



**Department of CSE**

**Course Code**

**CSE325**

**Course Title**

**Operating Systems**

**Project Report**

WILL & LOU'S CAFÉ PROBLEM

**Section : 02 Semester : Fall2024**



**Submitted By**

|  |  |  |
| --- | --- | --- |
| Group : 9 | | |
| Name | **ID** | **Roll** |
| Abrar Hossain Zahin | **2022-2-60-040** | **22** |



**Submitted To**

**Dr. Md. Nawab Yousuf Ali**

**Professor**

**Department of Computer Science and Engineering**

**Date of Report Submitted : 22 January, 2025**

Table of Contents :

|  |  |  |
| --- | --- | --- |
| Serial | Index Content | Page No. |
| 1 | **Acknowledgement** | **3** |
| 2 | **Abstract** | **3** |
| 3 | **Problem Statement** | **3** |
| 4 | **Project Description** | **4** |
| 5 | **Flow Chart** | **5** |
| 6 | **Code** | **6-8** |
| 7 | **Code Explanation** | **9-11** |
| 8 | **Limitation** | **12** |
| 9 | **Conclusion** | **12** |

**Acknowledgement**

We would like to express our deepest gratitude to Dr. Nawab Yousuf Ali Sir for his invaluable guidance, encouragement, and support throughout the development of this project, *"Will & Lou's Café Simulator."* His expertise and constructive feedback have been instrumental in enhancing our understanding of **multithreading**, **synchronization**, & **resource management**. Thank you, Sir, for inspiring us to explore and excel in the field of computer science.

**Abstract**

The program simulates the operations of a charming café, "Will & Lou's," where customers enjoy with limited tables and the unique ability to borrow books while savoring their treats. The simulation efficiently handles concurrent customer activities and ensuring smooth operations through effective synchronization techniques. **Semaphore-based Resource Management** ensures table availability for customers while managing a queue of waiting customers. **Threaded Customer Simulation e**ach customer’s actions (seating, borrowing books, and leaving) are handled in individual threads for realistic simulation. **Mutex-based Synchronization:** Prevents data inconsistencies in shared resources like book inventory and table statuses. **Interactive User Interface** Allows dynamic addition of customers, providing flexibility in testing various scenarios. The simulation emphasizes resource sharing, fair access, and a delightful experience reminiscent of the café’s charming atmosphere.

**Problem Statement**

‘Will and Lou’ are opened a charming café where customers can enjoy reading books while dining. To provide a seamless experience for every customer. So we’ve designed a system with the following guidelines :

1. A customer may enter the café only if a table is available for them to sit.
2. The system will maintain a record of book borrowing and returning to efficiently manage the book inventory.
3. Once customers have finished reading, eating, or drinking, they should vacate the table, allowing new customers to use it.

**Project Description**

We created a program to address the café management challenge. The program simulates the café environment, which includes a fixed number of tables and a fixed inventory of books. In the simulation, tables, books, and customers are treated as objects, each with its own attributes.

**Book** Bookname Bookstatus

**Table** Tablenumber Tablestatus

**Customer**

table

books

book\_count

Initially, the program initializes all the tables and books into their respective **global arrays** using threads. Afterward, it begins loading customers and ensures proper synchronization as they occupy tables, borrow and return books, and eventually leave the café.

**Customer enters and takes seats**

**Customer returns book, vacates the seat then exits the café**

**Customer borrow books**

This project leverages **semaphores**, **mutex locks**, and **threads** to handle real-world scenarios involving concurrency and resource management. The simulator Imitation the café's unique features, ensuring a smooth experience for every visitor.

