Name: Stanley Setiawan
SID: 430011232
UniKey: sset7405

COMP3520

Assignment 1 Report

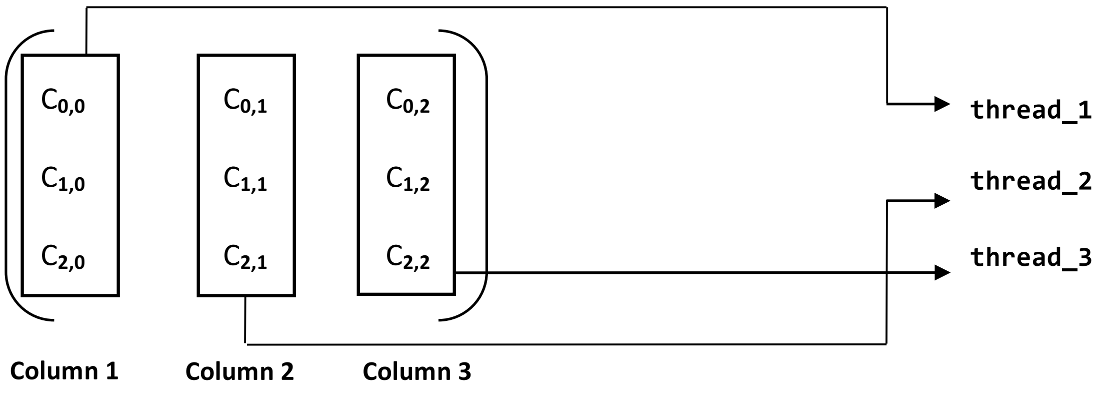
Parallel Matrix Multiplication – (Part A)

Overview

The Parallel Matrix multiplication program **`parallelMatrixMultiplication.c`** involves multiplying 2 matrices **`A`** and **`B`** and store the result in ***`matrix_C`*.** When the program begins, it asks user to enter 3 parameters:

- Parameter **M** specifies the number of rows for ***`matrix_A`***
- Parameter **K** specifies the number of columns in ***`matrix_A`*** as well as the number of columns of ***`matrix_B`***
- Parameter **N** specifies the number of columns of ***`matrix_B`***
- **`num_thrds`** which specifies the number of threads to be created

The result of the multiplication between matrix **A** and **B** will then be ***`matrix_C`*** with **M** *number of rows and* **N** *number of columns*. The multiplication between two matrices, however, shall utilize the **`pthread`** library in such a way where each thread is allocated with the task of calculating certain columns of ***`matrix_C`***.



Above is a representation of ***`matrix_C`*** with 3 rows and 3 columns where each element is denoted as **($C_{i,j}$)**. **`i`** denotes the row of the matrix element and **`j`** denotes the column of the matrix element. **`thread_1`**, in the example above, calculates the sum of multiplication between ***`matrix_A`*** and ***`matrix_B`*** elements and store the results in **$C_{0,0}$** | **$C_{1,0}$** | **$C_{2,0}$** with the algorithm below:

`multiThreadedMatrixMultiplication:`

```
for(z = startIndex to endIndex-1):
        for(x = 0 to m-1):
                C[x][z] = 0
                for(y = 0 to k-1):
                        C[x][z] = C[x][z] + (A[x][y] * B[y][z])
                end for
        end for
end for
```

- `startIndex` to `endIndex` indicates the range of columns that have to be processed by each thread
- `x` indicates the row number of *`matrix_A`* at each iteration of the loop (*and thus row number of* *`matrix_C`* at each iteration).
- `y` indicates the column number of *`matrix_A`* and also the row number of *`matrix_B`* at each iteration of the loop.
- `z` indicates *`matrix_B`* column number that will be processed at each iteration of the loop (*and thus* *`matrix_C`* column number at each iteration).

Each thread shall be assigned with a `startIndex` as well as `endIndex`. The thread will then start to execute the code above and populate matrix `C` columns starting from column `startIndex` up until column `(endIndex – 1)`. In some cases, it might not be possible to assign each thread the same number of columns to process and therefore some threads might get assigned 1 more column than other threads. This happens in the case where there are some leftover columns to be processed (columns cannot be divided equally between threads).

