## Mini-Project

**Course Title: Operating Systems**

**Course Code: CSE325**

**Semester: Spring 2025**

**Section: 02**

**Group No: 08**

## Project Title(8): Sweet Harmony

## Submitted to

**Dr. Md. Nawab Yousuf Ali**

Professor

Department of Computer Science & Engineering

East West University

## Submitted by

| Name | ID |
|---|---|
| Muhammad Abdul Kayum | 2023-2-60-082 |
| Afia Mahpara | 2023-2-60-011 |
| Md. Hasib Ali | 2023-3-60-186 |

**Date of Submission: 20th May 2025**

## Problem Description

The simulated bakery has a fixed number of tables (seats). Red and blue customer threads arrive and attempt to enter. The core fairness requirement is that the number of red and blue customers inside should remain balanced (differ by at most one) at all times. This prevents one color from continuously occupying the bakery while the other waits indefinitely. Without synchronization, threads could concurrently update the count of occupied seats or violate the balance rule, leading to a race condition. In this context, unsynchronized threads could allow too many of one color to enter, or miscount available tables. Thus the problem demands careful control: limit total occupants to table count, and enforce alternation (fairness) between colors.

## Objective

In our following project, we will implement a bakery with multiple tables using a multithreaded C program. Two types of customer threads (red and blue) attempt to enter the bakery, subject to limited seating capacity. Synchronization primitives (POSIX threads and semaphores) are employed to enforce mutual exclusion and balance. The goal is to ensure fairness – neither color dominates seating – while avoiding race conditions on shared state. The implementation uses counting semaphores to represent available tables and mutexes to protect shared counters. In summary, the program demonstrates thread synchronization, mutual exclusion, and fairness enforcement in a bounded-resource scenario.

## 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 the three main topics. Threads, process and semaphore.

**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:

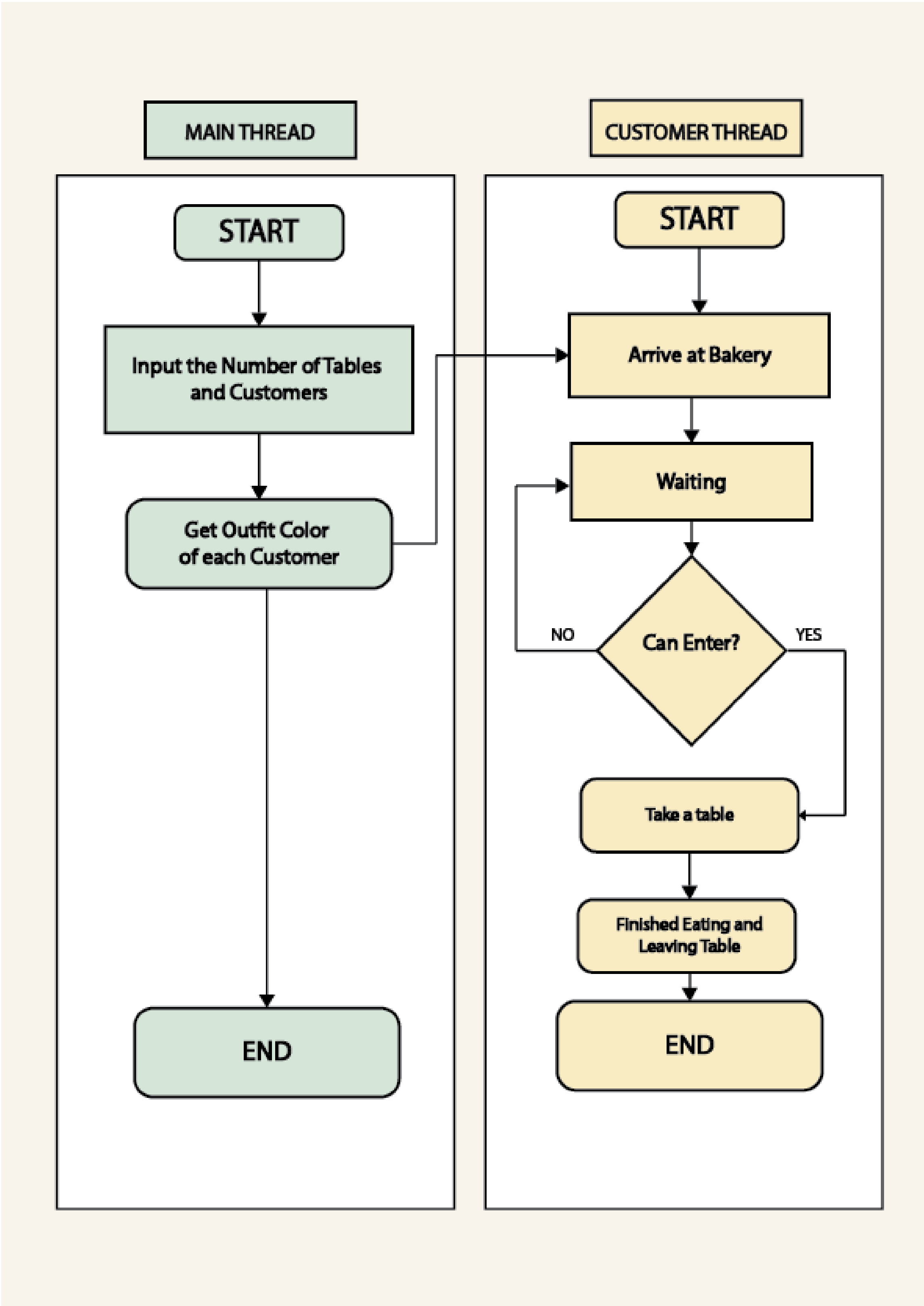
1. **Binary Semaphore –**
   This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
2. **Counting Semaphore –**
   Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Sweet Harmony is a quaint bakery in Pastelville known for its delicious pastries and warm ambiance. To preserve its pleasant atmosphere, the bakery enforces a special rule: at any moment, the number of customers wearing red outfits must equal those wearing blue outfits inside the bakery. This report details the design and implementation of a C program that simulates Sweet Harmony's customer flow, seating, and departure using concurrency primitives

