COMP 346 – Fall 2021
Assignment 2

Due date and time: Friday 5 November 2021, by midnight

Note:

1) You must submit the answers to all the questions below. However, only one or more questions, possibly chosen at random, will be corrected and will be evaluated to the full 50 marks.

2) Theory assignment to be **completed individually**.

Written Questions (50 marks):

Question # 1

What are the main differences between the user and kernel threads models? Which one of these models is likely to trash the system if used without any constraints?

Question # 2

Why are threads referred to as "light-weight" processes? What resources are used when a thread is created? How do they differ from those used when a process is created?

Question # 3

Does shared memory provide faster or slower interactions between user processes? Under what conditions is shared memory not suitable at all for inter-process communications?

Question # 4

a) Consider three concurrent processes A, B, and C, synchronized by three semaphores: *mutex*, *goB*, and *goC*, which are initialized to 1, 0 and 0 respectively:

```
Process A
---------
wait   (mutex)
...
signal (goB)
...
signal (mutex)
```

```
Process B
---------
wait   (mutex)
...
wait   (goB)
signal (goC)
...
signal (mutex)
```

```
Process C
---------
wait   (mutex)
...
wait   (goC)
...
signal (mutex)
```

Does there exist an execution scenario in which: (i) All three processes block permanently? (ii) Precisely two processes block permanently? (iii) No process blocks permanently? Justify your answers.

b) Now consider a slightly modified example involving two processes:

```
 Process A                    Process B
 ---------                    ---------
for (i = 0; i < m; i++) {     for (i = 0; i < n; i++) {
  wait (mutex);                wait (mutex);
  …                            …
  signal (goB);                 wait (goB);
  …                            …
  signal (mutex);               signal (mutex);
}                             }
```

(i) Let $m > n$. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

(ii) Now, let $m < n$. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

Question # 5

In a swapping/relocation system, the values assigned to the <base, limit> register pair prevent one user process from writing into the address space of another user process. However, these assignment operations are themselves privileged instructions that can only be executed in kernel mode.

Is it conceivable that some operating-system processes might have the entire main memory as their address space? If this is possible, is it necessarily a bad thing? Explain.

Question # 6

Sometimes it is necessary to synchronize two or more processes so that all process must finish their first phase before any of them is allowed to start its second phase.

For two processes, we might write:

```
semaphore s1 = 0, s2 = 0;

process P1 {              process P2 {
<phase I>                  <phase I>
V (s1)                     V (s2)
P (s2)                     P (s1)
<phase II>                <phase II>
        }                          }
```

a) Give a solution to the problem for three processes P1, P2, and P3.
b) Give the solution if the following rule is added: after all processes finish their first phase, phase I, they must execute phase II in order of their number; that is P1, then P2 and finally P3.

Question # 7

Generally, both P and V operation must be implemented as a critical section. Are there any cases when any of these two operations can safely be implemented as a non-critical section? If yes, demonstrate through an example when/how this can be done without creating any violations. If no, explain why these operations must always be implemented as critical sections.

Question # 8

What is the potential problem of multiprogramming?

Submission: Create a .zip file by name t*a2_studentID.zip* containing all the solutions, where ta2 is the number of the assignment and *studentID* is your student ID number. Upload the .zip file on moodle under *TA2.*

Programming assignment 2 (50 Marks)

Late Submission:	No late submission
Teams:	The assignment can be done individually or in teams of 2. Submit only one assignment per team.
Purpose:	The purpose of this assignment is to apply in practice the synchronization features of the Java programming language.

- **Specification.**

  In the first programming assignment, you have implemented the threads that allowed the operations of the client application and the server application to run concurrently. However, there were no critical section problem because only one server thread was updating the accounts and there was only one transaction on each account. In this programming assignment, the transaction file has been modified so that multiple transactions can be done on a single account. Therefore, if a thread blocks in a critical section while updating an account balance, then the result could be inconsistent if another thread also attempts another update operation on that account.

- **Problem.**

  The Java code provided is similar to that of PA1 but there have been some changes to adapt it to the requirements of PA2 as shown below. For this assignment, the server will use two concurrent threads to update the accounts and thus we may have inconsistent results in case the critical section is not well protected. In addition, the synchronization of the network buffers (i.e. ***inComingPacket***, ***outGoingPacket***) is using busy-waiting so you need now to block a thread when a buffer is full or empty.

