EAST WEST UNIVERSITY
**Department of Computer Science and Engineering**

# CSE 325- Mini project

Course Name: Operating Systems
Course Code: CSE 325
Section No: 01

**Name of the Project:** Sweet Harmony

**Date of submission:** 09 May 2023

**Submitted by: Group 08**

**Student's Name:** Sadia Islam Prova
**Student's ID**: 2020-3-60-012

**Student's Name:** Rokeya Akter Riya
**Student's ID**: 2020-1-60-145

**Student's Name:** Nafisa Siddiqua
**Student's ID**: 2019-3-60-036

**Submitted To:**

**Dr. Md. Nawab Yousuf Ali**
Professor,
Department of Computer Science & Engineering

## Abstract:

This problem involves simulating the operation of a bakery called "Sweet Harmony" in the town of Pastelville. The bakery has a unique rule that only allows an equal number of customers wearing red and blue outfits inside at any given time to maintain a pleasant ambiance. The staff must manage various aspects, such as allowing customers inside while adhering to the outfit rule, ensuring customers find a table, managing the queue of customers waiting outside, and handling customers leaving the bakery. To address these challenges, a C program can be developed using semaphores, processes, and thread mutex locks. Semaphores can help manage the outfit rule, processes can represent customers and their actions, thread mutex locks can synchronize available tables, and additional semaphores can manage the waiting queue outside. By utilizing these techniques, the program can simulate the operation of Sweet Harmony while addressing concurrency issues and the unique outfit rule.

## Introduction:

An operating system (OS) is software that manages computer hardware and software resources while also providing common functions to computer programs. Time-sharing operating systems plan tasks to make the most of the system's resources, and they may also contain accounting software for cost allocation of processor time, storage, printing, and other resources. Although application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it, the operating system acts as an intermediary between programs and the computer hardware for hardware functions such as input and output and memory allocation. From cellular phones and video game consoles to web servers and supercomputers, operating systems are found on many devices that incorporate a computer.

In our “sweet harmony’ problem”, we will focus on threads, process, mutex locks and semaphore. a C program can certainly be developed using semaphores, processes, and thread mutex locks to address the challenges faced by the Sweet Harmony bakery. Here's a brief overview of how each of these tools can be used to address the different aspects of the problem:

**Threads**: Within a process, a thread is a path of execution. Multiple threads can exist in a process. The lightweight process is also known as a thread. By dividing a process into numerous threads, parallelism can be achieved. Multiple tabs in a browser, for example, can represent different threads. MS Word makes use of numerous threads: one to format the text, another to receive inputs, and so on. Below are some more advantages of multithreading.

**Process:** A process is essential for running software. The execution of a process must be done in a specific order. To put it another way, we write our computer programs in a text file, and when we run them, they turn into a process that completes all of the duties specified in the program. A program can be separated into four components when it is put into memory and becomes a process:

stack, heap, text, and data. The diagram below depicts a simplified structure of a process in main memory.

**Semaphore:** Dijkstra proposed the semaphore in 1965, which is a very important technique for managing concurrent activities using a basic integer value called a semaphore. A semaphore is just an integer variable shared by many threads. In a multiprocessing context, this variable is utilized to solve the critical section problem and establish process synchronization. There are two types of semaphores:

1. Binary Semaphore – This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

2. Counting Semaphore – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

## Project implementation:

A C program can certainly be developed using semaphores, processes, and thread mutex locks to address the challenges faced by the Sweet Harmony bakery. Here's a brief overview of how each of these tools can be used to address the different aspects of the problem:

Semaphores: Semaphores can be used to manage the equal outfit rule, allowing only customers with matching outfits to enter. One semaphore can be used for red outfits, and another semaphore can be used for blue outfits. Each time a customer enters the bakery, they must wait on the appropriate semaphore until a matching customer leaves the bakery, which signals the semaphore to allow the waiting customer to enter.

Processes: Processes can be used to represent individual customers and their actions (entering the bakery, finding a table, waiting, and leaving). Each customer can be represented by a separate process that communicates with the other processes using pipes. When a customer enters the bakery, they can send a message to the bakery manager process indicating their outfit color and wait for a table to become available. When a table is available, the bakery manager process can send a message to the waiting customer process indicating which table to sit at.

