Department of Computer Science
COS226 - Concurrent Systems

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Practical Assignment 2

- ☐ **Date Issued:** Monday, 17 August 2020
- ☐ **Date Due:** Sunday, 23 August 2020, 18:00
- ☐ **Submission Procedure:** To be communicated within the week
- ☐ **Assessment:** To be communicated within the week
- ☐ This assignment consists of 3 tasks

---

1 Introduction

1.1 Objectives and Outcomes

From here on forward, the practical assignments assume that you know the basics of how threads work. 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 three 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.

Please note that we are working on moving the practical assessments online to allow you to get immediate feedback on your attempts. We will communicate where to upload your practical assignment during the course of the week. This should not deter you from working on the practical in the mean time.

The marks for this practical are allocated as follows:

Activity	Mark
Task 1	8
Task 2	12
Task 3	15
Total	**35**

2 Implementation

By now you have learnt a few different types of locking methods. For this practical you will be implementing the `Peterson Algorithm` and `Bakery Algorithm` that you have learnt in class.

You will also have to implement a variation of the `Bakery Algorithm` called the `Black-White Bakery Algorithm`.

2.1 Scenario - Family Account

An Account is shared between family members. Each thread represents a member of the family. When a member is finished withdrawing all the money they need out of the account, they deposit a certain amount back in at the end. If there is not enough money in the account they will not withdraw anything.

2.2 Task 1 - Peterson Lock

In this task you must code the Peterson lock algorithm (in the `PetersonLock.java` file) such that it enforces mutual exclusion on the critical sections of the `Account`'s `withdraw` and `deposit` functions. The `PetersonLock` class should implement the `Lock` interface. You are provided with the following files: `PetersonLock.java`, `Account.java`, `TThread.java`, `TThreadOptions.java`, `ThreadID.java`, `ThreadAccountDemo.java`, and `logger.properties`.

2.2.1 Implementation Instructions

Modify the given source code files as necessary such that your program supports the following functionality.

- Complete the `deposit(...)` and `withdraw(...)` functions of the `Account` class such that they increment and decrement the account balance by the given amount, respectively. The `withdraw` function should return a `boolean` value indicating whether or not a withdrawal transaction was successful.
- The primary methods of the `Lock` interface which you must implement include the `constructor`, `lock()`, and `unlock()` functions. You need to implement the `Peterson` lock algorithm as described in the textbook. Use the given `ThreadID` class to generate a local thread index for your `Peterson` lock implementation.

- To test your `Peterson` lock, you will make use of the given shared `Account` class. Make the necessary changes to the `Account.java` file to ensure that mutual exclusion is enforced through the `Peterson` lock. The `Account` class should be **lock-agnostic**. This means that you should be able to use any `Lock` object in the `Account` class without explicitly declaring the type of lock used in the `Account.java` file.
- You will have two (2) threads running in this system, as seen in `ThreadAccountDemo.java`. Implement the `TThread` class such that each thread withdraws Rx from the shared account over n iterations, and sleeps for t milliseconds in-between its transaction attempts. The aforementioned values can be specified through the following `TThreadOptions static` variables:
  - x: `TThreadOptions.withdrawAmount`
  - n: `TThreadOptions.numRuns`
  - t: `TThreadOptions.sleepTime`

  A withdrawal should only be successful if the account has sufficient funds.
- After exiting the loop, each thread should deposit Ry into the shared account using the `deposit(float amount, String threadName)` method. The amount y can be set through `TThreadOptions.depositAmount`.

2.2.2 Expected Output

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 4 logging levels to display different types of messages. The logging levels in descending order are: `INFO`, `FINE`, `FINER`, and `FINEST`. 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 levels to examine different types of thread behaviours.

Your program should produce the following outputs. They keywords enclosed in dollar signs (`$keyword$`) should be replaced with the associated data.

