INF214/2021, OBLIGATORY 2

Deadline: 23:59 on Monday, 27th of September, 2021

- There are 4 tasks in this obligatory assignment, some of them with subtasks.
- Maximum number of points for the tasks in this obligatory assignment: 20.
  - Task 1A gives 2 points.
  - Task 1B gives 3 points.
  - Task 2 gives 3 points.
  - Task 3 gives 4 points.
  - Task 4A gives 1 points.
  - Task 4B gives 7 points.
- You need at least 12 points to pass this obligatory assignment.
- **Submission: upload a zipped folder to *MittUiB* → *Assignments* → *Obligatory 2.***

Task 1 (use AWAIT language)

Consider the following routine written in the "AWAIT" language.

```
// this code is written in the AWAIT language

printer() {
   process P1 { write("1"); write("2"); }
   process P2 { write("3"); write("4"); }
   process P3 { write("5"); write("6"); }
}
```

part 1A

How many different outputs could `printer` produce? Explain your reasoning.

part 1B

Add semaphores to the program so that the six lines of output are printed in the order 1, 2, 3, 4, 5, 6. Declare and initialize any semaphores you need and add `P` and `V` operations to the above processes.

Task 2 (use Java)

A custom semaphore controls access to a shared resource by using *permits*. This variable `permits` is a sort of a counter. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Give an implementation of the Java class `Sem` for custom semaphores, with the following specifications.

- It has a variable `permits` in the code that stands for
  - the initial number of permits available, and
  - the number of threads that can access shared resource at a time.
- If `permits` is greater than zero, then the semaphore allows access to shared resources. Otherwise, the semaphore does not allow access to shared resources.

```java
public class Sem {

   private int permits;

   public Sem(int permits) {
      this.permits = permits;
   }

```

```
    public synchronized void acquire() throws InterruptedException {
        if (permits > 0){
            // ---------> TODO: Your solution goes here. <---------
        } else {
            // ---------> TODO: Your solution goes here. <---------
        }
    }

    public synchronized void release() {
        // ---------> TODO: Your solution goes here. <---------
    }
}
```

(this code can be copy-pasted from: `https://replit.com/@mikbar/Oblig2Semaphore#Sem.java`)

Task 3 (use Java)

In task 1B, you have used the AWAIT language. Translate your solution of task 1B from the AWAIT language to Java. Use threads and the class `Sem` that you implemented in task 2.

Task 4 (use Java)

A Toys factory is auditing its doll production in the winter coming. The process to produce one single doll consists of three different sequential stages: part assembly, painting process, and quality control. In the third stage, a machine assigns a quality score for a given doll between zero and ten. To keep the high reputation of its products, the company has decided to only keep toys with a score higher or equal than nine. This quality control is the second last stage. The last step consists of packaging dolls for delivering. We print out details of the toys which pass the process. We ask you to help the company with the logistic by implementing a safe concurrent Java solution. If you find a better approach to any piece of code given below, please comment out your solution.

part 4A

Give a **short** description of the Java concurrent classes used in the following imports.

```
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.BrokenBarrierException;
```

Consider the following Java class for the Doll objects. You can add the corresponding getters and setters that are ignored in the following Java definition.

```
class Doll {
    int id;
    int qualityScoreMachine;
    boolean imperfect, isPainted;
}
```

part 4B

Complete the missing parts in the following excerpts.

- The method `execution` attempts to produce a given number of dolls.
- There are three stages, A, B, C, representing the corresponding stages in the description above. To run a new stage, the previous stage must have already finished. The company considers a stage as "finished" when the complete set of dolls has gone through that stage. Finally, in the last part of the process we have D, the part where the packaging and reporting of good toys takes places.

```java
class DollFactory {
   List<Doll> dolls;
   private CyclicBarrier stageA, stageB, stageC;
   private void execution(int dollsNumber) throws InterruptedException {
      stageA = new CyclicBarrier(dollsNumber);
      stageB = new CyclicBarrier(dollsNumber);

      // ---------> TODO: stageC = new CyclicBarrier(...); <---------

      dolls = new ArrayList<>(dollsNumber);
      for (int i = 0; i < dollsNumber; i++) {
         Process task = new Process(i);

         // ---------> TODO: Your solution goes here <---------

      }
      try {
         stageC.await();
         System.out.println("Packaging process D");

         // ---------> TODO: print results <---------

      }
      catch (BrokenBarrierException e) {
         e.printStackTrace();
      }
   }

   // class Process implements Runnable { ...
}
```

– Complete the missing part in the Doll production task given below.

```java
class Process implements Runnable {
   int id;
   public Process(int id) {
      this.id = id;
   }
   public void run() {

      // ---------> TODO: Your solution goes here <---------

   }
   void painting(Doll d) {
      d.setPainted(true);
   }
   Doll assembly() {
      Random r = new Random();
      return new Doll(id, r.nextInt(4) + 7);
   }
   void qualityControl(Doll d) {
      if (d.getQualityScore() >= 9) {
         d.hasImperfections(false);
         dolls.add(d);
      }
   }
}
```

– Report the output of your production process.

```java
public static void main(String[] args) throws InterruptedException {
   DollFactory dcb = new DollFactory();
   dcb.execution(100); // Let's produce 100 dolls
}
```

(the code can be copy-pasted from: `https://replit.com/@mikbar/Oblig2DollFactory#Main.java`)
