(Archi4)
Architecture 4
Programmation répartie
Accès concurrentiels

Lecture *Architecture 4*
Programmation répartie 26 janvier 2021

Miklós MOLNÁR

contacter
molnar@lirmm.fr
Institut Universitaire de Technologie de Montpellier

Objectifs de ce module

Motivation

- Souvent, les ressources critiques sont accédées par plusieurs processus, programmes simultanément
  - Exemples : CPU, imprimante, clavier,...
    ==> Il faut organiser l'accès
- Des dépendances peuvent exister entre des tâches
  - Exemples : Un calcul doit précédé un autre,...
    ==> Il faut les synchroniser
- Des communications / coopérations entre des processus locaux mais aussi distants sont nécessaires
  - Exemples : récupérer des données, stocker des résultats,...
    ==> Il faut établir des communications
- Pour réduire le temps d'exécution de certains algorithmes, la programmation distribuée est nécessaire
  - Exemples : jeux interactifs, calcules scientifiques, simulations,...
    ==> Il faut segmenter et organiser le fonctionnement

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Objectifs de ce module

Nos investigations :

- Rafraîchir les connaissances sur les processus et sur les threads
- Etudier de problèmes issus du fonctionnement parallèle
  - Utilisation des ressources critiques et/ou limitées
  - Synchronisation des processus
  - Ordonnancement, etc.
- Etudier des communications / coopération entre des processus locaux mais aussi distants
  - Communications entre processus
  - Communications entre tâches (*threads*)
    - Réaliser certains services/communications

On va prendre les implémentations en JAVA

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Objectifs de ce module

Miklós MOLNÁR

contacter
molnar@lirmm.fr

De qui s'agit-il quand on parle de

- Concurrence
  - Activités qui existent et qui peuvent avoir besoin des mêmes ressources
    (pas forcement au même moment, mais
    "quasi-simultanément")
- Parallélisme
  - Activités qui sont exécutées au même moment
    (des activités concurrentielles peuvent être exécutées "quasi-parallèlement" avec des "time-slicing" et des interruptions)

Contenu

- Processus et tâches
- Utilisation concurrentielle des ressources
  - On étudie les problèmes de la concurrence
  - On définie les ressources/ sections critiques
  - On réalise des exclusions mutuelles, ...
- Outils
  - Verrouillage, verrous
  - Barrières
  - Sémaphores
- Synchronisation des processus
- Communication entre processus distants
  - Boîtes aux lettres
  - Sockets
  - Modèle client/serveur

Miklós MOLNÁR

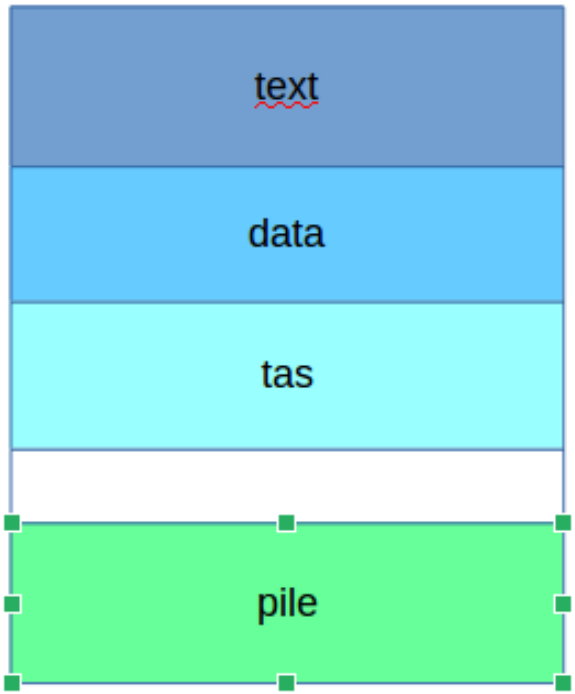
contacter
molnar@lirmm.fr

Processus

Processus :*un programme en cours d'exécution*
(un programme peut être lancé plusieurs fois)

Quels sont les segments d'un programme exécutable ?