**Flow Chart**

**A screenshot of a diagram

Description automatically generated**

**Code**

|  |  |
| --- | --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <pthread.h>  #include <semaphore.h>  #include <unistd.h>  #include <time.h>  #define MAX\_BOOK 10  #define MAX\_TABLE 5  #define MAX\_CUSTOMER 200  *// Data structures*  typedef **struct**{  **char** Bookname[200];  **int** Bookstatus; *// 1: Available, 0: Borrowed*  } Book;  typedef **struct**{  **int** Tablenumber;  **int** Tablestatus; *// 1: Available, 0: Occupied*  } Table;  typedef **struct**{  **int** table;  **int** books[3];  **int** book\_count;  } Customer;  *// Global variables*  Table tables[MAX\_TABLE];  Book book\_inventory[MAX\_BOOK];  Customer customers[MAX\_CUSTOMER];  **int** total\_tables = 0, total\_books = 0, total\_customers = 0;  *// Synchronization tools*  sem\_t table\_semaphore;  sem\_t customer\_semaphores[MAX\_CUSTOMER];  pthread\_mutex\_t table\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_mutex\_t book\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_mutex\_t customer\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  *// Function prototypes*  **void** initialize\_resources();  **void** load\_books();  **void** load\_tables();  **void** handle\_customer(**int** id);  **void** \*customer\_thread(**void** \*arg);  **void** show\_menu();  *// Initialize resources*  **void** initialize\_resources(){  sem\_init(&table\_semaphore, 0, 1);  for (**int** i = 0; i < MAX\_CUSTOMER; i++)  { sem\_init(&customer\_semaphores[i], 0, 0);  }  }  *// Load book inventory*  **void** load\_books(){  for (**int** i = 0; i < MAX\_BOOK; i++){ snprintf(book\_inventory[i].Bookname, sizeof(book\_inventory[i].Bookname), "Book %d", i + 1);  book\_inventory[i].Bookstatus = 1;  total\_books++;  } printf("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  printf("\n\* %d books loaded into the inventory \*\n", total\_books);  }  *// Load tables*  **void** load\_tables(){  for (**int** i = 0; i < MAX\_TABLE; i++){  tables[i].Tablenumber = i + 1;  tables[i].Tablestatus = 1;  total\_tables++;  } printf("\*====================================\*\n");  printf("\* %d tables loaded and available \*\n", total\_tables); printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  }  *// Customer handling*  **void** handle\_customer(**int** id){  printf("Customer %d searching for a table...\n", id);  while (1){  sem\_wait(&table\_semaphore); pthread\_mutex\_lock(&table\_mutex);  **int** table\_found = 0;  for (**int** i = 0; i < total\_tables; i++){  if (tables[i].Tablestatus == 1)  { tables[i].Tablestatus = 0;  customers[id].table = tables[i].Tablenumber; customers[id].book\_count = 0;  total\_customers++;  table\_found = 1;  printf("Customer %d assigned to table %d.\n", id, tables[i].Tablenumber);  break;  }  } | pthread\_mutex\_unlock(&table\_mutex);  sem\_post(&table\_semaphore);  if (table\_found)  break;  printf("#----------------------------------------#\n");  printf("No tables available. Customer %d waiting...\n", id);  sleep(1);  }  pthread\_mutex\_lock(&book\_mutex);  **int** books\_picked = 0;  for (**int** i = 0; i < total\_books && books\_picked < 3; i++){  if (book\_inventory[i].Bookstatus == 1)  {  book\_inventory[i].Bookstatus = 0;  customers[id].books[books\_picked++] = i;  }  }  customers[id].book\_count = books\_picked;  pthread\_mutex\_unlock(&book\_mutex);  printf("Customer %d borrowed %d books and is seated at table %d.\n", id, books\_picked, customers[id].table);  sleep(2);  pthread\_mutex\_lock(&table\_mutex);  tables[customers[id].table - 1].Tablestatus = 1;  pthread\_mutex\_unlock(&table\_mutex);  pthread\_mutex\_lock(&book\_mutex);  for (**int** i = 0; i < books\_picked; i++)  {  book\_inventory[customers[id].books[i]].Bookstatus = 1;  }  pthread\_mutex\_unlock(&book\_mutex);  printf("Customer %d has left table %d.\n", id, customers[id].table);  }  *// Customer thread function*  **void** \*customer\_thread(**void** \*arg){  **int** id = \*(**int** \*)arg;  free(arg);  handle\_customer(id);  return NULL;  }  *// Show menu*  **void** show\_menu(){  **int** choice;  while (1)  {  printf("\n|\*\*\*\*\*\*| Welcome To W&L's Cafe |\*\*\*\*\*\*|\n");  printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  printf("\*\*\*\*\*\*\*\*\* Chosse Option :- \*\*\*\*\*\*\*\*\*\n");  printf("\*\*\*\*\*\*\*\*\* 1. Add customers \*\*\*\*\*\*\*\*\*\n\*\*\*\*\*\*\*\*\* 2. Exit \*\*\*\*\*\*\*\*\*\n");  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  printf("\nEnter your choice : ");  scanf("%d", &choice);  switch (choice)  {  case 1:  {  **int** n;  printf("Simulation will be started\n");  printf("Enter number of customers : ");  scanf("%d", &n);  pthread\_t threads[n];  for (**int** i = 0; i < n; i++){  **int** \*id = malloc(sizeof(**int**));  \*id = i + total\_customers;  pthread\_create(&threads[i], NULL, customer\_thread, id);  }  for (**int** i = 0; i < n; i++)  {  pthread\_join(threads[i], NULL);  }  break;  }  case 2: printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  printf("\* %2d Customer Visited Today \*\n", total\_customers);  printf("\*\* Cafe Closed For Today ! \*\*"); printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");  return;  default:  printf("Invalid choice. Please try again.\n");  }  }  }  **int** main(){  srand(time(NULL));  initialize\_resources();  load\_books();  load\_tables();  show\_menu();  return 0;  } |
| **Input :** | **Output :** |

