Page de Jean-Michel Richer

Maître de Conférences en Informatique

<<<M1 - UE2

1 Rappels programmation concurrente et threads

Note : on pourra se référer à l'API JavaSE 7 pour plus de détails.

1. Thread

En informatique, un **thread d'exécution** (ou plus simplement thread) est la plus petite partie qui peut être traitée par un système d'exploitation lors de l'exécution. L'une des fonctionnalités de l'OS (*Operating System*) consiste à réaliser l'ordonnancement des processus à exécuter.

On qualifie généralement un thread de **processus** léger car il est contenu dans un processus.

Un processus (ou plus simplement *programme en exécution*) peut également être composé de plusieurs threads qui s'exécutent de manière concurrente. Les threads d'un même processus partagent des ressources (comme de la mémoire ou des canaux d'entrée / sortie).

Lorsque l'on programme avec Java, la gestion des threads est simplifiée : un programme, c'est à dire une classe dotée de la méthode `main`, possède un thread d'exécution.

1.1. création d'un thread

Pour créer d'autres threads au sein du programme principal, on définit des classes qui vont :

- soit hériter de la classe `java.lang.Thread`
- soit implémenter l'interface `java.lang.Runnable`

Dans les deux cas, il faudra définir la méthode `run` qui contient les instructions exécutées par le thread.

```
class MyRunnable extends MyClass implements Runnable {
 public void run() {
  // code to execute
 }
}

class MyThread extends Thread {
 public void run() {
  // code to execute
 }
}
```

Un thread peut être doté d'un nom (constructeur ou `setName()`) et/ou rattaché à une groupe de threads (`ThreadGroup` - cf plus loin dans le cours).

```
class MyThread extends Thread {

 public MyThread(String name) {
  super(name);
 }

 public void run() {
  while (true) {
   System.err.println("hello world !");
  }
 }
}
```

1.2. lancement de l'exécution d'un thread

On lance l'exécution du thread par appel de ma méthode `start`

```
class MyProgram {
```

```
 public static void main(String args[]) {
  MyThread thread = new MyThread("a thread");
  thread.start();
 }

}
```

ou encore :

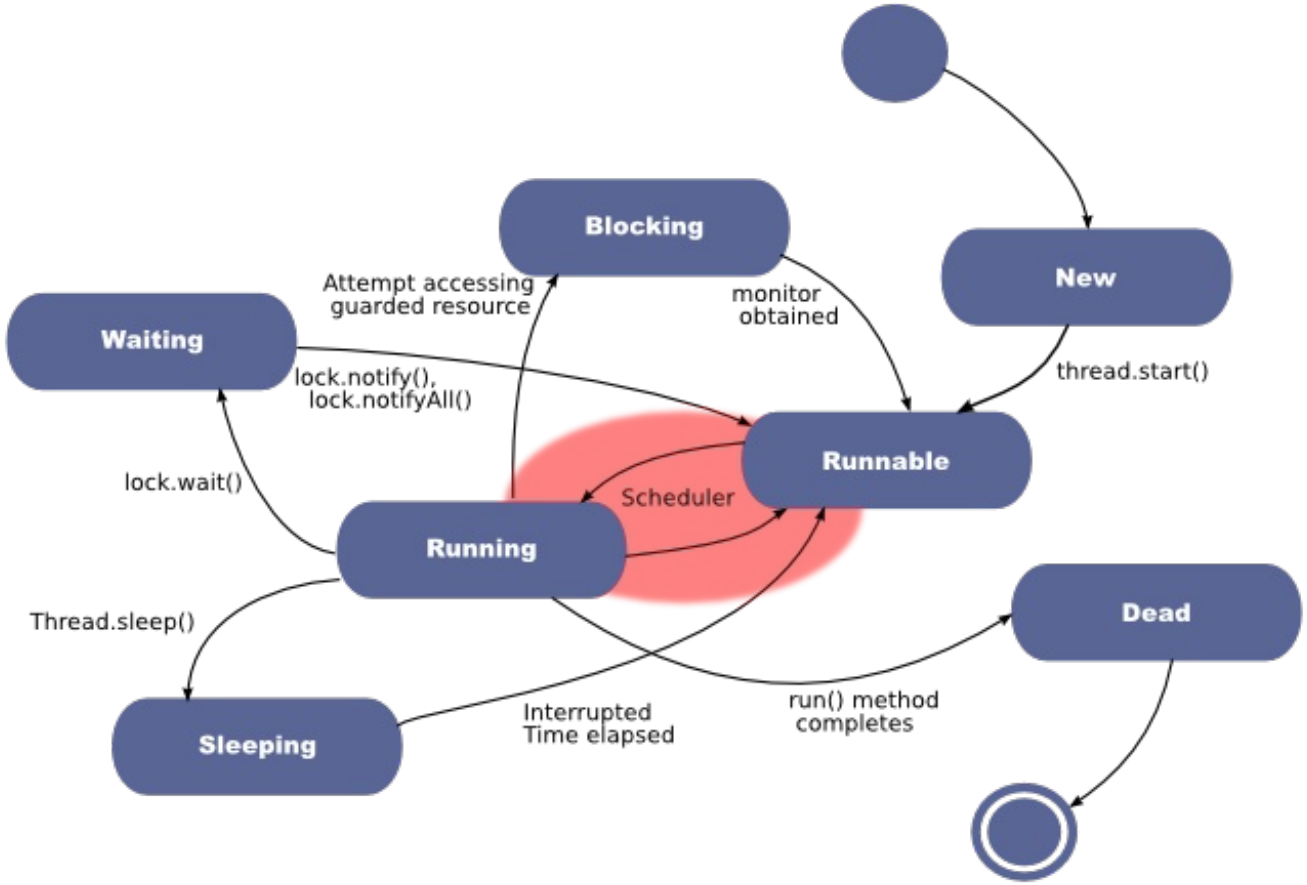
```
class MyProgram {

 public static void main(String args[]) {
  new Thread(new MyRunnable()).start();
 }

}
```