- For each account withdrawal, display a message that contains the name of the thread associated with a transaction, the transaction amount, and the remaining account balance. All `Account` messages should use the `INFO` logging level. Use the following format:

  ```
  $Thread-name$ $amount$ withdrawn from account, R$balance$ remaining
  ```

  In the case of insufficient funds, the following message should be displayed:

```
$Thread-name$ Not enough money, R$balance$ remaining
```

Sample withdrawal outputs:

```
Thread-0 R400.00 withdrawn from account, R1860.00 remaining
Thread-1 R400.00 withdrawn from account, R1460.00 remaining
Thread-0 R400.00 withdrawn from account, R1060.00 remaining
...
Thread-1 Not enough money, R260.00 remaining
```

- Log each deposit transaction as follows:

```
$Thread-name$ $amount$ deposited into account, R$balance$ remaining
```

Sample deposit output:

```
Thread-0 R500.00 deposited into account, R760.00 remaining
```

- To analyze thread behaviour under the Peterson lock algorithm, a message should be displayed when a thread becomes the victim, and when it is about to enter or leave the critical section (CS). The messages should be logged as FINE messages. The variable $\texttt{\$thread}_i\texttt{\$}$ shown below refers to the local thread ID generated by the lock.

```
[Lock] Thread $thread_i$ is victim
[Lock] Thread $thread_i$ entering CS (thread $victimID$ is victim)
[Lock] Thread $thread_i$ leaving CS
```

A sample output is shown below.

```
[Lock] Thread 0 is victim
[Lock] Thread 1 is victim
[Lock] Thread 0 entering CS (thread 1 is victim)
Thread-2 R800.00 withdrawn from account, R1460.00 remaining
[Lock] Thread 0 leaving CS
...
```

2.2.3 Execution Notes

Some questions to consider while testing your code:

- When you run your code, is the order of the victim display messages always as expected? Why or why not?
- Explore different thread conditions such as the number of runs, the time delay between thread runs, and withdrawal and deposit amounts. Take note of any thread behavioural changes to deepen your understanding of the Peterson lock algorithm.

2.2.4 Upload Instruction

Compress the following files into a single archive.

- `PetersonLock.java`

- `Account.java`
- `TThread.java`

Details of the upload slot will be communicated later.

2.3 Task 2 – Bakery Algorithm

In this task you have to implement the Bakery Lock algorithm as specified in the textbook. The Bakery lock algorithm should fully implement the `Lock` interface. Use the same `Account.java`, `TThread.java` and `ThreadAccountDemo.java` files from Task 1 and add your own `Bakery.java` file.

2.3.1 Implementation Instructions

- The Bakery class constructor should accept as input a single integer argument to indicate the maximum number of threads.
- All functions required to ensure that the Bakery algorithm functions correctly will need to be implemented. You may implement additional helper functions as needed.
- Test your Bakery lock implementation on the shared `Account` class.
- Use the same testing conditions as in Task 1. However, unlike Task 1, you will need to experiment with a variable number of threads.

2.3.2 Expected Output

Your program should produce the following outputs. The terms ticket and label are used interchangeably in this text.

- The `Account` class messages should be as specified in Task 1.
- To analyze thread behaviour under the Bakery lock algorithm, a message should be displayed when a thread is assigned a ticket, when a priority assessment is conducted to determine which thread should enter the critical section (CS) first, and when a thread enters or leaves a critical section.
- The messages have different logging levels. Adhere to the following level specifications:

Level	Message Type
`FINE`	Thread ticket assignment
`FINER`	Thread enters CS
`FINER`	Thread leaves CS
`FINEST`	Assess thread priority

- The formats for the message logs are as follows:
  - When a thread is assigned a ticket, display:
    `[LOCK]: (id:0,label:1) => ticket assigned`
    where 0 is the local thread ID, and 1 is the associated ticket number

- When a thread is about to enter the critical section, display:
  `[LOCK]: (id:0,label:1) => entering CS`
  where 0 is the local thread ID, and 1 is the associated ticket number
- When a thread leaves the critical section, display:
  `[LOCK]: (id:0,label:1) => left CS`
  where 0 is the local thread ID, and 1 is the associated ticket number
- Before entering the critical section, each thread runs a comparative analysis to determine whether it has the highest priority. A thread with a higher priority enters the critical section next. Display a message as follows:
  `[LOCK]: assess priority => (id:0,label:1) > (id:1,label:1)`
  Here, thread 0 with a label of 1 has higher priority than thread 1 with a label of 1. Only display messages for comparisons between threads that have **both raised** their flags.