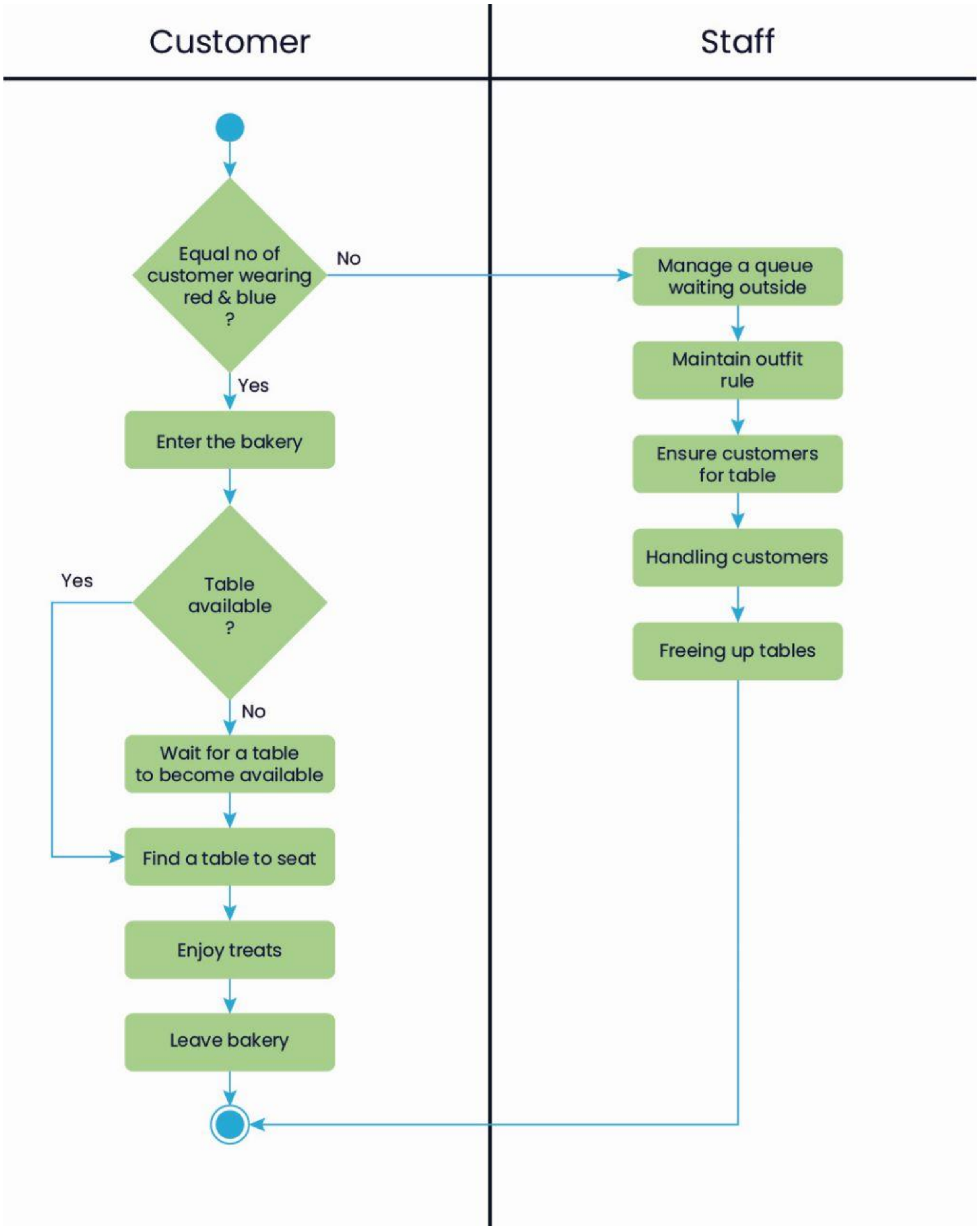
Thread mutex locks: Thread mutex locks can be used to synchronize the available tables, ensuring customers can only sit at a free table. Each table can be represented by a separate thread, and a mutex lock can be used to ensure that only one customer can sit at a table at a time.

When a customer leaves the bakery, the table thread can signal the bakery manager process, indicating that the table is now available for a new customer.

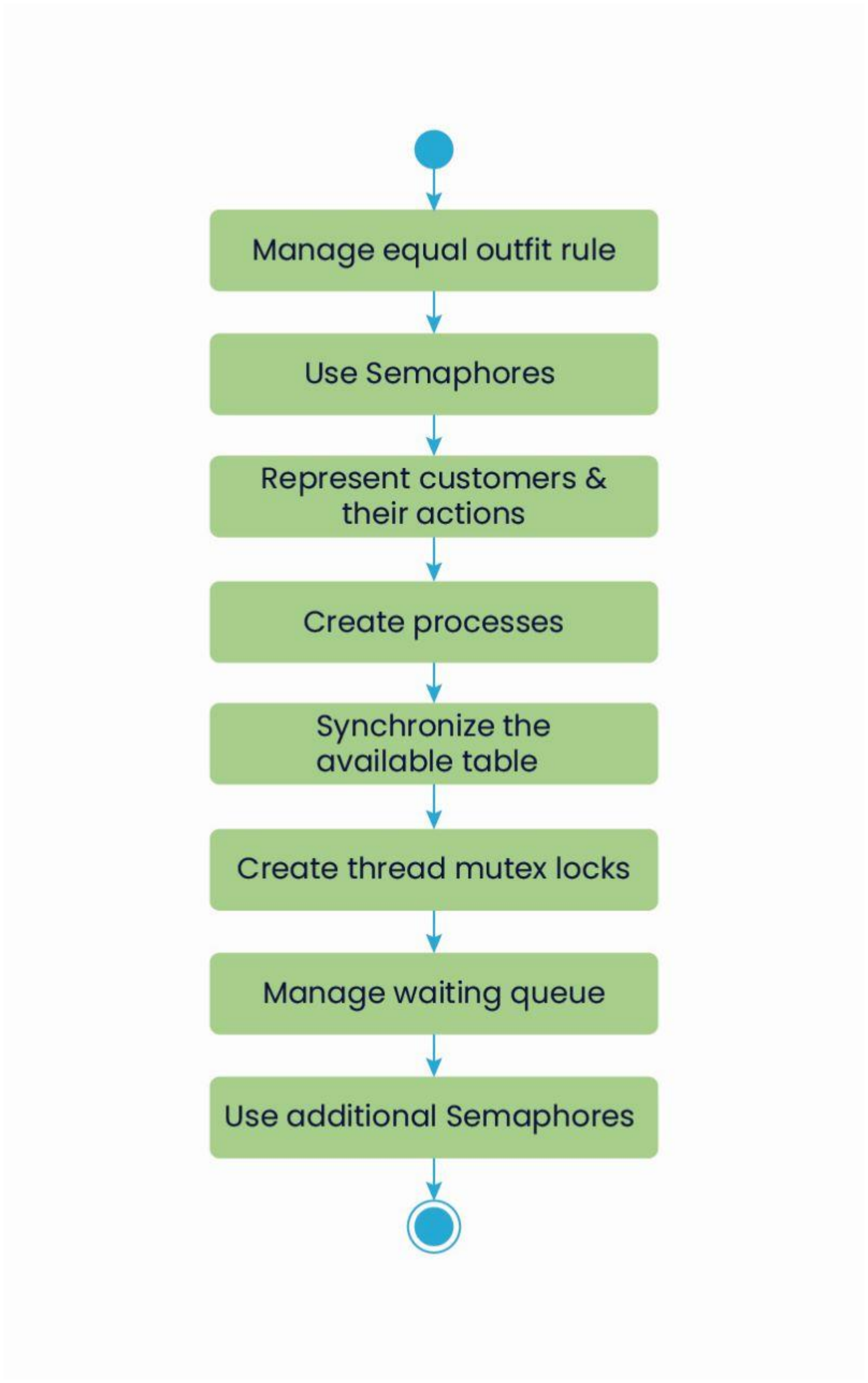
Additional semaphores: Additional semaphores can be used to manage the waiting queue outside the bakery. When a customer arrives and the equal outfit rule cannot be maintained, they can wait on a semaphore until a matching customer arrives, indicating that it is now their turn to enter the bakery.

