**Operating Systems**

**Synchronization Project**

Table of Contents

[Project Description 3](#_Toc121051072)

[Objectives 3](#_Toc121051073)

[Task 1 Part A 3](#_Toc121051074)

[Task 2 Part A 5](#_Toc121051075)

[Methodology 7](#_Toc121051076)

[Synchronized Keyword 8](#_Toc121051077)

[Lock 8](#_Toc121051078)

[Peterson’s Solution 8](#_Toc121051079)

[Semaphore 9](#_Toc121051080)

[Task 3 Part B (Method Level Synchronization) 10](#_Toc121051081)

[Synchronized Keyword 10](#_Toc121051082)

[Lock Variable (Reentrant Lock) 11](#_Toc121051083)

[Peterson’s Solution 12](#_Toc121051084)

[Semaphore 15](#_Toc121051085)

[Task 4 Part B (Block Level Synchronization) 16](#_Toc121051086)

[Synchronized Keyword 16](#_Toc121051087)

[Lock Variable 17](#_Toc121051088)

[Semaphore 19](#_Toc121051089)

[Conclusion 20](#_Toc121051090)

[Appendix 21](#_Toc121051091)

[Peterson’s Solution (Method Level) 21](#_Toc121051092)

[Lock variable (Block Level) 27](#_Toc121051093)

[Lock variable (Method Level) 34](#_Toc121051094)

[Semaphores (Block Level) 40](#_Toc121051095)

[Semaphores (Method Level) 46](#_Toc121051096)

[Synchronized Keyword (Block Level) 53](#_Toc121051097)

[Synchronized Keyword (Method Level) 59](#_Toc121051098)

# 

# Project Description

Processes are programs currently in execution and loaded into memory and they are required to perform instructions for the system and the operating system. Threads are a unit of execution of a process. When accessing shared resources, threads and processes require synchronization to control access to the shared variable to preserve consistency. Otherwise, race condition will occur which happens if two or more threads try to access a shared resource or variable at the same time. The threads will read the same value from the shared variable and perform their operations based on the value. The thread that writes to the shared variable last, will have its value preserved while the others is overwritten. To eliminate the race condition, the threads must be synchronized using one of the synchronization techniques.

An account array, holding 10 accounts was created. Each account had 2 threads; a deposit thread for depositing and a withdraw thread for withdrawing, resulting in 20 total threads. When the threads start, they access the shared variable, balance, through deposit() and withdraw() at the same time, which causes race condition to occur, resulting in inconsistent account balances. To eliminate the race condition and preserve the consistency of the accounts, the team will use different block and method level synchronization techniques. These techniques include using the synchronized keyword solution, the lock variable solution, Peterson’s solution, and semaphore’s solution.

# Objectives

The main objective of the project is to analyse why and where the race condition occurs in the initially faulty code and solve it using various techniques of method level and block level synchronization to remove the race condition. Thus, achieving consistency of the account balances before and after execution.

# Task 1 Part A

For each account, we can see that the withdrawer thread is withdrawing 10 from the balance ten million times, and the depositor thread is depositing 10 to the balance ten million times. What is expected is for the accounts to have the same initial and final account balances, but as we can see from Figure 1 below, the initial account balance is different from the final account balance. This is due to the threads accessing a shared variable at the same time without mutual exclusion. Which causes race condition to occur. Resulting in the threads reading invalid values for balance and using it in the calculation giving us wrong final accounts balance.

To fix the error in the final accounts balance, the race condition must be eliminated using synchronization methods. Which provides the threads with exclusive access to the shared variable. Which allows the threads to read the correct value of balance, giving us the correct final account balances.

