Laboratory Practices
Shared use of a Pool (2 sessions)

Concurrency and Distributed Systems

Introduction

The main goal of this practice is to complete a concurrent program that contains different synchronization conditions. When you complete the proposed tasks you will have learned:

- How to compile and execute concurrent programs.
- Detect synchronization errors in concurrent programs.
- Analyse synchronization requirements that appear in concurrent programs.
- Design solutions that satisfy a set of requirements.
- Implement those solutions.

You will be using Java as your programming language.

You will need approximately two weeks to complete the practice assignments. Each week you must attend one lab session, where the professor can help you with technical questions you might find. You might also need additional time to complete the practice.

In this document you will see some activities to be done. You should solve them and write the results, so as to facilitate further study of the content of the practice.

The Shared Pool problem

In this practice, we model the way a pool is shared among kids and swimming instructors. Kids learn to swim under instructors' supervision.

- Therefore there are two types of swimmers:
  - **Kid**, learning to swim.
  - **Instructor**, supervising one or more kids as they learn to swim.

- Swimming facilities have two areas:
  - Swimming pool (where people swim). – Every swimmer (either kid or instructor) goes into the water and swims some time, after which they exit the pool and rest.
  - Terrace. - External area where swimmers rest between swimming periods.

Swimmers execute this pseudo-code:

Swimmer
Repeat a number of iterations swim rest

Swimmers are coded as Threads. Several concurrent threads of both types (kid and instructor) are executed. The number of kids and instructors is configurable. Time consumed by kids and instructors to swim and to rest is random.

The swimming pool is modelled as the shared resource accessed by the different threads. Each thread must execute specific pool methods to enter and exit the pool.

Swimming pool Abstract Class *(Pool)*

Kids and instructors must follow specific pool usage rules. When kids or instructors try to perform an action that violates a particular pool rule, they must wait until pool usage requirements are met again (conditional synchronization).

There are 5 different types of pools where differences reside in which rules must be fulfilled to swim and to rest. Each pool is more restrictive than previous ones. This means, for instance, that to follow Pool3 rules implies following Pool2 and Pool1 rules.

Pool Type	*Rules for K kids and I instructors*
Pool0	Free access to the pool (no rules)
Pool1	Kids are not allowed to swim alone (instructors must be with them in the pool)
Pool2	There is a maximum number of kids per instructor (*max kids/instructor*)
Pool3	There is a maximum number of swimmers that can be in the pool at the same time (*maximum pool capacity*)
Pool4	If there are instructors waiting to exit the pool, no additional kids can enter the pool.

For instance, rule stated for Pool1 (kids cannot swim alone) must be satisfied at every upper pool (Pool2, Pool3 and Pool4).

