COMP 346 – Summer 2021
Assignment 1

Due date and time: Friday July 16, 2021 by midnight

Written Questions (50 marks):

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**.

Question # 1

I. What is an operating system? What are the main purposes of an operating system?

II. Define the essential properties of the following types of operating systems:
- Batch
- Time sharing
- Dedicated
- Real time
- Multiprogramming

III. Under what circumstances would a user be better of using a time-sharing system rather than a PC or single-user workstation?

Question # 2

Consider a computer system with a single-core processor. There are two processes to run in the system: P_1 and P_2. Process P_1 has a life cycle as follows: CPU burst time of 15 units, followed by I/O burst time of minimum 10 units, followed by CPU burst time of 10 units. Process P_2 has the following life cycle: CPU burst time of 10 units, followed by I/O burst time of minimum 5 units, followed by CPU burst time of 15 units. Now answer the following questions:

a) Considering a *single programmed* operating system, what is the minimal total time required to complete executions of the two processes? You should explain your answer with a diagram.

b) Now considering a *multiprogrammed* operating system, what is the minimal total time required to complete executions of the two processes? You should explain your answer with a diagram.

c) *Throughput* is defined as the number of processes (tasks) completed per unit time. Following this definition, calculate the throughputs for parts

a) and b) above. How does multiprogramming affect throughput? Explain your answer.

Question # 3

I. What is the performance advantage in having device drivers and devices synchronized by means of device interrupts, rather than by polling (i.e., device driver keeps on polling the device to see if a specific event has occurred)? Under what circumstances can polling be advantageous over interrupts?

II. Is it possible to use a DMA controller if the system does not support interrupts? Explain why.

III. The procedure *ContextSwitch* is called whenever there is a switch in context from a running program A to another program B. The procedure is a straightforward assembly language routine that saves and restores registers and must be atomic. Something disastrous can happen if the routine *ContextSwitch* is not atomic.

(a) Explain why *ContextSwitch* must be atomic, possibly with an example.
(b) Explain how the atomicity can be achieved in practice.

Question # 4

I. If a user program needs to perform I/O, it needs to trap the OS via a system call that transfers control to the kernel. The kernel performs I/O on behalf of the user program. However, systems calls have added overheads, which can slow down the entire system. In that case, why not let user processes perform I/O directly, without going through the kernel?

II. Consider a computer running in the user mode. It will switch to the monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt vector.

(a) A smart, but malicious, user took advantage of a certain serious loophole in the computer's protection mechanism, by which he could make run his own user program in the monitor mode! This can cause disastrous effects. What could have he possibly done to achieve this? What disastrous effects could it cause?
(b) Suggest a remedy for the loophole.

Question # 5

Suppose that a multiprogrammed system has a load of N processes with individual execution times of t_1, t_2, …, t_N. Answer the following questions:

a) How would it be possible that the time to complete the N processes could be as small as:
maximum (t_1, t_2, …, t_N)?

b) How would it be possible that the total execution time, $T > t_1 + t_2 + \ldots + t_N$? In other words, what would cause the total execution time to exceed the sum of individual process execution times?

Question # 6

Which of the following instructions should be privileged? Explain why.

(i) Read the system clock,
(ii) Clear memory,
(iii) Reading from user space
(iv) Writing to user space
(v) Copy from one register to another
(vi) Turn off interrupts, and
(vii) Switch from user to monitor mode.

Question # 7

Assume you are given the responsibility to design two OS systems, a Network Operating System and a Distributed Operating System. Indicate the primary differences between these two systems. Additionally, you need to indicate if there are any possible common routines between these systems? If yes, indicate some of these routines. If no, explain why common routines between these two particular systems do not make sense.

Submission: Create a .zip file by name t*a1_studentID.zip* containing all the solutions, where ta1 is the number of the assignment and *studentID* is your student ID number. Upload the .zip file on moodle under *TA1.*

Programming assignment 1 (50 marks)

