

|  |  |
| --- | --- |
| **CSE-325**  **Operating Systems**  **Section: 02** | |
| **Project No: 03** | |
| **Project Name: Restaurant Table Reservation System** | |
| **Submitted Date:** 27 May, 2024 | |
| **Semester & Year:**  Spring 2024 | **Group No:**  03 |
| **Under the Guidance of:**  Dr. Md. Nawab Yousuf Ali Professor  Department of Computer  Science and Engineering  East West University, Bangladesh | Shamima Sharmin Ananna  ID: 2022-1-60-148  Roll: 27  Tasnuva Tasnim Nova  ID: 2021-2-60-266  Roll: 30  Taniz Fatema Jarin  ID: 2022-1-60-065  Roll: 26 |

# Introduction:

The Restaurant Table Reservation System is designed to efficiently manage table reservations in a restaurant setting using a multithreading approach. The system utilizes POSIX threads, semaphores, and mutexes to ensure synchronized access to shared resources and provide a seamless experience for customers wishing to reserve and release tables. This project was developed in C and tested on the Ubuntu operating system.

# Objectives:

The objectives of this project are given below:

* **Efficient Table Management:** To manage the reservation and release of tables dynamically as customers arrive and leave.
* **Concurrency Handling:** To ensure that multiple customers can access the system concurrently without conflicts or data inconsistencies.
* **User Interaction:** To allow users to cancel or modify their reservations through an interactive command-line interface.
* **Resource Optimization:** To use synchronization primitives like semaphores and mutexes for optimal resource management and to prevent race conditions.

# Methodology:

* **Multithreading:** Each customer is represented as a thread. The threads perform actions like reserving and releasing tables concurrently.
* **Semaphore Usage:** A semaphore is used to keep track of the number of available tables, ensuring that no more than the available number of tables can be reserved at any time.
* **Mutex Locks:** Mutexes are used to protect critical sections where shared resources (like the table status array) are modified to prevent race conditions.
* **Dynamic Memory Allocation:** Memory is dynamically allocated for customer data to ensure that each customer's reservation details are maintained correctly.
* **User Interaction:** A command-line interface is provided to allow users to interact with the system, making or modifying reservations.

# Source Code:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

int tablesAvailable;

sem\_t tableSem;

pthread\_mutex\_t mutex;

int \*tableStatus;

int tableNum;

typedef struct {

long id;

int tableNum;

} Customer;

void \*leaveTable(void \*arg) {

Customer \*customer = (Customer \*) arg;

long id = customer->id;

int tableNum = customer->tableNum;

sleep(2);

pthread\_mutex\_lock(&mutex);

tableStatus[tableNum - 1] = -1;

tablesAvailable++;

printf("Customer[%ld]\tLeft the table. Tables Available: %d\n", id, tablesAvailable);

pthread\_mutex\_unlock(&mutex);

sem\_post(&tableSem);

free(customer);

pthread\_exit(NULL);

}

Customer \*reserveTable(void \*arg) {

long id = (long) arg + 1;

printf("Customer[%ld]\tArrives and wants to reserve a table.\n", id);

sem\_wait(&tableSem);

pthread\_mutex\_lock(&mutex);

int reservedTable = -1;

for (int i = 0; i < tableNum; i++) {

if (tableStatus[i] == -1) {

tableStatus[i] = id;

tablesAvailable--;

reservedTable = i + 1;

printf("Customer[%ld]\tReserved table %d. Tables Available: %d. Tables: ", id, reservedTable, tablesAvailable);

for (int j = 0; j < tableNum; j++) {

if (tableStatus[j] == -1) {

printf("%d ", j + 1);

}

}

printf("\n");

break;

}

}

pthread\_mutex\_unlock(&mutex);

Customer \*customer = (Customer \*) malloc(sizeof(Customer));

customer->id = id;

customer->tableNum = reservedTable;

return customer;

}

void displayAvailableTables() {

pthread\_mutex\_lock(&mutex);

printf("Available Tables: ");

int available = 0;

for (int i = 0; i < tableNum; i++) {

if (tableStatus[i] == -1) {

printf("%d ", i + 1);

available++;

}

}

if (available == 0) {

printf("No tables available");

}

printf("\n");

pthread\_mutex\_unlock(&mutex);

}

int main() {

int customerNum;

printf("Enter number of customers: ");

scanf("%d", &customerNum);

printf("Enter number of tables: ");

scanf("%d", &tableNum);

if (customerNum > tableNum) {

printf("Not enough tables available for all customers.\n");

return 0;

}

printf("Restaurant Table Reservation System\n");

printf("Customers: %d, Tables: %d\n", customerNum, tableNum);

printf("---------------------------------------------\n");

sem\_init(&tableSem, 0, tableNum);

pthread\_mutex\_init(&mutex, NULL);

pthread\_t customerThread[customerNum];

tableStatus = (int\*) malloc(tableNum \* sizeof(int));

tablesAvailable = tableNum;

for (int i = 0; i < tableNum; i++) {

tableStatus[i] = -1;

}

Customer \*customers[customerNum];

for(long id = 0; id < customerNum; ++id) {

customers[id] = reserveTable((void\*)id);

usleep(500);