## Proposed Solution

We use a combination of counting semaphores and mutexes to meet these requirements. A counting semaphore (`table_sem`) is initialized to the number of tables, so that only that many customers can be seated simultaneously. Each customer thread performs `sem_wait(&table_sem)` before entering, which blocks if all tables are taken. A mutex protects two shared counters (`redInside`, `blueInside`) that track the current number of seated reds and blues. Threads check whether entry would violate balance and wait accordingly on color-specific condition variables. This solution guarantees mutual exclusion and fair scheduling to prevent starvation.

# Flow Chart


MAIN THREAD
START
Input the Number of Tables and Customers
Get Outfit Color of each Customer
END
CUSTOMER THREAD
START
Arrive at Bakery
Waiting
Can Enter?
NO
YES
Take a table
Finished Eating and Leaving Table
END

## C Program Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#define MAX_CUSTOMERS 100
#define MAX_TABLES 20

int red_count = 0;
int blue_count = 0;
int total_tables = 0;
int bakery_capacity = 0;
sem_t table_sem;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_equal = PTHREAD_COND_INITIALIZER;

typedef struct {
    int id;
    char color[6];
} Customer;

void get_string_input(const char* prompt, char* buffer, size_t size)
{
    printf("%s", prompt);
    if (fgets(buffer, size, stdin) != NULL) {
        size_t ln = strlen(buffer) - 1;
        if (ln<size && buffer[ln] == '\n') buffer[ln]='\0';
    }
}

int get_int_input(const char* prompt, int min, int max)
{
    char buffer[32];
    int value;
    int valid;
    do {
        valid = 1;
        get_string_input(prompt, buffer, sizeof(buffer));
        for (size_t i=0; buffer[i]!='\0'; i++)
            if (!isdigit(buffer[i])) {
                valid = 0;
                break;
            }
        if(!valid){
             printf("Invalid input! Please enter a number.\n");
             continue;
        }
        value = atoi(buffer);
        if (value <min || value > max) {
            printf("Invalid input! Please enter a number between %d and %d.\n", min, max);
            valid = 0;
        }
    } while (!valid);
    return value;
}
```

