

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.)

```

import java.util.concurrent.locks.ReentrantLock;

class Thread1 extends Thread {
    public void run() {
        TP1.printer.lock();
        TP1.network.lock();
        // work
        TP1.network.unlock();
        TP1.printer.unlock();
    }
}

class Thread2 extends Thread {
    public void run() {
        TP1.network.lock();
        TP1.hdrive.lock();
        // work
        TP1.hdrive.unlock();
        TP1.network.unlock();
    }
}

class Thread3 extends Thread {
    public void run() {
        TP1.hdrive.lock();
        TP1.printer.lock();
        // work
        TP1.printer.unlock();
        TP1.hdrive.unlock();
    }
}

public class TP1 {
    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 TP1

```

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 as exactly the same speed, and you have access to one (and only one) fair lock, 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 are still fair, 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. Your solution should use two locks, but no shared variable. What type of locks do you use? Are they re-entrant (aka recursive)? Why?

4 – The above solution assumes 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 solve the above problems if your locks are re-entrant? Or if they are not, but one thread can not 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 cannot deadlock? (Hint: You might want to focus on the order in which threads take resources, and use a proof by contradiction.)