- Chaque processus possède :
  - un segment (zone d'adressage contiguë) de texte (du programme) qui est invariant
  - un segment de données qui s'évolue avec l'exécution



Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Processus

Processus :*un programme en cours d'exécution*
(un programme peut être lancé plusieurs fois)

Pour un processus et une exécution :

- *Contexte du processus* : il doit donner les informations sur l'exécution (des fonctions par exemple)
  - réalisé sous forme d'une pile d'exécution (pour sauvegarder les contextes, les valeurs passées...)
- *Contexte du processeur* : il doit donner les informations sur l'état du processeur
  - contenu des registres, du compteur ordinal, drapeaux...
- Pour les exécuter, il faut connaître le contexte du processus et celui du processeur

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Processus

L'espace d'adressage (composée des segments) : ce qu'un exécutable peut gérer

- processus
  - chaque processus gère son espace
- thread
  - les threads partagent une espace, sauf la pile
- protothread
  - threads mais sans une pile privée (ils partagent l'espace entière)
- co-routines
  - threads coopératifs (non préamptifs)

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Processus

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Exécution de plusieurs activités (processus)

- coopérative
  - les programmes (processus) sont présents en mémoire et gèrent seuls leur exécution
    - Danger qu'un processus ne rend pas la main
- préamptive
  - les programmes sont régulièrement interrompus
    - un ordonnanceur (scheduler) existe
    - Round Robin ou "interruption based"

Création des processus

- L'appel système `fork()` crée un processus fils par une copie de son père (zone du texte et de données)

```
int main(int argc, char **argv){
int s = fork();
 if (s != 0 )
   printf("[père] : %d \n",s);
 else
   printf("[fils] : %d \n",s);
   }
```

- Le fils "hérite" des fichiers, des redirections, mais utilise des variables, des tampons différents
- Dans le code
  - le fils créé reçoit le code de retour = 0
  - le père reçoit le PID de son fils
  - pas de communication entre les deux par les variables (chacun a une copie non partagée des variables)

Les processus peuvent partager le code mais pas les données

Comment peuvent-ils communiquer ?

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Communication entre processus (rappel)

Différentes communications sont possibles entre les processus (potentiellement concurrentiels dans un même système)

- par code de retour (`exit(n)`)
- par paramètres passés (arguments d'un programme)
- par fichiers
- par tubes (anonymes, nommés, processus)
- par files de messages
- par signaux

Si les processus qui veulent communiquer tournent sur des machines différentes, ils doivent

- être connectés (via le réseau)
- utiliser les services du réseau
- résoudre des éventuels problèmes de communications, synchronisation et de la concurrence

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Etats d'un processus

A un moment donné de son existence, un processus est dans un des états possibles suivants

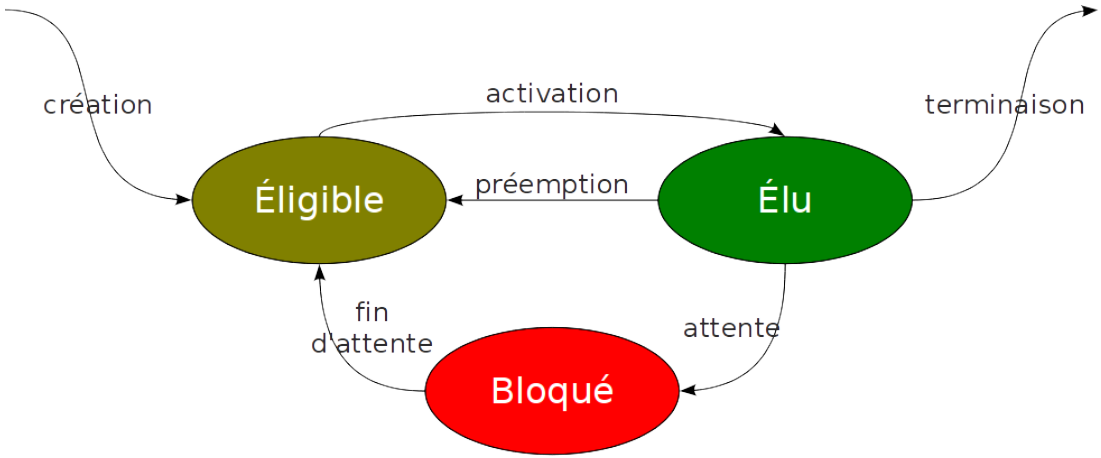


FIGURE – Etats-transitions (simplifiés) d'un processus

- Simplifications : pas de swap, pas d'exécution en mode noyau...

Qui décide l'exécution d'un processus ?

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Processus et JAVA

- Ici, on utilise JAVA qui s'évolue sans cesse
  - Des éléments de la programmation distribuée et des conteneurs génériques sont apparus
  - Nous restons à un niveau simple de la programmation JAVA

Comment créer un "exécutable" en JAVA (par exemple prog.class) ?

Comment exécuter un "exécutable" en JAVA ?