For example, if *`matrix_C`* is a `3 X 7` matrix instead of a `3 X 3` matrix then the 3 threads in the example above will have to decide which thread will get one more column to process. In this particular case, that means:

- `thread_1` will have to process *3 columns* (column **1-3**)
- `thread_2` will process *2 columns* (column **4-5**)
- `thread_3` will process *2 columns* (column **5-7**)

Testing

In order to test whether or not the multithreaded matrix multiplication is working as intended, the program also has a sequential matrix multiplication function which will store the matrix multiplication result of *`matrix_A`* and *`matrix_B`* in *`matrix_D`*:

```
for(x = 0 to m-1):                              sequentialMatrixMultiplication:
       for(z = 0 to n-1):
              C[x][z] = 0
              for(y = 0 to k-1):
                     C[x][z] = C[x][z] + (A[x][y] * B[y][z])
              end for
       end for
end for
```

Where `x`, `y` and `z` signify the exact same thing as they did before with the parallel matrix multiplication thread function. Once the program has finished executing the multi-threaded matrix multiplication and sequential matrix multiplication, it compares each element of *`matrix_C`* and *`matrix_D`* to see if they are the same by using the comparison function:

```
matricesAreSame = true                                          compareMatrices:
for(i = 0 to row-1):
       for(j = 0 to column-1):
              if(sequentialMatrix[i][j] is not equal to parallelMatrix[i][j]
                     matricesAreSame = false
       end for
end for
```

- `i` indicates the row number of the two matrices at each iteration
- `j` indicates the column number of the two matrices at each iteration

If **matricesAreSame** is **true** then the program prints out **"The two matrices are the same!"**. Otherwise if it's ~~false~~ it prints out **"The two matrices are different, there's an error!"**.

Limitation and improvement

Despite producing the correct result, the program is not without its fault. For one, it can take a lot of time to produce the result matrix as there are overheads in the thread creation stage. This means that given a small matrix, sequential matrix multiplication might produce the result faster than the multithreaded matrix multiplication method. For instance, if ***matrix_C*** is a **1 X 3** matrix and the program wants to calculate it with the multithreaded matrix multiplication method it will need to:

- create each thread separately and that also means allocating memory for user defined ids
- calculate the **startIndex** and **endIndex** for each thread to give them a range of columns to process
- join all the threads
- free the allocated memory for user defined ids

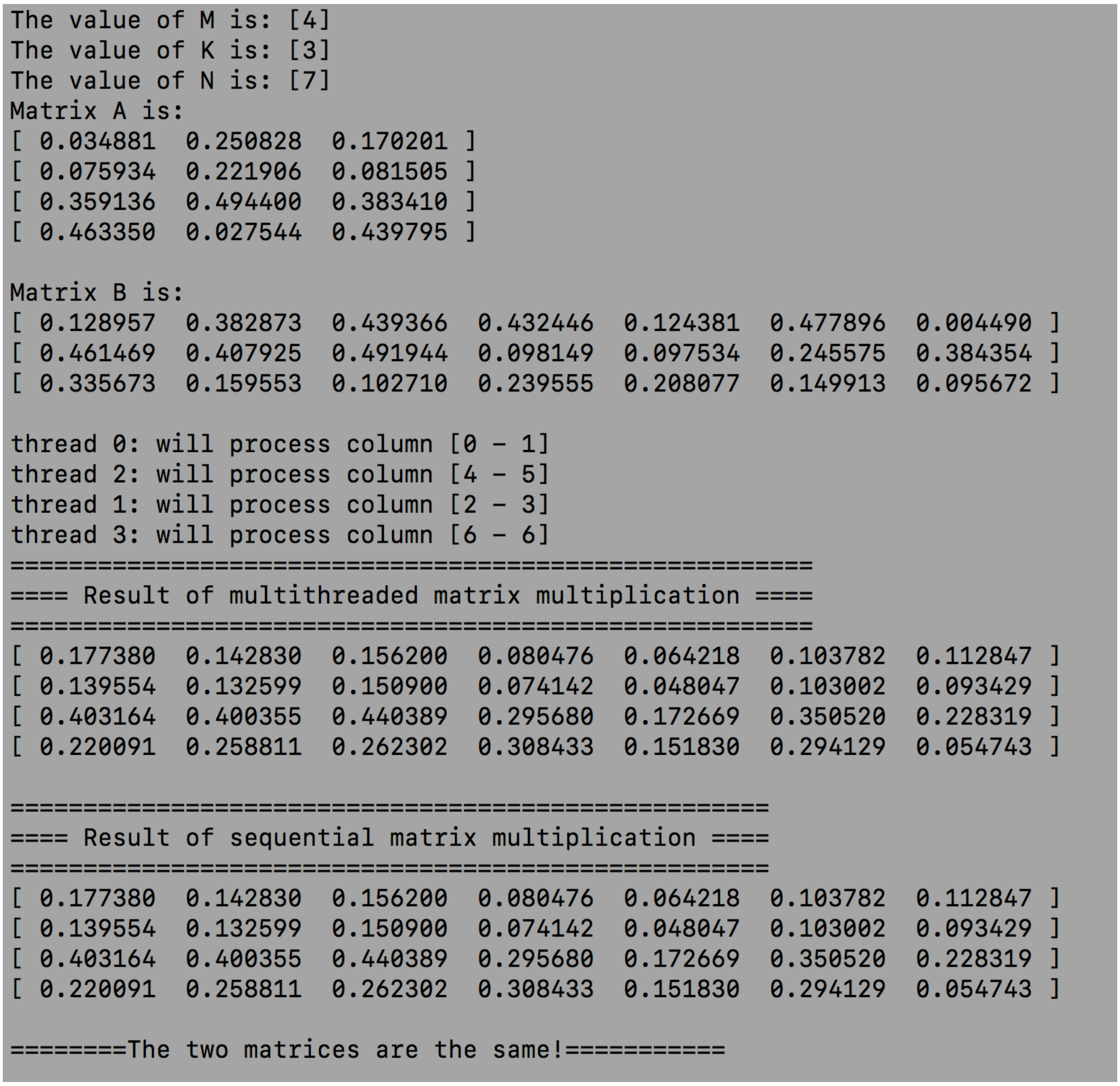
Furthermore, the program can also be improved by making each matrix representation a 1-dimensional array instead of 2 dimensional. 2-dimensional array is a list of pointers where each member of the list points to another pointer. This also means that getting an element from a 2-dimensional array will take significantly more time than getting an element from a 1-dimensional array. Example of a **3 X 3 matrix_C** represented in a normal 1-dimensional array:

$$\textbf{int * A =} \Big\{\underbrace{\overset{A[0]}{C_{0,0}}\quad\overset{A[1]}{C_{0,1}}\quad\overset{A[2]}{C_{0,2}}}_{\textbf{First Row}} \;\Big|\; \underbrace{\overset{A[3]}{C_{1,0}}\quad\overset{A[4]}{C_{1,1}}\quad\overset{A[5]}{C_{1,2}}}_{\textbf{Second Row}} \;\Big|\; \underbrace{\overset{A[6]}{C_{2,0}}\quad\overset{A[7]}{C_{2,1}}\quad\overset{A[8]}{C_{2,2}}}_{\textbf{Third Row}}\Big\}$$

Compiling source code and running the program

parallelMatrixMultiplication.c is a C source code file that shall be compiled using a **makefile**. User only needs to type **make** on the command line to compile it and run the program by typing **./parallelMatrixMultiplication** on the command line. This program does not accept additional argument.

Output



```
The value of M is: [4]
The value of K is: [3]
The value of N is: [7]
Matrix A is:
[ 0.034881  0.250828  0.170201 ]
[ 0.075934  0.221906  0.081505 ]
[ 0.359136  0.494400  0.383410 ]
[ 0.463350  0.027544  0.439795 ]

Matrix B is:
[ 0.128957  0.382873  0.439366  0.432446  0.124381  0.477896  0.004490 ]
[ 0.461469  0.407925  0.491944  0.098149  0.097534  0.245575  0.384354 ]
[ 0.335673  0.159553  0.102710  0.239555  0.208077  0.149913  0.095672 ]

thread 0: will process column [0 - 1]
thread 2: will process column [4 - 5]
thread 1: will process column [2 - 3]
thread 3: will process column [6 - 6]
=======================================================
==== Result of multithreaded matrix multiplication ====
=======================================================
[ 0.177380  0.142830  0.156200  0.080476  0.064218  0.103782  0.112847 ]
[ 0.139554  0.132599  0.150900  0.074142  0.048047  0.103002  0.093429 ]
[ 0.403164  0.400355  0.440389  0.295680  0.172669  0.350520  0.228319 ]
[ 0.220091  0.258811  0.262302  0.308433  0.151830  0.294129  0.054743 ]

====================================================
==== Result of sequential matrix multiplication ====
====================================================
[ 0.177380  0.142830  0.156200  0.080476  0.064218  0.103782  0.112847 ]
[ 0.139554  0.132599  0.150900  0.074142  0.048047  0.103002  0.093429 ]
[ 0.403164  0.400355  0.440389  0.295680  0.172669  0.350520  0.228319 ]
[ 0.220091  0.258811  0.262302  0.308433  0.151830  0.294129  0.054743 ]

========The two matrices are the same!===========
```

