Compte Rendu TP5

BOURENNANE Amine

Exercice 1

1. Le programme incrémente un compteur pendant 100 millisecondes théoriquement mais en réalité il ne s'arrête sûrement jamais car la condition dans le if ne sera jamais lu en mémoire.
2. Effectivement le code ne s'arrête jamais car la thread qui fait le décompte ne voit jamais la valeur de stop dans la mémoire.
3. On peut faire un bloc synchronized qui force la lecture et l'écriture des variables en RAM et non dans le cache.

```
public class Bogus {
  private boolean stop;
  private final Object lock = new Object();

  public void runCounter() {
    var localCounter = 0;
    for(;;) {
      synchronized (lock) {
        if (stop) {
          break;
        }
      }
      localCounter++;
    }
    System.out.println(localCounter);
  }

  public void stop() {
    synchronized (lock) {
      stop = true;
    }
  }
}
```

4. Les implémantations qui n'utilise ni les blocks synchronized, ni les locks sont appelées *lock-free*

```
public class BogusVolatile {
    private volatile boolean stop;

    public void runCounter() {
        var localCounter = 0;
        for(;;) {
            if (stop) {
                break;
```

```
            }
            localCounter++;
        }
        System.out.println(localCounter);
    }

    public void stop() {
        stop = true;
    }
}
```

Exercice 2

1. Réentrant signifie que l'on peut réutiliser la partie du code qui a une "barrière" avec un jeton d'attente.
2. Si le programme est thread-safe on s'attendrai que les deux thread compte ensemble jusqu'à `1 000 000` et que le counter de fin soit à `2 000 000`. Même si on déclare *counter* en volatile cela ne fonctionne pas car je pense que le fait que ça soit dans une boucle la lecture n'est pas ne RAM.
3. Si l'on arrive pas à avoir le lock il faudrait faire attendre la thread, mais on ne sait quand on peut réveiller la thread. Car celui permet de laisser la main aux autres threads quand on le désire. Cela permet d'attendre d'avoir la bonne instance de la thread avant de faire les calculs, malgré les pipeline cela permet d'avoir un certain contrôle sur celle-ci
4.

```
public class SpinLock {
  private volatile boolean lock = false;
  private final static VarHandle LOCK_REF;
  static {
    Lookup lookup = MethodHandles.lookup();
    try {
      LOCK_REF = lookup.findVarHandle(SpinLock.class,
              "lock", boolean.class);
    } catch (NoSuchFieldException | IllegalAccessException e) {
      throw new AssertionError(e);
    }
  }

  public void lock() {
    for (;;) {
      if (LOCK_REF.compareAndSet(this, false, true))
        return;
      Thread.onSpinWait();
    }
  }

  public void unlock() {
    lock = false;
  }
}
```

Exercice 3

1. Un générateur pseudo-aléatoire est une suite de valeur qui change de valeur si l'on lui donne une seed différente. L'API n'est pas thread-safe car elle fait des opérations non atomique sur la variable x, elle peut donc ne pas avoir la même valeur en RAM pendant l'écriture que la lecture.
2.

```
public class RandomNumberGeneratorAtomicLong {
  private AtomicLong x = new AtomicLong();

  public RandomNumberGeneratorAtomicLong(long seed) {
    if (seed == 0) {
      throw new IllegalArgumentException("seed == 0");
    }
    x.set(seed);
  }

  public long next() {  // Marsaglia's XorShift
    for (;;) {
        var current = x.get();
        var newValue = current;
        newValue ^= newValue >>> 12;
        newValue ^= newValue << 25;
        newValue ^= newValue >>> 27;
        if (x.compareAndSet(current, newValue))
            return x.get()
    }
  }
}
```

3. On peut l'utiliser dans la méthode `next()` à la place d'un `compareAndSet()`.

```
public class RandomNumberGeneratorAtomicLong {
  private AtomicLong x = new AtomicLong();

  public RandomNumberGeneratorAtomicLong(long seed) {
    if (seed == 0) {
      throw new IllegalArgumentException("seed == 0");
    }
    x.set(seed);
  }

  public long next() {  // Marsaglia's XorShift
    return x.getAndUpdate(x -> {
      x ^= x >>> 12;
      x ^= x << 25;
      x ^= x >>> 27;
      return x * 2685821657736338717L;
    });
```

```
    }
}
```

4.

```
public class RandomNumberGeneratorVarHandler {
    private volatile long x;
    private final static VarHandle X_REF;
    static {
        Lookup lookup = MethodHandles.lookup();
        try {
            X_REF =
lookup.findVarHandle(RandomNumberGeneratorVarHandler.class,
                    "x", long.class);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            throw new AssertionError(e);
        }
    }

    public RandomNumberGeneratorVarHandler(long seed) {
        if (seed == 0) {
            throw new IllegalArgumentException("seed == 0");
        }
        X_REF.set(this, seed);
    }

    public long next() {  // Marsaglia's XorShift
        for (;;) {
            var current = this.x;
            var newValue = current;
            newValue ^= newValue >>> 12;
            newValue ^= newValue << 25;
            newValue ^= newValue >>> 27;
            if (X_REF.compareAndSet(this, current, newValue))
                return newValue;
        }
    }
}
```

Exercice 4

1.

```
public class ReentrantSpinLock {
    private volatile int lock;
    private volatile Thread ownerThread;
    private final static VarHandle LOCK_REF;
    static {
        MethodHandles.Lookup lookup = MethodHandles.lookup();
```

```
        try {
            LOCK_REF = lookup.findVarHandle(ReentrantSpinLock.class,
                    "lock", int.class);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            throw new AssertionError(e);
        }
    }

    public void lock() {
        for (;;) {
            var thread = Thread.currentThread();
            if (LOCK_REF.compareAndSet(this, 0, 1)) {
                ownerThread = thread;
                return;
            }
            if (thread == ownerThread) {
                lock++;
                return;
            }
            Thread.onSpinWait();
        }
    }

    public void unlock() {
        if (Thread.currentThread() != ownerThread) {
            throw new AssertionError("Bad Thread");
        }
        if (lock == 1) {
            ownerThread = null;
        }
        lock--;
    }
}
```

2.

```
public class ReentrantSpinLock {
    private volatile int lock;
    private Thread ownerThread;
    private final static VarHandle LOCK_REF;
    static {
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        try {
            LOCK_REF = lookup.findVarHandle(ReentrantSpinLock.class,
                    "lock", int.class);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            throw new AssertionError(e);
        }
    }

    public void lock() {
        for (;;) {
```

```
            var thread = Thread.currentThread();
            if (LOCK_REF.compareAndSet(this, 0, 1)) {
                ownerThread = thread;
                return;
            }
            if (thread == ownerThread) {
                lock++;
                return;
            }
            Thread.onSpinWait();
        }
    }

    public void unlock() {
        if (Thread.currentThread() != ownerThread) {
            throw new AssertionError("Bad Thread");
        }
        if (lock == 1) {
            ownerThread = null;
        }
        lock--;
    }
}
```