By utilizing semaphores, processes, and thread mutex locks, the C program can effectively simulate the charming environment at Sweet Harmony bakery, while addressing the concurrency issues and unique outfit rule that arise in this scenario.

## Flowchart of proposed work:



## Flowchart of the solution:


Manage equal outfit rule
Use Semaphores
Represent customers & their actions
Create processes
Synchronize the available table
Create thread mutex locks
Manage waiting queue
Use additional Semaphores

## C Program code:

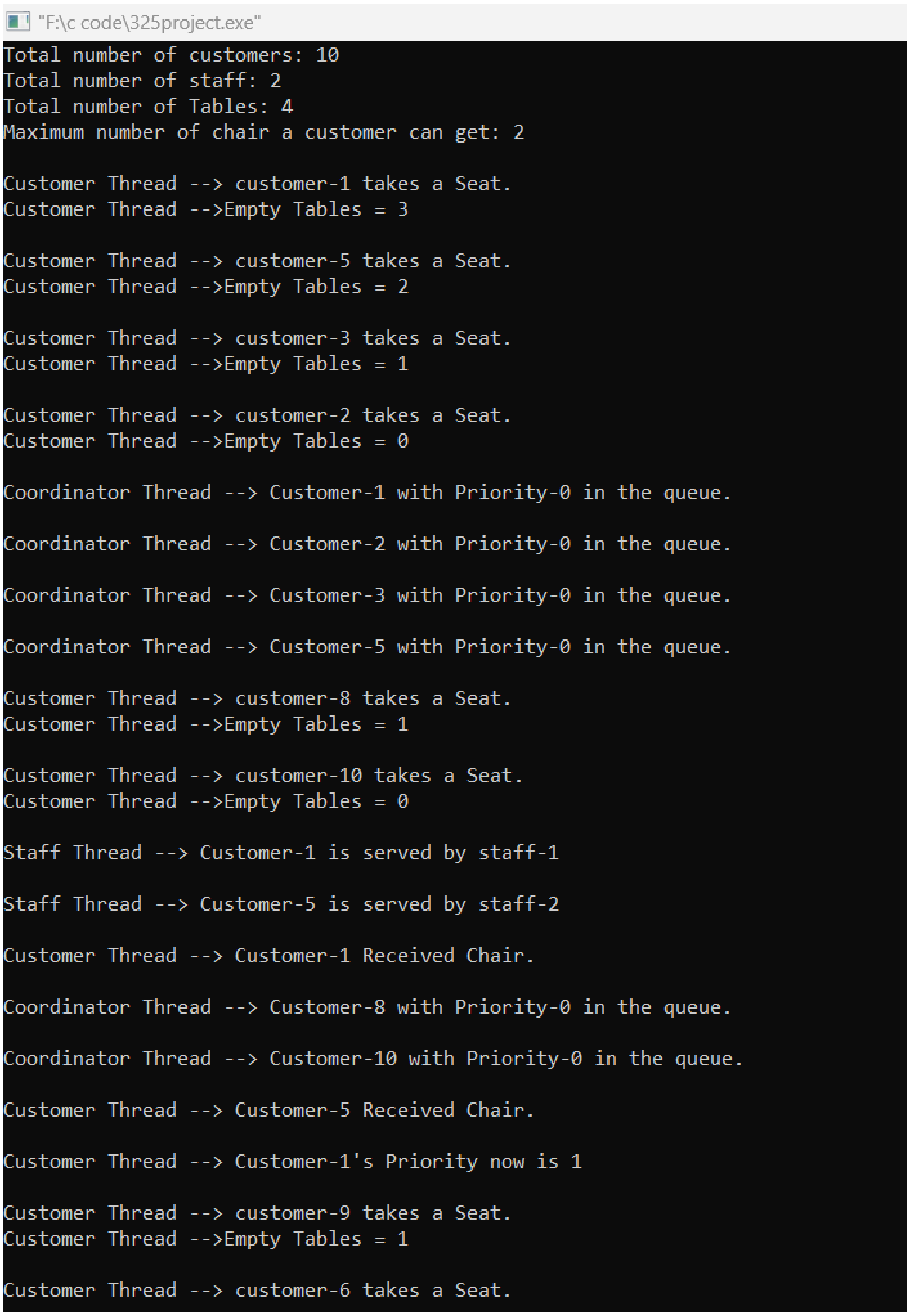
```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define MAX 100000

int

mark_visited[MAX],customer_priority[MAX],arrival_time[MAX],total_visit_of_customer[MA
X],customer_id[MAX],staff_id[MAX];

int

requesting_order=0,customer_done=0,occupied_table=0,total_table,total_customer,total_staff,ch
air_limit;

sem_t customer,coordinator,served_customer[MAX],mutexLock;

void *customer_thread(void *customer_id)

{

 int c_id=*(int*)customer_id; //converting void to int

 while(1)

 {

 if(total_visit_of_customer[c_id-1] == chair_limit)

 {

 sem_wait(&mutexLock);

 customer_done++; //how many customers have served totally

 sem_post(&mutexLock);

 printf("\n\t customer-%d terminates\n",c_id);

 if(customer_done == total_customer) printf("\n\t All customers Have Recieved Chair\n");

 sem_post(&customer); //notify coordinate to terminate

 pthread_exit(NULL); //thread ends

 }

 sem_wait(&mutexLock); //mutex was initially 1, when we call sem_wait(), it turns 0, it is locked
right now
```

```
 if(occupied_table == total_table)
 {
 sem_post(&mutexLock); //mutex unlockes it, by turning the value 1
 continue;
 }
 occupied_table++;
 requesting_order++; //request number of customer, it keeps the track of when the customer came
 mark_visited[c_id-1]=requesting_order;
 printf("\nCustomer Thread --> customer-%d takes a Seat.\nCustomer Thread -->Empty Tables =
%d\n",c_id,total_table-occupied_table);
 sem_post(&mutexLock); //unlocked for other customers
 sem_post(&customer); //notify coordinator that customer seated.
 sem_wait(&served_customer[c_id-1]); //wait to be serverd.
 printf("\nCustomer Thread --> Customer-%d Received Chair.\n",c_id);
 sem_wait(&mutexLock);
 total_visit_of_customer[c_id-1]++; //tutored times++ after served.
 printf("\nCustomer Thread --> Customer-%d's Priority now is %d\n",c_id,
total_visit_of_customer[c_id-1]);
 sem_post(&mutexLock);
 }
}
void *coordinator_thread()
{
 while(1)
 {
 if(customer_done==total_customer) //if all customers finish, customers threads and coordinate
thread terminate.
 {
 for(int i=0; i<total_staff; i++) sem_post(&coordinator); //notify staffs to terminate
 printf("\n\t coordinator terminates\n"); //terminate coordinate itself
 pthread_exit(NULL); //thread ends
 }
 sem_wait(&customer); // waiting for customers signal
 sem_wait(&mutexLock);
 for(int i=0; i<total_customer; i++) //find the customers who just seated and push them into the
priority queue
 {
 if(mark_visited[i]>-1)
 {
 customer_priority[i] = total_visit_of_customer[i];
 arrival_time[i] = mark_visited[i]; //to save the time when the customer came
 printf("\nCoordinator Thread --> Customer-%d with Priority-%d in the
queue.\n",customer_id[i],total_visit_of_customer[i]);
 mark_visited[i]=-1;
 sem_post(&coordinator); //notify staff that customer is in the queue.
 }
 }
 sem_post(&mutexLock);
 }
}
void *staff_thread(void *staff_id)
{
 int s_id=*(int*)staff_id;
 while(1)
 {
 if(customer_done==total_customer) //if all customers finish, staffs threads terminate.
 {
 sem_wait(&mutexLock);
```

```
printf("\n\t staff-%d terminates\n",s_id);
sem_post(&mutexLock);
pthread_exit(NULL); //thread ends
}
int max_request=total_customer*chair_limit+1; //this is the maximum serial a customer can get
int max_priority = chair_limit-1;
int c_id=-1; //it is a flag.
sem_wait(&coordinator); //wait coordinator's notification
sem_wait(&mutexLock);
for(int i=0; i<total_customer; i++) //find customer with highest priority from priority queue. If customers with same priority, who come first has higher priority
{
if(customer_priority[i]>-1 && customer_priority[i]<= max_priority)
{
if(arrival_time[i]<max_request)
{
max_priority=customer_priority[i];
max_request=arrival_time[i]; //who comes first, here we are trying to find the minimum arrival time if the priotiy is same
c_id=customer_id[i];
}
}
}
if(c_id==-1) //in case no customer in the queue.
{
sem_post(&mutexLock);
continue;
}
customer_priority[c_id-1] = -1; //reset the priority queue
```