Terminal Problem – (Part B)

Overview

The second program called **`terminalProblem.c`** is a program that is designed to simulate a synchronization problem in a multi-threaded program. A number of customers wish to use one of a limited number of terminals. Each customer who wishes to use a terminal need to wait in a waiting room with a limited number of seats. Suppose the room is full and all seats are occupied, a customer who just arrives will not bother waiting to get access to a terminal and leaves.

On top of that, an attendant is available in the waiting room and their job is to assign a terminal to one of the waiting customers. One customer can only occupy one terminal and each terminal can only be occupied by one customer at a time. In the case when there is no customer available, the attendant will wait and go to sleep, otherwise it will keep assigning a terminal to each waiting customer. Once all customers are served the attendant can leave.

First of all, the program asks user 5 parameters which will be used throughout the program lifetime:

- Total **`no_of_customers`** who wish to get access to a terminal.
- Total **`no_of_seats`** available in the waiting room so customers can seat before they get served by the attendant.
- **`no_of_terminals`** available
- **`customers_arrival_rate`** to specify the delay between each customer thread creation.
- **`terminal_usage_time`** to specify the amount of time each customer is going to spend on the terminal.

Each customer is represented by a *customer thread* and the attendant who will serve the customers is represented by the *attendant thread*. The program then initialises the *attendant thread* and enters a loop where it shall create **`no_of_customers`** *customer threads*. Between the creation of each customer thread is a **`sleep()`** function which will delay the creation of the next customer thread. The delay can range from **`0`** *to* **`customers_arrival_rate`** *seconds*.

Each *attendant thread* executes the **`attendant_routine`** and each *customer thread* executes **`customer_routine`**. Once all *customer threads* have finished their execution, they are going to be joined at the program main thread and the program shall terminate the *attendant thread*. Last but not least, the program shall free any memory allocated to each *customer thread* **`customer_id`** and will proceed to destroy the *mutex* and *condition variables* used to synchronize the operation between threads.

Below are pseudocodes for **attendant_routine** as well as **customer_routine** which will be executed by the *attendant thread* and each *customer thread* respectively.

```
while(true):                                                    attendant_routine:
    lock(mutex)
    if(all seats are unoccupied):
        print "No customers and I'm waiting"
        wait until there is customer available – condVar[customerAvailable]
    end if
    print "try to find a free terminal for the customer"
    if(all terminals are being used):
        print "All terminals are occupied"
        wait until one terminal is available – condVar[terminalAvailable]
        print "There are free terminals now"
    end if
    no_of_free_seats++
    print "Call one customer who's waiting"
    signal the customer that the attendant is now ready to serve them – condVar[attendantServesCustomer]
    unlock(mutex)
    lock(mutex)
    no_of_free_term--
    print "Assign one terminal to the customer"
    unlock(mutex)
end while
```

```
print "Customer[customer_id] arrives                             customer_routine:
lock(mutex)
if(all seats are occupied):
    print "All seats have been taken and I'll leave now!"
    unlock(mutex)
else:
    print "I'm lucky to get a free seat!"
    no_of_free_seats--
    signal the attendant that there's a customer available – condVar[customerAvailable]
    print "I'm waiting to be served"
    wait for the attendant to get ready serving the customer – condVar[attendantServesCustomer]
    print "I'm getting a terminal now"
    unlock(mutex)
    customer uses the terminal for a certain period of time
    lock(mutex)
    no_of_free_terms++
    print "I'm finished using the terminal and leaving."
    signal the attendant that there is now a terminal available – condVar[terminalAvailable]
    unlock(mutex)
end if

terminate this customer thread
```

- **no_of_free_seats++** indicates that there is one more free seat (a customer is being served by the attendant) whereas **no_of_free_seats--** indicates that there is one less free seat (a new customer occupies that seat).
- **no_of_free_terms++** indicates that there is one more free terminal (a customer is done using the terminal) whereas **no_of_free_terms--** indicates that there is one less free terminal (the attendant assigns one more terminal to another customer).
- The period of time a customer thread spends on a terminal is implemented by the **sleep()** function and it varies between **0** to **terminal_usage_time** *seconds* for each customer.