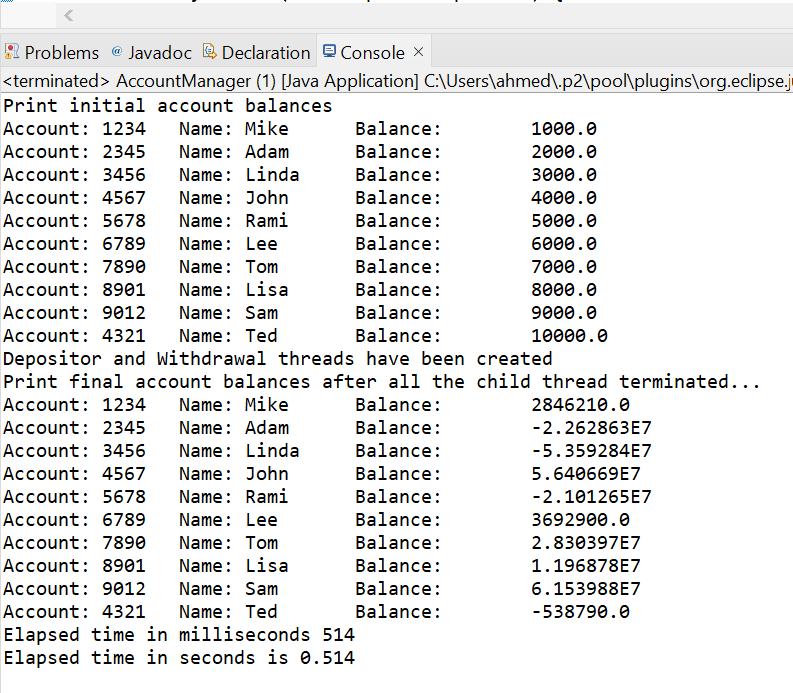


Figure 1: Initial and final account balances

In Figures 2 and 3, we can see the code where incorrect reading of the balance value may happen, due to the race condition.

Graphical user interface, text, application

Description automatically generated

Figure 2: Withdraw critical section

Graphical user interface, text, application

Description automatically generated

Figure 3: Deposit critical section

# Task 2 Part A

We can see in Figure 4 below that for each account, the deposit thread starts first, then immediately after, the withdraw thread starts. This is done for each account as the loop iterates from account0 to account9.

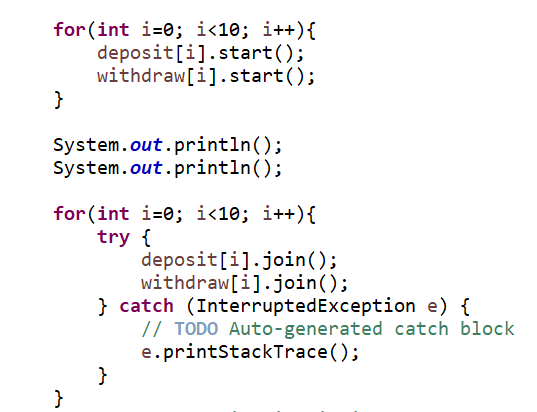


Figure 4: Start order of the threads

The lifetime of a thread included the following states: new, runnable, waiting, timed waiting, terminated. A newly created thread starts at the new state, where it waits in this state until the thread is started. A thread moves to the runnable state after the thread start() method is invoked, where the thread starts executing its task. A thread goes to the waiting state when it is waiting for another thread to perform a certain task, the thread goes back to the runnable state after it receives a signal that the other thread has performed its task. Timed waiting is a state where threads are waiting for a specified interval of time, the thread transitions back to the runnable state after the time interval ends or when an event happens. The final state in the lifetime of a thread is terminated, which happens when a thread has finished executing its task. In our program, we can see that a withdraw and a deposit threads are created and appended to each account and then wait until they are started, In the loop shown in Figure 4 above, we can see that the deposit and withdraw threads are invoking the start() method where they enter the runnable state. The threads then start executing and perform their tasks where they withdraw and deposit money to the accounts. After execution is done the threads are then terminated. In the second loop shown in Figure 4, we can see that the threads are calling the join() method, which makes the current thread stop executing until the thread which invoked the join() method is terminated.

