Operating Systems Project 2

Alejandro Lopez

Nova Southeastern University

Table of Contents

[Overview 3: What is this program?](#_Overview:)

[Requirements 3: Requirements met](#_Requirements)

[Software Design 4: How did I design each part of the Software](#_Software_Design:)

* Semaphores 4 – Critical Section
* Thread Design 4
* Tuition and Student Loan Threads 5
* Producer Thread 6
* Consumer Thread 7
* Account Balance 8
* Credit Card System 8
* Months and Semesters System 9

[Classes & Methods 9: Logic behind the parts of the code](#_Classes(Main_bullet_points))

[Pseudo Code 14: Code structure](#_Pseudo_Code:_1)

[Default Values for Testing 23: Default values to test the program](#_Default_Values_for)

[User Manual 24: How to use the program](#_User_Manual)

# **Overview:**

A software that consists of synchronization and mutual exclusion for multiple threads accessing one bank account using semaphores.

# **Requirements**

Automated bank account: 

Producer threads add funds to bank account: 

Consumer threads debit the bank account: 

Credit card that can be debited up to a certain amount: 

One year period of college: 

3 Academic Terms: 

Financial Aid Received each semester: 

Tuition withdrawn each semester: 

Housing and Food withdrawn each month: 

Semaphores that coordinate the threads: 

No over-draft or closing of the account: 

Producer and Consumer threads run separate: 

Included print statements: 

