UFR SSI
M1 INFO - INF 2160u CONCURRENCE
Session II - 2021-2022

Luc Courtrai

date : 15 juin 2022 – durée : 1h30

barème à titre indicatif

1 Question de cours (6 pts)

-1- Une variable globale (en dehors d'une fonction) d'un programme C est-elle partagée par 2 threads POSIX. Pourquoi ?

-2- Comment partager un tableau d'entiers entre deux processus lourds en C sous Linux sans passer par le système de fichier ?

-3- Dans le système de processus sous Linux à quoi sert l'attribut nice et comment peut-on l'affecter ?

-4- En Java, les sémaphores java.util.concurrent.Semaphore respectent-ils l'ordre FIFO de sortie d'un appel à acquire ? Si oui comment le spécifier ?

-5- En Java, peut-on interrompre un thread en attente dans la méthode `wait()` sans utiliser un notify ou notifyAll ? Si oui comment ?

-6- En Java, une méthode d'instance avec la clause synchronized peut-elle appeler une autre méthode, elle même synchronized, sur le même objet ? Pourquoi ?

-7- C'est quoi l'état Zombi d'un processus sous Linux ?

-8- Sous Linux, une variable placée dans le segment de Data (issue d'un malloc) est-elle partagée par un processus et ses processus fils (fork). Pourquoi ?

-9- Pourquoi l'appel aux méthodes Wait et Notify de Java doivent-ils être encadrés dans un bloc synchronized ?

-10- C'est quoi un point d'annulation dans la programmation avec des threads ? Un mécanisme similaire existe t-il dans les threads en Java. Si oui lequel ?

2 Println (4 pts)

```
class Th extends Thread {
  static int sval=0;
  int val; int result=1;
  public Th(int v) {val =v;};
  public void run() {
    sval++;
    System.out.println("sval:" + sval+" val:" + val +" result:" +result);
    while(val >0 ) {
      val --;
      Th unT = new Th (val); unT.start();
      try {unT.join();} catch (InterruptedException e){};
      result +=unT.result;
      System.out.println("sval:" + sval+" val:" + val +" result:" +result);
    }
  }
  public static void main(String args[]){
        Th th=new Th(2); th.start();
        try {th.join();} catch (InterruptedException e){};
        System.out.println("fin result : " + th.result);
  }
```

Quel est le résultat de l'exécution de ce programme JAVA sur la sortie standard ?

3 ReentrantLock (10 pts)

Reécrire la classe java.concurrent.locks.ReentrantLock pour des applications Java multi-theadées en utilisant les méthodes wait et notify. (voir la javadoc page suivante)

3.1 Version non équitable

Ecrire l'implantation du ReentrantLock :

```
ReentrantLock()
Crée une instance de ReentrantLoc
```

donc aucun ordre n'est garanti pour les threads en attente du verrou.

3.2 Version équitable FIFO

Ecrire maintenant la version équitable (séparément de la première version)

```
ReentrantLock(boolean fifo)
Crée une instance de ReentrantLock avec un comportement  donné
  fifo :  true  ordre FIFO
```

