Department of Computer Science
COS226 - Concurrent Systems

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Practical Assignment 5

- ☐ **Date Issued:** Tuesday, 29 September 2020
- ☐ **Date Due:** Monday, 19 October 2020, 18:00
- ☐ **Submission Procedure:** Upload your tasks on the CS Assignments Portal website
- ☐ **Assessment:** The practical will be marked online through the Assignments Portal and slots will open in the second week of the practical.
- ☐ This assignment consists of 4 tasks

---

1 Introduction

1.1 Objectives and Outcomes

You must complete this assignment individually.

You may ask the Teaching Assistants for help but they will not be able to give you the solutions. They will be able to help you with coding, debugging and understanding the concepts explained both in the textbook and during lectures.

1.2 Source Code

Before you begin, you should download the source code from the ClickUP website. You are allowed to make changes to all the files in the source code. Ensure that your **student number** and **name** appears in the comments section at the top of each file that you submit.

1.3 Mark Allocation

This assignment is divided into four tasks. For each task:

(a) your program must produce the expected output;

(b) your program must not throw any exceptions; and

(c) your program must implement the correct mechanisms or algorithms, as specified by the assignment.

Your code will be assessed online through a script. Please note that the same marking rubric is used for auto-assessment and manual marking. If you do wish to have the structure of you code marked offline you can query after you receive your marks.

The marks for this practical are allocated as follows:

Activity	Mark
Task 1	6
Task 2	8
Task 3	8
Task 4	10
Total	**32**

2 Program Logger

The program output will be displayed using a Logger object from the java.util.logging package. Each class that needs to display an output needs to declare a global Logger variable. For example:

```
private static final Logger LOGGER = Logger.getLogger("global");
```

The display configurations are specified in the logger.properties file.

This practical assignment will make use of two logging levels to display messages, `INFO` and `FINE`.

When you set a level, all messages belonging to that level and higher levels will be displayed. For example, if the level is set to `FINER`, messages logged as `FINER, FINE`, and `INFO` will be displayed. Please make sure to implement your output messages using the **correct logging method**. Failure to do so may result in you not being awarded marks for certain items, as the marking script will use the logging level to examine different types of thread behaviours.

The keywords enclosed in square brackets (**[keyword]**) should be replaced with the associated data.

3 Implementation

By now you should know the implementation of a few different types of locking methods. This practical will require you to implement different types of locks from Chapter 7 and Chapter 8.

- Task 1 requires you to implement a TTAS (*test-and-test-and-set*) Lock.
- Task 2 requires you to implement an Anderson Queue Lock.
- Task 3 requires you to implement a Semaphore.
- Task 4 requires you to implement a Fair Readers-Writers Lock.

3.1 Task 1 - Store Limit Simulator

3.1.1 What to Implement

You are a store owner, and because of the pandemic you limit your store to one customer at a time. You need to use a **TTAS Lock** to simulate this.

- `TTASLock`
  - This class is given to you. You will need to implement the following functions in this class:
    - `void lock()`
    - `void unlock()`
  - The given `TTASLock` class extends an `AbstractLock` class (also given) to make the code implementation compact. You can opt to have your `TTASLock` class directly implement the `java.util.concurrent.locks.Lock` interface.
- `Customer`
  - You will need to create this class and implement the following functions:
    - `Customer(TTASLock lock)`
    - `void run()`

3.1.2 Implementation Notes

- In order to test your code you will need to simulate the above scenario. The lock has to work for n threads. Each thread represents a customer.
- Each customer (thread) must try enter the store, this is simulated by trying to acquire the lock. If the lock is acquired, the customer (thread) can enter the store. If the lock can't be acquired, the customer must wait.
- Each thread will need to do the following:

  - sleep for a random amount of time between 10 (inclusive) and 50 (inclusive). This simulates people coming to the store at different times.
  - call `lock()`
  - if lock is acquired, enter the store (critical section)
  - sleep for a random amount of time between 10 (inclusive) and 50 (inclusive) milliseconds to simulate a customer in the shop
  - call `unlock()` after sleeping

- This sequence of events needs to happen x times. **The `Main` class is responsible for simulating several runs of this scenario.**
- The configurable scenario values can be specified through the following `CustomerOptions` static variables:
  - n: `CustomerOptions.customers`
  - x: `CustomerOptions.runs`
  - w: `CustomerOptions.wait`
- The `CustomerOptions.wait` value will generally be between 5 and 10.
- Please do not hardcode any values as our marking scripts tests with our own values. All values used need to be done in a dynamic nature.
- **Please note**: Only one customer (thread) is allowed in the store (critical section) at any given time

3.1.3 Expected Output

A full example of output is shown at the end of this section. Your output statements should be done in the TTASLock class.