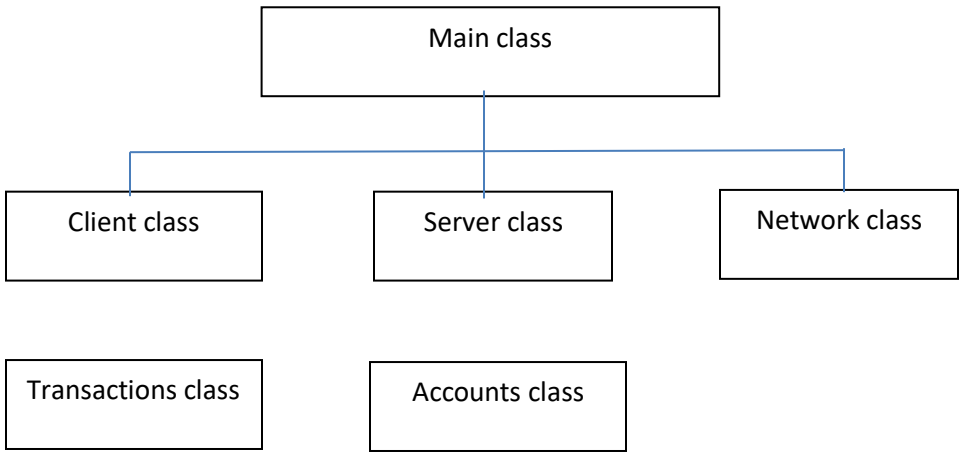
Late Submission:	No late submission
Teams:	The assignment can be done individually or in teams of 2. Submit only one assignment per team.
Purpose:	The purpose of this assignment is to apply in practice the multi-threading features of the Java programming language.

- **Problem specification.**

  In a client-server system, the client application sends requests to a server application through a network connection. In such system, the user interface is implemented in the client and the database is stored in the server.
  You are required to implement a client-server application to process banking transactions such as withdrawals and deposits. In modern banking systems, a costumer accesses a bank account using an access card at an ATM, at the counter or on the web.

- **Implementation.**

  The following diagram illustrates the classes for the client-server banking application.



- **Network class**:

  The **Network** class provides the infrastructure to allow the client and the server to process the transactions. The client and the server need to be connected (using ***connect()***) to the network prior to an exchange. The Network class also implements an input buffer (***inComingPacket[]***) and an output buffer (***outGoingPacket[]***) to respectively receive transactions from the client and to return updated transactions to the client. The capacity of these buffers is 10 elements, so the network indicates whether they are full or empty.

- **Client class**:

  The **Client** class reads all the transactions from a file (*transaction.txt*) and saves them in an array (***transaction[]***). A transaction is implemented by the **Transactions** class.
  Using the ***send()*** method of **Network** class the client transfers the transactions to the network input buffer and it yields the cpu in case the network input buffer is full.
  Also, using the ***receive()*** method of **Network** class the client retrieves the updated transactions from the network output buffer and yields the cpu in case the buffer is empty. Each updated transaction received is displayed immediately on the console.

- **Server class:**

  The **Server** class reads all the accounts from a file (*account.txt*) and saves them in an array (***account[]***). An account is implemented by the **Accounts** class.
  Using the ***transferrIn()*** method of **Network** class the server retrieves the transactions from the network input buffer and perform the operations (withdraw, deposit, query) on the specific accounts. It yields the cpu in case the buffer is empty.
  Each updated transaction is transmitted to the network output buffer using the ***transferOut( )*** method of **Network** class and the server yields the cpu in case the buffer is full.

- **Problems:**

  - You need to complete the Java program that is provided by implementing 4 threads so that the client, the server and the network all run concurrently. The client has 2 threads, one for sending the transactions and another for receiving the completed transactions.
    In case the input and output network buffers are full or empty each client or server thread must yield the cpu using the Java method ***Thread.yield()***. The network thread executes an infinite loop that ends when both client and server threads have disconnected. In case the client or sever threads are still connected the network thread must continuously yield the cpu.

- You must record the running times of both client threads and the server thread using the Java method ***System.currentTimeMillis()***.

- You need to provide output test cases with the appropriate running times for the client and the server threads. Perform 3 different runs of the program and explain why there is a difference in the running times.

- **Sample output test cases:**

  - See attached text files.

- **Evaluation:**

  You will be evaluated mostly on the implementation of the required methods, the implementation of the threads, the measurements of the running times and the voluntary sharing of the cpu by the threads.

  - Evaluation criteria

Criteria	Marks
Implementation of the 4 threads	35%
Implementation of the main class	15%
Answer to a question during the demo	10%
Implementation of the yield( ) method	10%
Implementation of the measurements of the running times	10%
Output test cases including running times	20%

- **Required documents:**

  - Source codes in Java.
  - Output test cases.
  - 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*a1_studentID*, where pa1 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*a1_studentID1_studentID2* where *studentID1* and *studentID2* are the student ID numbers of each student.
  - Upload your zip file on moodle under the *PA1*.

- **Disclaimer:**
  - Note that the provided code has been tested on Windows. You may need to make changes if you would like to run it on other OS.

