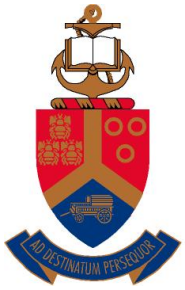

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science
COS 226 - Concurrent Systems Date Issued: 24 August 2021

Practical Assignment 1

- **Due Date:** 2 September 2021
- **Assessment:** The practical will be assessed via a live demo during the practical sessions on the 2nd of September
- This practical consists of 3 tasks. Read each task **carefully**!

1 Information

1.1 Objectives

This practical aims to introduce the topic of Java threads as well as basic lock functionality within Java. Please try hard to familiarize yourself with all aspects of Java threading as from here onwards, all future practicals will assume you are familiar with them. It is already assumed you are able to create programs within the Java langauge.

You must complete this assignment individually.

1.2 Provided Code

Some example code to demonstrate the functionality of threads within Java as well as two different implementations of threads have been provided. Please download these examples from either ClickUp or the Discord server before continuing.
While future practicals may contain starting code for you to build on, for the purposes of this practical, all code must be written from scratch.

1.3 Mark Allocation

For each task in this practical, in order to achieve any marks, the following must hold:

- Your code must produce console output. (As this is not marked by fitchfork, formatting is not that strict)
- Your code must not contain any errors. (No exceptions must be thrown)
- Your code may not use any external libraries.
- You must be able to explain your code live to a tutor and answer any questions asked.

The mark allocation is as follows:

Task Number	Marks
Task 1	5
Task 2	5
Task 3	5
Total	15

2 Assignment

Java supports multiprocessing via a wide array of methods, however for the purposes of this course, we will be focusing on threads specifically.
Java encompasses the idea of a thread into a suitably named class, the **Thread** class.
There are a couple of methods we can use to implement this class, which will be introduced to you in this course.

No matter which method you choose to use, the end result is the creation of an explicit **Thread** object in Java.

The first way (and perhaps the easiest way) to create this object is by extending the **java.lang.Thread** class. In the code provided to you, the **TDemo** class is an example of this method.

The second way to create a **Thread** object is by creating a class that implements the **java.lang.Runnable** interface, and have an instance of this class passed to the constructor of a new **Thread** object.
The **RDemo** class is an example of this implementation.

The **Main** class in the example given covers creation of the **Thread** objects and the process of starting the threads.

Please study the example carefully.

Notes:

- No matter which method you use, your class will need to contain a **run()** method. This method serves as the 'action' of the thread that will occur when you call the thread's **start()** method.
- Try playing around with the example to gain a better understanding of threads.

- See what happens if you comment out lines 18-22 in the **Main** class of the demo.
- Some useful functions for the Thread class:
  - **void sleep(long)** will cause the thread to cease execution for the number of milliseconds given. (Note that this method throws an **InterruptedException** which will need to be caught)
  - **String getName()** will return the name of the thread. (This is very useful for output).
  - **Thread currentThread()** will return a reference to the currently executing thread. (This can be replaced with the '**this**' keyword if you are extending the **Thread** class)
- Further information about the **Thread** class and **Runnable** interface can be obtained via the online Java Documentation.

2.1 Task 1 - Thread Creation

For the following task you will need to implement threading in Java based on the above information. You may use **either** method for your implementation.

The following must be completed:

- A class named **Counter** must be created, this class must consist of:
  - A single integer variable initialized to a value of 0.
  - A single method that increments the value of the stored integer by 1 and returns the new value.
- A class that implements a thread, which must consist of:
  - A variable to hold a reference to a **Counter** object.
  - A constructor that takes in a **Counter** object and stores it in the variable.
  - The **run()** method, which should call the **Counter** objects method a minimum of 5 times and display output as follows:
    [Thread Name] Counter: [Counter Value]
    Example: Thread-0 Counter: 3
- Two threads need to be initialized and run using the class you have created and the shared Counter object passed to them.
  Example of creation if the **MyThread** class extends **Thread**:

```
Counter c = new Counter();
Thread t1 = new MyThread(c);
Thread t2 = new MyThread(c);
```

- If the method of the **Counter** method is called 10 times, the final output should logically be 10, however notice this may not be the case in your code.

2.2 Task 2 - Locking

For the next task you will need to extend your previous implementation to make use of a lock. For this task, we will be implementing a lock built into Java. Namely, the **java.util.concurrent.locks.ReentrantLock** class.

We will be using this lock to enforce mutual exclusion the critical section of the previous task, so that only one thread may access the critical section at a time. The output of the counter will now be logically correct. i.e. if the method of **Counter** is called 10 times, the final output will be 10.

The following code may assist in creation and use of the lock: (You may also consult the documentation online)

```
Lock l = ....;
.....
l.lock();
try{
    //critical section....
}
finally{
   l.unlock();
}
```

Notes/Hints:

- ONE **ReentrantLock** object must be created in the program, otherwise the **Lock.lock()** method will not function correctly.
- The output must now be logically correct. (The output order may be mixed up but no two outputs may output the same counter number)

2.3 Task 3 - Peterson Lock

For the final task, you must replace the **ReentrantLock** with YOUR OWN implementation of the Peterson Lock.

Notes:

- Create a new class for the Peterson Lock that **implements** the **Lock** interface from Java, similar to how you would implement the **Runnable** interface in thread creation.
- Copying the code directly from the textbook will probably not work, as Java handles Thread IDs differently, you will have to find a way to use the **Thread Name** of a thread to differentiate the threads.

3 Final Notes

You will have to demonstrate each task of this practical separately, so be sure to create copies of your source code.
Good Luck!