```c
void get_color_input(char* color)
{
    while (1) {
        get_string_input("  Enter outfit color (red/blue): ", color, 8);
        for (int i = 0; color[i]; i++) color[i] = tolower(color[i]);
        if (strcmp(color, "red") == 0 || strcmp(color, "blue") == 0) break;
        printf("  Invalid color! Please enter 'red' or 'blue'.\n");
    }
}

void* customer_routine(void* arg) {
    Customer* cust = (Customer*)arg;
    pthread_mutex_lock(&count_mutex);
    printf("[Queue] %s customer %d arrived, waiting to enter.\n",
           cust->color,cust->id);

    while (1)
    {
        int current_inside = red_count + blue_count;
        int can_enter = 0;

        if (current_inside < bakery_capacity)
        {
            if (strcmp(cust->color, "red") == 0) {
                if (red_count <= blue_count) {
                    can_enter = 1;
                }
            } else {
                if (blue_count <= red_count) {
                     can_enter = 1;
                }
            }
        }

        if (can_enter) break;
        printf("[Wait ] %s customer %d waiting outside (Red: %d, Blue: %d, Inside: %d/%d capacity).\n",
                cust->color, cust->id, red_count, blue_count, current_inside, bakery_capacity);
        pthread_cond_wait(&cond_equal, &count_mutex);
    }
    if (strcmp(cust->color, "red") == 0) red_count++;
    else blue_count++;
    int current_inside_after_entry = red_count + blue_count;

    printf("[Enter] %s customer %d ENTERED bakery. (Red: %d, Blue: %d, Inside: %d/%d capacity).\n",
           cust->color, cust->id, red_count, blue_count, current_inside_after_entry, bakery_capacity);

    pthread_mutex_unlock(&count_mutex);

    sem_wait(&table_sem);

    int eating_time = rand() % 3 + 1;
    sleep(eating_time);

    printf("[Leave] %s customer %d finished eating, leaving table (%d sec).\n", cust->color, cust->id, eating_time);
    sem_post(&table_sem);

    pthread_mutex_lock(&count_mutex);
    if (strcmp(cust->color, "red") == 0) red_count--;
    else blue_count--;
    int current_inside_after_leaving = red_count + blue_count;

    printf("[Exit ] %s customer %d LEFT bakery. (Red: %d, Blue: %d, Inside: %d/%d capacity).\n",
            cust->color, cust->id, red_count, blue_count, current_inside_after_leaving, bakery_capacity);

    pthread_cond_broadcast(&cond_equal);
    pthread_mutex_unlock(&count_mutex);
    free(cust);
    pthread_exit(NULL);
}
```

```c
int main(){
    srand(time(NULL));
    printf("\n=== Sweet Harmony Bakery Simulation ===\n");

    total_tables = get_int_input("Enter the number of tables in the bakery (1-20): ", 1, MAX_TABLES);
    bakery_capacity = total_tables * 2;
    int n_customers = get_int_input("Enter the total number of customers arriving (1-100): ", 1, MAX_CUSTOMERS);

    Customer* customer_data[n_customers];
    pthread_t threads[n_customers];

    printf("Enter outfit color for each customer:\n");
    for (int i = 0; i<n_customers; i++) {
        char color[8];
        printf("Customer %d:\n", i + 1);
        get_color_input(color);

        customer_data[i] = malloc(sizeof(Customer));
        if (customer_data[i] == NULL) {
            perror("Failed to allocate memory for customer data");
            for(int j = 0; j < i; ++j) free(customer_data[j]);
            return 1;
        }
        customer_data[i]->id = i + 1;
        strcpy(customer_data[i]->color, color);
    }
    if (sem_init(&table_sem, 0, total_tables) != 0) {
        perror("Semaphore initialization failed");
        for(int i = 0; i < n_customers; ++i) free(customer_data[i]);
        return 1;
    }

    printf("\n--- Starting Customer Arrivals (Bakery Capacity: %d, Tables: %d) ---\n", bakery_capacity, total_tables);
    for (int i = 0; i < n_customers; i++) {
        if (pthread_create(&threads[i], NULL, customer_routine, customer_data[i]) != 0) {
            perror("Failed to create thread");
            sem_destroy(&table_sem);
            pthread_mutex_destroy(&count_mutex);
            pthread_cond_destroy(&cond_equal);
            for(int j = 0; j < i; ++j) free(customer_data[j]);
            exit(1);
        }
        usleep(100000 + rand()%300000);
    }

    for (int i = 0; i < n_customers; i++)
        pthread_join(threads[i], NULL);

    sem_destroy(&table_sem);
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&cond_equal);
    printf("\n=== Simulation Complete! All customers have visited Sweet Harmony. ===\n");
    return 0;
}
```

# Function description

## 1. get_string_input() function