  - **Changes in PA1.**

    - Use of static methods in class **Network** in order to call the methods using the class Name (i.e. **Network**) instead of using the instance variable **objNetwork** in classes **Client** and **Server**. Thus, the argument context in the constructor of the class **Network** is no more required.
    - Member variable ***serverThreadId*** in class **Server** identifies one of the two server threads. Member variables ***serverThreadRunningStatus1*** and ***serverThreadRunningStatus2*** indicate the current status of the two server threads. Also, the appropriate accessor and mutator methods have been added.
    - Member variables of the **Server** class that have shared values with the two receiving threads are now static.
    - A server thread is blocked in the ***deposit( )*** method before updating the 10^{th}, 20^{th}, …, 70^{th} account balance in order to simulate an inconsistency situation.
    - Transaction file now includes two transactions for accounts in index positions 9, 19, … 69 (i.e. 10^{th}, 20^{th}, …70^{th} accounts). Also, there are no transactions for accounts in positions 10, 20, …, 70.

- **Implementation.**

  This problem will be implemented in two phases, phase (i) will synchronize the access to the critical section for updating the accounts and phase (ii) will coordinate the threads when accessing a full or an empty network buffer.

  - Phase (i).

    - First, you must adapt the Java code provided to your solution of PA1. In case your PA1 code doesn't work properly, you will be provided help by the lab instructors.
    - Implement a second server thread in the ***main()*** method by respecting the changes already made in the constructor of the **Server** class. Consequently you need to modify the ***run()*** method of the **Server** class to accommodate the two threads and also to display the running time of each thread. The server can disconnect only when both threads have terminated.
    - Now execute the program with DEBUG flags and notice the accounts with inconsistent results as shown in the file *comp346-pa2-output-unsynchronized.txt*. The accounts 60520, 22310 and 91715 should be inconsistent but that may sometimes change depending on the sequence of execution. We have forced inconsistency by sleeping for 100 ms a thread accessing the critical section in the ***deposit()*** method of the **Server** class.
    - Next, using synchronized methods or synchronized statements, protect properly the critical section of the methods ***deposit()***, ***withdraw()*** and ***query()***. Execute the program again and there should be no inconsistent results. Explain your choice of using either synchronized methods or synchronized statements.

  - Phase (ii).

    - The network buffers (i.e. ***inComingPacket***, ***outGoingPacket***) are synchronized using busy-waiting by constantly yielding the CPU until an empty buffer or a full buffer is available. This is good for the performance of the thread as it can respond quickly to the event, but it is not good for the overall system performance as useful CPU cycles are wasted.
    - Using the methods ***acquire()*** (similar to ***wait()*** or ***P()***) and ***release()*** (similar to ***signal()*** or ***V()***) of the class **Semaphore**, synchronize the operations of the network input buffers. The semaphores must be implemented in the methods ***send()***, ***receive()***, ***transferrIn()*** and ***transferOut()*** of the class **Network**. Sample output for this phase is in the file *comp346-pa2-output-semaphores.txt*.
    - Execute the program and comment about the running times of the server threads compared to using busy-waiting in phase (i).

- **Sample output test cases.**

    - See attached text files.

- **Evaluation.**

    You will be evaluated mostly on the implementation of the required methods and the use of the synchronization tools.

    - Evaluation criteria

Criteria	Marks
Implementation of the server threads in the main method	5%
Implementation of the run() method in the class Server	10%
Implementation of the synchronized keyword	20%
Answer to a question during the demo	10%
Implementation of the semaphores	30%
In Phase(i), explain why you chose synchronized methods or synchronized statements to protect the critical section.	5%
Output test cases including running times	20%

- **Required documents.**

    - Source codes in Java.
    - Output test cases for phase (i) (unsynchronized and synchronized) and for phase (ii).
    - We have included DEBUG flags in the source code in order to help you trace the program but once your program works properly you should put the DEBUG flags in comments.

- **Submission.**

    - Create one zip file, containing the necessary files (.java, .txt and test cases). If the assignment is done individually, your file should be called p*a2_studentID*, where pa2 is the number of the assignment and *studentID* is your student ID number. If the work is done in a team of 2 people, the zip file should be called p*a2_studentID1_studentID2* where *studentID1* and *studentID2* are the student ID numbers of each student.
    - Upload your zip file on moodle under the *PA2*.

- **Notice.**

    - Note that this code has been tested on Windows using Eclipse. You may need to make changes if you would like to run on other OS.
    - You must not modify the original Java code provided nor change the size of the arrays but simply implement the required elements.