Le programme ne terminera que lorsque le thread aura terminé son exécution.

1.3. états d'un thread



- après création le thread se trouve dans l'état **new**
- puis passe dans l'état **runnable** après appel à la méthode start
- l'ordonnanceur (scheduler) choisit le prochain thread à exécuter et le passe dans l'état **running**, puis le repassera dans l'état **runnable**
- le thread peut appeler la méthode `yield()` pour signigier à l'ordonnanceur qu'il désire revenir à l'état **runnable** pour laisser les autres threads s'exécuter

On peut connaître l'état d'un thread en appelant la méthode `Thread.State getState()`.

1.4. suspendre et reprendre l'exécution d'un thread

Il existe plusieurs moyens pour suspendre temporairement l'exécution d'un thread.

- la méthode `static void sleep(long millis)` force le thread à suspendre son exécution pendant un temps exprimé en

milli secondes.

- la méthode `void wait()` associée à la classe `Object` force le thread à suspendre son exécution jusqu'à ce qu'un autre thread appelle la méthode `void notify()` ou `notifyAll()`
- la méthode `void wait(long timeout)` possède le même comportement que la méthode précédente mais permet de reprendre l'exécution si la notification n'a pas été déclenchée après un certain laps de temps.

1.5. synchronisation des threads

Il existe plusieurs niveaux de synchronisation pour les threads.

- on peut utiliser le mot clé `synchronized` devant un nom de méthode, ce qui permet de poser un verrou sur la méthode, elle ne sera exécutée que par un seul thread à la fois.

```
1.  public class BankAccount {
2.    private float total;
3.
4.    public BankAccount() {
5.      total = 0.0f;
6.    }
7.
8.    public synchronized void deposit(float f) {
9.      total += f;
10.   }
11.
12.   public synchronized float withdraw(float f) {
13.     total -= f;
14.     return total;
15.   }
16. }
17.
```

- on peut utiliser le mot clé `synchronized` à l'intérieur d'une méthode pour établir une exclusion mutuelle sur des données à modifier :

```
1.  public class BankAccount2 {
2.    private float total;
3.
4.    public BankAccount2() {
5.      total = 0.0f;
6.    }
7.
8.    public void deposit(float f) {
9.      synchronized(this) {
10.       total += f;
11.     }
12.   }
13.
14.   public float withdraw(float f) {
15.     synchronized(this) {
16.       total -= f;
17.     }
18.     return total;
19.   }
20. }
21.
```

- on peut utiliser la méthode `void join()` qui permet d'attendre la fin de l'exécution d'autres threads avant de poursuivre un traitement

```
1.  import java.util.*;
2.
3.  class JoinThread extends Thread {
4.    String name;
5.    Random random;
6.
7.    public JoinThread(String name) {
8.      this.name = name;
9.      random = new Random();
10.   }
11.
12.   public void run() {
13.     try {
14.       for (int i=1; i< 5; ++i) {
15.         System.err.println(name+":"+i);
16.         sleep(random.nextInt(2000));
17.       }
18.     }catch(Exception exc) {}
19.   }
20. }
21.
22. public class JoinExample {
23.   public static void main(String args[]) {
24.     try {
25.       JoinThread thread[] = new JoinThread[3];
26.       for (int i=0; i<3; ++i) {
27.         thread[i] = new JoinThread("thread"+i);
28.         thread[i].start();
29.       }
30.
```

```
31.      for (int i=0; i<3; ++i) {
32.        thread[i].join();
33.       }
34.       System.err.println("JOIN HERE");
35.
36.     } catch(Exception exc) {
37.       System.out.println(exc);
38.     }
39.   }
40. }
41.
42.
```

Le résultat de l'exécution est par exemple :

```
thread0:1
thread1:1
thread2:1
thread1:2
thread0:2
thread2:2
thread2:3
thread1:3
thread2:4
thread0:3
thread1:4
thread0:4
JOIN HERE
```

1.6. priorités d'un thread

Chaque thread possède une priorité d'exécution, héritée par défaut de son thread parent. On peut cependant modifier la priorité d'un thread grâce à `void setPriority(int newPriority)`. Celle-ci varie entre :

- `Thread.MIN_PRIORITY = 1`
- `Thread.NORM_PRIORITY = 5`

- `Thread.MAX_PRIORITY = 10`

On obtient la priorité d'un thread grâce à `int getPriority()`.