```c
void get_string_input(const char* prompt, char* buffer, size_t size)
{
    printf("%s", prompt);
    if (fgets(buffer, size, stdin) != NULL) {
        size_t ln = strlen(buffer) - 1;
        if (ln<size && buffer[ln] == '\n') buffer[ln]='\0';
    }
}
```

This function takes user input as a string with a custom prompt. It ensures the newline character at the end is removed. It is mainly used to bring clean user input for things like colors.

## 2. `get_int_input()` function

```c
int get_int_input(const char* prompt, int min, int max)
{
    char buffer[32];
    int value;
    int valid;
    do {
        valid = 1;
        get_string_input(prompt, buffer, sizeof(buffer));
        for (size_t i=0; buffer[i]!='\0'; i++)
            if (!isdigit(buffer[i])) {
                valid = 0;
                break;
            }
        if(!valid){
             printf("Invalid input! Please enter a number.\n");
             continue;
        }
        value = atoi(buffer);
        if (value <min || value > max) {
            printf("Invalid input! Please enter a number between %d and %d.\n", min, max);
            valid = 0;
        }
    } while (!valid);
    return value;
}
```

This function is used to take numeric input from the user. It keeps asking the user until a valid number within the given range is entered. It prevents invalid or non-numeric entries and helps keep the program robust.

## 3. `get_color_input()` function

```c
void get_color_input(char* color)
{
    while (1) {
        get_string_input("  Enter outfit color (red/blue): ", color, 8);
        for (int i = 0; color[i]; i++) color[i] = tolower(color[i]);
        if (strcmp(color, "red") == 0 || strcmp(color, "blue") == 0) break;
        printf("  Invalid color! Please enter 'red' or 'blue'.\n");
    }
}
```

This function asks the user to enter a customer's outfit color, which must be either "red" or "blue". It loops until a valid color is entered, making sure only these two inputs are allowed (case-insensitive).

## 4. `customer_routine()` function

```c
void* customer_routine(void* arg) {
    Customer* cust = (Customer*)arg;
    pthread_mutex_lock(&count_mutex);
    printf("[Queue] %s customer %d arrived, waiting to enter.\n",
           cust->color,cust->id);

    while (1)
    {
        int current_inside = red_count + blue_count;
        int can_enter = 0;

        if (current_inside < bakery_capacity)
        {
            if (strcmp(cust->color, "red") == 0) {
                if (red_count <= blue_count) {
                    can_enter = 1;
                }
            } else {
                if (blue_count <= red_count) {
                     can_enter = 1;
                }
            }
        }

        if (can_enter) break;
        printf("[Wait ] %s customer %d waiting outside (Red: %d, Blue: %d, Inside: %d/%d capacity).\n",
               cust->color, cust->id, red_count, blue_count, current_inside, bakery_capacity);
        pthread_cond_wait(&cond_equal, &count_mutex);
    }
    if (strcmp(cust->color, "red") == 0) red_count++;
    else blue_count++;
    int current_inside_after_entry = red_count + blue_count;

    printf("[Enter] %s customer %d ENTERED bakery. (Red: %d, Blue: %d, Inside: %d/%d capacity).\n",
           cust->color, cust->id, red_count, blue_count, current_inside_after_entry, bakery_capacity);

    pthread_mutex_unlock(&count_mutex);

    sem_wait(&table_sem);

    int eating_time = rand() % 3 + 1;
    sleep(eating_time);

    printf("[Leave] %s customer %d finished eating, leaving table (%d sec).\n", cust->color, cust->id, eating_time);
    sem_post(&table_sem);

    pthread_mutex_lock(&count_mutex);
    if (strcmp(cust->color, "red") == 0) red_count--;
    else blue_count--;
    int current_inside_after_leaving = red_count + blue_count;

    printf("[Exit ] %s customer %d LEFT bakery. (Red: %d, Blue: %d, Inside: %d/%d capacity).\n",
           cust->color, cust->id, red_count, blue_count, current_inside_after_leaving, bakery_capacity);

    pthread_cond_broadcast(&cond_equal);
    pthread_mutex_unlock(&count_mutex);
    free(cust);
    pthread_exit(NULL);
}
```

This is the thread function for each customer. It simulates the customer's entire visit to the bakery. The steps include:

- Waiting outside if the bakery is full or if the red-blue count is unbalanced.
- Entering the bakery once allowed.

- Waiting for an available table.
- Simulating eating by sleeping for a few seconds.
- Leaving the table and exiting the bakery.

## 5. main()