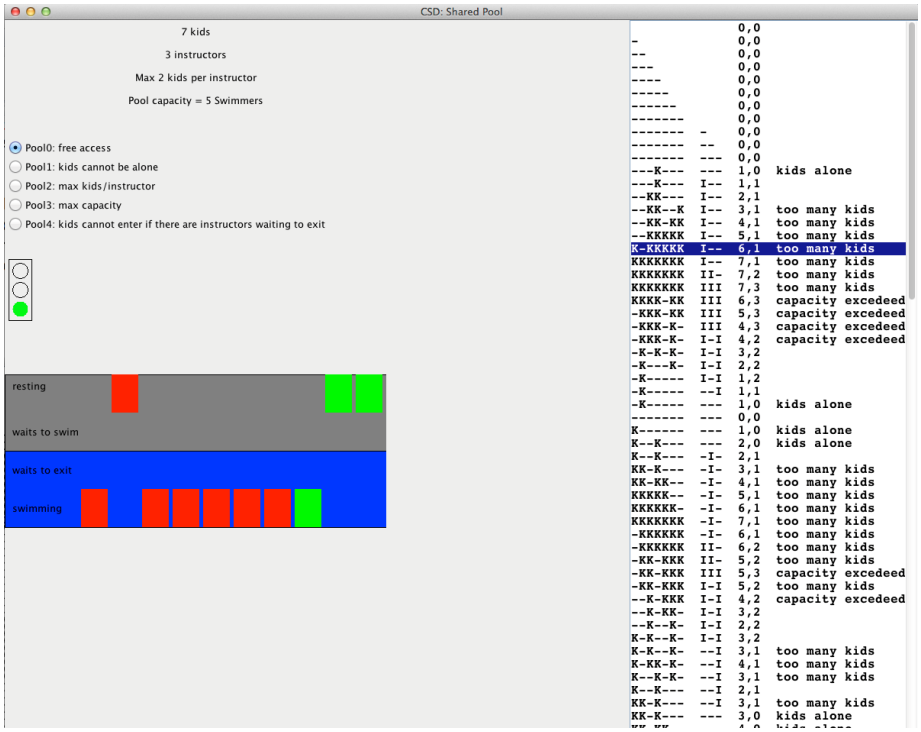
For this practice, you have a complete environment to execute pools and swimmers and Pool0 is already implemented. You can download the provided code from the poliformaT site (file “PPool.jar” in Practice 1). The goal of this practice is to gradually modify this code so as to give support to the other types of pools.

Code already implemented

Application is complete as provided. You can compile and execute it. However only Pool0 is implemented, so even if you select different types of pools, only Pool0 code is actually executed. You must implement the other pool types to complete the application.

The application includes a project description for BlueJ. However it is not mandatory to use it. You can use whichever Java environment you prefer, or you could just use a simple text editor and the java compiler. To open the project using BlueJ, select the menu option “Project/Open” and select file *PPool.jar*. To execute it, select class *PPool* as the class that contains the main() method. To execute the application from command line, just execute the command *java -jar PPool.jar*

Whichever the method you use to execute the application, you will see the application user interface as shown in the following picture. Using the interface, you can select the pool type. Once you select it, the simulation starts and it finishes when the semaphore is shown in green. If you observe that the traffic light stays red indefinitely and that the right of the screen has stopped writing, then there is a situation of deadlock (some threads have blocked between them).



The application window has the following areas:

At the upper left corner you can see the simulation parameters: 7 kids, 3 instructors, a maximum of 2 kids per instructor and pool capacity of 5 swimmers.

• Default parameters are 7 kids and 3 instructors. You can change them using program arguments. (e.g.: to have 12 kids and 4 instructors, execute: *java -jar PPool.jar* **12 4**).

• The number of kids must be within the range [5..20]; and instructors within range [2..5]. If not, default values will be used.

• The maximum number of kids per instructor is a derived parameter, obtained from operation K/I (i.e. numKids/numInstructors).

• Pool capacity is another derived parameter that is obtained from operation (K+I/2), (i.e. numKids + numInstructors)/2).

Under the parameter area you can find the area for pool selection. Whenever you select a new pool type, a new simulation is started. Notice that the provided code just contains implementation for Pool0, so whichever pool you select, just Pool0 executes. You must implement and test the other pools.

The right hand side area is the log area. It is used to show the simulation state as text lines with the following format:

• As many characters as kids (numKids). Possible characters are:
  - Whitespace: Kid thread has not started yet (or has terminated).
  - '-': Kid is resting.
  - '*': Kid is waiting to enter into the pool.
  - 'k': kid is waiting to exit the pool.
  - 'K': kid is swimming.

• As many characters as Instructors. Possible character values are:
  - Whitespace: Instructor thread has not started (or has terminated).
  - '-': Instructor is resting.
  - '*': Instructor is waiting to enter into the pool.
  - 'i': instructor is waiting to exit the pool.
  - 'I': instructor is swimming.

• The number of kids and instructors inside the pool (either swimming or waiting to exit the pool).