Quel est le processus qui s'exécute ?

Comment faire simplement des programmes distribués ?

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Processus légers - threads

Thread : sous-processus qui s'exécutent au sein d'un processus

- Chaque processus a une tâche initiale (main)
- Chaque thread possède :
  - son propre mot d'état (PSW), son propre contexte du processeur
  - une pile (variables locales)
- Les threads partagent :
  - le code
  - les données globales et/ou statiques
  - le tas
- Intérêt : si plusieurs processeurs existent dans le système, des parties peuvent être exécutées "indépendamment"
- Comme certaines données sont partagées, il faut éviter les conflits et les incohérences

Quelles sont des données partagées ?

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Processus légers en C

Comment les threads sont construits en C ?

- Quelques éléments (selon la proposition POSIX) :
  - *pthread_t desc ; // type*
  - *pthread_create((&desc), NULL, fonction, NULL) ; // creation*
  - *pthread_join(desc, NULL) ; // attente*
- Exemple

```
 include <pthread.h>
void* fonction(void *arg) { // activité
while (1) { sleep(50); printf("Bonjour"); }
return NULL;}
int main() {
pthread_t thread;
...
if (pthread_create(&thread, NULL, fonction, NULL) < 0) {
perror("pb de thread"); exit(1); }
...
pthread_join(thread, NULL);return 0; }
```

Quand sera le thread lancé ?

Que se passe-t-il sans *pthread_join* ?

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Processus légers en Java

- Les threads sont gérées de base par la classe `Thread`
- On peut les créer de deux façons différentes
  - par héritage de la classe `Thread`
  - par implémentation de l'interface `Runnable`

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Création des threads, héritage de la classe Thread

- En Java, la classe qui hérite de la classe `Thread` doit surcharger la méthode `run()`

```
class A_tache extends Thread {
    A_tache() { ... }  // constructeur
    public void run() { ...  } //  activité
}
```

- Une instance de `A_tache` est lancée avec la méthode `start()` définie par la classe `Thread`

```
A_tache a = new A_tache();
a.start();
```

- Après `start()`, l'objet passe à l'état "Eligible" (mais ne se démarre pas forcement immédiatement). C'est la machine virtuelle Java qui démarre le fonctionnement par l'exécution de la méthode `run()`
- Le thread se termine quand sa méthode `run()` se termine

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Création des threads, héritage de la classe Thread

- Exemple

```
public class exemple extends Thread {
 public void run() {System.out.println("Bonjour." );
          ... }
 public static void main(String args[]) {
      exemple T1 = new exemple(1);
      T1.start();
      System.out.println("main : T1 lancee" );
      ... }
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

Thread par implémentation de Runnable

- On peut aussi créer un thread par une classe qui implémente l'interface `Runnable` qui nécessite la méthode `run()`

```
class B_tache implements Runnable {
      B_tache() { ... } // constructeur
public void run() { ...  } //  activité
}
```

cette classe n'est pas un thread...

- La classe `Thread` a un constructeur qui prend une instance implémentant `Runnable` en argument. Sur une telle instance de `Thread`, on appelle la méthode `start()` pour exécuter la tâche

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

- Le thread se termine quand sa méthode `run()` se termine

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Méthodes utiles de la classe Thread

La classe `Thread` possède des méthodes pour contrôler le comportement des threads

- `static void sleep(long ms)`

  le thread sera bloqué pendant `ms` millisecondes (d'autres threads peuvent alors s'exécuter)
  Attention : appel avec `Thread.sleep(400)`

- `boolean isAlive()`

  elle retourne vrai si le thread est vivant (sa méthode `run()` n'est pas encore terminée)

- `Thread.State getState()`

  elle retourne l'état

Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Méthodes utiles de la classe Thread

Gestion de la priorité

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

  traitent la priorité du thread
- Constantes pour les priorités

  ```
  int  Thread.MIN_PRIORITY
  int  Thread.MAX_PRIORITY
  int  Thread.NORM_PRIORITY
  ```

- pas de garantie sur l'exécution des threads.
- Sous LINUX, le thread qui a la plus grande priorité a accès au processeur s'il n'est pas bloqué...
- En général, une priorité supérieure permet d'augmenter les chances d'exécution.