Furthermore, in order to mitigate synchronization problem between the attendant thread and customer threads, *1 mutex* and *3 condition variables* were used. The table below outlines the role that the *mutex* and each *condition variable* played in both **`attendant_routine`** and **`customer_routine`**.

`Mutex`	
The mutex is used to give only one thread access on a critical section. This ensures that no race condition is going to happen between the thread and other threads when modifying a shared variable in the critical section. Mutex is also used as the second argument when a thread called **`pthread_cond_wait()`** function to make sure that it unlocks the mutex, allowing another thread to gain access to the lock and continues their execution.	
`condVar[customerAvailabe]`	
`attendant_routine`	At the start of the program, the attendant is waiting for customers to arrive. Therefore, the **`customerAvailable`** *condition variable* is used by the attendant to wait for a customer to arrive.
`customer_routine`	When a customer arrives, it shall signal the attendant with the **`customerAvailable`** *condition variable* regardless if the attendant is still waiting or not.
`condVar[terminalAvailable]`	
`attendant_routine`	When there are customers available in the waiting room, the attendant checks whether a free terminal is available for the one of the customers to use before serving them. There will be times when all terminals are occupied by previous customers and in such cases the attendant waits until one terminal is available for the next customer with **`terminalAvailable`** *condition variable*.
`customer_routine`	Whenever a customer is finished using the terminal, it will signal the attendant to tell them that their terminal is now available to be used by the next waiting customer using **`terminalAvailable`** *condition variable*.
`condVar[attendantServesCustomer]`	
`attendant_routine`	The attendant will signal the customer that they are ready to serve them. Just like others, the signalling is done with a *condition variable* and in this case the **`attendantServesCustomer`** *condition variable* is used
`customer_routine`	When the customer arrives, they will occupy one seat in the waiting room and waits until the attendant is ready to serve them. The **`attendantServesCustomer`** *condition variable* is used in this waiting scenario by the customer.

Testing

User of the program needs to be able to keep track of the program output to see whether the threads execute themselves in the manner expected and if the program has correct interaction between threads. In order to do that, each thread keeps track of all variables which value is modified during its execution and outputs a message to the console.

For the **`attendant_routine`**:

- *"Attendant: The number of free seats is* **`no_of_free_seats`**. *No customers and I'm waiting."* message will be printed when there's no customer waiting in the waiting room.
- *"Attendant: The number of free seats now is* **`no_of_free_seats`**. *try to find a free terminal."* message will be printed when there are customers waiting in the room and the attendant checks if there is a terminal available for waiting customers.
- *"Attendant: The number of free terminal s is* **`no_of_free_terms`**. *All terminals are occupied."* message will be printed when there is no terminal available for the waiting customer (all terminals are occupied). Therefore, the attendant needs to wait until a terminal is available.
- *"Attendant: The number of free terminals is* **`no_of_free_terms`**. *There are free terminals now."* will be printed out when a previous customer has just finished using the terminal. This means that the attendant can now assign a terminal to the next waiting customer.
- *"Attendant: Call one customer. The number of free seats is now* **`no_of_free_seats`**.*"* message will be printed when the attendant is ready to serve the next waiting customer.
- *"Attendant: Assign one terminal to the customer. The number of free terminals is now* **`no_of_free_terms`***"* will be printed when the attendant assigns one terminal to the next waiting customer.

For the **`customer_routine`**:

- *"Customer* **`customer_id`** *arrives"* is printed when a new customer arrives.
- *"Customer* **`customer_id`**: *oh no! all seats have been taken and I'll leave now!"* will be printed when all seats in the waiting room have already been occupied by previous customers who came earlier. The customer will then proceed to leave and not bother using the terminal.
- *"Customer* **`customer_id`**: *I'm lucky to get a free seat from* **`no_of_free_seats`***"* will be printed when there are free seats available and the customer occupies an empty seat.
- *"Customer* **`customer_id`**: *I'm waiting to be served"* will be printed when the customer is waiting for the attendant to serve them
- *"Customer* **`customer_id`**: *I'm getting a terminal now."* will be printed when it is the customer turned to be served by the attendant. The attendant will then proceed to assign the customer a terminal to use and the customer will use the terminal for a period of time.
- *"Customer* **`customer_id`**: *I'm finished using the terminal and leaving."* is printed when the customer is finished using the terminal and leaves the waiting room.