# **Software Design:**

* Semaphores: The semaphore logic is inside the SemaphoreBank.java file. There are two types of semaphores in this project. There is a consumer thread semaphore and a producer thread semaphore. Each time a consumer performs an action like withdrawing from the account, the consumer thread is acquired. The consumer thread performs its actions based on which method was called and once the method is finished, the producer thread gets released. This is due to the structure of this program which executes a consumer thread first, then a producer thread. A consumer thread cannot execute right after another, and a producer thread must follow right after that consumer thread. The same applies for when a producer executes a method like deposit. The producer thread acquires the producer thread semaphore, performs the actions of depositing to the account, and then the consumer thread is released. Each semaphore has a speed for which to run the thread and this speed is defined through each individual thread but the value is given inside RunAccount.java which is the main class.
* Thread Design: Each thread has a speed, a max amount, a bank account instantiation, a thread name, and a count variable. These variables make up the structure of the threads. The speed variable constitutes the speed at which each semaphore will run each thread. The max amount variable places a restriction on the max amount that can be either deposited or withdrawn. These values are determined at random but are limited by the max amount variable. The bank account instantiation allows the thread to use the methods inside the SemaphoreBank.java which are initialized inside the java interface called Bank.java. The threads will execute which ever methods were called through the account instantiation through a try catch. In the try the thread is executed and in the catch there is an exception that signals the pool of threads to shut down that is called printStackTrace. An important aspect of this design is that each thread must have the same amount of account instantiations inside them and there must be the same amount of consumer threads as producer threads. This due to the design choice of having a consumer thread execute right after a producer thread while a consumer thread cannot execute right after another consumer thread. If this design implementation is broken, then the software will not work. The number of producers and consumers is decided at RunAccount.java which is the main class. As an example, if there are 2 consumers which must have 2 producer threads to balance the operations then there will be a consumer 1 and a consumer 2. Consumer 1 will use the same thread each time they deposit/Job and will be identified through the thread name variable which is just the ID to recognize a thread. An example of an execution would be: Producer 1 deposits $56 or Job 1 deposits $200 which both come from the Producer thread #1.
* Tuition and Student Loan Threads: Tuition is a consumer thread while Student Loan is a producer thread. I created these separately because since the requirements ask for multiple threads withdrawing and depositing from the account then there would be multiple instances of the consumer and producer threads such as Bill 1 withdraws $200 and Bill 2 withdraws $200. If there are 2 consumer threads and the account instantiation of Tuition is inside the consumer thread then 2 instances of Tuition would be created due to the design of my software. It would look like Tuition 1 withdraws $1500 and Tuition 2 withdraws $1500 from the account. This would not be accurate since the design requirements asks for a single tuition. This would work if there would for example be 2 people that used the account, each with their own university which would mean 2 tuitions and 2 student loans to make up for the tuition. However, the design requirements imply that there must only be 1 university for the account.
* Producer Thread: This thread has two different types and those are the producer thread itself defined in Producer.java and StudentLoan.java. I am describing both together because the producer threads always deposit or contribute to the account. A producer thread is made up of the following:
  + Job: A Job method that is called through a Producer Thread and this method is defined through SemaphoreBank.java which generates deposits to the account with the value of $200 emulating a job that the consumer might have.
  + Deposit: A Deposit method that is called through a Producer Thread and this method is defined through SemaphoreBank.java which generates random value deposits limited through the variable max deposits. This method emulates random deposits that the consumer might make during the months.
  + Student Loan (Student Loan Thread): Student Loan method that is called through a Student Loan thread and this method is defined through SemaphoreBank.java which generates a deposit to the account every time a new semester starts. The Student Loan value of $1500 is a fixed value that matches the Tuition value.
  + Pay Credit (Student Loan Thread): Pay Credit method that is called through a Student Loan thread and this method is defined through SemaphoreBank.java which generates the payment for the credit owed which happens each month. The credit owed is only paid in 20% as opposed to the full amount. This procedure is explained in the Credit Card System section.
* Consumer Thread: This thread has two different types and those are the consumer thread itself defined in Consumer.java and Tuition.java. I am describing both together because the consumer threads always withdraw from the account. A consumer thread is made up of the following:
  + Bills: A bill method that is called through a Consumer Thread and this method is defined through SemaphoreBank.java which generates withdrawals from the account with the value of $200 emulating monthly bills that must be paid.
  + Withdraw: A withdraw method that is called through a Consumer Thread and this method is defined through SemaphoreBank.java which generates random value withdraws limited through the variable max withdrawals. This method emulates random withdrawals that the consumer might make during the months.
  + Tuition (Tuition Thread): Tuition method that is called through a Tuition thread and this method is defined through SemaphoreBank.java which generates a withdrawal from the account every time a new semester starts. The Tuition value of $1500 is a fixed value that matches the Student Loan value.
  + Housing and Food (Tuition Thread): Housing method that is called through a Tuition thread and this method is defined through SemaphoreBank.java which generates a withdrawal from the account at the beginning of every month with a set value of $200. This method emulates the housing and food costs of living in a college dorm.
* Account balance: The account balance is measured through a private integer variable inside the SemaphoreBank.java. This variable gets modified according to which method is called through either a consumer thread or a producer thread. If withdraw is called through a producer thread then this variable will decrease if the method withdraws succeeded executing, else it will have the same value. As for the deposit method which is called through a consumer thread, it has no restrictions from executing and therefore this method will always add the account balance variable. This same logic applies for the rest of the producer and consumer threads except for the Pay Credit method which is a producer thread.
* Credit Card System: The logic for the credit card system is defined inside SemaphoreBank.java. This system consists of 2 private integer variables which are credit balance and credit debt. Credit balance is the amount that the credit card can be debited up to and credit debt is the variable that holds the value that the consumer owes after using the credit card. This system is called through both the consumer and producer threads. Using withdraw again as an example to represent producer threads, if there is no account balance to withdraw from the account then the consumer will use the credit card to withdraw from the account. This withdraw method will increase the credit debt variable by the amount withdrawn and it will decrease the credit balance by the amount withdrawn. As for paying the credit card debt, this is called by the consumer thread because this is a method that is contributing to the account and is reducing the amount owed. This is done through a consumer thread method called Pay Credit that only pays 20% of the credit debt and not in full. The decision to only pay 20% of the credit card was because there are many transactions happening which keeps the account balance at a relatively low amount, therefore, paying the balance in full would make it so that the consumer can never pay the with credit card. As for the functionality, if there are no funds to pay the credit card debt then block the payment, else pay 20% of the debt using the account balance.
* Months and Semesters System: In the main class called RunAccount.java there is a for loop that runs n months times decided through the months variable instantiated at the top of the main class. Inside this loop each time it executes then the counter variable is used to print out the current month that is being executed. Inside this for loop there is an if statement that determines if count is 1, 4, or 8 then mark the beginning of a new semester. Furthermore, inside this for loop there is a try catch which executes each type of threads in the try and inside the catch it shuts down the app once the pool of threads are executed once. The loop then initiates the pool of threads one more time once the loop restarts.