- ```
  static void yield();
  ```

  le thread appelant passe de l'état "Elu" (Running) à l'état "Eligible" (Runnable)
  (elle donne plus de chance aux autres)

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Threads et programme principale

En C, quand le programme principale se termine, on quite l'exécutable

- pour attendre la fin de l'exécution du thread T1 dans le main() d'un programme en C :

```
int main() {

  pthread_t T1;
   int  iret1 = pthread_create( &T1, NULL, th_function,
                                (void*) message1);
...
   /* attendre que le thread T1 se termine   */
   pthread_join( T1, NULL);
  exit(0);
}
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

Threads et programme principale

Contrairement aux normes POSIX, les `Threads` en Java ne sont pas arrêtés quand le programme principale (le thread main) se termine

- En Java, le programme se termine quand les Threads lancés sont terminés

```
public class ex2 extends Thread {
public void run() {... des impressions : Je suis T1... }

public static void main(String args[]) {
    ex2 T1 = new ex2(1);        ex2 T2 = new ex2(2);
    T1.start();          T2.start();
    System.out.println("main : T1, T2 sont lancees" );
    System.out.println("main : termine" ); }
```

```
>>> TD1$ java ex2
main : T1, T2, T3 sont lancees
main : termine
Je suis T1 1 ...
Je suis T2 1 ...
```

- Cependant, un Thread peut attendre la fin de l'exécution d'un autre Thread (par ex. t2) en utilisant la méthode `join()`

```
t2.join();
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Partage des variables entre threads

Les threads d'un même processus partagent la mémoire

- Ils peuvent accéder aux variables globales
- Ils ont une pile (des variables locales propres) chacun
- Ils instancient une (des) classe(s) ; les règles de l'accessibilité des membres sont les règles connues pour les classes
- Rappels
  - données propres à l'objet vs. données statiques (partagées)
  - données privées, protégées et publiques

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Exemple du partage

```
public class partage extends Thread {
    private static String chaine = "";
    private String nom;
    partage ( String s ) {
        nom = s;
    }
    public void run() {
       for (int i = 0; i<10; i++)
        {
        chaine = chaine + nom;
        try {
             Thread.sleep(100);  // milliseconds
            } catch(InterruptedException e) {}
        }
    }
    public static void main(String args[]) {
        Thread T1 = new partage( "T1" );
        Thread T2 = new partage( "T2" );
        T1.start();
        T2.start();
        try {
             sleep(1000);  // milliseconds
            } catch(InterruptedException e) {}
        System.out.println( chaineCommune );
    }
}
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Résultats de l'exécution

- ```
  molnar@molnar-laptop $ java partage
  T1T2T1T2T2T2T2T1T2T1T2T1T2T1T2
  ```
- Au lieu de 20 éléments, il n y a que 15

Comment expliquer ces résultats ?

