Department of Computer Science
COS226 - Concurrent Systems

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Practical Assignment 4

- ☐ **Date Issued:** Tuesday, 8 September 2020
- ☐ **Date Due:** Sunday, 27 September 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	8
Task 2	7
Task 3	8
Task 4	7
Total	**30**

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

For this practical you will be implementing **Consensus Protocols** that make use of different mechanisms to achieve an agreement among competing threads. You will explore objects with differing consensus numbers: those with a consensus number of 2, and others with an infinite consensus number.

3.1 Task 1 - Decision Making Simulator (8 marks)

Alice and Bob always fight about how much time they should spend watching a specific show. Devise a **stack-based consensus protocol** that will help them decide who gets to choose how long they should watch the show for.

If you are not sure where to start, take a look at how consensus protocols are done in the book and slides, specifically at how a `Queue Consensus Protocol` is implemented, and consider how this can be adapted to work with a stack instead of a queue.

3.1.1 What to Implement

- `ConsensusProtocol`
  - This class is given to you. You will need to implement the functions given in this file. Use an appropriate data structure to hold the proposed values.
- `ConsensusThread`
  - You will need to create this class and implement the following functions:
    - `ConsensusThread(Consensus<Integer> consensusObject)`
    - `void run()`
- `StackConsensus`
  - You will need to create this class and implement the following functions:
    - `StackConsensus(int threadCount)`
    - `void propose(Integer value)`
    - `void decide()`

Note: These are just classes/functions that are needed to be implemented. You may implement any other classes and functions for your own use, if needed.

3.1.2 Implementation Notes

- In order to test your code you will need to simulate the above scenario. The consensus protocol has to work for n threads. In this scenario, n will be 2. The two threads represent Alice and Bob. Alice is associated with the local thread ID 0, and Bob with the local thread ID 1.
- Each thread (i.e. Alice and Bob) should propose an amount of time for watching a show. The proposed time should be random integer number between 0 (inclusive) and 50 (exclusive). This number has to be generated for each run.

- Each thread will need to do the following:
  - call `propose(...)` with the randomly generated number
  - sleep for t milliseconds
  - call `decide()` to reach a consensus on the amount of time that both Alice and Bob should watch a show for
- This sequence of events needs to happen x times. The `Main` class is responsible for simulating several runs of this scenario. Study the given `Main.java` file for implementation guidelines.
- The configurable scenario values can be specified through the following `ConsensusOptions` static variables:
  - `n : ConsensusOptions.threads`
  - `t : ConsensusOptions.delay`
  - `x : ConsensusOptions.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**: The value decided on has to be the same for all threads.

3.1.3 Expected Output

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

- When calling `propose()`, you need to output the following:

  ```
  [Name] wants to watch the program for [length] minutes
  ```

  (*Note: The above must be logged using the* ***INFO*** *level.*)
- When calling `decide()`, you need to output the following two statements:

  ```
  [Name] drew a [outcome] from the stack
  ```

  (*Note: The above must be logged using the* ***FINE*** *level.*)

  ```
  [Name] decided on [length] minutes
  ```

  (*Note: The above must be logged using the* ***INFO*** *level.*)

An example of one run of the program is shown below. As you can see in this run, Bob won and Alice lost:

```
Alice wants to watch the program for 23 minutes
Bob wants to watch the program for 31 minutes
Alice drew a LOSE from the stack
Alice decided on 31 minutes
Bob drew a WIN from the stack
Bob decided on 31 minutes
```

3.1.4 Execution Notes

Some questions to consider while testing your code:

- Run your code a couple of times with different experiment configurations (i.e. `ConsensusOptions` configurations). Can you see how a winner is decided each time around?
- Test your code with more than two threads and study why the protocol fails in such cases.
- What would be the effect of swapping around the items in the stack during its initialization?
- What would be the difference in the winning strategy if the LIFO stack was to be replaced with a FIFO queue? *Note:* you do not have to implement this comparison, do a mental comparison of the two data structures.

3.1.5 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 - Raffle Simulator: Stack Consensus (7 marks)

The local school is having a raffle to raise funds for a school trip. Each ticket bought has a random number associated with it, and the winner of the raffle is determined by randomly picking one of the raffle ticket numbers. Modify the stack consensus program used in the previous task to simulate the raffle. Note that consensus should be reached for **more than 2 threads**.