# **Classes(Main bullet points) & Methods(Indented):**

* RunAccount.java: This class instantiates all the variables that are used by the threads and those are the number of months that the bank account will run for, the number of universities in the account, the number of consumers, the number of producers, the withdrawal speed, the maximal withdrawal amount, the deposit speed, and the maximum deposit amount. In this class the Bank account is instantiated so that each thread can execute their given methods through the semaphore bank. Then there is a for loop that loops with the number of months given and inside each iteration the threads are executed through a try catch which has an exception to shut down the app if the threads signal that they are done. Each time the loop runs, a new thread pool is initiated.
* Bank.java: An interface file that contains all the methods that each thread calls and their logic if written inside SemaphoreBank.java.
* SemaphoreBank.java: This class contains the semaphores that allow the synchronization and multithreading in the application as well as the logic for the methods that each thread uses, the account balance, the credit balance, the credit debt, and the format for which the data is displayed on the command line.
  + Withdraw: Acquires a consumer thread semaphore and then executes if there is enough balance to withdraw from the account. This method attains the random value attained through the consumer thread and subtracts it from the account balance to then print the transaction on the command line and release a Producer Thread. If there is not enough balance to subtract from the account then display on the command line that there are insufficient funds and then use the credit only if there is enough credit balance to credit the transaction. If there is not enough credit balance then block the transaction. If there is sufficient credit then subtract the value from the withdrawal from the credit balance and add the withdrawal to the credit debt to finally print the transaction on the command line.
  + Deposit: Acquires a producer thread semaphore and then executes. This method attains the random value attained through the Producer thread and adds it to the account balance to then print the transaction on the command line and release a Consumer Thread semaphore.
  + Student Loan: Acquires a producer thread semaphore and then executes. This method attains the $ attained through the Student Loan thread and adds it to the $1500 account balance to then print the transaction on the command line and release a Consumer Thread semaphore. The difference in this one is that it only executes every time a new semester starts. There is an if statement that uses the count on the main class to mark when it equals 1, 4, or 8 which is when a new semester starts and the student loan method executes in those cases.
  + Tuition: Acquires a consumer thread semaphore and then executes if there is enough balance to withdraw Tuition from the account. This method attains the $1500 attained through the consumer thread and subtracts it from the account balance to then print the transaction on the command line and release a Producer Thread. If there is not enough balance to subtract the Tuition from the account then display on the command line that there are insufficient funds and then use the credit only if there is enough credit balance to credit the transaction. If there is not enough credit balance then block the transaction. If there is sufficient credit then subtract the value from the withdrawal from the credit balance and add the withdrawal to the credit debt to finally print the transaction on the command line. The difference in this one is that it only executes every time a new semester starts. There is an if statement that uses the count on the main class to mark when it equals 1, 4, or 8 which is when a new semester starts and the Tuition method executes in those cases.
  + Housing and Food: Acquires a consumer thread semaphore and then executes if there is enough balance to withdraw the Housing and Food from the account. This method attains the $200 attained through the consumer thread and subtracts it from the account balance to then print the transaction on the command line and release a Producer Thread. If there is not enough balance to subtract from the account then display on the command line that there are insufficient funds and then use the credit only if there is enough credit balance to credit the transaction. If there is not enough credit balance then block the transaction. If there is sufficient credit then subtract the value from the Housing and Food charge from the credit balance and add the withdrawal to the credit debt to finally print the transaction on the command line.
  + Job: Acquires a producer thread semaphore and then executes. This method attains the $200 attained through the Producer thread and adds it to the account balance to then print the transaction on the command line and release a Consumer Thread semaphore.
  + Bill: Acquires a consumer thread semaphore and then executes if there is enough balance to withdraw the Bill from the account. This method attains the $200 attained through the consumer thread and subtracts it from the account balance to then print the transaction on the command line and release a Producer Thread. If there is not enough balance to subtract from the account then display on the command line that there are insufficient funds and then use the credit only if there is enough credit balance to credit the transaction. If there is not enough credit balance then block the transaction. If there is sufficient credit then subtract the value from the withdrawal from the credit balance and add the withdrawal to the credit debt to finally print the transaction on the command line.
  + Pay Credit: This method pays 20% of the credit debt if there is one every single month. This is done by acquiring 20% of the credit debt and storing it inside a variable and then grabbing the difference between the credit debt and the 20% of the debt and storing it inside a variable. Then a producer thread is acquired to then check if there are enough funds to pay the credit debt. If there is not enough balance then the payment is blocked. If there is no credit to be paid then nothing is done. If there is a credit to be paid and balance to pay it then subtract the 20% of the credit debt from the account balance and then subtract the twenty percent form the credit debt. Finally, print the amount of the credit debt that was paid.
  + Print State: Instead of using a System.out.println I constructed a way to print the transactions a way that they would be divided by consumer, producer, and balance. His is done through tabs and a formatting structure.
* Consumer.java, Producer.java, Tuition.java, StudentLoan.java: These all have the same structure so I will discuss them together. Each thread contains a speed to wait for the semaphores, a max amount for the random value generated to either subtract or add to the account balance, the instantiation of the bank account to be able to call all the thread methods, the thread name to identify each thread, and the count to check where in the year the thread is running. Once the constructor for the thread is defined, there is a method called run which has a try catch. Inside the try the thread waits for the semaphore and after that the thread executes its methods which may be deposit if it is for producer or withdraw if for consumer. Finally, in the catch the exception sends a signal to the main class with the ex.printStackTrace that the thread finished running.