Miklós MOLNÁR

contacter
molnar@lirmm.fr
  ```

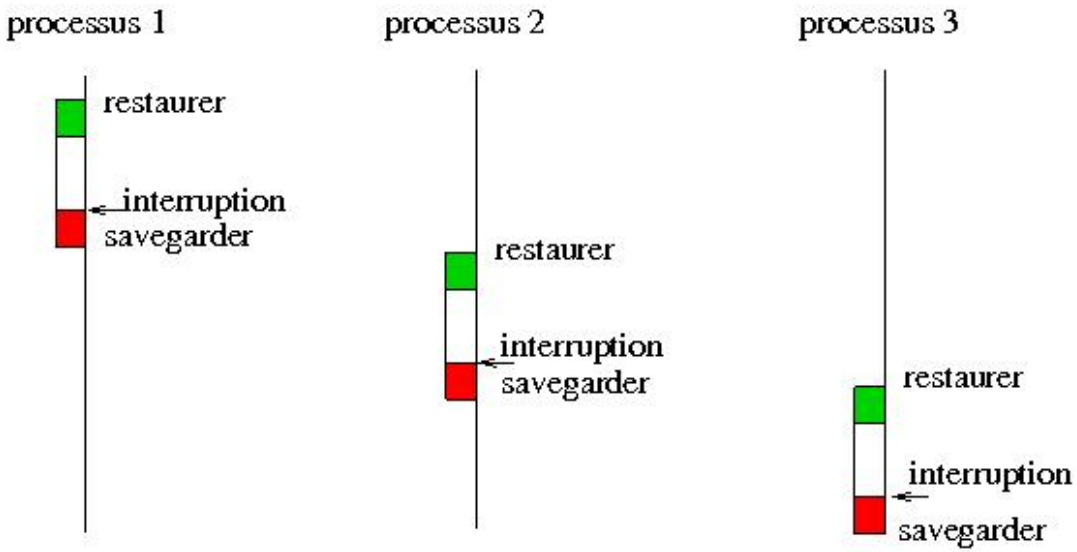
Exécution des programmes /threads concurrentiels

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Cas de base

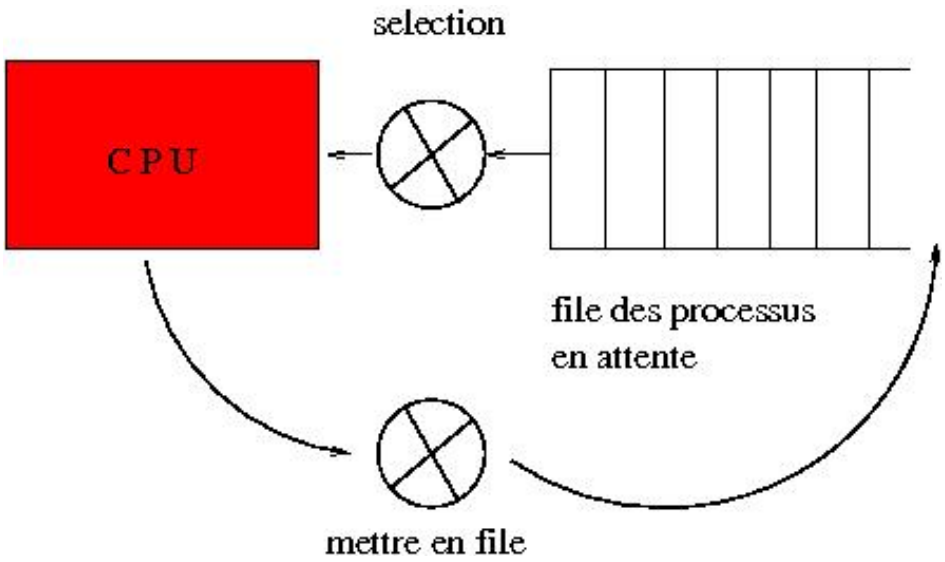
- Supposons un seul processeur partagé par plusieurs processus / threads



- Le contrôle passe d'un thread à l'autre
  - seul le contexte des threads (par exemple la pile) doit être changé

Exécution des programmes /threads concurrentiels

Ordonnancement



- Il existe un ordonnanceur (scheduler) pour gérer les processus /threads
- Les tâches initialisées, interrompues, débloquées sont mises dans une file
- Des politiques différentes peuvent être envisagée pour gérer les files d'attente
  - réaliser un tourniquet (fair play)
  - sélection basée sur des priorités ...

Que fait la commande *nice* sous linux ?

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Ordonnancement

- Le thread JAVA à exécuter est choisi parmi les threads qui sont éligibles.
- L'ordonnanceur JAVA est
  - *préemptif* (l'ordonnanceur peut interrompre une tâche pour donner les ressources à une autre tâche)
  - basé sur une *priorité* (l'ordonnanceur essaye de donner les ressources à une tâche prioritaire)
- L'ordonnanceur dépend de l'implémentation de JVM :
  - *green thread :* c'est la JVM qui implémente l'ordonnancement (UNIX)
  - *thread natif :* c'est le système d'exploitation hôte de la JVM qui effectue l'ordonnancement

Architecture 4
Programmation réparti

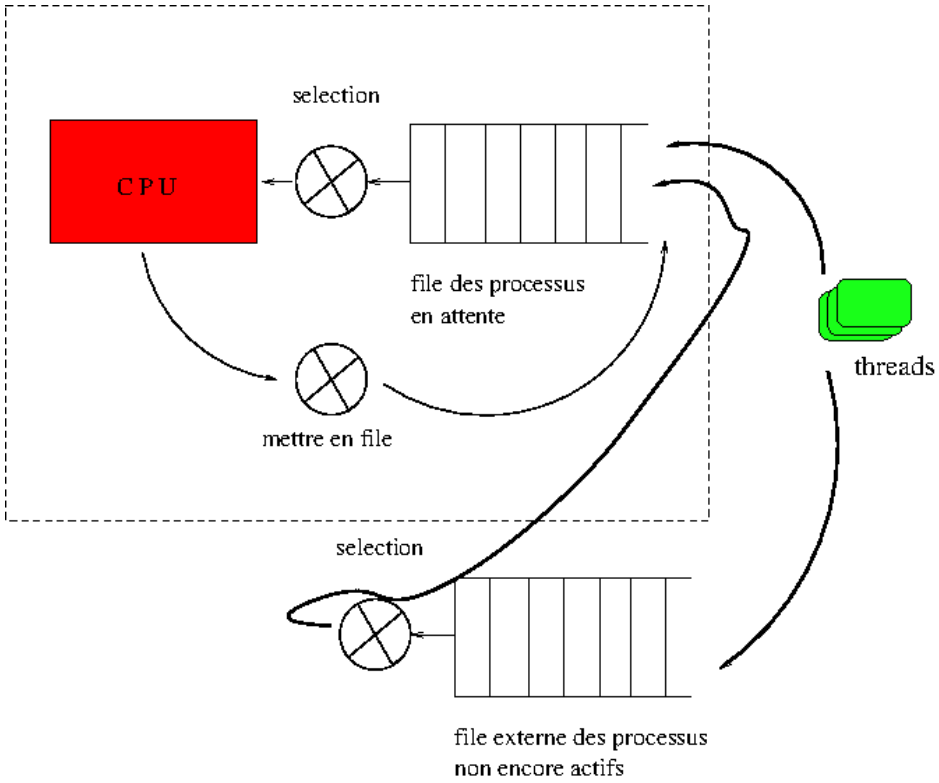
Miklós MOLNÁR

contacter
molnar@lirmm.fr

Un ensemble de tâches

Thread Pool

- Le système gère les tâches selon sa propre politique et selon l'état des tâches
- Un nombre important de threads peut encombrer la JVM
- On peut avoir besoin des gestionnaires différents
- Créer des ensembles de threads, gérés par un gestionnaire à part : thread pools



Miklós MOLNÁR

contacter
molnar@lirmm.fr

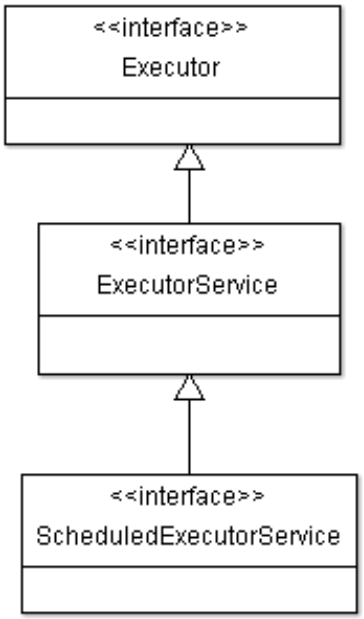
I.U.T MONTPELLIER

Gestionnaires d'ensembles de tâches

- Il est possible de créer des gestionnaires de tâches qui gèrent un ensemble de tâches
  - La politique, la gestion des threads peut être paramétrée
- **Les comportements sont définis par les interfaces** :