• If some particular pool rule is broken, there will be a text line explaining the situation. If current pool implementation should implement that rule, **text will be printed in Red color.** In that case, the semaphore will be in yellow.

• If all the kids have finished, the message "finished" is shown. If there are no instructors left (all of them have finished), but there are still active kids, the text "out of instructors" is displayed.

At the bottom left side of the window we can see a graphical representation of current state.

• Each swimmer is represented as a colored rectangle. Red for kids and green for instructors. Each swimmer appears at the same column but it changes its vertical position to reflect state changes. If a particular thread has not started or it has terminated, its rectangle is not shown.

• The grey area represents the terrace where swimmers rest. If they try to enter the pool but are forced to wait, they are shown in the bottom part of the terrace.

• Blue area is the swimming pool. Swimmers at the upper limit are waiting to exit the pool. Notice that swimmers at Pool0 never wait, so they are always shown swimming or resting.

Implementation classes

You are provided with three types of classes:

- Opaque classes. - They are required for the application to run, but their study is not required. **They must not be modified**.
  - *State, Box, StateRenderer, Light.*

- Translucent classes. - Students **must understand interfaces**, but implementation details inside classes are not important to develop the practice. **They must not be modified.**
  - *Pool.* – Abstract class with abstract operations, **whose code must be implemented in Pool0..Pool4**. These operations include the *init* method, which initializes the pool parameters, and operations invoked by a swimmer when requires a change of state (see code of *Swimmer*, *Instructor* and *Kid*).

```
public abstract void init(int ki, int cap);
public abstract void kidSwims() throws InterruptedException;
public abstract void kidRests() throws InterruptedException;
public abstract void instructorSwims()  throws InterruptedException;
public abstract void instructorRests()  throws InterruptedException;
```

  - *Log*.- This class contains important methods, some of which must be invoked when there is a change of state of a swimmer (e.g. when he/she enters the pool, when he/she has to wait for swimming, etc.). Thus, the application can graphically show the right state of swimmers.
    In this practice you must invoke the following methods of this class:

```
public void waitingToSwim()
public void swimming()
public void waitingToRest()
public void resting()
```

- Transparent classes.- These classes must be studied but **they must not be modified**.
  - *Swimmer*.- abstract class that implements a generic swimmer. `Kid` and `Instructor` will specialize it. Its `run()` method is the main code for each thread.

```
public abstract class Swimmer extends Thread{
  final int DELAY=60;
  Random rd=new Random();
  Pool pool; // pool to be used
  protected void delay() throws InterruptedException {
        Thread.sleep(DELAY+rd.nextInt(DELAY));}
  public Swimmer(int id0, Pool p) { super(""+id0); pool=p;}
  public void run() {     //Thread code
     try{
        pool.begin(); delay();
        for (int i=0; i<6 && !this.isInterrupted(); i++) {
          swims();  delay();
          rests();   delay();
        }
        pool.end();
    }catch (InterruptedException ex) {}
  }
   //Kid and instructor must implement this
  abstract void swims() throws InterruptedException;
  abstract void rests() throws InterruptedException;
}
```

- *Instructor*

```
public Instructor(int id, Pool p) {super(id,p);}
void swims() throws InterruptedException{pool.instructorSwims(); }
void rests() throws InterruptedException {pool.instructorRests(); }
```

- *Kid*

```
public Kid(int id, Pool p) {super(id,p);}
void swims() throws InterruptedException { pool.kidSwims(); }
void rests() throws InterruptedException {pool.kidRests(); }
```

- *PPool*.- This is the **main class**. It is interesting to know some code of the method simulate, specifically:
  - Initialization of the pool, calling to the *init* method of the Pool class, and passing as arguments the maximum numer of kids per instructor (KI) and the maximum capacity of the pool (CAP).

```
p.init(KI,CAP);
```