- When a thread calls `lock()`, you need to output the following:

  `Customer [thread-id] is trying to enter the store`

  Note: This must be logged using the INFO level.

  - When a thread successfully acquires the lock, you need to output

    `Customer [thread-id] has entered the store`

    Note: This must be logged using the INFO level.

  - When a thread tries to acquire the lock, but sees that another thread is in possession of the lock, the thread should sleep for w millisecond (specified by `CustomerOptions.wait`) and then output:

    `Customer [thread-id] tried to enter the store but sees customer [thread-id who holds lock] is already in the store`

Note: This must be logged using the INFO level. A thread should display the above output once per lock acquisition attempt. The thread should thereafter spin until the lock becomes available.

- When a thread calls `unlock()`, you need to output:

  `Customer [thread-id] has left the store`

  Note: This must be logged using the INFO level. You need to output this message **before any update of a shared variable** in the `unlock()` method. The effect of this is that the notification of the next customer to enter the store will only be displayed **after** the current customer's store exit message.

Full example output of 1 run with 4 threads:

```
Customer 3 is trying to enter the store
Customer 2 is trying to enter the store
Customer 0 is trying to enter the store
Customer 1 is trying to enter the store
Customer 3 has entered the store
Customer 2 tried to enter the store but sees customer 3 is already in the store
Customer 0 tried to enter the store but sees customer 3 is already in the store
Customer 1 tried to enter the store but sees customer 3 is already in the store
Customer 3 has left the store
Customer 0 has entered the store
Customer 0 has left the store
Customer 2 has entered the store
Customer 2 has left the store
Customer 1 has entered the store
Customer 1 has left the store
```

3.1.4 Upload Instructions

You must archive your files and upload them to the Task 1 submission slot on the CS Assignments Portal. Some points to take note of:

- You need to upload all of your files, except `Main.java`.
- Ensure there are no binary files in your archive (such as `.class` files).
- You can archive your files using `zip`, `tar`, or `tar.gz`.
- Make sure your files are at the root of your archive. Do not archive a folder.

3.2 Task 2 - Fair Store Limit Simulator

3.2.1 What to Implement

You have received complaints from waiting customers that there is no clear order in which they have to wait before accessing the store. If, for example, a customer tries to enter the shop and finds it full, then that customer has to wait, but they are currently just standing around, without any order. When a space then becomes available, waiting customers have to compete to be able to enter the store and it is possible for waiting customers to overtake one another. You need to use an **Anderson Queue Lock** to simulate this.

- `AndersonQueueLock`
  - This class is given to you. You will need to implement the following functions in this class:
    - `AndersonQueueLock(int numThreads)`
    - `void lock()`
    - `void unlock()`
  - The given `AndersonQueueLock` class extends an `AbstractLock` class (also given) to make the code implementation compact. You can opt to have the `AndersonQueueLock` class directly implement the `java.util.concurrent.locks.Lock` interface.
- `Customer`
  - You will need to create this class and implement the following functions:
    - `void run()`
    - `Customer(AndersonQueueLock lock)`

3.2.2 Implementation Notes

- In order to test your code you will need to simulate the above scenario. The lock has to work for n threads. Each thread represents a customer.
- Each customer (thread) must try to enter the store, this is simulated by trying to acquire the lock. If the lock is acquired, the customer (thread) can enter the store. If the lock can't be acquired, the customer must wait.
- Each thread will need to do the following:
  - Sleep for `((ThreadID.get()+1) * 100 % 75)` milliseconds. This simulates people coming to the store at different times
  - The thread must then enter the store (call `lock()`)
  - The thread must sleep for a random value between 10 (inclusive) and 50 (inclusive) milliseconds to simulate a customer in the shop
  - The thread must leave the shop (call `unlock()`)
- This sequence of events needs to happen x times. The `Main` class is responsible for simulating several runs of this scenario.

- The configurable scenario values can be specified through the following `CustomerOptions` static variables:
  - n: `CustomerOptions.customers`
  - x : `CustomerOptions.runs`
- Please do not hardcode any values as our marking scripts tests with our own values. All values used need to be done in a dynamic nature.
- **Please note**: Only one customer (thread) is allowed in the store (critical section) at any given time, and they must follow the FIFO principle.

3.2.3 Expected Output

- When the customer enters the store, output the following:

  `Customer [thread-ID] enters the store`

  Note: This is done using logging level INFO.

- When the customer leaves the store, output the following:

  `Customer [thread-ID] leaves the store`

  Note: This is done using logging level INFO.