```
Executor  (pour Runnable)
ExecutorService  (pour Runnable et Callable)
ScheduledExecutorService  (pour exécution périodique)

===>   import java.util.concurrent.Executor;
```



Miklós MOLNÁR

contacter
molnar@lirmm.fr

I.U.T MONTPELLIER

Gestionnaires d'ensembles de tâches

Un `Executor` gère une file d'attente bloquante de tâches à effectuer.

- Idée : *dissocier la soumission des traitements de leur exécution*
- En général, le nombre de tâches actives et la politique de la gestion de la file et des tâches actives peuvent être paramétrés.

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Gestionnaires d'ensembles de tâches

Interface Executor

- Elle prévoit de "gérer" simplement des tâches
- Une seule méthode dans l'interface "mère" :

```
void execute(Runnable command)
```

- La tâche peut être exécutée dans un thread dédié ou dans le thread courant
- Une implémentation simple dans le thread courant :

```
class MonExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```

- Une autre qui crée explicitement un thread par appel :

```
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Gestionnaires d'ensembles de tâches

`Interface ExecutorService`

Des ajouts :

- Pour gérer la durée de vie du gestionnaires :

```
...
boolean isTerminated()
...
boolean awaitTermination( ...) // bloquant
...
void    shutdown()    // fermeture mais pas arrêt
...
List<Runnable>  shutdownNow()
...
```

- Pour avoir des informations sur les tâches lancées : elle permet de gérer des `Runnables` (execute)
mais aussi les `Callables` (submit)
(on va les voir plus loin)

