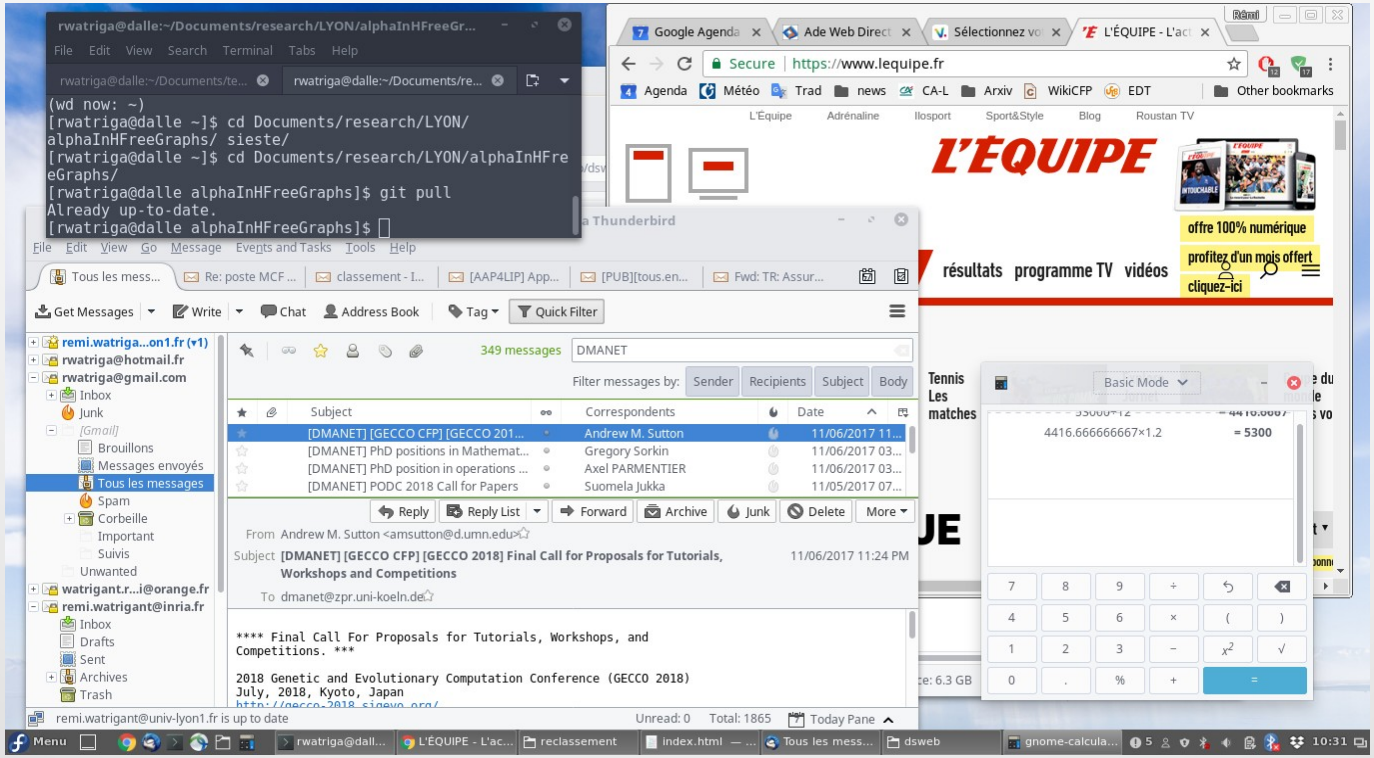
Licence Pro DevOps

Java :
Multi-threading

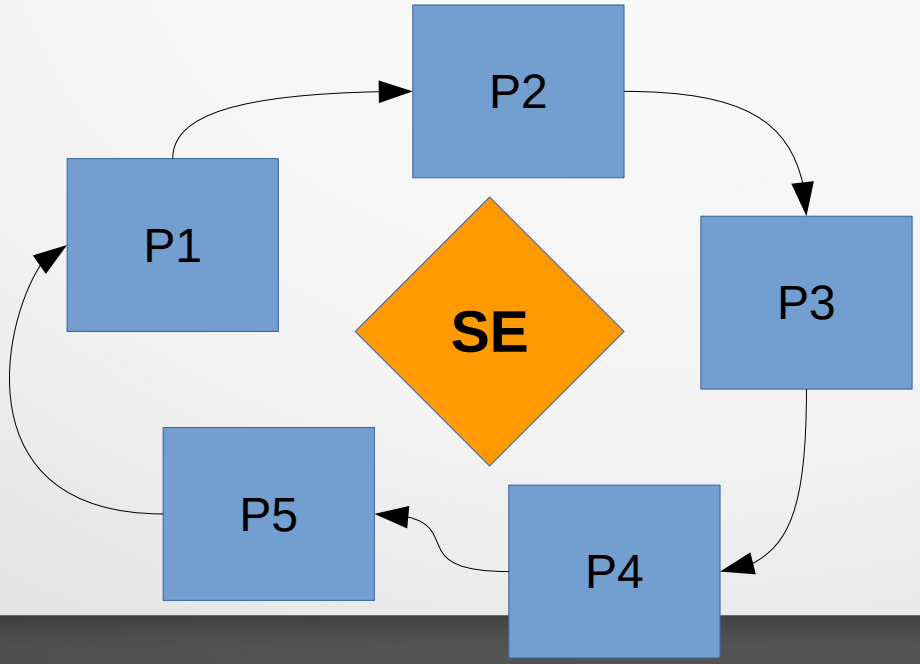
Introduction

- Un processeur n’est capable d’exécuter qu’une seule opération à la fois
- Pourtant, on veut pouvoir exécuter plusieurs tâches simultanément



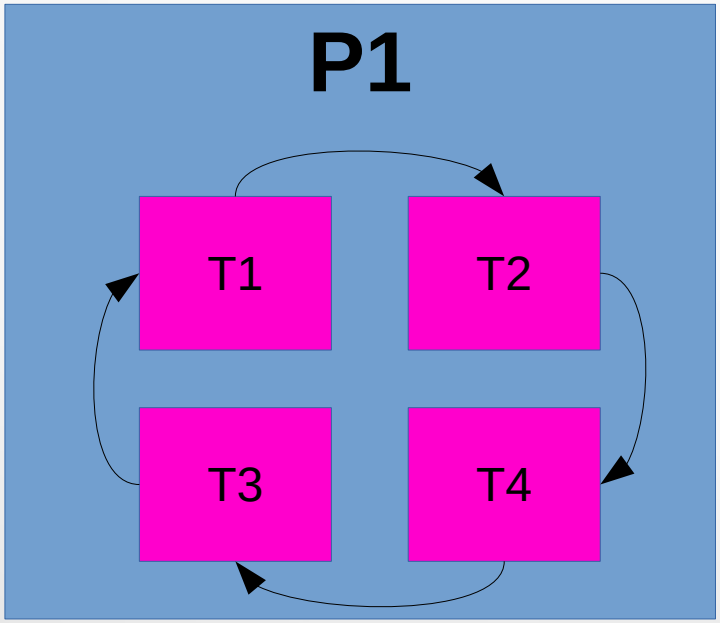
Introduction

- Un processeur n’est capable d’exécuter qu’une seule opération à la fois
- Pourtant, on veut pouvoir exécuter plusieurs tâches simultanément
- Pour cela, le système d’exploitation gère plusieurs processus, et leur donne la main à tour de rôle



Introduction

- Possibilité de reproduire le même schéma au sein d'un même processus.
- On appelle cela des **processus legers (threads)**



Introduction

- Possibilité de reproduire le même schéma au sein d'un même processus.
- On appelle cela des **processus legers (threads)**
- Différences avec les processus "classiques"
  - Chaque processus dispose de sa zone mémoire
    - Pile, tas, variables globales, environnement…
  - Les threads partagent une zone mémoire commune (appartenant au processus qui les a créés)
- Tout programme Java est lancé par la méthode **main**, le code est exécuté dans le **thread principal** qui peut alors lancer d'éventuels d'autres threads

Créer un thread