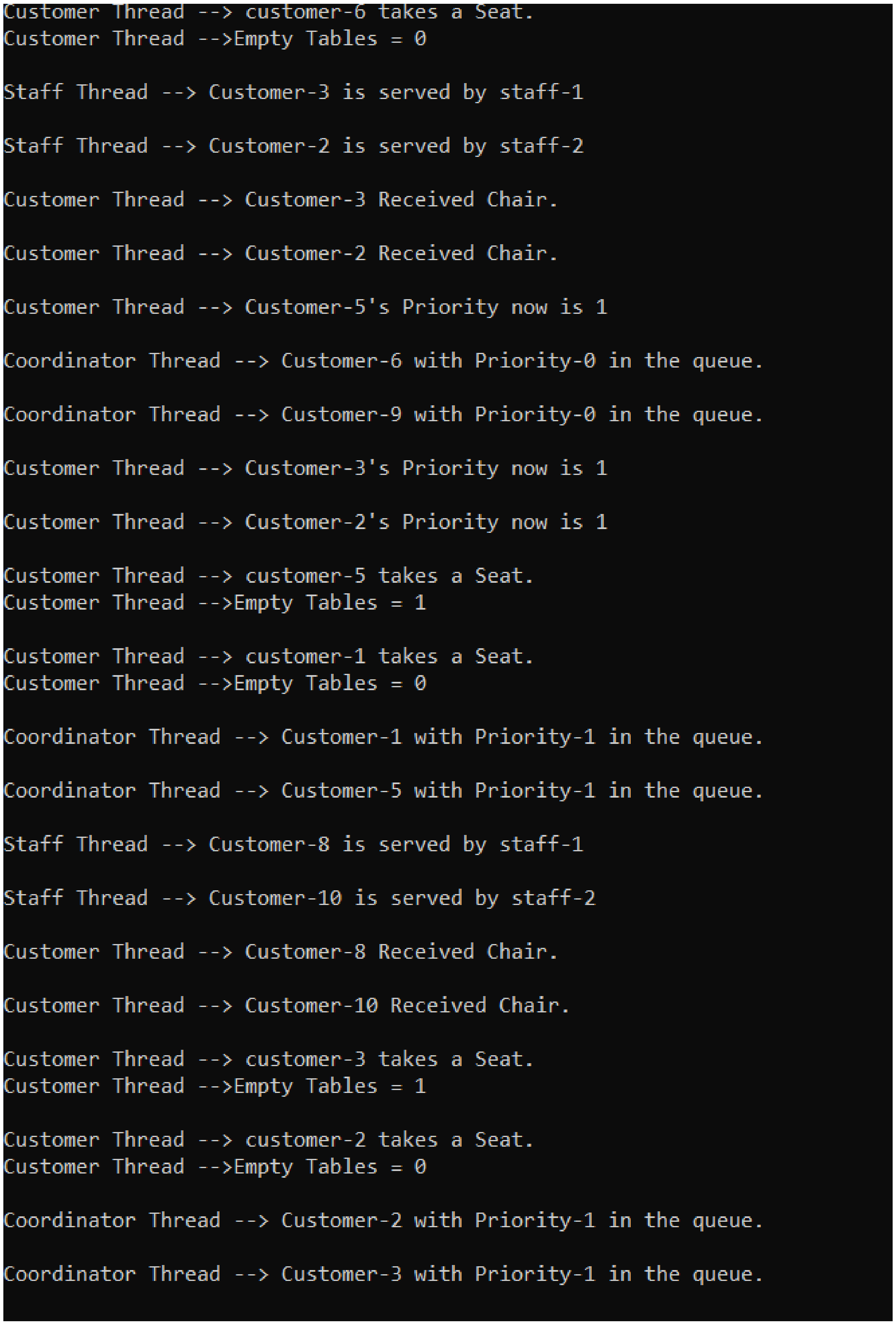
```
 arrival_time[c_id-1] = -1;
 occupied_table--;
 sem_post(&mutexLock);
 sem_wait(&mutexLock); //after tutoring
 printf("\nStaff Thread --> Customer-%d is served by staff-%d\n",c_id,s_id);
 sem_post(&mutexLock);
 sem_post(&served_customer[c_id-1]); //update shared data so customer can know who served
him.
 }
}
int main()
{
 printf("Total number of customers: ");
 scanf("%d", &total_customer);
 printf("Total number of staff: ");
 scanf("%d", &total_staff);
 printf("Total number of Tables: ");
 scanf("%d", &total_table);
 printf("Maximum number of chair a customer can get: ");
 scanf("%d", &chair_limit);
 for(int i=0; i<total_customer; i++)
 {
 mark_visited[i]=-1;
 customer_priority[i] = -1;
 arrival_time[i] = -1;
 total_visit_of_customer[i]=0;
 }
 sem_init(&customer,0,0);
 sem_init(&coordinator,0,0);
 sem_init(&mutexLock,0,1);
 for(int i=0; i<total_customer; i++) sem_init(&served_customer[i],0,0);
 pthread_t customers[total_customer],staff[total_staff],coordinator; //allocate threads
 for(int i = 0; i < total_customer; i++) //create threads for customer
 {
 customer_id[i] = i + 1; //saved in array
 pthread_create(&customers[i], NULL, customer_thread, (void*) &customer_id[i]);
 }
 for(int i = 0; i < total_staff; i++) //create threads for staffs
 {
 staff_id[i] = i + 1;
 pthread_create(&staff[i], NULL, staff_thread, (void*) &staff_id[i]);
 }
 pthread_create(&coordinator,NULL,coordinator_thread,NULL); //create thread for coordinator
 //join threads, to connect the threads to main program
 for(int i =0; i < total_customer; i++) pthread_join(customers[i],NULL);
 for(int i =0; i < total_staff; i++) pthread_join(staff[i],NULL);
 pthread_join(coordinator, NULL);
}
```