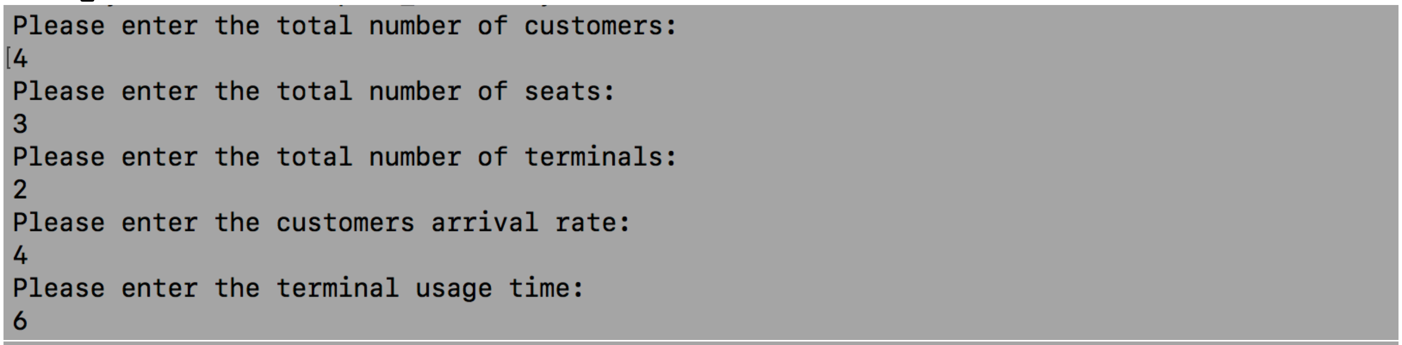
Limitation and Improvement

The program involves using a *mutex* and 3 *condition variables*. The problem with the *mutex*, however, is that we cannot specify the default state of the *mutex*. A *mutex* will always have a default state of locked. Instead of using a *mutex*, a better way to design the program is to use a *binary semaphore*. *Binary semaphore* allows us to set the initial condition of the lock by setting it to either 1 for unlock or 0 for locked.