Changing the start order of the threads will not fix the result. Because the threads are still accessing the critical region at the same time and are reading incorrect values of balance, causing the final balance output to be both incorrect and inconsistent. In Figure 5 the start order of the threads was changed to start from account9 threads to account0 threads, but the final balance of the accounts was still inconsistent. Additionally, in Figure 6 the withdraw thread was started before the deposit thread this time, but the final balance of the accounts remains inconsistent. We can see that the consistency of the accounts will not be preserved unless the race condition is eliminated, and the threads are synchronized.

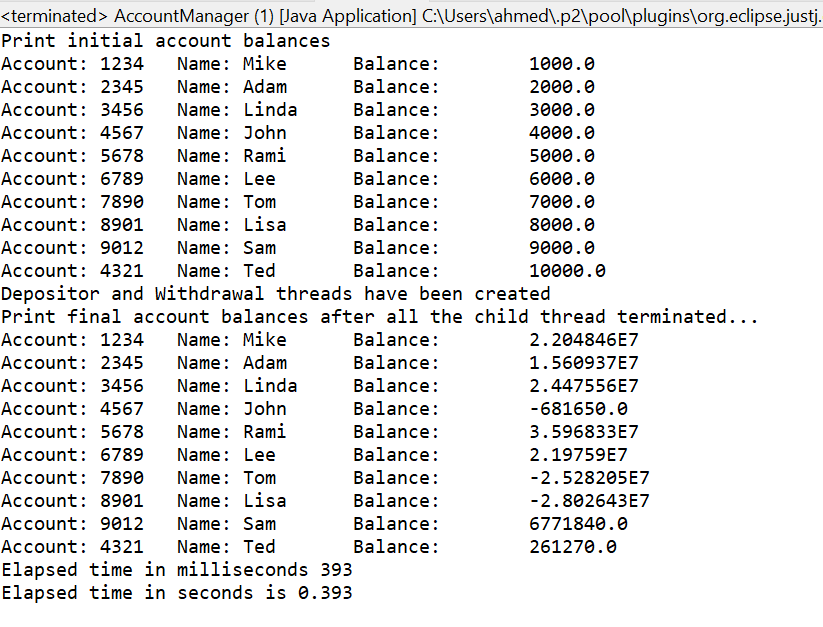


Figure 5: Final balance after starting from account 9 to account 0

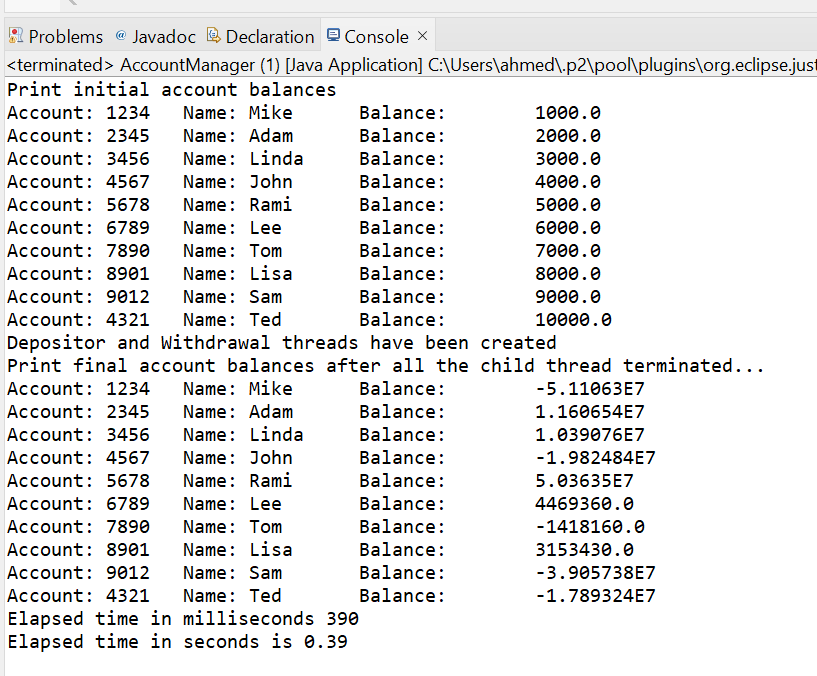


Figure 6: Starting the withdraw thread before deposit thread

# **Methodology**

This section will include a detailed breakdown of how we solved this issue using our various methods.