}

while (1) {

int choice, customerId, newTableNum;

printf("Enter 1 to cancel a reservation, 2 to modify table number, 0 to proceed: ");

scanf("%d", &choice);

if (choice == 0) break;

switch (choice) {

case 1:

printf("Enter customer ID to cancel reservation: ");

scanf("%d", &customerId);

pthread\_mutex\_lock(&mutex);

int cancelled = 0;

for (int i = 0; i < tableNum; i++) {

if (tableStatus[i] == customerId) {

tableStatus[i] = -1;

tablesAvailable++;

sem\_post(&tableSem);

printf("Reservation for customer[%d] cancelled. Table %d is now available.\n", customerId, i + 1);

cancelled = 1;

break;

}

}

if (!cancelled) {

printf("No reservation found for customer[%d].\n", customerId);

}

pthread\_mutex\_unlock(&mutex);

break;

case 2:

printf("Enter customer ID to modify table reservation: ");

scanf("%d", &customerId);

displayAvailableTables();

printf("Enter new table number: ");

scanf("%d", &newTableNum);

pthread\_mutex\_lock(&mutex);

int modified = 0;

for (int i = 0; i < tableNum; i++) {

if (tableStatus[i] == customerId) {

if (tableStatus[newTableNum - 1] == -1) {

tableStatus[newTableNum - 1] = customerId;

tableStatus[i] = -1;

printf("Customer[%d] moved from table %d to table %d.\n", customerId, i + 1, newTableNum);

modified = 1;

} else {

printf("Table %d is not available.\n", newTableNum);

}

break;

}

}

if (!modified && tableStatus[newTableNum - 1] != -1) {

printf("No reservation found for customer[%d].\n", customerId);

}

pthread\_mutex\_unlock(&mutex);

break;

default:

printf("Invalid choice. Please try again.\n");

}

displayAvailableTables();

}

for(int id = 0; id < customerNum; ++id) {

if (customers[id] != NULL && customers[id]->tableNum != -1 && customers[id]->id != 1) {

pthread\_create(&customerThread[id], NULL, leaveTable, (void\*)customers[id]);

}

}

for(int id = 0; id < customerNum; ++id) {

if (customers[id] != NULL && customers[id]->tableNum != -1 && customers[id]->id != 1) {

pthread\_join(customerThread[id], NULL);

}

}

sem\_destroy(&tableSem);

pthread\_mutex\_destroy(&mutex);

free(tableStatus);

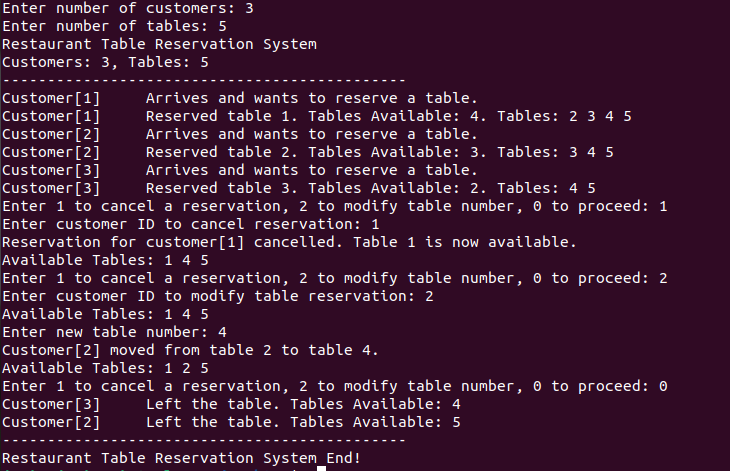
printf("---------------------------------------------\n");

printf("Restaurant Table Reservation System End!\n");

return 0;

}

**Output:**

****

# Advantages:

* Utilizes semaphores and mutexes for optimal utilization of available tables.
* Ensures that no more than the available number of tables can be reserved at any given time, preventing overbooking.
* Allows multiple customers to reserve and release tables simultaneously.
* Uses multithreading to handle concurrent access to the reservation system, improving efficiency during peak times.
* Provides an interactive command-line interface for customers to reserve, cancel, or modify their reservations.
* Offers real-time updates on table availability, enhancing user experience and satisfaction.
* Dynamically allocates memory for customer data and table status arrays, accommodating varying numbers of customers and tables.
* Supports flexible reservation modifications, allowing customers to change their table assignments as needed.
* Easily scalable to accommodate additional tables or customers by adjusting input parameters.
* Can handle increased customer traffic without compromising performance or reliability.
* Ensures efficient use of system resources by releasing tables promptly after customers leave.
* Prevents resource wastage by freeing tables when reservations are canceled or modified.
* Reduces wait times for customers by efficiently managing table reservations and minimizing conflicts.
* Enhances customer satisfaction by providing a seamless and responsive reservation experience.

# Limitations:

* The system currently operates only through a command-line interface, which may not be user-friendly for all users.
* The dining time is fixed, which may not accurately reflect real-world scenarios.
* The system has limited error handling, particularly regarding invalid user inputs.
* The current implementation is designed for a single system and does not support distributed operations.

# Conclusion:

The Restaurant Table Reservation System successfully demonstrates the use of multithreading and synchronization techniques to manage table reservations efficiently. By using semaphores and mutexes, the system ensures consistent and conflict-free access to shared resources. While the system has certain limitations, it provides a solid foundation for further development and enhancements, such as a graphical user interface and support for distributed environments. This project highlights the importance of concurrency control in real-time reservation systems and serves as a practical application of threading and synchronization concepts in C programming on the Ubuntu operating system.

**END**