- La tâche à effectuer en parallèle sera représentée par une classe qui devra implémenter l'interface **Runnable** du package **java.lang**
- Cette interface ne comporte qu'une méthode à implémenter

```
public void run();
```

- Joue le rôle du "main" pour le nouveau thread
- La classe peut également contenir d'autres données (attributs et méthodes)

Créer un thread

```
public class ProcessusLegerParallele implements Runnable {
	@Override
	public void run() {
		for (int i = 0 ; i < 100 ; i++) {
			System.out.println("Je suis le nouveau thread !");
			System.out.flush();
		}
	}
}
```

Créer un thread

```
public class MonApplication {
   public static void main(String[] args) {
      ProcessusLegerParallele tache = new ProcessusLegerParallele();
      Thread thread = new Thread(tache);
      thread.start();
      for (int i = 0 ; i < 100 ; i++) {
         System.out.println("Je suis le thread principal !");
         System.out.flush();
      }
   }
}
```

Thread est une classe de la librairie standard java.lang.Thread

Créer un thread

Thread principal (début dans le main)

```
ProcessusLegerParallele tache = new ProcessusLegerParallele();

Thread thread = new Thread(tache);

thread.start();




for (int i = 0 ; i < 100 ; i++) {

       System.out.println("Je suis le thread principal !");

       System.out.flush();

}
```