1.7. groupes de threads

Il est parfois intéressant de regrouper les threads par groupes. Sur l'exemple suivant on crée deux groupes de threads, le groupe g1 continuera à s'exécuter alors que g2 sera interrompu par le thread t3. Une exception `InterruptedException` est levée car les threads de g2 sont interrompus alors qu'ils sont dans l'état **waiting** engendré par l'instruction `sleep`

```
1.  import java.util.*;
2.
3.  class T1 extends Thread {
4.    String name;
5.    ThreadGroup group;
6.    public T1(ThreadGroup g, String name) {
7.      super(g, name);
8.      group=g;
9.      this.name = name;
10.   }
11.   public void run() {
12.     try {
13.       while (true) {
14.         sleep(1000);
15.         yield();
16.         System.err.println(name+" running");
17.       }
18.     } catch(Exception exc) {
19.       System.err.println(name+":"+exc);
20.     }
21.   }
22. }
23.
```

```
24.
25. class T2 extends Thread {
26.   String name;
27.   ThreadGroup group;
28.   public T2(ThreadGroup g, String name) {
29.     super(g, name);
30.     group=g;
31.     this.name = name;
32.   }
33.   public void run() {
34.     try {
35.       while (true) {
36.         sleep(1000);
37.         yield();
38.         System.err.println(name+" running");
39.       }
40.     } catch(Exception exc) {
41.       System.err.println(name+":"+exc);
42.     }
43.   }
44. }
45.
46. class T3 extends Thread {
47.   String name;
48.   ThreadGroup groupToInterrupt;
49.   public T3(ThreadGroup g, String name) {
50.     super(name);
51.     groupToInterrupt=g;
52.     this.name = name;
53.   }
54.   public void run() {
```

```
55.    try {
56.     sleep(2000);
57.     System.err.println(name+":active =
    "+groupToInterrupt.activeGroupCount() );
58.     synchronized(groupToInterrupt) {
59.      groupToInterrupt.interrupt();
60.     }
61.    } catch(Exception exc) {
62.     System.err.println(name+":"+exc);
63.    }
64.   }
65. }
66.
67. public class ThreadGroups {
68.   public static void main(String args[]) {
69.    try {
70.     ThreadGroup g1 = new ThreadGroup("g1");
71.     ThreadGroup g2 = new ThreadGroup("g2");
72.
73.     for (int i=1; i<5; ++i) {
74.      T1 t = new T1(g1, "g1.t1-"+i);
75.      t.start();
76.     }
77.     for (int i=1; i<5; ++i) {
78.      T2 t = new T2(g2, "g2.t2-"+i);
79.      t.start();
80.     }
81.     T3 t = new T3(g2, "g2.t3");
82.     t.start();
83.
84.
```

```
85.     } catch(Exception exc) {
86.       System.err.println(exc);
87.     }
88.   }
89. }
90.
91.
```

Le résultat de l'exécution est par exemple :

```
g1.t1-1 running
g1.t1-4 running
g2.t2-1 running
g1.t1-3 running
g2.t2-2 running
g2.t2-4 running
g1.t1-2 running
g2.t2-3 running
g1.t1-1 running
g1.t1-4 running
g1.t1-3 running
g2.t2-1 running
g2.t2-4 running
g1.t1-2 running
g2.t2-2 running
g2.t3:active = 0
g2.t2-3 running
g2.t2-1:java.lang.InterruptedException: sleep interrupted
g2.t2-4:java.lang.InterruptedException: sleep interrupted
g2.t2-3:java.lang.InterruptedException: sleep interrupted
g2.t2-2:java.lang.InterruptedException: sleep interrupted
g1.t1-1 running
g1.t1-4 running
g1.t1-3 running
g1.t1-2 running
```

2. Programmation concurrente

Le principe de la programmation concurrente consiste à faire interagir plusieurs threads pour accomplir une tâche.

On peut distinguer deux approches :

- calculs indépendants : on répartit la charge de travail sur plusieurs threads (simple)
- calculs coopératifs : les threads coopèrent pour accomplir la tâche (complexe)

Exemple 1 : on désire réaliser le calcul de la somme des évaluations d'une fonction **f(x)** , pour différentes valeurs de **x** , sur l'intervalle **$[x_1, x_2]$** mais le calcul de **f(x)** prend du temps. Les calculs étant indépendants on décide de multithreader l'application. On utilisera par exemple **k** threads qui se répartiront la tâche : feront les calculs en parallèle et reporteront leurs résultats à l'application principale.

Exemple 2 : on souhaite résoudre un problème d'optimisation combinatoire et on crée plusieurs threads qui recherchent la meilleure solution possible et échangent de l'information : exploration, intensification.