- A sample output is shown below.

```
[LOCK]: (id:0,label:1) => ticket assigned
[LOCK]: (id:2,label:2) => ticket assigned
[LOCK]: (id:1,label:1) => ticket assigned
[LOCK]: assess priority => (id:0,label:1) > (id:1,label:1)
[LOCK]: assess priority => (id:0,label:1) > (id:2,label:2)
[LOCK]: (id:0,label:1) => entering CS
Thread-3 R800.00 withdrawn from account, R1460.00 remaining
[LOCK]: (id:0,label:1) => left CS
...
```

2.3.3 Execution Notes

Some questions to consider while testing your code:

- Explore different thread conditions such as the number of threads, number of runs, etc. Take note of any thread behavioural changes to deepen your understanding of the Bakery lock algorithm.
- What would happen to the ticket numbers if you were to run this algorithm for a total of 2.3 billion iterations? How would this influence the execution order of the threads?

2.3.4 Upload Instructions

Compress the following files into a single archive.

- `Bakery.java`
- `Account.java`
- `TThread.java`

Details of the upload slot will be communicated later.

2.4 Task 3 – Black-White Bakery Algorithm

To circumvent the unbounded label number problem of the Bakery Algorithm, Taubenfeld [1] proposed the use of one additional shared bit to bound the amount of space required in the Bakery algorithm. Taubenfeld uses the analogy of a ticket instead of the `Label` concept described in the course textbook. The shared bit indicates a colour of a ticket issued to a thread or process, where the colour $\in$ {`black`, `white`}. Instead of issuing numbered tickets that grow without end, his algorithm issues a coloured ticket (i.e. a black or white ticket) with a number that can only grow up to n, where n is the number of processes or threads.

2.4.1 Algorithm Overview

Here is a summary of Taubenfeld's algorithm.

- Initialize the shared colour bit (`color`) to either `black` or `white`.
- When thread i starts the lock acquisition process, it is issued a coloured ticket (`mycolor[i]`) with a number (`number[i]`) as follows:
  - the ticket is assigned the same colour as that of the shared colour bit (i.e. `mycolor[i]` = `color`).
  - the ticket number (`number[i]`) is set to 1 + the count of tickets that have the same colour as thread i's ticket.
- Once thread i has a ticket, it waits until its ticket is the *lowest* before it can enter the critical section.
- The order of the coloured tickets is defined as follows:
  - If two tickets have different colours, the ticket whose colour is not the same as that of the shared colour bit (`color`) is deemed the *lowest*.
  - If two tickets have the same colour, the ticket with the smaller number is the *lowest*.
  - If tickets of two threads have the same colour and number, then the thread with the smaller local thread ID, is deemed the *lowest*.
  - A thread that is deemed the *lowest* has a higher priority, i.e., it is allowed to enter the critical section first.
- When thread i leaves its critical section, its first task should be to set the shared colour bit (`color`) to a colour that is not the same as its ticket (thread i's ticket). This way, thread i gives priority to waiting threads that hold tickets with the same colour as its ticket.
- Intuitively, if thread i's ticket is `black`, when it leaves the critical section, the shared colour bit will become `white`. This means that any new thread will be assigned a colour of `white`. All waiting threads (before the change of the shared colour bit) would have been assigned `black` tickets. According to the *first* rule of the order of tickets (defined above), if a waiting thread (`black` ticket) is compared to newly-arrived thread (`white` ticket), the waiting thread will be allowed to enter the critical

section first because its `black` ticket is different from the (now `white`) shared colour bit.

The pseudo-code of the algorithm is shown below.

Algorithm 1: The Black-White Bakery Algorithm

Shared variables:

`color`: a bit of type $\{\texttt{black}, \texttt{white}\}$
`flag`$[1..n]$: boolean array
`number`$[1..n]$: array of type $\{0, \ldots, n\}$
`mycolor`$[1..n]$: array of type $\{\texttt{black}, \texttt{white}\}$

Initialization:

$\forall i : 1 \leq i \leq n$: set `flag[i]` := `false` and `number[i]` := 0

```
1   flag[i] := true                                          // beginning of doorway
2   mycolor[i] := color
3   number[i] := 1 + max({number[j] | (1 ≤ j ≤ n) ∧ (mycolor[j] = mycolor[i])})
4   flag[i] := false                                         // end of doorway
5   for j = 1 to n do
6       while flag[j] do
7           // wait for ticket to be assigned
8       end while
        /* wait until all threads "ahead" of me are served */
9       if mycolor[j] = mycolor[i] then
10          while (number[j] ≠ 0) ∧ ((number[j], j) ≪ (number[i], i)) ∧
11                (mycolor[j] = mycolor[i]) do
12          end while
13      else
14          while (number[j] ≠ 0) ∧ (mycolor[i] = color) ∧
15                (mycolor[j] ≠ mycolor[i]) do
16          end while
17      end if
18  end for
    /* critical section */
19  if mycolor[i] = black then
20      color := white
21  else
22      color := black
23  end if
24  number[i] := 0
```

The flag in the Black-White Bakery algorithm plays a different role to that of the Bakery algorithm described in the course textbook. Here, the flag is used to indicate the start and end of a *doorway*, where a thread is assigned a ticket.

