ESIR SPP – TD1

Exercise 1:

1) Consider the following multi-threaded program, where account is a shared data structure that is thread-safe (i.e. read(..) and write(..) are atomic).

```
shared int account = 1
```

```
thread t1 is
var a int ;
begin
    a := read(account);
    a := a+10         ;
    write(account,a)  ;
end
```

```
thread t2 is
var b int ;
begin
    b := read(account);
    b := b*2          ;
    write(account,b)  ;
end
```

What are the possible values of account after both t1 and t2 have executed?

2) Consider this new version, where account is a shared variable.

```
thread t1 is
var a int ;
begin
    account+=10 ;
end
```

```
thread t2 is
var b int ;
begin
    account*=2 ;
end
```

2.1 – What are the possible values of account after both t1 and t2 have executed?

2.2 – Is this program safe? Why?

2.3 – How would you correct this program?

Exercise 2:

Consider the following Java program.

1 – Look at the program and try to understand the different possible executions it can take.

2 – Is it safe? Is it lively? Find an example of a problematic execution. What will the user see at the end of the execution? How much CPU will the program use at the end of this execution?

3 – Will the above problem occur in every execution? Why?

4 – Provide a modified version that corrects the problem. (Just explain which lines you would change, how you would change them, and why this corrects the problem.)

```java
import java.util.concurrent.locks.ReentrantLock;

class Thread1 extends Thread {
  public void run() {
      TD1.printer.lock();
      TD1.network.lock();
      // work
      TD1.network.unlock();
      TD1.printer.unlock();
  }
}
class Thread2 extends Thread {
  public void run() {
      TD1.network.lock();
      TD1.hdrive.lock();
      // work
      TD1.hdrive.unlock();
      TD1.network.unlock();
  }
}
class Thread3 extends Thread {
  public void run() {
      TD1.hdrive.lock();
      TD1.printer.lock();
      // work
      TD1.printer.unlock();
      TD1.hdrive.unlock();
  }
}
public class TD1 {
  public static ReentrantLock printer = new ReentrantLock();
  public static ReentrantLock network = new ReentrantLock();
  public static ReentrantLock hdrive  = new ReentrantLock();

  public static void main(String[] args) {
    Thread t1 = new Thread1();
    Thread t2 = new Thread2();
    Thread t3 = new Thread3();
    t1.start();
    t2.start();
    t3.start();
  }
} // EndClass TD1
```

Exercise 3:

We want to write a parallel program with two threads, so that both threads alternatively write "ping" (for the first thread) and "pong" (for the second) on the console. A first unsynchronised sketch of the program looks like this:

```
thread t1 is
begin
    while(true) { println("ping") }
end
```

```
thread t2 is
begin
    while(true) { println("pong") }
end
```

1 – Assuming both threads execute at exactly the same speed, and you have access to one (and only one) fair lock that grants access to the critical section in the order in which it is reserved, write a synchronised version of this program that fulfils the above requirements. (Your program should only use **one** lock, no more.)

2 – Does your solution still works if:

i) Locks still grant access to the critical section in the order in which they are reserved, but the 2 threads work at different speed. Explain why.

ii) The 2 threads still work at the same speed, but the locks are not fair. Explain why.

3 – Find a general solution to the above problems that neither assumes threads work at the same speed or that locks are fair and relies on semaphores. Your solution should use two binary semaphores (mutex semaphores), but no shared variable. Could you use locks instead? Could these locks be re-entrant (aka recursive)? Why?

4 – Using locks instead of semaphores in the above solution requires that one thread can unlock the lock held by another thread. Some platforms prevent this. Other platforms (Java for instance) only have re-entrant locks (*). Using a shared variable now, how would you solve the above problem if your locks are re-entrant? Or if they are not, but one thread cannot unlock the lock held by another thread?

(*) But Java has semaphores that can be use as non-reentrant locks.

(Optional) Exercise 4:

How would you prove (i.e. give a formal reasoning) that your solution to exercise 3.3 cannot deadlock? (Hint: You might want to focus on the order in which threads access locks, and use a proof by contradiction by counting how many times the lock and unlock operations have been invoked on the two locks.)