## Synchronized Keyword

This is a built-in Java keyword that prevents multiple threads to accessing the method at the same time. This will prevent the threads from accessing the critical region of the account balance. Before a thread can enter the critical region/method, it must acquire the lock. A thread without the lock cannot access the method. This works well as a solution to our bank account problem. The deposit and withdraw threads can run in synchronization, and mutual exclusion is satisfied. The only downside for using this method, is that only one thread can access the shared resource at a time.

## Lock

The lock variable solution works the same as the synchronized keyword solution described above. Instead of using the keyword however, Java as a built-in Lock class. This Lock class allowed for the creation of a lock variable object that will function as a lock with the ability to lock and unlock. As said before, a thread with the lock will be able to enter the critical region and perform its actions. Once this thread leaves, and the lock *unlocks* then another thread may enter. The locking is done by calling the lock() function on the lock object, and then it is unlocked by calling unlock().

This lock variable solution satisfies mutual exclusion and progress but may not always satisfy bounded waiting. If a thread in the critical region were to never unlock the lock, then no other thread could ever enter violating bounded waiting.

## Peterson’s Solution

Peterson’s solution is a mutual exclusion by busy waiting technique where two threads or processes are synchronized via alteration of a turn variable as well as a check of the willingness of the thread to execute its corresponding set of instructions. In this solution, a flag array which a size of 2 is stored with each index corresponding to one of the two process and a turn variable that indicates the process that is currently in execution. The flag array and turn variable were stored in the Account class as instantiation variables. Then, the deposit class/thread was assigned process 0, while the withdraw class/thread was assigned to process 1 as this must be defined prior to execution. Then the Peterson’s solutions template was adopted where the process loops indefinitely until the other process has executed and then it executes itself while the other process then takes on the waiting role. This wait process allows for synchronization and was improved by Thread.yield() which would hint to the scheduler it can release its current thread as another one is waiting but the scheduler does not have to listen, thus improving the functionality. Finally, once execution of process ends, the flag array is updated to indicate that the process no longer is willing to execute.

Due to it being mutual exclusion with busy waiting, it is slower than usual processes and wastes resources due to the busy waiting. By using sleep instead of busy wait, efficiency did improve. This form of block level synchronization was effective in solving the issue but was not synchronized in nature which was resulting in slightly lower speed due to the manual synchronization whereas the method level synchronization was preforming much faster.

## Semaphore

Semaphore is a technique used to synchronize threads or processes, where the number of threads accessing a resource is controlled and restricted. There are two types of semaphores, binary semaphores and counting semaphores. In binary semaphores, the value of the semaphore can be either 0 or 1, allowing only a single thread to access the resource. In counting semaphores, we can use a non-negative integer for the semaphore allowing only that number of threads to access the resource. Semaphores uses two atomic operations, called acquire (wait) and release (signal). The wait operation tests the semaphore, if the value is less than or equal to 0, then the thread gets blocked for a period of time and waits in a queue until one of the permits is released, to ensure that mutual exclusion is satisified. The queue will work on a first come first serve basis, where the first thread in the queue will enter if a permit is released. If the thread is blocking the queue, then it will be moved to become the last element in the queue. This helps in achieveing bounded waiting and progress. If the value of the semaphore is more than 0, then the semaphore variable is decremented by 1 and the thread enters the critical region where it can access the resource. The other operation is release, which is called when the thread is done executing in the critical region. The release operation increments the semaphore value by 1. Using these two operations, semaphores provide us with a solution to synchronization.