## Output:

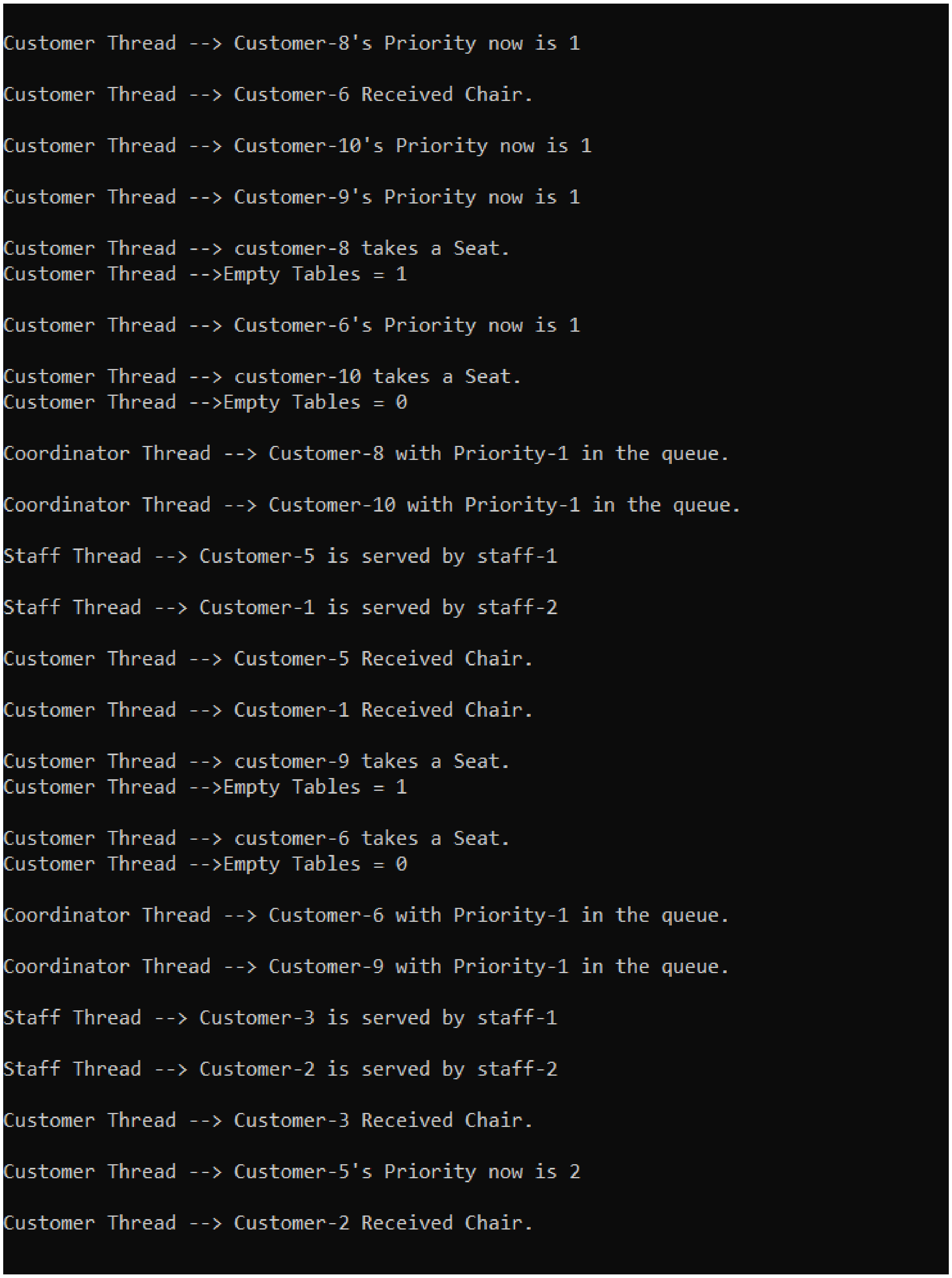


"F:\c code\325project.exe"

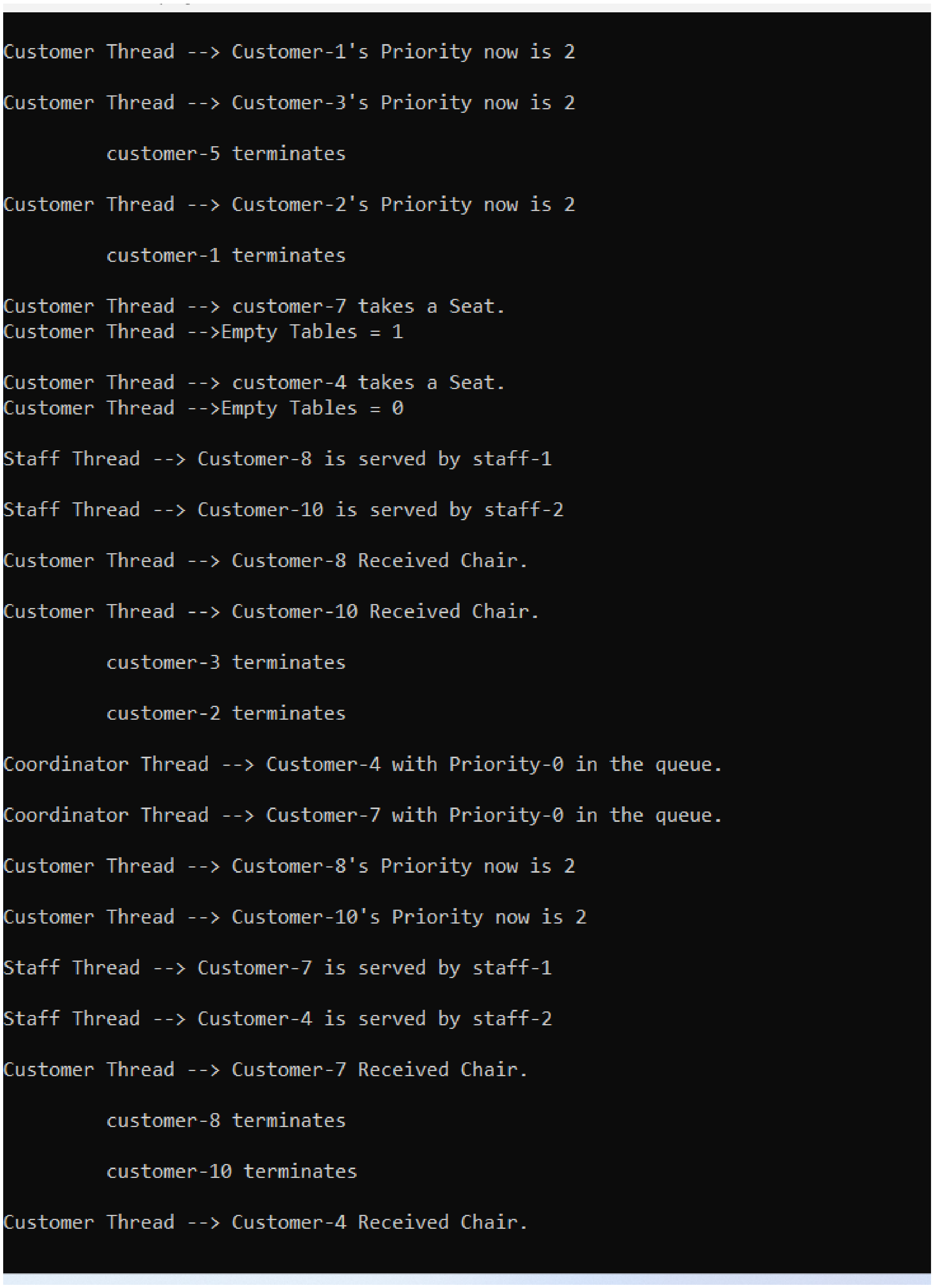
```
Total number of customers: 10
Total number of staff: 2
Total number of Tables: 4
Maximum number of chair a customer can get: 2

Customer Thread --> customer-1 takes a Seat.
Customer Thread -->Empty Tables = 3

Customer Thread --> customer-5 takes a Seat.
Customer Thread -->Empty Tables = 2

Customer Thread --> customer-3 takes a Seat.
Customer Thread -->Empty Tables = 1

Customer Thread --> customer-2 takes a Seat.
Customer Thread -->Empty Tables = 0

Coordinator Thread --> Customer-1 with Priority-0 in the queue.

Coordinator Thread --> Customer-2 with Priority-0 in the queue.

Coordinator Thread --> Customer-3 with Priority-0 in the queue.

Coordinator Thread --> Customer-5 with Priority-0 in the queue.

Customer Thread --> customer-8 takes a Seat.
Customer Thread -->Empty Tables = 1

Customer Thread --> customer-10 takes a Seat.
Customer Thread -->Empty Tables = 0

Staff Thread --> Customer-1 is served by staff-1

Staff Thread --> Customer-5 is served by staff-2

Customer Thread --> Customer-1 Received Chair.

Coordinator Thread --> Customer-8 with Priority-0 in the queue.

Coordinator Thread --> Customer-10 with Priority-0 in the queue.

Customer Thread --> Customer-5 Received Chair.

Customer Thread --> Customer-1's Priority now is 1

Customer Thread --> customer-9 takes a Seat.
Customer Thread -->Empty Tables = 1

Customer Thread --> customer-6 takes a Seat.
```