```
<T> Future<T> submit(Callable<T> task)
...
```

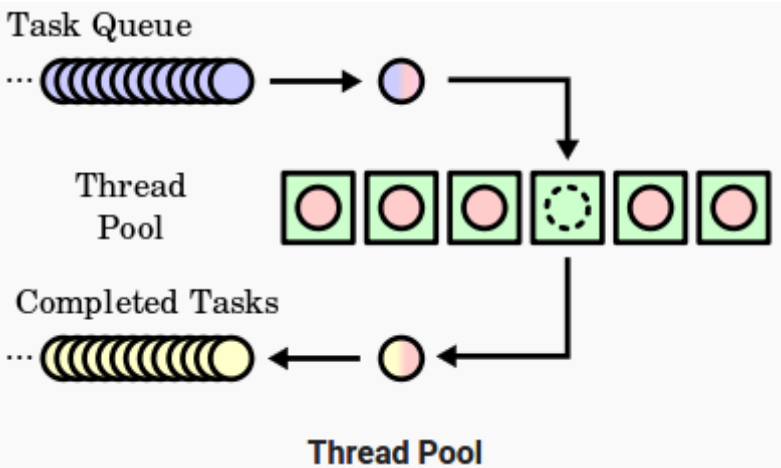
Gestionnaires d'ensembles de tâches

`Interface ExecutorService`

- Exemples des classes qui l'implémentent :

```
public abstract class AbstractExecutorService
          extends Object implements ExecutorService
...
public class ThreadPoolExecutor
          extends AbstractExecutorService
```

- Pour optimiser la performance (avec N threads au max) :



Miklós MOLNÁR

contacter molnar@lirmm.fr

Gestionnaires d'ensembles de tâches

- Des instances de gestionnaires peuvent être crées par la fabrique `Executors`

```
                              AbstractExecutorService
                              ThreadPoolExecutor
                              ScheduledThreadPoolExecutor ...

 ===>    import java.util.concurrent.Executors;
```

- Exemples de créations d'`Executor` par la classe `Executors`

```
static ExecutorService  newSingleThreadExecutor()
...
static ExecutorService  newFixedThreadPool(int nThreads)
...
static ScheduledExecutorService
   newScheduledThreadPool(int corePoolSize,
                          ThreadFactory threadFactory)
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Exemple de Executor

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
class ex_thread implements Runnable {
  private int nbr;
    // constructeur
  ex_thread (int nb) { nbr=nb; }
  public void run() {
    for (int nombre=1; nombre <10;nombre++) {
      System.out.println("appel de thread"+nbr+" "+nombre);
      try {    Thread.sleep(50); // milliseconds
      } catch (InterruptedException e) {   }
    }
  }
}
public class executor_ex {
    public static void main(String args[]) {
     Executor executor = Executors.newSingleThreadExecutor();
        //fabrique d'executor
     ex_thread ex1 = new  ex_thread(1);
     ex_thread ex2 = new  ex_thread(2);
     executor.execute( ex1);
     executor.execute( ex2);
     }
}
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Résultats de l'exécution

```
miklos@miklos-laptop $ java executor_ex
appel de thread 1 1
appel de thread 1 2
appel de thread 1 3
appel de thread 1 4
appel de thread 1 5
appel de thread 1 6
appel de thread 1 7
appel de thread 1 8
appel de thread 1 9
appel de thread 2 1
appel de thread 2 2
appel de thread 2 3
appel de thread 2 4
appel de thread 2 5
appel de thread 2 6
appel de thread 2 7
appel de thread 2 8
appel de thread 2 9
```

- Ce gestionnaire exécute les activités de manière asynchrone vis-à-vis du thread courant, mais assure qu'elles seront appelées dans l'ordre de leurs lancements

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Un autre Executor

- `ExecutorService es = Executors.newFixedThreadPool(int nbTaches)`
  - nbTaches : nombre maximum d'exécutions simultanées
  - une file d'attente aux autres tâches pour attendre qu'une active se termine
- `execute(obj)` pour lancer une tâche
- `shutdown()` pour terminer l'exécution du groupe de tâches, n'accepte plus de nouvelles tâches mais exécution de celles de la file d'attente
- `shutdownNow()` arrêt immédiat
- `isTerminated()` true si toutes les tâches sont terminées => attente active