```c
int main(){
    srand(time(NULL));
    printf("\n=== Sweet Harmony Bakery Simulation ===\n");

    total_tables = get_int_input("Enter the number of tables in the bakery (1-20): ", 1, MAX_TABLES);
    bakery_capacity = total_tables * 2;
    int n_customers = get_int_input("Enter the total number of customers arriving (1-100): ", 1, MAX_CUSTOMERS);

    Customer* customer_data[n_customers];
    pthread_t threads[n_customers];

    printf("Enter outfit color for each customer:\n");
    for (int i = 0; i<n_customers; i++) {
        char color[8];
        printf("Customer %d:\n", i + 1);
        get_color_input(color);

        customer_data[i] = malloc(sizeof(Customer));
        if (customer_data[i] == NULL) {
            perror("Failed to allocate memory for customer data");
            for(int j = 0; j < i; ++j) free(customer_data[j]);
            return 1;
        }
        customer_data[i]->id = i + 1;
        strcpy(customer_data[i]->color, color);
    }
    if (sem_init(&table_sem, 0, total_tables) != 0) {
        perror("Semaphore initialization failed");
        for(int i = 0; i < n_customers; ++i) free(customer_data[i]);
        return 1;
    }

    printf("\n--- Starting Customer Arrivals (Bakery Capacity: %d, Tables: %d) ---\n", bakery_capacity, total_tables);
    for (int i = 0; i < n_customers; i++) {
        if (pthread_create(&threads[i], NULL, customer_routine, customer_data[i]) != 0) {
            perror("Failed to create thread");
            sem_destroy(&table_sem);
            pthread_mutex_destroy(&count_mutex);
            pthread_cond_destroy(&cond_equal);
            for(int j = 0; j < i; ++j) free(customer_data[j]);
            exit(1);
        }
        usleep(100000 + rand()%300000);
    }
```

```c
    for (int i = 0; i < n_customers; i++)
        pthread_join(threads[i], NULL);

    sem_destroy(&table_sem);
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&cond_equal);
    printf("\n=== Simulation Complete! All customers have visited Sweet Harmony. ===\n");
    return 0;
}
```

The `main()` function handles the setup and overall coordination of the simulation. It:

- Takes user input for the number of tables and the number of customers.
- Asks for the outfit color of each customer.
- Initializes synchronization tools like semaphores and mutexes.
- Creates threads for each customer and simulates their arrival with small delays.
- Waits for all customers to finish their visit.
- Cleans up all resources after the simulation ends.

## Key Concepts Used

- **Threads (`pthread`)**: Each customer is handled by a separate thread to simulate real-time arrival and activity.
- **Mutex**: Ensures only one thread can update shared counters (`red_count`, `blue_count`) at a time.
- **Condition Variable**: Makes customers wait outside if the red-blue count condition is not met.
- **Semaphore**: Used to manage the number of available tables inside the bakery.

## Testing and Observations:

- **Test Cases** 1:
    1. Fewer tables than the number of customers.
    2. Random color sequences.

```
=== Sweet Harmony Bakery Simulation ===
Enter the number of tables in the bakery (1-20): 1
Enter the total number of customers arriving (1-100): 4
Enter outfit color for each customer:
Customer 1:
  Enter outfit color (red/blue): red
Customer 2:
  Enter outfit color (red/blue): red
Customer 3:
  Enter outfit color (red/blue): blue
Customer 4:
  Enter outfit color (red/blue): blue
```

**OUTPUT**