Thread secondaire

```
public void run() {

           for (int i = 0 ; i < 100 ; i++) {
                System.out.println("Je suis le nouveau thread !");
                System.out.flush();
           }




     }
```

Créer un thread

- **Sortie :**

```
Je suis le thread principal !
 Je suis le nouveau thread !
Je suis le thread principal !
Je suis le thread principal !
Je suis le thread principal !
Je suis le thread principal !
Je suis le thread principal !
Je suis le thread principal !
 Je suis le nouveau thread !
 Je suis le nouveau thread !
 Je suis le nouveau thread !
 Je suis le nouveau thread !
 Je suis le nouveau thread !
 Je suis le nouveau thread !
 Je suis le nouveau thread !
 Je suis le nouveau thread !
 Je suis le nouveau thread !
            ...
```

Quelques méthodes...

- Pour avoir des informations ou agir sur le thread dans lequel on est : appels de **méthodes static** de la classe **Thread**
  - **static int activeCount()** : renvoie le nombre total de threads actifs actuellement (dans le processus)
  - **static Thread currentThread()** : référence vers le thread courant
  - **static void sleep(long millis)** : met le thread courant en attente pendant un nombre de millisecondes
- Pour avoir des informations ou agir sur un thread qu'on a lancé : appels de **méthodes** (d'un **objet de type Thread**)
  - **getPriority(), setPriority(int)** : priorité des threads
  - **join() :** attend que le thread meure
  - **wait(long timeout)** : permet d'attendre un certain laps de temps (permet aussi de fixer des rendez-vous entre threads grâce à la méthode **notify()**)

Partage de données, accès concurrent

- On a dit : “les threads partagent une zone de mémoire commune”

```
public class EncapsuleDonnee {
   private int x;
   public EncapsuleDonnee() {
      x = 0;
   }
   public void afficheEtIncremente(String message) {
      System.out.println(message+" x="+x);
      System.out.flush();
      x++;
   }
}
```

Partage de données, accès concurrent

```
public class ProcessusLegerParallele implements Runnable {
   private EncapsuleDonnee enc;
   public ProcessusLegerParallele(EncapsuleDonnee enc) {
      this.enc = enc;
   }

   @Override
   public void run() {
      for (int i = 0 ; i < 1000 ; i++) {
         enc.afficheEtIncremente("thread secondaire");
      }
   }
}
```

Partage de données, accès concurrent

```
public class MonApplication {


   public static void main(String[] args) {
      EncapsuleDonnee enc = new EncapsuleDonnee();
      ProcessusLegerParallele tache = new ProcessusLegerParallele(enc);
      Thread thread = new Thread(tache);
      thread.start();
      for (int i = 0 ; i < 1000 ; i++) {
            enc.afficheEtIncremente("thread principal");
      }
   }
}
```

Partage de données, accès concurrent

Sortie :

```
thread principal x=0
thread secondaire x=0
thread principal x=1
thread secondaire x=2
thread principal x=3
thread secondaire x=4
thread principal x=4
thread principal x=5
thread secondaire x=6
thread principal x=6
       ...
thread principal x=1997
thread secondaire x=1998
thread secondaire x=1999
```

Partage de données, accès concurrent

- Explication :
  - La méthode **afficheEtIncremente** de la classe **EncapsuleDonnee** est appelée par deux threads différents A et B
  - Le thread A peut exécuter la première instruction de la méthode (afficher x) puis passer la main au thread B qui exécute aussi la première instruction (ils affichent donc la même valeur)
  - Puis le thread A reprend la main, incrémente x
  - Puis le thread B reprend la main, incrémente x
  - ...etc
- On ne contrôle à priori pas l'ordonnancement

Partage de données, accès concurrent

- Que faire si la méthode est une **“zone sensible”** ?
- Mot-clé **synchronized :**

```
public class EncapsuleDonnee {

    private int x;

    public EncapsuleDonnee() {
        x = 0;
    }

    public synchronized void afficheEtIncremente(String message) {
        System.out.println(message+" x="+x);
        System.out.flush();
        x++;
    }
}
```

- Contraint la méthode à n’être exécutée que par un seul thread à la fois

Partage de données, accès concurrent

- Sortie :

```
thread principal x=0
thread principal x=1
thread principal x=2
thread principal x=3
thread secondaire x=4
thread secondaire x=5
thread secondaire x=6
thread secondaire x=7
thread secondaire x=8
thread secondaire x=9
thread secondaire x=10
thread secondaire x=11
thread secondaire x=12
thread secondaire x=13
thread secondaire x=14
thread principal x=15
thread principal x=16
thread principal x=17
thread principal x=18
        ...
```