la javadoc de la classe ReentrantLock

```
java.util.concurrent.locks
Class ReentrantLock

Un verrou réentrant a le même comportement et la  même sémantique que le bloc
ou la méthode  synchronized.

Le verrou appartient au dernier thread qui l'a verrouillé,  mais qui ne l'a
pas encore déverrouillé. Le retour de l'appel à la méthode lock s'effectue
quand le verrou n'appartient plus à un autre thread.
Le retour est immédiat si le thread possède déja le verrou.

Le conctucteur accepte un paramètre booléen d'équité. Quand le paramètre est
vrai, lors de l'attente d'obtention, le verrou garantit l'accès au plus vieux
thread en attend. Dans l'autre cas, aucun ordre n'est garanti.

class X {
private final ReentrantLock lock = new ReentrantLock();
// ...

  public void m() {
    lock.lock();  // Bloquant tant que le verrou est détenu par un autre
    // Si le thread courant possède déjà le verrou il n'est
      // pas bloqué (REENTRANT)
    ..
    lock.unlock();
  }
}

CONSTRUCTOR SUMMARY

ReentrantLock()
  Crée une instance de ReentrantLock
ReentrantLock(boolean fair)
  Crée une instance de ReentrantLock avec un comportement  donné
  fifo :  true  ordre FIFO
METHOD SUMMARY
void  lock()
   demande le verrou
void  unlock()
  libère le verrou.
boolean  tryLock()
  prend le verrou que si aucun autre thread ne le possède
```

```
class Th extends Thread {
    static int sval=0;
    int val;
    int result=1;
    public Th(int v) {val =v;};
    public void run() {
        sval++;
        System.out.println("sval:" + sval+" val:" + val +" result:" +result);
        while(val >0 ) {
              val --;
              Th unT = new Th (val);
              unT.start();
              try {unT.join();} catch (InterruptedException e){};
              result +=unT.result;
              System.out.println("sval:" + sval+" val:" + val +" result:" +result);
        }
    }
    public static void main(String args[]){
        Th th=new Th(2);
        th.start();
        try {th.join();} catch (InterruptedException e){};
        System.out.println("fin result:" + th.result);
    }
}

/*
sval:1 val:2 result:1
sval:2 val:1 result:1
sval:3 val:0 result:1
sval:3 val:0 result:2
sval:3 val:1 result:3
sval:4 val:0 result:1
sval:4 val:0 result:4
result : 4


*/
```

```
public class LockReentrant {



    boolean isLock;
    Thread owner;
    public LockReentrant() {
        isLock = false;
        owner=null;
    }
    public  synchronized void lock(){

        while (isLock && ! Thread.currentThread().equals(owner))  {
            try {
                wait();
            }  catch (InterruptedException e) {};
        }

        owner=Thread.currentThread();
        isLock=true;
    }

    public synchronized  void unlock(){
        if (Thread.currentThread().equals(owner)) {
            notify();  // reveille  le thread bloque
            isLock = false; // suivthread ne doit pas etre bloque
            owner=null;
        }
    }

     public  synchronized  boolean tryLock(){
         if (isLock && ! Thread.currentThread().equals(owner))
             return false;
         else {
             owner=Thread.currentThread();
             isLock=true;
             return true;
         }

     }


}
```

```
import java.util.LinkedList;


public class FIFOLockReentrant {


        boolean isLock; // Le premier thread doit etre bloque

        private LinkedList<Thread> lockThreads;
         Thread owner;

        public FIFOLockReentrant() {
           isLock = false;
           lockThreads = new LinkedList<Thread>();
           owner=null;
        }

        public   void lock(){

                 Thread self = Thread.currentThread();
                 boolean stop =false;
                 synchronized(self) {
                     synchronized(this){
                         if (isLock && ! self.equals(owner)) {
                             lockThreads.add(self);
                             stop=true;
                         } else {
                             isLock=true;
                         }
                     }
                     if ( stop){
                             try {
                                     self.wait();

                             } catch (InterruptedException e){};
                     }
                     owner=self;
                 }
                 //   OU   pour eviter stop
                 /*
                 synchronized(self) {
                     synchronized(this){
                             if (! isLock) {
                                     isLock=true;
                                     return;
                             }
                             lockThreads.add(self);
                     }

                     try {
                             self.wait();

                     } catch (InterruptedException e){};
                     owner=self;

                 }
                 */

        }

         public synchronized  void unlock(){
             if (Thread.currentThread().equals(owner)) {
                     if (! lockThreads.isEmpty()) {
                         Thread first = lockThreads.removeFirst();
                         synchronized (first){
                                 first.notify();
                         }
                     } else {
                         isLock = false;
                         owner=null;
                     }
                 }
             }

             public  synchronized  boolean tryLock(){
                  Thread self = Thread.currentThread();
                  if (isLock && ! self.equals(owner))  return false;
                  isLock=true;
                  owner=self;
             }

            /**
             * @param args
             */
            public static void main(String[] args) {
                    // TODO Auto-generated method stub

            }
}
```