt();
   }
}
```

Arrêter un Thread (fin)

```
public static void main ( String args[ ] )  {
    tt = new Train ( ) ;
    tt.start ( ) ;
    try{
        Thread.sleep(2000);
        tt.arreter( ) ; // on arrête le Thread tt
        tt.join( ) ;
    } catch (Exception ex)  { … }
    System.out.println ("fin du main") ;
}
```