# **Pseudo Code:**

**SemaphoreBank.java**

* PrintState()
  + PRINT each item with a TAB(\t) -> deposit string, withdraw string, balance
* Withdraw()
  + ACQUIRE Consumer Thread
  + IF (There is not enough money)
    - CALL PrintState()
      * PRINT Insufficient Funds
    - IF (There is not enough Credit balance to do the transaction)
      * CALL PrintState()
        + PRINT Insufficient Credit
    - ELSE
      * Use Credit for Transaction
      * CALL PrintState()
        + PRINT Credited Transaction , Credit Owed and Credit Left
      * RELEASE Producer Thread
  + ELSE
    - Withdraw from account balance
    - CALL PrintState()
      * PRINT Withdrawal and Balance left
    - RELEASE Producer Thread
* Housing and Food()
  + ACQUIRE Consumer Thread
  + IF (There is not enough money)
    - CALL PrintState()
      * PRINT Insufficient Funds
    - IF (There is not enough Credit balance to do the transaction)
      * CALL PrintState()
        + PRINT Insufficient Credit
    - ELSE
      * Use Credit for Transaction
      * CALL PrintState()
        + PRINT Credited Transaction , Credit Owed and Credit Left
      * RELEASE Producer Thread
  + ELSE
    - Withdraw from account balance
    - CALL PrintState()
      * PRINT Withdrawal and Balance left
    - RELEASE Producer Thread
* Tuition()
  + IF((Count IS 1) OR (Count IS 4) OR (Count IS 8))
    - ACQUIRE Consumer Thread
    - IF (There is not enough money)
      * CALL PrintState()
        + PRINT Insufficient Funds
      * IF (There is not enough Credit balance to do the transaction)
        + CALL PrintState()

PRINT Insufficient Credit

* + - * ELSE
        + Use Credit for Transaction
        + CALL PrintState()