# Task 3 Part B (Method Level Synchronization)

## Synchronized Keyword

|  |  |
| --- | --- |
| One way to achieve synchronization by adding the keyword *synchronized* to the function implementation.  This keyword will synchronize the entire method. The entire method will become the critical region and only one thread can access it at a time. The thread inside the critical region will have the lock and prevent other threads from entering. Once thread exits, it will release the lock and another thread can enter.  The synchronized keyword is used for each method so it might seem like the deposit function can execute at the same time as the withdraw, however the internal lock for synchronized is shared between all threads, not just threads trying to access the same method. | Text  Description automatically generated |
| **Output and results**  We can see that the account balances are consistent after the synchronized keyword. |  |

## Lock Variable (Reentrant Lock)

|  |  |
| --- | --- |
| The method level implementation of the lock variable solution was done in the withdrawer and depositor class.  For each account object, the constructer will instantiate a lock object. |  |
| These classes have a for-loop of 10 million iterations calling the withdraw or deposit function for the respective accounts. To make this solution work on the method level, the lock object is locked before the call for withdraw or deposit and then unlocked after. This ensures that to access the deposit or withdraw functions the thread must have the lock.    As the call of the deposit or withdraw happens in between the lock and unlock, the entire method is in the critical region, and this is a method level implementation. | Graphical user interface, text, application  Description automatically generated  Text  Description automatically generated |
| **Output and results**  We can see that the account balances are consistent after the use of the lock variable. |  |

## Peterson’s Solution

|  |  |
| --- | --- |
| Peterson’s solution is an improvement from Strict Alteration where two processes take turns executing in the shared memory. The issue why it is not reliable and why we decided not to use it is that it doesn’t satisfy the progress condition as sometimes the currently executing process may have nothing to execute so the other process must wait until the process in the region ends its use of the region, and this can be indefinite! So, this is solved by using Peterson’s solution where the switch of processes can only be done if it is the turn of the process, and it has an execution pending and needs the region. | A picture containing graphical user interface  Description automatically generated |
| This was implemented by our project by:  Firstly, a turn variable was added to the account class that tracked which process was currently waiting to execute next. Deposit was set as process 0 and withdraw was set at process 1. We then added a flag array which indicated whether a process had an execution pending. If process 0, which is depositor, was waiting to execute, flag[0] would turn true and similarly withdraw with flag[1]. |  |
| Depositor was named process 0 and withdraw was set as process 1 | A picture containing text  Description automatically generated  A picture containing logo  Description automatically generated |
| Every time withdraw or deposit wanted to be performed, its flag would turn to true to inform the program that it has a usage for the critical region. The turn would be set to the other process as the other process is currently executing/using the region. A while loop was used to check if the other process had ended yet by either the process ending execution or having nothing to execute so its flag would be false. If the while loop continued to execute, the scheduler would be “hinted” to that it can release the currently executing process to ensure progress is not failed by any bugs. The scheduler does not have to listen to the hint, in case the currently executing process has not ended. Once the process waiting to be executed is allowed to enter by the region being freed, we sleep the thread so that it can get sufficient time to execute and not have any interferences. This is the start of the block level synchronization as you can see, the synchronization occurs after the method has already been run rather than the nature of the method being synchronized already. | Graphical user interface, text, application  Description automatically generated |
| Once it has ended its execution, it turns its flag back to false to indicate it has nothing to execute anymore. This is the end of the block level synchronization |  |
| **Output and results**  We can see that the account balances are consistent after using Peterson’s Solution for synchronized method. | Table  Description automatically generated |