3.2.1 Implementation Notes

- The sequence of events are similar to those of Task 1. The only difference is that you need to implement a stack consensus protocol for more than 2 threads. You are only required to make changes to the `StackConsensus` and `Main` classes.
- *Hint:* You will need to use a `peek()` method.
- Each thread should get a ticket with a random number. The random number should should be an integer between 0 (inclusive) and 50 (exclusive). This number has to be generated for each run.
- You will need to add the function `void displayStackContents()` to the `StackConsensus` class. This function must print out a space-delimited list of the stack contents, from the bottom element to the top element. This function should only be called from the `Main` class, nowhere else.

3.2.2 Expected Output

A full example of the output is shown at the end of this section. Your output statements should be implemented in the `StackConsensus` class.

- When calling `propose()`, you need to output the following:

  ```
  Participant [ThreadID] buys a raffle ticket with random number [number]
  ```

  (*Note: The above must be logged using the* ***INFO*** *level.*)

- When calling `decide()`, you need to output the following:

  ```
  Participant [ThreadID] sees the winning raffle number is [number]
  ```

  (*Note: The above must be logged using the* ***INFO*** *level.*)

- After all the threads have finished executing, call the `displayStackContents()` method from the `Main` class. The method should output the following:

  ```
  Stack :  [values in the stack]
  ```

  (*Note: The above must be logged using the* ***FINE*** *level.*)

An example of one run of the program, as you can see there are 5 threads:

```
Participant 0 buys a raffle ticket with random number 29
Participant 2 buys a raffle ticket with random number 1
Participant 1 buys a raffle ticket with random number 30
Participant 4 buys a raffle ticket with random number 21
Participant 3 buys a raffle ticket with random number 34
Participant 0 sees the winning raffle number is 21
Participant 2 sees the winning raffle number is 21
Participant 1 sees the winning raffle number is 21
Participant 4 sees the winning raffle number is 21
Participant 3 sees the winning raffle number is 21
Stack:  1 2 0 3 4
```

3.2.3 Execution Notes

Some questions to consider while testing your code:

- What are the elements of the stack in this scenario? How does this differ from the *consensus-2* implementation?
- How is the winner decided in this protocol?
- How does the `peek()` method help transform this protocol to have an infinite consensus number?

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 - Raffle Simulator: Compare and Swap Consensus (8 marks)

3.3.1 Implementation Notes

- As in Task 2, this consensus protocol has to work with more than 2 threads, but this has to be implemented using the **compare-and-swap** protocol. The protocol makes use of the `compareAndSet()` operation to determine the winning thread (or raffle participant).
- Study Chapter 5 for implementation guidelines of the compare-and-swap protocol.
- You are only required to implement a `CompareAndSwapConsensus` class (you can make changes to and rename the `StackConsensus` class). You will also need to make changes to the `Main` class to test this protocol.

3.3.2 Expected Output

A full example of the output is shown at the end of this section. Your output statements should be implemented in the `CompareAndSwapConsensus` class.

- When calling `propose()`, you need to output the following:

  `Participant [ThreadID] buys a raffle ticket with random number [number]`

  (*Note: The above must be logged using the* ***INFO*** *level.*)

- When calling `decide()`, you will need to output two different statements as follows:

  `Participant [ThreadID] sees the winning raffle number is [number]`

  (*Note: The above must be logged using the* ***INFO*** *level.*)

  `Thread [ThreadID] - register:  [before value] => [after value]`

  (*Note: The* `before value` *is the register's value before a decision has been computed, and the* `after value` *is the register's value after consensus has been reached. The above output statement must be logged using the* ***FINE*** *level.*)

An example of one run of the program, as you can see there are 5 threads:

```
Participant 4 buys a raffle ticket with random number 26
Participant 1 buys a raffle ticket with random number 33
Participant 2 buys a raffle ticket with random number 25
Participant 0 buys a raffle ticket with random number 37
Participant 3 buys a raffle ticket with random number 47
Thread 4 - register:  -1 => 4
Participant 4 sees the winning raffle number is 26
Thread 1 - register:  4 => 4
```

```
Participant 1 sees the winning raffle number is 26
Thread 2 - register:  4 => 4
Thread 3 - register:  4 => 4
Thread 0 - register:  4 => 4
Participant 3 sees the winning raffle number is 26
Participant 2 sees the winning raffle number is 26
Participant 0 sees the winning raffle number is 26
```

3.3.3 Execution Notes

Some questions to consider while testing your code:

- What is the role of the register in this protocol? How does it achieve its purpose?
- Execute several runs of your program with different `ConsensusOptions` configurations. How is the winner decided each time?
- Set the `ConsensusOptions.delay` variable to 0 for one of your runs (do this in the `Main` class!). What do you notice? Can you give a reason for the observed behaviour?