- Detailed implementation of the classes

  - Client class

Client
- numberOfTransactions : int /* total number of transactions to process */ - maxNbTransactions : int /* maximum number of transactions */ - transactions : Transactions[ ] /* transaction array */ - clientOperation : String /* sending, receiving */ - objNetwork : Network /* handle to access network methods */
Client(String operation) +getNumberOfTransactions( ) : int +getClientOperation() : String +setNumberOfTransactions(int nbOfTrans) : void +setClientOperation(String operation) : void +readTransactions( ) : void +sendTransactions( ) : void +receiveTransactions(Transactions transact) : void +toString( ) : String +run() : void

  - Transactions class

Transactions
- accountNumber : String /* account number */ - operationType : String /* deposit, withdrawal, query */ - transactionAmount : double /* transaction amount */ - transactionBalance : double /* updated account balance */ - transactionError : String /* NSF, invalid amount or account, none */ - transactionStatus : String /* pending, sent, received, done */
Transactions( ) + getTransactionType( ) : String + getAccountNumber( ) : String + getTransactionAmount( ) : double + getTransactionBalance( ) : double + getTransactionError( ) : String + getTransactionStatus( ) : String + setAccountNumber(String accNumber) : void + setTransactionType(String opType) : void + setTransactionAmount(double transAmount) : void + setTransactionBalance(double transBalance) : void + setTransactionError(String transError) : void + setTransactionStatus(String transStatus) : void + toString( ) : String

- Server class

Server
- numberOfTransactions : int /* total number of transactions received */ - numberOfAccounts : int /* total number of accounts saved */ - maxNbAccounts : int /* maximum number of accounts */ - transaction : Transactions /* a transaction to process */ - objNetwork : Network / * handle to access network methods */ - account : Accounts[] /* account array */
Server( ) +getNumberOfTransactions( ) : int +getNumberOfAccounts( ) : int +getMaxNbAccounts() : int +setNumberOfTransactions(int nbOfTrans) : void +setNumberOfAccounts(int nbOfAcc) : void +setMaxNbAccounts(int nbOfAcc) :void +initializeAccounts( ) : void +findAccount(String accNumber) : int +processTransactions(Transaction trans) : boolean +deposit(int i, double amount) : double +withdraw(int i, double amount) : double +query(int i) : double +toString( ) : String +run() : void

- Accounts class

Accounts
- accountNumber : String /* unique account number */ - accountType : String /* chequing, saving, credit */ - firstName : String /* first name of account holder */ - lastName : String /* last name of account holder */ - balance : double /* account balance */
+ getAccountNumber( ) : String + getAccountType( ) : String + getFirstName( ) : String + getLastname( ) : String + getBalance( ) : double + setAccountNumber(double accNumber) : void + setAccountType(String accType) : void + setFirstName(String fName) : void + setLastname(String lName) : void + setBalance(double bal) : void + toString( ) : String

- Network class

Network
- clientIP : string /* IP of client application */ - serverIP : string /* IP of server application */ - portID : int /* port ID of client application */ - clientConnectionStatus : String /* connected, disconnected */ - serverConnectionStatus : String /* connected, disconnected */ - maxNbPackets : int /* capacity of network buffers */ - inComingPacket : Transactions[10] /* network input buffer */ - outGoingPacket : Transactions[10] /* network output buffer */ - inBufferStatus, outBufferStatus : String /* normal, full, empty */ - inputIndexClient, inputIndexServer, outputIndexServer, outputIndexClient : int /* buffer index position */ - networkStatus : String /* active, inactive */
Network(String context) + getClientIP( ) : String + getServerIP( ) : String + getPortID( ) : integer + getClientConnectionStatus( ) : String + getServerConnectionStatus( ) : String + getInBufferStatus( ) : string + getOutBufferStatus( ) : string + getNetworkStatus( ) : String + getInputIndexClient( ) : int + getInputIndexServer( ) : int + getIOutputIndexClient( ) : int + getOutputIndexServer( ) : int + setClientIP(String cip) : void + setServerIP(String sip) : void + setPortID(int pid) : void + setClientConnectionStatus(String connectStatus) : void + setServerConnectionStatus(String connectStatus) : void + setNetworkStatus(String netStatus) : void + setInBufferStatus(String inBufStatus) : void + setOutBufferStatus(String outBufStatus) : void + setInputIndexClient(int i1) : void + setInputIndexServer(int i2) : void + setOutputIndexClient(int o2) : void + setOutputIndexServer(int o1) : void + connect(String IP) : boolean + disconnect(String IP) : boolean + send(Transactions inPacket) : boolean + receive(Transactions outPacket) : boolean + transferOut(Transactions outPacket) : boolean + transferIn(Transactions inPacket) : boolean + toString( ) : String +run() : void