## Semaphore

|  |  |
| --- | --- |
| Using binary semaphore for method level synchronization, we first initialize the value of the semaphore to 1, then pass it to the constructer for each account. |  |
| In method level synchronization, the method should be locked if a thread already entered the method. Therefore, we call the semaphore acquire method to test, if the semaphore value is less or equal to 0 then the thread is blocked and waits in a queue to be called later when a permit is released, if the value is more than 0, then the thread accesses the method, and the semaphore is decremented by 1. After the execution ends, the semaphore release method is called, which increments the value of the semaphore by 1. Allowing other threads to enter the method. This is done in both the Depositor and the Withdrawer classes. | Text  Description automatically generated |
| **Output and results**  We can see that the account balances are consistent after using semaphores for synchronized method. |  |

# Task 4 Part B (Block Level Synchronization)

## Synchronized Keyword

|  |  |
| --- | --- |
| Another way to achieve synchronization by adding the keyword *synchronized* as a function block in-line with the code.  This in-line function takes an object as the parameter such as the lock object passed in the code on the right. This implementation of the synchronized keyword works the same as its method level counterpart. The thread enters the critical region will take the lock, which will prevent other threads from entering. Once the thread leaves, only then can another thread take the lock again and enter.  The main difference is that instead of the entire method acting as the critical region, only the code encapsulated inside the function will be the critical region. | Graphical user interface, text, application  Description automatically generated |
| **Output and results**  We can see that the account balances are consistent after the use of the synchronized keyword variable. |  |

## Lock Variable

|  |  |
| --- | --- |
| The method level implementation of the lock variable solution was done in the withdrawer and depositor class.  For each account object, the constructer will instantiate a lock object. | Text, letter  Description automatically generated |
| Here the lock object is used in both methods, deposit and withdraw.  At the beginning of both methods, the lock object calls the lock() function for a thread that will enter this critical region. After that line, a *try* block is used and everything that is inside this block will be inside the critical region. At the end a *finally* is used to let the thread exit the critical region and call unlock() on the lock object.  This works as a block level implementation because not everything inside the method needs to be in this block. Any code outside of this critical region code, would not function synchronously. | Graphical user interface, text, application  Description automatically generated |
| **Output and results**  We can see that the account balances are consistent after the use of the lock variable. |  |

## Semaphore

|  |  |
| --- | --- |
| Using binary semaphore for block level synchronization, we first initialize the value of the semaphore to 1, then pass it to the constructer for each account. | Text  Description automatically generated with medium confidence |
| In block level synchronization, we synchronize a part of the code which becomes our synchronized block. In our binary semaphore implementation. Inside the deposit method and the withdraw method, we surround the block of code we want to synchronize, which is the operation on the balance in this case. The semaphore acquire method is called first to test, if the semaphore value is less or equal to 0 then the thread is blocked and waits in a queue to be called later when a permit is released, if the value is more than 0, then the thread accesses the block and gets access to the resource and the semaphore is decremented by 1. After the execution ends, the semaphore release method is called, which increments the value of the semaphore by 1. | Text  Description automatically generated  Text  Description automatically generated |
| **Output and results**  We can see that the account balances are consistent after using semaphores for synchronized block. | Table  Description automatically generated |

# Conclusion

The team employed 4 techniques to synchronize the bank account depositor and withdrawer threads. The synchronization keyword is a built-in Java technique and is the most common form of synchronization due to its simplicity. This procedure uses the lock variable system for blocking threads, meaning only the thread with the lock is allowed to enter the critical region and all other threads are suspended. This method satisfies all conditions with the only downside being that it can only have one thread in the shared region at a time. Moreover, the team also implemented a proper lock variable procedure from the Java ReentrantLock class. This works the same the synchronized keyword and will suspend all threads without the lock. The lock variable implementation worked well for our solution; it may not always satisfy the bounded-waiting condition. Additionally, the team implemented Semaphores for synchronization. Peterson’s Solution does not violate any condition, but the implementation is not efficient because of busy waiting. It also did not work on a block level implementation because the busy waiting would make it run for too long. Binary semaphores were used and satisfied all 3 conditions, mutual exclusion, progress, and bounded waiting successfully. Semaphore also provides the flexibility of allowing multiple threads to enter the critical region.