PRINT Credited Transaction , Credit Owed and Credit Left

* + - * + RELEASE Producer Thread
    - ELSE
      * Withdraw from account balance
      * CALL PrintState()
        + PRINT Withdrawal and Balance left
      * RELEASE Producer Thread
* Bill()
  + ACQUIRE Consumer Thread
  + IF (There is not enough money)
    - CALL PrintState()
      * PRINT Insufficient Funds
    - IF (There is not enough Credit balance to do the transaction)
      * CALL PrintState()
        + PRINT Insufficient Credit
    - ELSE
      * Use Credit for Transaction
      * CALL PrintState()
        + PRINT Credited Transaction , Credit Owed and Credit Left
      * RELEASE Producer Thread
  + ELSE
    - Withdraw from account balance
    - CALL PrintState()
      * PRINT Withdrawal and Balance left
  + RELEASE Producer Thread
* StudentLoan()
  + IF((Count IS 1) OR (Count IS 4) OR (Count IS 8))
    - ACQUIRE Producer Thread
    - Deposit to account balance
    - CALL PrintState()
      * PRINT Deposit amount and Balance left
    - RELEASE Consumer Thread
* Deposit()
  + ACQUIRE Producer Thread
  + Deposit to account balance
  + CALL PrintState()
    - PRINT Deposit amount and Balance left
  + RELEASE Consumer Thread
* Job()
  + ACQUIRE Producer Thread
  + Deposit to account balance
  + CALL PrintState()
    - PRINT Deposit amount and Balance left
  + RELEASE Consumer Thread
* PayCredit()
  + GET 20% of Credit Debt
  + GET Credit Debt – 20% of Credit Debt
  + ACQUIRE Producer Thread
  + IF(There are not enough funds to pay credit debt)
    - CALL PrintState()
      * PRINT Insufficient Funds
  + ELSE IF(There is not debt to be paid)
    - Do nothing
  + ELSE
    - Pay 20% of credit debt using account balance
    - CALL PrintState()
      * PRINT Paid credit 20% and Credit Owed

**RunAccount.java**

* GET number of months
* GET number of universities
* GET number of consumer threads
* GET number of producer threads
* GET withdrawal speed (semaphore wait time)
* GET maximum withdrawal amount
* GET deposit speed (semaphore wait time)
* GET maximum deposit amount
* IF(The number of consumers is not equal to the number of producers)
  + EXIT()
* PRINT Grid to organize output:
  + Producer: Consumer: Balance:
* FOR(Number of Months)
  + EXECUTE Thread Pool (APP)
  + IF((Count IS 1) OR (Count IS 4) OR (Count IS 8))
    - PRINT Semester Started and Month
  + ELSE
    - PRINT Current Month
  + EXECUTE University Threads (Tuition and Student Loans)
  + EXECUTE Producer Threads
  + EXECUTE Consumer Threads
  + SHUTDOWN(APP) and Threads

**Bank.java (Interface for SemaphoreBank.java)**

* DEFINE Withdraw()
* DEFINE Deposit()
* DEFINE StudentLoan()
* DEFINE Tuition()
* DEFINE HousingAndFood()
* DEFINE Job()
* DEFINE Bill()
* DEFINE PayCredit()

**Consumer.java**

* Consumer()
  + SET Speed
  + SET Max Amount
  + INSTANTIATE Bank Account
  + SET Thread Name
  + SET Count
* Consumer()
  + Constructor
* Run()
  + WAIT For Semaphore
  + INSTANTIATE Account.Bill:
    - Value $200
  + INSTANTIATE Account.Withdraw:
    - Value random

**Producer.java**

* Producer()
  + SET Speed
  + SET Max Amount
  + INSTANTIATE Bank Account
  + SET Thread Name
  + SET Count
* Producer()
  + Constructor
* Run()
  + WAIT For Semaphore
  + INSTANTIATE Account.Job:
    - Value $200
  + INSTANTIATE Account.Deposit:
    - Value random

**Tuition.java**

* Tuition()
  + SET Speed
  + SET Max Amount
  + INSTANTIATE Bank Account
  + SET Thread Name
  + SET Count
* Tuition()
  + Constructor
* Run()
  + WAIT For Semaphore
  + INSTANTIATE Account.Tuition:
    - Value $1500
  + INSTANTIATE Account.Withdraw:
    - Value $200

