**Obligatory 2 – INF214**

Kasper Melheim

**Task 1)**

**Part 1 A)**

Here we have 3 processes running concurrently.

The way the code is now, it is not certain that you will get the same order of of numbers in the output each time.

Process 3 can write before process 1, process 2 before process 3 etc. Let us assume the 2 write calls in each process will always be called after each other, and no write call from a different process can interweave and come in between.

Using this logic, we know that the number 1 has to come before 2, the number 5 before 6, and so on.

There are 6 possible outputs by using this logic: **123456, 125634, 561234, 563412, 341256, 345612**

If we then assume that a write call from a different process can come in between the two write calls inside a different process, one can end up with many extra combinations.

We need to also consider that a 2 before a 1 for example is illegal, and we can’t calculate possible outcomes just by doing "6!"

By removing illegal outcomes, we end up with **90** different combinations.

**Part 1 B)**

**sem arrive1 = 1;**

**sem arrive2 = 0;**

**sem arrive3 = 0;**

printer() {

process P1 { **P**(**arrive1**); write("1"); write("2"); **V**(**arrive2**); }

process P2 { **P**(**arrive2**); write("3"); write("4");  **V**(**arrive3**}; }

process P3 { **P**(**arrive3**); write("5"); write("6"); **V**(**arrive1**); }

}

Adding semaphores like these will ensure the correct order. This solution will loop aswell.

**Task 2)**

When acquire is called, it will decrease permits by one if permits are greater than zero. If greater is lesser than or equal to zero it will wait till permits have been increased. Release increases permits by one, notifying the semaphore that it is ready to be reused.

public class Sem {  
 private int permits;  
  
 public Sem(int permits) {  
 this.permits = permits;  
 }  
  
 public synchronized void acquire() throws InterruptedException {  
 if (permits <= 0) {  
 while (permits <= 0) {

}  
 }  
 permits--;  
 }  
  
 public synchronized void release() {  
 permits++;  
 }  
}

**Task 3)**

Using the Sem class we implemented above, and converting the code written in await language on part 1B) to Java language, we end up with something like this:

class Printer {  
  
 private static Sem *a1*; private static Sem *a2*; private static Sem *a3*;  
  
 public static void main(String[] args) {  
  
 *a1* = new Sem(1);  
 *a2* = new Sem(0);  
 *a3* = new Sem(0);  
  
 Thread t1 = new Thread(() -> {  
 while (true){  
 try {  
 *a1*.acquire();  
 System.*out*.println("1");  
 System.*out*.println("2");  
 *a2*.release();  
 }  
 catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 Thread t2 = new Thread(() -> {  
 while (true){  
 try {  
 *a2*.acquire();  
 System.*out*.println("3");  
 System.*out*.println("4");  
 *a3*.release();  
 }  
 catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 Thread t3 = new Thread(() -> {  
 while (true){  
 try {  
 *a3*.acquire();  
 System.*out*.println("5");  
 System.*out*.println("6");  
 *a1*.release();  
 }  
 catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 try {  
 t1.start();  
 t2.start();  
 t3.start();  
 t1.join();  
 t2.join();  
 t3.join();  
 }  
 catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

This will infinitely run while printing out the numbers in this order:

1

2

3

4

5

6

**Task 4)**

**Part 4 A)**

CyclicBarrier:

CyclicBarrier is a helpful tool when working with synchronization. It allows a set of threads to all wait for each other to reach a common barrier point. Useful in programs where all threads must finish their tasks until one can proceed.

BrokenBarrierException:

This is an exception that can be thrown when a thread tries to wait upon a barrier in a broken state, or which enters the broken state while the thread is waiting.

**Part 4 B)**

import java.util.ArrayList;  
import java.util.Iterator;  
import java.util.List;  
import java.util.Random;  
import java.util.concurrent.CyclicBarrier;  
import java.util.concurrent.BrokenBarrierException;  
  
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);  
 stageC = new CyclicBarrier(dollsNumber+1);  
  
 dolls = new ArrayList<>(dollsNumber);  
  
 for (int i = 0; i < dollsNumber; i++) {  
 Thread t = new Thread(new Process(i));  
 t.start();  
 }  
  
 try {  
 this.stageC.await();  
 System.*out*.println("Packaging process D");  
 Iterator i = this.dolls.iterator();  
  
 while(i.hasNext()) {  
 Doll doll = (Doll) i.next();  
 System.*out*.println("Doll " + doll.getId() + " is ready for packaging");  
 }  
 System.*out*.println("Dolls produced: " + this.dolls.size());  
 }  
 catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 DollFactory dcb = new DollFactory();  
 dcb.execution(100); // Let's produce 100 dolls  
 }  
  
  
 // nested class Process  
 class Process implements Runnable {  
 int id;  
  
 public Process(int id) {  
 this.id = id;  
 }  
  
 public void run() {  
  
 Doll doll = this.assembly();  
 System.*out*.println("Doll " + doll.getId() + " produced");  
  
 try {  
 stageA.await();  
 }

catch (BrokenBarrierException | InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 this.painting(doll);  
 System.*out*.println("Doll " + doll.getId() + " painted");  
  
 try {  
 stageB.await();  
 }

catch (BrokenBarrierException | InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 this.qualityControl(doll);

if (doll.getImperfections()) {  
 System.*out*.println("Doll " + doll.getId() + " failed inspection");  
 }

else {  
 System.*out*.println("Doll " + doll.getId() + " passed inspection");  
 }  
  
 try {  
 stageC.await();  
 }

catch (BrokenBarrierException | InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 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);  
 }  
 else {  
 d.hasImperfections(true);  
 }  
 }  
 }  
}  
  
  
//////////////////////////////////////////  
  
class Doll {  
 int id;  
 int qualityScoreMachine;  
 boolean imperfect, isPainted;  
  
 public Doll(int id, int qualityScoreMachine) {  
 this.id = id;  
 this.qualityScoreMachine = qualityScoreMachine;  
 }  
  
 public int getQualityScore() {  
 return this.qualityScoreMachine;  
 }  
  
 public void setQualityScore(int qualityScoreMachine) {  
 this.qualityScoreMachine = qualityScoreMachine;  
 }  
  
 public int getId() {  
 return this.id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public boolean getImperfections() {  
 return this.imperfect;  
 }  
  
 public void hasImperfections(boolean imperfect) {  
 this.imperfect = imperfect;  
 }  
  
 public boolean isPainted() {  
 return this.isPainted;  
 }  
  
 public void setPainted(boolean painted) {  
 this.isPainted = painted;  
 }  
}