CPO-Thread et Serveur

La programmation concurrente permet de paralléliser des taches, des traitements, des calculs etc…

Pour Faire de la programmation concurrente on peut utiliser deux méthodes : les processus ou les threads. L'utilisations des processus est lourde à mettre en place car chaque processus a sa mémoire. L'utilisations de threads est donc plus simple car les threads ont une mémoire partagée.

Les threads

Pour qu'une classe JAVA puisse utiliser les threads il faut quelle implémente l'interface Runnable. Lorsqu'une classe utilise cette interface elle doit redéfinir la méthode run. C'est les instructions contenues dans ce run qui vont être exécuté par le thread.

```
public class IncDeux implements Runnable {
        private static int x = 0;
        private static int y = 0;

        @Override
        public void run() {
                while(true) {
                        x++;
                        y++;

                        if(x != y)
                                System.out.println("x = " + x + ", y = " + y);
                }
        }

        public static void main(String[] args) {
                IncDeux inc1 = new IncDeux();
                Thread t1 = new Thread(inc1);
                t1.start();

                IncDeux inc2 = new IncDeux();
                Thread t2 = new Thread(inc2);
                t2.start();

                while(true) {
                }
        }
}
```

Dans cette exemple la classe « IncDeux » implémente l'interface « Runnable » et redéfini la méthode « run ». Une instance de cette classe peut donc être donner en paramètre à un thread pour qu'il exécute les instructions de la méthode « run ».

Dans l'exemple on créer deux instances de la classe « IncDeux ». Les variables x et y de ces deux classes sont partagé. Lorsque les deux thread qui on reçut en paramètre les deux instances de classe sont démarré par « .start() ; », les deux

threads exécute les instructions du « run » en même temps. L'incrémentation de x et y se fait donc en même temps dans les deux threads.

On voit donc par cette cet exemple que lorsque on utilise les threads il faut faire attentions aux ressources partagées. En effet par exemple, l'opération d'incrémentation d'une variable n'est pas atomique. C'est-à-dire qu'entre le moment ou un thread commence l'incrémentation et le moment où il la fini, un autre thread peut lire / modifier la même variable.

Pour résoudre se problème il y a plusieurs méthodes : les verrous ou rendre certaines variables atomiques.

Variables atomiques

On peut rendre des variables atomiques comment dans l'exemple si dessous, x et y sont des « AtomicInteger ». L'utilisations de threads de pose donc plus de problème lors de l'incrémentation de x et de y. Cependant le reste du traitement n'est pas protégé, le reste des instructions peuvent se passer dans n'importe quel ordre entre les threads.

```
public class IncDeuxSync implements Runnable {
        private static AtomicInteger x = new AtomicInteger(0);
        private static AtomicInteger y = new AtomicInteger(0);

        @Override
        public void run() {
                while(true) {
                        x.getAndIncrement();
                        y.getAndIncrement();

                        if(x.get() != y.get())
                                System.out.println("x = " + x.get() + ", y = " + y.get());
                }
        }
```

Les verrous

En Java, tout objet peut être utilisé comme un verrou

A chaque instant, au plus une section de code d'un même objet (verrou) peut être exécutée

S'il y a un autre thread qui possède le verrou, on doit attendre qu'il ne quitte la section « synchronized »

```
public class IncDeuxLock implements Runnable {
        private static int x = 0;
        private static int y = 0;
        private static Object Lock = new Object();

        @Override
        public void run() {
                while(true) {
                        synchronized(lock) {
                                x++;
                                y++;

                                if(x != y)
                                        System.out.println("x = " + x + ", y = " + y);
                        }
                }
        }
```

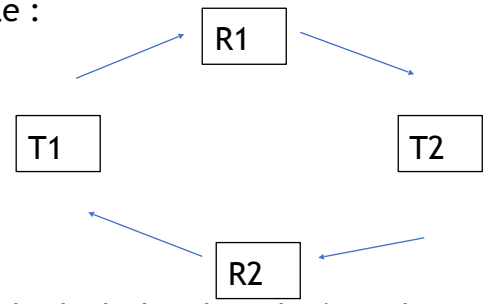
Dans cette exemple tout le traitement du run qui utilise x et y est mis dans un verrou. De cette manière on à plus de problème d'accès simultané aux variables x et y cependant notre code n'est plus parallélisé puisque chaque thread à besoin du même verrou et doivent donc s'attendre.

Il faut donc utiliser les verrous au minimum et sur une portion minimale du code. Pour cela il faut donc détecter les sections critiques du code se qui rend la programmation plus complexe. Dans notre exemple l'ensemble du run est critique ce n'est donc pas un traitement favorable à l'utilisation de threads.

Les verrous : Deadlock

L'utilisation de plusieurs verrous peut entrainer un Deadlock, c'est-à-dire que les threads peuvent se bloquer l'un l'autres. Exemple :

- Le thread T1 utilise la ressource R2
- Le thread T1 attend la ressource R1
- Le thread T2 utilise la ressource R1
- Le thread T2 attend la ressource R2



Dans ce cas, si les ressources sont protégées par des locks les threads s'interloque.

Tester des programmes concurrents

Problèmes de reproductibilité :

L'exécution des threads est aléatoire, on ne peut pas prédire l'ordre d'exécution. Il est donc difficile de trouver la source d'un problème quand chaque exécution n'a pas exactement le même comportement.

Heisenbugs

Le mot Heisenbugs vient du Principe d'Incertitude d'Heisenberg. Un concept de physique quantique qui dit que les observateurs affectent ce qu'ils observent par le simple acte d'observation.

L'Heisenbugs est donc un bug très difficile à détecter car l'lorsqu'on le cherche on modifie le comportement du programme (par des affichages ou des pauses par exemple). L'Heisenbugs est en partie due à la non reproductibilité.

Objet thread-Safe :

- Pas de problèmes de thread (Section à risque dans des « syncronized »).
- C'est un Objet qui satisfait la spécification même si plusieurs threads l'appellent en même temps (immuabilité peut aider à satisfaire la spécification).

LES SERVEURS

Client TCP en Java

• créer une instance de la classe Socket en indiquant l'adresse et le port du server

```
Socket socket = new Socket(address, port);
```

• utiliser les deux streams socket.getOutputStream() et socket.getInputStream() pour envoyer/recevoir des messages

```
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
```

• terminer la connexion avec socket.close()

```
socket.close();
```

Serveur TCP en Java

• créer une instance de la classe ServerSocket en indiquant le port

```
ServerSocket listener = new ServerSocket(DEFAULT_PORT);
```

• appeler ssocket.accept() pour attendre une connexion client dans une boucle infinie

```
while (true) {
        new Thread(new Responder(listener.accept())).start();
}
```

• interagir avec le nouveau client dans un nouveau thread

• faire attention aux exceptions