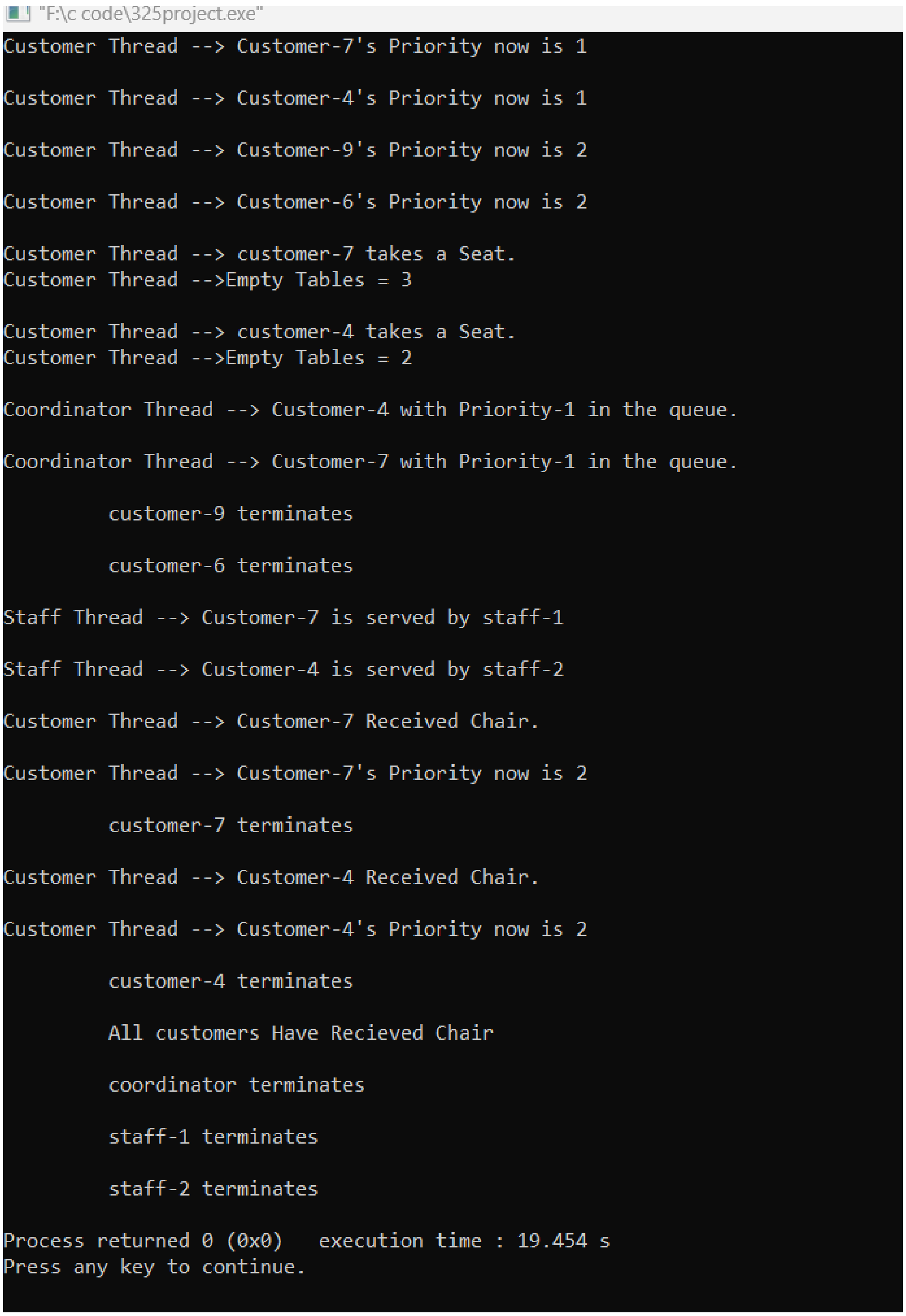
```
Customer Thread --> customer-6 takes a Seat.
Customer Thread -->Empty Tables = 0

Staff Thread --> Customer-3 is served by staff-1

Staff Thread --> Customer-2 is served by staff-2

Customer Thread --> Customer-3 Received Chair.

Customer Thread --> Customer-2 Received Chair.

Customer Thread --> Customer-5's Priority now is 1

Coordinator Thread --> Customer-6 with Priority-0 in the queue.

Coordinator Thread --> Customer-9 with Priority-0 in the queue.

Customer Thread --> Customer-3's Priority now is 1

Customer Thread --> Customer-2's Priority now is 1

Customer Thread --> customer-5 takes a Seat.
Customer Thread -->Empty Tables = 1

Customer Thread --> customer-1 takes a Seat.
Customer Thread -->Empty Tables = 0

Coordinator Thread --> Customer-1 with Priority-1 in the queue.

Coordinator Thread --> Customer-5 with Priority-1 in the queue.

Staff Thread --> Customer-8 is served by staff-1

Staff Thread --> Customer-10 is served by staff-2

Customer Thread --> Customer-8 Received Chair.

Customer Thread --> Customer-10 Received Chair.

Customer Thread --> customer-3 takes a Seat.
Customer Thread -->Empty Tables = 1

Customer Thread --> customer-2 takes a Seat.
Customer Thread -->Empty Tables = 0

Coordinator Thread --> Customer-2 with Priority-1 in the queue.

Coordinator Thread --> Customer-3 with Priority-1 in the queue.
```



```
Customer Thread --> Customer-8's Priority now is 1

Customer Thread --> Customer-6 Received Chair.

Customer Thread --> Customer-10's Priority now is 1

Customer Thread --> Customer-9's Priority now is 1

Customer Thread --> customer-8 takes a Seat.
Customer Thread -->Empty Tables = 1

Customer Thread --> Customer-6's Priority now is 1

Customer Thread --> customer-10 takes a Seat.
Customer Thread -->Empty Tables = 0

Coordinator Thread --> Customer-8 with Priority-1 in the queue.

Coordinator Thread --> Customer-10 with Priority-1 in the queue.

Staff Thread --> Customer-5 is served by staff-1

Staff Thread --> Customer-1 is served by staff-2

Customer Thread --> Customer-5 Received Chair.

Customer Thread --> Customer-1 Received Chair.

Customer Thread --> customer-9 takes a Seat.
Customer Thread -->Empty Tables = 1

Customer Thread --> customer-6 takes a Seat.
Customer Thread -->Empty Tables = 0

Coordinator Thread --> Customer-6 with Priority-1 in the queue.

Coordinator Thread --> Customer-9 with Priority-1 in the queue.

Staff Thread --> Customer-3 is served by staff-1

Staff Thread --> Customer-2 is served by staff-2

Customer Thread --> Customer-3 Received Chair.

Customer Thread --> Customer-5's Priority now is 2

Customer Thread --> Customer-2 Received Chair.
```



```
Customer Thread --> Customer-1's Priority now is 2

Customer Thread --> Customer-3's Priority now is 2

         customer-5 terminates

Customer Thread --> Customer-2's Priority now is 2

         customer-1 terminates

Customer Thread --> customer-7 takes a Seat.
Customer Thread -->Empty Tables = 1

Customer Thread --> customer-4 takes a Seat.
Customer Thread -->Empty Tables = 0

Staff Thread --> Customer-8 is served by staff-1

Staff Thread --> Customer-10 is served by staff-2

Customer Thread --> Customer-8 Received Chair.

Customer Thread --> Customer-10 Received Chair.

         customer-3 terminates

         customer-2 terminates

Coordinator Thread --> Customer-4 with Priority-0 in the queue.

Coordinator Thread --> Customer-7 with Priority-0 in the queue.

Customer Thread --> Customer-8's Priority now is 2

Customer Thread --> Customer-10's Priority now is 2

Staff Thread --> Customer-7 is served by staff-1

Staff Thread --> Customer-4 is served by staff-2

Customer Thread --> Customer-7 Received Chair.

         customer-8 terminates

         customer-10 terminates

Customer Thread --> Customer-4 Received Chair.
```