  - Creating and starting the threads:

```
// K=number of kids, I=number of instructors
Swimmer[] sw= new Swimmer[K+I]; //declares and creates swimmers
....
for (int i=0; i<K+I; i++)
    sw[i]= i<K? new Kid(i,p): new Instructor(i,p);
....
for (int i=0; i<K+I; i++)
    sw[i].start();//start swimmers
```

- *Pool0*.- Simple implementation of a free access pool (type 0). Notice that any other Pool that we implement **must extend the abstract class Pool**. Notice how methods of Log class (*log* object) are used in this Pool0 class.

```
public class Pool0 extends Pool {
  public void init(int ki, int cap)     {}
  public void kidSwims() { log.swimming(); }
  public void kidRests() { log.resting();}
  public void instructorSwims() {log.swimming();}
  public void instructorRests() {log.resting();}
}
```

- **Classes to be implemented**:
  - *Pool1, Pool2, Pool3, Pool4*. Initially, they have the same code as Pool0. Activities 1 to 4 explain what has to be implemented.

Activity 0 (Pool0)

Execute the *Pool0* simulation several times. Notice that each simulation produces different log output (different duration times for swimming/resting; possibly different scheduling decisions, etc.). Notice also that there are many situations that would break rules for upper level Pools (pool types 1,2,3,4).

Next table shows information about *Pool0* methods. For each method, it states when the method will block (i.e. the method cannot be completed because the pool state does not allow it), how it modifies the pool state and who this method must awake.

Pool0	wait if …	updates state …	Awakes …
kidSwims()	*Never waits*	*There is no state*	*No one*
kidRests()	*Never waits*	*There is no state*	*No one*
instructorSwims()	*Never waits*	*There is no state*	*No one*
instructorRests()	*Never waits*	*There is no state*	*No one*

Notice that Pool0 has no important internal state; therefore there is no state to be updated. No method blocks and no method awakes other threads. It is a simple class with no conditional synchronization.

Questions for Pool0 (free access):

1) Check whether the ***synchronized*** label has been used at the implementation of the methods of Pool0 (*kidSwims*, *kidRests*, *instructorSwims*, *instructorRests*). Would you observe any difference in execution of these methods when using or not using the *synchronized* label?

2 Can there be race conditions? Why?

3) How is the internal state of the pool represented? Why is it like that?

4) In Pool0, what state transitions can occur? Can kids and instructors go from "resting" to "swimming" directly, without passing through any intermediate states? Why?

Activity 1 (Pool1)

Activity 1.1: Specify which variables you use to represent the Pool1 state. Complete the following table to reflect your implementation details so that it implements the required rules for pool type 1.

Pool1	wait if …	Updates state …	Awakes …
kidSwims()	*There are no instructors*		
kidRests()			
instructorSwims()			
instructorRests()			

To wait until rules are fulfilled, we will use the following scheme:

```
while (condition to wait) {
     wait();
}
Update state
Awake threads waiting for this state
```

As the *wait*() statement can return an interrupt (if the thread is interrupted during its wait), the expression "throws InterruptedException" must be added to the declaration of the *kidSwims(), kidRests(), instructorSwims(), instructorRests()* methods when necessary.

For example, method *kidSwims*() from class Pool1 should be as follows (**the student must complete what is missing**):

```
public synchronized void kidSwims() throws InterruptedException {
   while (condition to wait) {  //COMPLETE the condition
       log.waitingToSwim();//to show the swimmer state
       wait();
   }
   ...   //Update state   (COMPLETE)
   ...   //If needed, awake threads waiting for this state
         //using notifyAll();
   log.swimming();  //to show the swimmer state
}
```

Activity 1.2: Complete *kidSwims* from class *Pool1* following the above-mentioned scheme. Threads invoking this code must wait if there are no instructors swimming. You must include your own variables **to keep track of the instructors and kids** entering and leaving the pool. If necessary, modify other methods of Pool1, apart from *KidSwims* method, to update these variables.

Compile and execute. You will observe that kids are kept waiting if they want to enter the pool when there are no instructors in it. However, there are still illegal situations.