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Un autre Executor

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
 class ex_thread implements Runnable {
    private int nbr;
    ex_thread (int nb) {  this.nbr=nb; }
    public void run() {
       for (int nombre=1; nombre <10;nombre++) {
            System.out.println("appel de thread " + nbr + " " + nombre);
            try {    Thread.sleep(50);
               // ou encore  Thread.sleep(nbr*50);  <<<<<<<<<<
            } catch (InterruptedException e) {   }
        }
    }
}
public class executor_ex2 {
    public static void main(String args[]) {
     Executor executor = Executors.newFixedThreadPool(2);
         //fabrique d'executor
     ex_thread ex1 = new  ex_thread(1);
     ex_thread ex2 = new  ex_thread(2);
     ex_thread ex3 = new  ex_thread(3);
     executor.execute( ex1);
     executor.execute( ex2);
     executor.execute( ex3);
     }
}
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Résultats de l'exécution

```
miklos@miklos-laptop $ java executor_ex2
appel de thread 1 1
appel de thread 2 1
appel de thread 2 2
appel de thread 1 2
appel de thread 2 3
appel de thread 1 3
appel de thread 2 4
appel de thread 1 4
appel de thread 2 5
appel de thread 1 5
appel de thread 2 6
appel de thread 1 6
appel de thread 2 7
appel de thread 1 7
appel de thread 1 8
appel de thread 2 8
appel de thread 2 9
appel de thread 1 9
appel de thread 3 1
appel de thread 3 2
appel de thread 3 3
appel de thread 3 4
appel de thread 3 5
appel de thread 3 6
appel de thread 3 7
appel de thread 3 8
appel de thread 3 9
```

- avec les mêmes délais dans "sleep"

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Résultats de l'exécution

```
miklos@miklos-laptop $ java executor_ex2
appel de thread 1 1
appel de thread 2 1
appel de thread 1 2
appel de thread 2 2
appel de thread 1 3
appel de thread 1 4
appel de thread 2 3
appel de thread 1 5
appel de thread 1 6
appel de thread 2 4
appel de thread 1 7
appel de thread 1 8
appel de thread 2 5
appel de thread 1 9
appel de thread 3 1
appel de thread 2 6
appel de thread 2 7
appel de thread 3 2
appel de thread 2 8
appel de thread 3 3
appel de thread 2 9
appel de thread 3 4
appel de thread 3 5
appel de thread 3 6
appel de thread 3 7
appel de thread 3 8
appel de thread 3 9
^C   //pourquoi ???
miklos@miklos-laptop $
```

- avec les délais dans "sleep" proportionels au numéro de thread

pour réaliser les TDs, des objets Executor sont fortement conseillés

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Problème de l'interface Runnable

- L'interface a deux limitations :
  - La méthode run() ne peut renvoyer aucune valeur (cf. void)
  - On ne peut lancer aucune exception

- Nouvelle interface dans (java.util.concurrent.Callable)

```
import java.util.concurrent.Callable;

public interface Callable<V> {
    public V call() throws Exception;
}
```

- un résultat du type V
  (dans le sens de la programmation template)

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Exemple de l'interface Callable<V>

```
import java.util.concurrent.Callable;
public class MonCallable implements Callable<Integer> {

    public Integer call() throws Exception  {
        try {
            Thread.sleep(50);
            //ici c'est le traitement.
            System.out.println("␣traitement␣Callable");
        }
        catch(InterruptedException e){
            throw new Exception("Thread␣interrompu␣;␣cause␣"
                                + e.getMessage());
        }
        return Integer.valueOf(1); //On retourne un entier
    }
}
```

Utilisation des gestionnaires pour des Callable<V>

- On peut utiliser les *ExecutorService*s pour soumettre des *Callables* en utilisant une méthode

  submit (au lieu de *execute*).

```
public class Main {
  public static void main(String[] args) {
    ExecutorService ex = Executors.newSingleThreadExecutor();
    Future<Integer>  res = ex.submit(new MonCallable());  // Retourne un

    . . .

    ex.shutdown();
    }
```

Architecture 4
Programmation réparti

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Utilisation des gestionnaires pour des Callable<V>

- L'interface *Future<V>* donne des fonctionnalités pour gérer le cycle de vie de l'exécution d'une tâche.

```
boolean cancel(boolean)

V   get() //bloquant pour attendre le resultat du type V

V   get(long timeout, TimeUnit unit) // avec timeout

boolean isCancelled()

boolean isDone()
```

- la méthode get() permet de récupérer le résultat via l'objet Future

Miklós MOLNÁR

contacter
molnar@lirmm.fr

Utilisation des gestionnaires pour des Callable<V>

- Exemple de *Future<V>*

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
...
    Integer  resultat;
    Future<Integer>  res = ex.submit(new MonCallable());
                     // Retourne un Future
...
while (!res.isDone()) {
 System.out.println("attente ");
  try { Thread.sleep(200);} catch (InterruptedException e){ }
  }
...
try { resultat = res.get();} catch (ExecutionException e){ }
                            catch (InterruptedException e){ }
```


Miklós MOLNÁR

contacter
molnar@lirmm.fr