**StudentLoan.java**

* StudentLoan()
  + SET Speed
  + SET Max Amount
  + INSTANTIATE Bank Account
  + SET Thread Name
  + SET Count
* StudentLoan()
  + Constructor
* Run()
  + WAIT For Semaphore
  + INSTANTIATE Account.StudentLoan:
    - Value $1500
  + INSTANTIATE Account.PayCredit

# **Default Values for Testing**

Default values for Testing:

**account**.Bill(200, **threadName**, **count**); *//Using consumer to pay Bills: withdraw from balance***account**.withdraw(1 + *generator*.nextInt(**max**), **threadName**, **count**); *//Using Consumer to withdraw from balance*

**account**.Job(200, **threadName**, **count**); *// Deposits money earned from job to Balance***account**.deposit(1 + *generator*.nextInt(**max**), **threadName**, **count**); *//Using Producer to deposit to Balance*

**account**.Tuition(1500, **threadName**, **count**); *//Tuition gets withdrawn from the balance***account**.HousingAndFood(200, **threadName**, **count**); *//Housing and Food withdraws from Balance*

**account**.studentLoan(1500, **threadName**, **count**); *//Using Producer's student Loan to deposit to Balance***account**.PayCredit(1 + *generator*.nextInt(**max**), **threadName**, **count**); *//Pay credit through Producer*

**public static void** main(String[] args)  
{  
 *//Number of months* **int** months = 12;  
 *//Number of Universities* **int** u = 1;  
 *//Number of consumers* **int** c = 2;  
 *//Number of producers* **int** p = 2;  
 *//Withdrawal speed* **int** wSpeed = 1;  
 *//Maximum withdrawal amount* **int** wMax = 150;  
 *//Deposit speed* **int** dSpeed = 1;  
 *//Maximum deposit amount* **int** dMax = 200;  
  
 *//Check if number of producers equals number of consumers* **if**(c != p)  
 {  
 System.***out***.println(**"Error: Number of Producers must be equal to number of Consumers"** + **"\nExiting...."**);  
 System.*exit*(0);  
 }

# **User Manual**

1. Setting up:
   1. To set up only RunAccount.java needs to be accessed.
      1. To change the number of months for which the app runs change the months variable.
      2. To adjust the number of consumers, producers or universities change the variables c, p, and u respectively. Keep in mind that the number of consumers must equal the number of producers.
      3. To adjust the wait time for the semaphores adjust the speed variables for both deposit and withdraw
      4. To adjust the max amount that gets either withdrawn or deposited then please modify wMax and dMax respectively.
   2. To change the value of the Tuition, Bills, Housing, Job, or StudentLoan
      1. Access StudentLoan.java, Tuition.java, Producer.java, and Consumer.java which are the threads and change the respective value. Example:
         1. **account**.Job(200, **threadName**, **count**); *// Deposits money earned from job to Balance*
2. Things to Consider:
   1. The program was written in JAVA using IntelliJ
   2. The values for depositing, withdrawing, tuition, student loan, bills, and jobs are already set up in a way that testing the app will show accurate results like in real life. It is up to the Program’s user if they want to change the values.
3. Understanding the output file:

Producer: Consumer: Balance:

--------- --------- --------

Each column has a name which makes the output file easier to read

------------------------------------------------------------------------------------------------

Month: 1 - Semester Started

------------------------------------------------------------------------------------------------

Every four months a new semester will start

Producers will appear on the left. Examples:

* Student Loan 1 deposits $1500
* Job 1 deposits $200
* Producer 1 deposits $27
* Pay Credit 20% 1 $54

Consumers will appear in the middle. Examples:

* Tuition 1 withdraws $1500
* Housing 1 withdraws $200
* Bill 1 withdraws $200
* Consumer 1 withdraws $71
* Consumer 1 Credit $71

Account Balance will appear on the right. Examples:

* Balance: $1500
* Withdraw Blocked: Insufficient Funds
* Credit left: $1429, Credit owed: $71
* Withdraw Blocked: Insufficient Credit