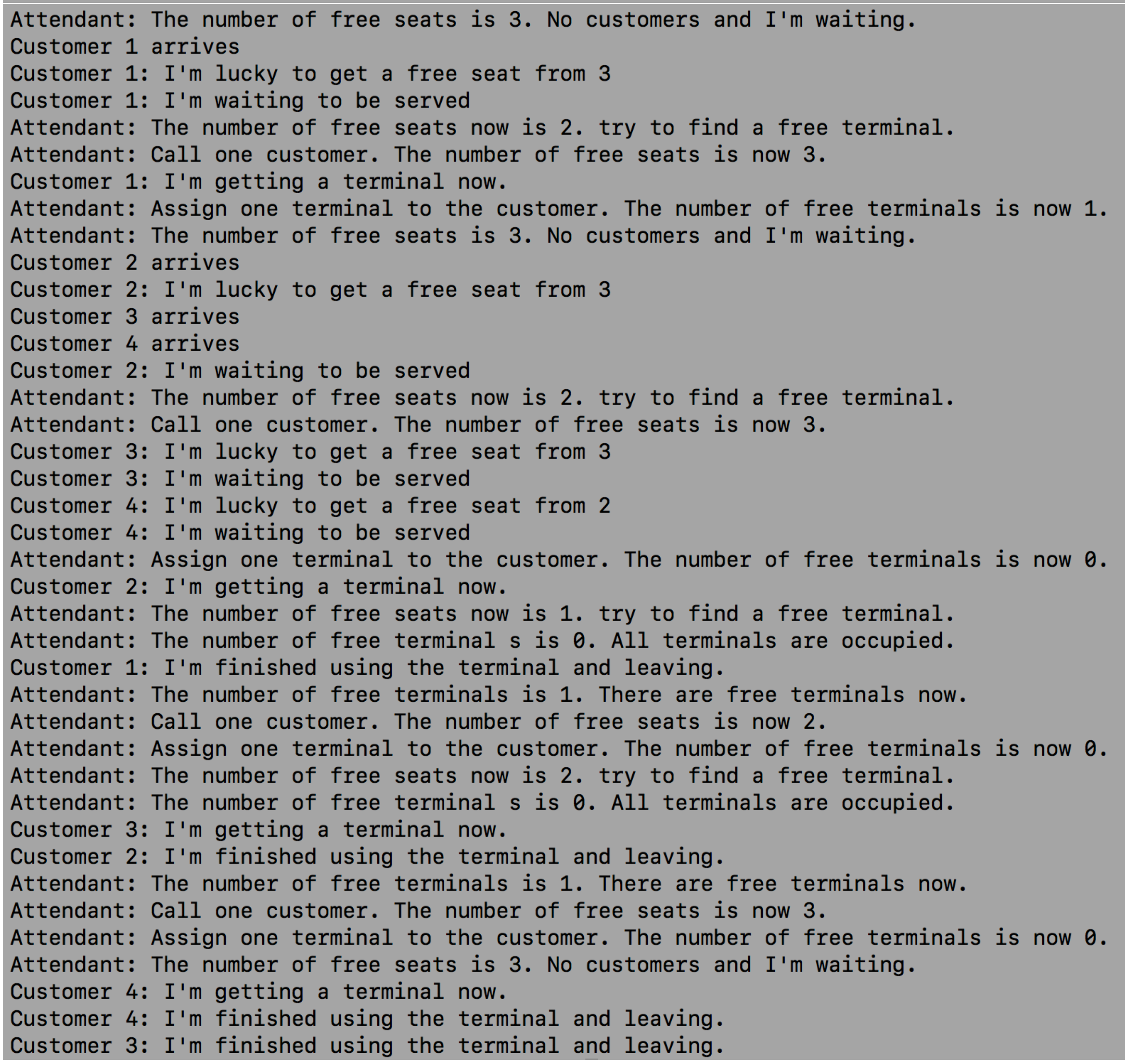
Compilation and running the program

`terminalProblem.c` is a C source code file which will be compiled using a **`makefile`**. User only needs to type **`make`** on the command line to compile it and run the program by typing **`./terminalProblem`** on the command line. It is also important to note that the program does not accept any additional argument.

Output



```
Please enter the total number of customers:
[4
Please enter the total number of seats:
3
Please enter the total number of terminals:
2
Please enter the customers arrival rate:
4
Please enter the terminal usage time:
6
```



```
Attendant: The number of free seats is 3. No customers and I'm waiting.
Customer 1 arrives
Customer 1: I'm lucky to get a free seat from 3
Customer 1: I'm waiting to be served
Attendant: The number of free seats now is 2. try to find a free terminal.
Attendant: Call one customer. The number of free seats is now 3.
Customer 1: I'm getting a terminal now.
Attendant: Assign one terminal to the customer. The number of free terminals is now 1.
Attendant: The number of free seats is 3. No customers and I'm waiting.
Customer 2 arrives
Customer 2: I'm lucky to get a free seat from 3
Customer 3 arrives
Customer 4 arrives
Customer 2: I'm waiting to be served
Attendant: The number of free seats now is 2. try to find a free terminal.
Attendant: Call one customer. The number of free seats is now 3.
Customer 3: I'm lucky to get a free seat from 3
Customer 3: I'm waiting to be served
Customer 4: I'm lucky to get a free seat from 2
Customer 4: I'm waiting to be served
Attendant: Assign one terminal to the customer. The number of free terminals is now 0.
Customer 2: I'm getting a terminal now.
Attendant: The number of free seats now is 1. try to find a free terminal.
Attendant: The number of free terminal s is 0. All terminals are occupied.
Customer 1: I'm finished using the terminal and leaving.
Attendant: The number of free terminals is 1. There are free terminals now.
Attendant: Call one customer. The number of free seats is now 2.
Attendant: Assign one terminal to the customer. The number of free terminals is now 0.
Attendant: The number of free seats now is 2. try to find a free terminal.
Attendant: The number of free terminal s is 0. All terminals are occupied.
Customer 3: I'm getting a terminal now.
Customer 2: I'm finished using the terminal and leaving.
Attendant: The number of free terminals is 1. There are free terminals now.
Attendant: Call one customer. The number of free seats is now 3.
Attendant: Assign one terminal to the customer. The number of free terminals is now 0.
Attendant: The number of free seats is 3. No customers and I'm waiting.
Customer 4: I'm getting a terminal now.
Customer 4: I'm finished using the terminal and leaving.
Customer 3: I'm finished using the terminal and leaving.
```