- When the customer obtains the lock:

  `Thread [thread-ID] has slot [slot number]`

  Note: This is done using logging level FINE.

Full example output of 1 run with 5 threads:

```
Thread 2 has slot 0
Customer 2 enters the store
Customer 2 leaves the store
Thread 0 has slot 2
Thread 3 has slot 1
Customer 3 enters the store
Thread 4 has slot 4
Thread 1 has slot 3
Customer 3 leaves the store
Customer 0 enters the store
Customer 0 leaves the store
Customer 1 enters the store
Customer 1 leaves the store
Customer 4 enters the store
Customer 4 leaves the store
```

3.2.4 Upload Instructions

You must archive your files and upload them to the Task 2 submission slot on the CS Assignments Portal. Some points to take note of:

- You need to upload all of your files, except `Main.java`.
- Ensure there are no binary files in your archive (such as `.class` files).
- You can archive your files using `zip`, `tar`, or `tar.gz`.
- Make sure your files are at the root of your archive. Do not archive a folder.

3.3 Task 3 - Fair Store Simulator With an Increased Limit

3.3.1 What to Implement

You have just heard that the pandemic restrictions have been eased and you are now allowed to have more than one person in your store at a time. Use a **Semaphore** to simulate this scenario. The store should still be fair.

- `Semaphore`
  - You will need to implement this class and the following functions in this class:
    - `Semaphore(int capacity)`
    - `void acquire()`
    - `void release()`
- `Customer`
  - You will need to create this class and implement the following functions:
    - `Customer(Semaphore semaphore)`
    - `void run()`

3.3.2 Implementation Notes

- In order to test your code you will need to simulate the above scenario. The lock has to work for n threads. Each thread represents a customer.
- Each customer (thread) must try enter the store, this is simulated by trying to acquire the semaphore with capacity c. If the semaphore is not at its capacity, the customer (thread) can enter the store. If the semaphore is at its capacity, the customer must wait.
- Each thread will need to do the following:
  - sleep for a random amount of time between 10 (inclusive) and 50 (inclusive) milliseconds. This simulates people coming to the store at different times
  - call `acquire()`
  - if the lock is acquired, enter the store (critical section)
  - sleep for a random amount of time between 10 (inclusive) and 50 (inclusive) milliseconds to simulate a customer in the shop
  - call `release()` after sleeping
- This sequence of events needs to happen x times. The `Main` class is responsible for simulating several runs of this scenario.
- The configurable scenario values can be specified through the following `CustomerOptions` static variables:
  - n: `CustomerOptions.customers`

- x : `CustomerOptions.runs`
- c : `CustomerOptions.capacity`

Note that c is given as input to the `Semaphore` class constructor.

- Please do not hardcode any values as our marking scripts tests with our own values. All values used need to be done in a dynamic nature.
- **Please note**: All customers (threads) should enter the store (critical section) eventually and your program should not deadlock.

3.3.3 Expected Output

A full example of output is shown at the end of this section. Your output statements should be done in the Semaphore class.

- When a thread calls `acquire()`, you need to output the following:

  `Customer [thread-id] is trying to enter the store`

  Note: This must be logged using the INFO level.

  You also need to output the following:

  - The number of available spaces **before** the current thread enters the store:

    `Available spaces in the store:  [amount]`

    Note: This must be logged using the FINE level.

  - When a thread successfully enters the store, output:

    `Customer [thread-id] has entered the store`

    Note: This must be logged using the INFO level.

  - When a thread does not successfully enter the store, you need to output:

    `Customer [thread-id] tried to enter the store but sees the store is at its capacity`

    Note: This must be logged using the INFO level.

    And wait until more spaces in the store become available.

- When a thread calls `release()`, you need to output:

  `Customer [thread-id] has left the store`

  Note: This must be logged using the INFO level.

Full example output of 1 run with 5 threads, and a capacity of 3:

```
Customer 0 is trying to enter the store
Available spaces in the store:  3
Customer 0 has entered the store
Customer 4 is trying to enter the store
Available spaces in the store:  2
Customer 4 has entered the store
Customer 3 is trying to enter the store
Available spaces in the store:  1
Customer 3 has entered the store
Customer 2 is trying to enter the store
Available spaces in the store:  0
Customer 2 tried to enter the store but sees the store is at its capacity
Customer 1 is trying to enter the store
Available spaces in the store:  0
Customer 1 tried to enter the store but sees the store is at its capacity
Customer 3 has left the store
Customer 2 has entered the store
Customer 0 has left the store
Customer 1 has entered the store
Customer 4 has left the store
Customer 2 has left the store
Customer 1 has left the store
```

3.3.4 Upload Instructions

You must archive your files and upload them to the Task 3 submission slot on the CS Assignments Portal. Some points to take note of:

- You need to upload all of your files, except `Main.java`.
- Ensure there are no binary files in your archive (such as `.class` files).
- You can archive your files using `zip`, `tar`, or `tar.gz`.
- Make sure your files are at the root of your archive. Do not archive a folder.