"F:\c code\325project.exe"

```
Customer Thread --> Customer-7's Priority now is 1

Customer Thread --> Customer-4's Priority now is 1

Customer Thread --> Customer-9's Priority now is 2

Customer Thread --> Customer-6's Priority now is 2

Customer Thread --> customer-7 takes a Seat.
Customer Thread -->Empty Tables = 3

Customer Thread --> customer-4 takes a Seat.
Customer Thread -->Empty Tables = 2

Coordinator Thread --> Customer-4 with Priority-1 in the queue.

Coordinator Thread --> Customer-7 with Priority-1 in the queue.

         customer-9 terminates

         customer-6 terminates

Staff Thread --> Customer-7 is served by staff-1

Staff Thread --> Customer-4 is served by staff-2

Customer Thread --> Customer-7 Received Chair.

Customer Thread --> Customer-7's Priority now is 2

         customer-7 terminates

Customer Thread --> Customer-4 Received Chair.

Customer Thread --> Customer-4's Priority now is 2

         customer-4 terminates

         All customers Have Recieved Chair

         coordinator terminates

         staff-1 terminates

         staff-2 terminates

Process returned 0 (0x0)   execution time : 19.454 s
Press any key to continue.
```

## **Conclusion**:

Context switching is the process of storing a process's context or state such that it can be reloaded and execution continued from the same point as before. A "Context Switch" is the act of transitioning from one process to another. A computer system often has multiple duties to complete. So, if one activity requires some I/O, we want to start the I/O operation before moving on to the next process. We'll go through it again later. We should pick up where we left off when we return to a process. For all intents and purposes, this process should never be aware of the switch, and it should appear as if it were the only one in the system