The article describing the Black-White Bakery algorithm can be found at this link: https://www.cs.tau.ac.il/~afek/gadi.pdf.

2.4.2 Implementation Instructions

In this task you have to implement the Black-White Bakery Lock algorithm as specified in the previous sub-section.

Use the same `Account.java`, `TThread.java` and `ThreadAccountDemo.java` files from Task 1 and add your own `BlackWhiteBakery.java`.

- The `BlackWhiteBakery` class constructor should accept as input a single integer argument to indicate the maximum number of threads.
- All functions required to ensure that the Black-White Bakery algorithm functions correctly will need to be implemented. You may implement additional helper functions as needed.
- Test your Black-White Bakery lock implementation on the shared `Account` class.
- Use the same testing conditions as in Task 2.

2.4.3 Expected Output

Your program should produce the following outputs.

- The Black-White Bakery lock algorithm has similar output messages as that of the Bakery lock algorithm.
- To analyze thread behaviour under the Black-White Bakery lock algorithm, a message should be displayed when a thread is assigned a ticket, when a priority assessment is conducted to determine which thread should enter the critical section (CS) first, and when a thread enters or leaves a critical section.
- The messages have different logging levels. Adhere to the following level specifications:

Level	**Message Type**
`FINE`	Thread ticket assignment
`FINER`	Thread enters CS
`FINER`	Thread leaves CS
`FINEST`	Assess thread priority

- The formats for the message logs are as follows:
  - When a thread is assigned a ticket, display:
    `[LOCK]: (id:0,number:1,color:Black) => ticket assigned`
    where 0 is the local thread ID, 1 and `Black` are the associated ticket number and colour.
  - When a thread is about to enter the critical section, display:
    `[LOCK]: (id:0,number:1,color:Black) => entering CS`
    where 0 is the local thread ID, 1 and `Black` are the associated ticket number and colour.

- When a thread leaves the critical section, display:
  `[LOCK]: (id:0,number:0,color:Black) => left CS, shared color:White`

  where 0 is the local thread ID, the second 0 is the updated ticket number, `Black` is the ticket colour, and `White` is the updated shared colour bit.
- Before entering the critical section, each thread runs a comparative analysis to determine whether it has the highest priority. A thread with the highest priority enters the critical section next. Display the following message:

`[LOCK]: assess priority => (id:0,number:1,color:Black) > (id:2,number:3,color:Black)`

  Here, thread 0 has a higher priority than thread 2.

- A sample output is shown below.

```
[LOCK]: (id:1,number:2,color:Black) => ticket assigned
[LOCK]: (id:2,number:3,color:Black) => ticket assigned
[LOCK]: (id:0,number:1,color:Black) => ticket assigned
[LOCK]: assess priority => (id:0,number:1,color:Black) > (id:1,number:2,color:Black)
[LOCK]: assess priority => (id:0,number:1,color:Black) > (id:2,number:3,color:Black)
[LOCK]: (id:0,number:1,color:Black) => entering CS
Thread-2 R800.00 withdrawn from account, R1460.00 remaining
[LOCK]: (id:0,number:0,color:Black) => left CS, shared color:White
...
```

2.4.4 Execution Notes

Some questions to consider while testing your code:

- Explore different thread conditions such as the number of threads, number of runs, etc. Take note of any thread behavioural changes to deepen your understanding of the Black-White Bakery lock algorithm.
- What are the lowest and highest numbers that can be assigned to a ticket? Why?

2.4.5 Upload Instructions

Compress the following files into a single archive.

- `BlackWhiteBakery.java`
- `Account.java`
- `TThread.java`

Details of the upload slot will be communicated later.

References

[1] Gadi Taubenfeld. "The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms", *International Symposium on Distributed Computing*, Springer, pp. 56–70, 2004. https://www.cs.tau.ac.il/~afek/gadi.pdf