3.4 Task 4 - Poster Simulator

3.4.1 What to Implement

You have now decided to have a poster outside your store where you write how many spaces are still available in the store. You will not be implementing the entire scenario, but rather a **Fair Readers-Writers Lock** that will be used to demonstrate multiple threads reading from and writing to a shared resource.

- `FIFOReadWriteLock`
  - This class is given to you. You will need to implement the following functions in this class:
    - `FIFOReadWriteLock()`
    - `Lock readLock()`
    - `Lock writeLock()`
  - This class will also contain the following classes, each with their respective functions to be implemented:
    - `ReadLock`
      - `void lock()`
      - `void unlock()`
    - `WriteLock`
      - `void lock()`
      - `void unlock()`
  - The `ReadLock` and `WriteLock` classes extend the given `AbstractLock` class to make the code implementation compact. You can opt to have these classes directly implement the `java.util.concurrent.locks.Lock` interface.
- `Customer`
  - You will need to create this class and implement the following functions:
    - `Customer(FIFOReadWriteLock lock, Boolean isWriter)`
    - `void run()`

3.4.2 Implementation Notes

- There will be w writer threads and r reader threads. A Boolean value passed to the thread's constructor will be used to determine whether the thread is a writer or reader thread, where:
  - A value of `true` represents a writer thread
  - A value of `false` represents a reader thread
- Each thread will need to do the following:
  - call the respective lock's `lock()` method

  - if the lock is acquired, sleep for a random amount of time between 10 (inclusive) and 50 (inclusive) milliseconds to simulate the thread reading or writing
  - call the respective lock's `unlock()` method after sleeping

- This sequence of events needs to happen x times. The `Main` class is responsible for simulating several runs of this scenario.
- The configurable scenario values can be specified through the following `CustomerOptions` static variables:
  - w: `CustomerOptions.writers`
  - r: `CustomerOptions.readers`
  - x: `CustomerOptions.runs`
- Please do not hardcode any values as our marking scripts tests with our own values. All values used need to be done in a dynamic nature.
- **Please note**: Your program should not deadlock.

3.4.3 Expected Output

A full example of output is shown at the end of this section.

- When a thread's `run()` method is called, you need to output the following:

  ```
  Thread [thread-id] is a [reader/writer]
  ```

  Note: This must be logged using the FINE level.

- When a thread calls the respective lock's `lock()` method, output:

  ```
  Thread [thread-id] wants to acquire the [read/write] lock
  ```

  Note: This must be logged using the INFO level.

- When a thread calls successfully acquires the respective lock, output the following:

  ```
  Thread [thread-id] acquired the [read/write] lock
  ```

  Note: This must be logged using the INFO level.

- When a thread calls the respective lock's `unlock()` method, and therefore release that lock, output:

  ```
  Thread [thread-id] released the [read/write] lock
  ```

  Note: This must be logged using the INFO level.

Full example output of 1 run with 2 writer threads and 4 reader threads:

```
Thread 1 is a writer
Thread 0 is a writer
Thread 4 is a reader
Thread 2 is a reader
Thread 3 is a reader
Thread 5 is a reader
Thread 1 wants to acquire the write lock
Thread 1 acquired the write lock
Thread 0 wants to acquire the write lock
Thread 4 wants to acquire the read lock
Thread 2 wants to acquire the read lock
Thread 3 wants to acquire the read lock
Thread 5 wants to acquire the read lock
Thread 1 released the write lock
Thread 0 acquired the write lock
Thread 0 released the write lock
Thread 4 acquired the read lock
Thread 2 acquired the read lock
Thread 3 acquired the read lock
Thread 5 acquired the read lock
Thread 5 released the read lock
Thread 2 released the read lock
Thread 4 released the read lock
Thread 3 released the read lock
```

3.4.4 Upload Instructions

You must archive your files and upload them to the Task 4 submission slot on the CS Assignments Portal. Some points to take note of:

- You need to upload all of your files, except `Main.java`.
- Ensure there are no binary files in your archive (such as `.class` files).
- You can archive your files using `zip`, `tar`, or `tar.gz`.
- Make sure your files are at the root of your archive. Do not archive a folder.