Activity 1.3: Analyze the execution trace and determine which problems and illegal situations appear, and which other situations have been solved.

Activity 1.4: Review the rest of methods of Pool1 class (i.e. *kidRests, instructorSwims* and *instructorRests)* and update them as needed, to rightly show pool entrance/exit of kids and instructors.

Activity 1.5: Check that no Pool1 rule is broken. You must not see any red warning. Repeat simulations changing the number of kids and instructors and check whether your simulation is still correct.

Important! Check that your program never gets blocked. If so, review thread waiting and awaking conditions.

Remember that the **color of the traffic light** indicates:

- **Green light**: correct execution. All the rules of the pool are met.
- **Yellow light**: execution is finished, but some of the rules of the pool are not met.
- **Red light**: at least one of the threads (kids or instructors) has been locked indefinitely, so the execution can not be completed.

Questions Pool1 (kids cannot swim alone)

1) Check whether the **synchronized** label has been used at the implementation of Pool1 methods. Would you observe any difference in execution of these methods when using or not using the *synchronized* label?

2) Can there be race conditions? Why?

3) How is the internal state of the pool represented? Why is it like that?

4) We employ *notifyAll*() to notify of a new state of the pool. Could you employ *notify*() instead? Why?

5) At the end of which methods of Pool1 have you employed *notifyAll*()? Have you employed it at the end of all methods? If so, analyze whether it is compulsory to use *notifyAll*() for all methods, or if it is preferable (or more efficient) to use it only on the required ones.

Activity 2 (Pool2)

Activity 2.1: Explain which variables you need to reflect the sate needed for pools of type 2. Recall that Pool2 must follow Pool1 rules too. To that end, complete the following table:

Pool2	wait if …	updates state …	Awake …
kidSwims()			
kidRests()			
instructorSwims()			
instructorRests()			

Activity 2.2: Update *Pool2* class to follow its required rules.

Activity 2.3: Check that your Pool2 implementation works when you change the number of kids and instructors.

Questions Pool2 (maximum kids per instructor)

1) To represent the state of the shared object, is it enough to use an integer variable (e.g. *nSwimkids*) and a Boolean variable (e.g. *instInPool*)? Why ?

2) Which Pool2 methods have you updated?

3) When an instructor goes inside Pool2, must he invoke *notifyAll*()? Why?

Activity 3 (Pool3)

Activity 3.1: Explain which variables you need as the Pool3 state. Complete the following table:

Pool3	wait if …	updates state …	Awakes …
kidSwims()			
kidRests()			
instructorSwims()			
instructorRests()			

Activity 3.2: Complete the implementation of Pool3 class so as to follow its required rules.

Activity 3.3: Verify that your implementation for Pool3 works executing it with different number of kids and instructors.

Questions Pool3 (maximum capacity):

1) To represent the state of pool, do you need to add any other variable to those used at the implementation of Pool2? If so, which one? Why?

2) Can you say that Pool3 is acting as a "monitor"? And previous pools, are they also acting as "monitors"?

Activity 4 (Pool4)

Activity 4.1: Explain which variables are needed to implement Pool4. Complete the following table:

Pool4	Wait if …	Updates state …	Awakes …
kidSwims()			
kidRests()			
instructorSwims()			
instructorRests()			

Activity 4.2: Complete implementation of Pool4 class so as to follow its required rules.

Activity 4.3: Check that your implementation satisfies rules for pools of type 4. Change the number of kids and instructors and verify it works correctly.

Questions Pool4 (if instructors waiting, kids cannot enter)

1) To represent the state of this pool, do you need to add any other variable to those used at the implementation of Pool3? If so, which one? Why?

2) At the end of which methods of Pool4 have you employed *notifyAll*()? Have you employed it at the end of all methods? If so, analyze whether it is compulsory to use *notifyAll*() for all methods, or if it is preferable (or more efficient) to use it only on the required ones.