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 - Raffle Simulator: Memory-to-Memory Swap (7 marks)

This task is a continuation of Tasks 2 and 3, where the raffle scenario is used to explore various consensus protocols for $n > 2$ threads. Task 4 concerns the development of a **memory-to-memory swap** protocol that has an infinite consensus number.

3.4.1 Implementation Notes

- Implement the memory-to-memory swap protocol in a newly created class called `MemorySwapConsensus` that extends the `ConsensusProtocol<Integer>` abstract class. You may use the contents of the `CompareAndSwapConsensus` class as your starting base.
- The execution events of the threads are the same as that of Task 2 and 3, with the exception of the `decide()` method.
- The memory-to-memory swap protocol is defined as follows:
  - The n threads **share** an array of registers $a[1 \,..\, n]$ whose elements are initialized to 0, and a single register r which is initialized to 1.
  - Each thread will propose a value as prescribed in previous tasks.
  - To reach consensus among competing threads, the first thread to **swap** the register value r of 1 into the shared register array a wins. For example: suppose that $n = 4$, if the thread with ID 1 enters the `decide()` function and executes the swapping event first, the resulting register contents will be: $a = [\,0\,1\,0\,0\,]$, $r = 0$. Participant 1's proposed value will thereafter be the winning raffle number.
  - *Hint:* You may need to implement your own `swap` function that atomically swaps the contents of the registers r and $a[i]$ for some thread i.
- Make changes to the `Main` class to test your `MemorySwapConsensus` protocol implementation under different experiment configurations.
- In addition to the `propose(...)` and `decide()` functions for your `MemorySwapConsensus` class, you will need to implement a public function with the signature: `void printRegisterContents()` whose objective is to print the contents of the shared register array a. You will need to call this function inside the `decide()` function after the consensus value has been determined. The format of the output is given in subsection 3.4.2 below.

3.4.2 Expected Output

A full example of the output is shown at the end of this section. Your output statements should be implemented in the `MemorySwapConsensus` class.

- When calling `propose()`, you need to output the following:

  `Participant [ThreadID] buys a raffle ticket with random number [number]`

  (*Note: The above must be logged using the* ***INFO*** *level.*)

- When calling `decide()`, you need to output the following:

  `Participant [ThreadID] sees the winning raffle number is [number]`

  (*Note: The above must be logged using the* ***INFO*** *level.*)

- When calling `printRegisterContents()`, you need to output the following:

  `Register @ Participant [ThreadID]: [array` *a* `elements]`

  (*Note: The above must be logged using the* ***FINE*** *level. The array elements should be* ***enclosed in square brackets****.*)

An example of one run of the program (with the number of threads set to 4):

```
Participant 0 buys a raffle ticket with random number 19
Participant 1 buys a raffle ticket with random number 37
Participant 2 buys a raffle ticket with random number 2
Participant 3 buys a raffle ticket with random number 22
Participant 1 sees the winning raffle number is 37
Register @ Participant 1:  [ 0 1 0 0 ]
Participant 3 sees the winning raffle number is 37
Participant 2 sees the winning raffle number is 37
Register @ Participant 3:  [ 0 1 0 0 ]
Register @ Participant 2:  [ 0 1 0 0 ]
Participant 0 sees the winning raffle number is 37
Register @ Participant 0:  [ 0 1 0 0 ]
```

3.4.3 Execution Notes

Some questions to consider while testing your code:

- Are the register array outputs consistent across all participants? If not, think about how the interleaving of threads could affect program execution? How can you resolve this problem?

- How can *atomicity* be implemented to ensure consistency in the decision value (consensus value)?

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 that 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.