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**Code Explanation**

Firstly I declared the data structures. Then global variables, shared variables semphores and mutex.

|  |  |
| --- | --- |
| A screenshot of a computer code  Description automatically generated | A screenshot of a computer code  Description automatically generated |

All the semaphore are being initiated in initialize\_resources() function. Prepares semaphores and mutexes for use.

A close-up of text

Description automatically generated

The load\_books() and load\_tables() functions work by initializing their respective resources: the book inventory and the seating tables. In load\_books(), a loop iterates through the book\_inventory array, assigning each book & marking it as available by setting its status to 1. Similarly, load\_tables() iterates through the tables array, assigning each table a unique number and marking it as available. Both functions update global counters to track the total resources initialized and provide user feedback through printed messages, confirming the successful setup of books and tables for the simulation.

A computer code with colorful text

Description automatically generated

The handle\_customer() function manages a customer's visit by first searching for an available table, using a **semaphore** and **mutex** to ensure thread-safe access. Once a table is found, it is marked as occupied, and the customer's table and book data are initialized. The customer then borrows up to **3** available books, updating their status to borrowed. After a simulated stay sleep(2), the customer releases the table and returns the books, restoring their availability. The function uses synchronization tools like semaphores and mutexes to handle **concurrent** access to shared resources efficiently and ensure proper customer flow.

|  |  |
| --- | --- |
| A screen shot of a computer code  Description automatically generated | A screenshot of a computer program  Description automatically generated |

The \*customer\_thread(*void* \*arg)function retrieves the customer ID from the argument, frees the memory, and calls handle\_customer() to manage the customer's actions. It enables multithreaded handling of customer interactions.

A computer code with text

Description automatically generated

This below code handles the simulation of customer interactions. It prompts the user to enter the number of customers (n) and creates an array of pthread\_t to store thread identifiers. For each customer, it dynamically allocates memory for their ID and starts a thread using pthread\_create() to run the customer\_thread() function. After all threads are created, it waits for their completion using pthread\_join() ensuring all customer interactions finish before proceeding.

A computer code with text

Description automatically generated with medium confidence

A green and purple rectangles with black text

Description automatically generated

Firstly Customer 0: process with sem\_wait and pthread\_mutex\_lock being used to ensure exclusive access to a table. Customer 0 searches for a free table and occupies it ( Tablestatus

= 0). Then Customer 1 is blocked, waiting for the semaphore to become available as no free table exists.

A green and purple rectangles with black text

Description automatically generated

Then Once seated, Customer 0 proceeds to borrow books (Bookstatus= 0 for up to 3 books). Mutex ensures thread-safe access to the book inventory then Customer 1 finds a free table and occupies it while Customer 0 is handling books.

A diagram of a customer service

Description automatically generated with medium confidence

Finally, After using the table and books, Customer 0 leaves, releasing the table (Tablestatus = 1) and resetting book statuses (Bookstatus= 1) then Customer 1 proceeds to borrow books in the same synchronized manner after acquiring a table.

**Limitation**

1. **No Timeout for Waiting :** Customers wait indefinitely for tables, with no timeout mechanism to leave the queue after prolonged waiting.
2. **Concurrency Conflicts :** Insufficient optimization for managing concurrent access to resources (e.g., overlapping book assignments under heavy load).
3. **Unoptimized Queue Management :** The system lacks customer prioritization or termination logic if tables remain unavailable.
4. **Fixed Resource Limits :** The hardcoded number of books and tables restricts scalability. Allowing dynamic resource adjustment could overcome this.
5. **Limited Error Handling :** The program doesn't validate or handle exceptional scenarios like invalid inputs or system resource exhaustion effectively.

**Conclusion**

Our system provides a fundamental multi-threaded approach to managing tables and books, utilizing semaphores and mutexes effectively. It ensures customers are assigned tables and books and allows them to exit after their allotted time. However, the lack of dynamic interfaces limits the system's potential. Enhancing these aspects could make the simulation more realistic and efficient, aligning it more closely with a real-world café environment.

**“THE END”**