```
--- Starting Customer Arrivals (Bakery Capacity: 2, Tables: 1) ---
[Queue] red customer 1 arrived, waiting to enter.
[Enter] red customer 1 ENTERED bakery. (Red: 1, Blue: 0, Inside: 1/2 capacity).
[Queue] red customer 2 arrived, waiting to enter.
[Wait ] red customer 2 waiting outside (Red: 1, Blue: 0, Inside: 1/2 capacity).
[Queue] blue customer 3 arrived, waiting to enter.
[Enter] blue customer 3 ENTERED bakery. (Red: 1, Blue: 1, Inside: 2/2 capacity).
[Leave] red customer 1 finished eating, leaving table (3 sec).
[Exit ] red customer 1 LEFT bakery. (Red: 0, Blue: 1, Inside: 1/2 capacity).
[Enter] red customer 2 ENTERED bakery. (Red: 1, Blue: 1, Inside: 2/2 capacity).
[Queue] blue customer 4 arrived, waiting to enter.
[Wait ] blue customer 4 waiting outside (Red: 1, Blue: 1, Inside: 2/2 capacity).
[Leave] blue customer 3 finished eating, leaving table (3 sec).
[Exit ] blue customer 3 LEFT bakery. (Red: 1, Blue: 0, Inside: 1/2 capacity).
[Enter] blue customer 4 ENTERED bakery. (Red: 1, Blue: 1, Inside: 2/2 capacity).
[Leave] red customer 2 finished eating, leaving table (3 sec).
[Exit ] red customer 2 LEFT bakery. (Red: 0, Blue: 1, Inside: 1/2 capacity).
[Leave] blue customer 4 finished eating, leaving table (3 sec).
[Exit ] blue customer 4 LEFT bakery. (Red: 0, Blue: 0, Inside: 0/2 capacity).

=== Simulation Complete! All customers have visited Sweet Harmony. ===
```

- **Test Cases** 2:
    1. Fewer customers than the number of tables.
    2. Random color sequences.

```
=== Sweet Harmony Bakery Simulation ===
Enter the number of tables in the bakery (1-20): 5
Enter the total number of customers arriving (1-100): 3
Enter outfit color for each customer:
Customer 1:
  Enter outfit color (red/blue): red
Customer 2:
  Enter outfit color (red/blue): blue
Customer 3:
  Enter outfit color (red/blue): blue
```

**OUTPUT**

```
--- Starting Customer Arrivals (Bakery Capacity: 10, Tables: 5) ---
[Queue] red customer 1 arrived, waiting to enter.
[Enter] red customer 1 ENTERED bakery. (Red: 1, Blue: 0, Inside: 1/10 capacity).
[Queue] blue customer 2 arrived, waiting to enter.
[Enter] blue customer 2 ENTERED bakery. (Red: 1, Blue: 1, Inside: 2/10 capacity).
[Queue] blue customer 3 arrived, waiting to enter.
[Enter] blue customer 3 ENTERED bakery. (Red: 1, Blue: 2, Inside: 3/10 capacity).
[Leave] red customer 1 finished eating, leaving table (3 sec).
[Exit ] red customer 1 LEFT bakery. (Red: 0, Blue: 2, Inside: 2/10 capacity).
[Leave] blue customer 2 finished eating, leaving table (3 sec).
[Exit ] blue customer 2 LEFT bakery. (Red: 0, Blue: 1, Inside: 1/10 capacity).
[Leave] blue customer 3 finished eating, leaving table (3 sec).
[Exit ] blue customer 3 LEFT bakery. (Red: 0, Blue: 0, Inside: 0/10 capacity).

=== Simulation Complete! All customers have visited Sweet Harmony. ===
```

- **Test Cases** 3:
    1. Equal number of customers and tables.
    2. Random color sequences.

```
=== Sweet Harmony Bakery Simulation ===
Enter the number of tables in the bakery (1-20): 4
Enter the total number of customers arriving (1-100): 4
Enter outfit color for each customer:
Customer 1:
  Enter outfit color (red/blue): blue
Customer 2:
  Enter outfit color (red/blue): blue
Customer 3:
  Enter outfit color (red/blue): blue
Customer 4:
  Enter outfit color (red/blue): red
```

OUTPUT

```
--- Starting Customer Arrivals (Bakery Capacity: 8, Tables: 4) ---
[Queue] blue customer 1 arrived, waiting to enter.
[Enter] blue customer 1 ENTERED bakery. (Red: 0, Blue: 1, Inside: 1/8 capacity).
[Queue] blue customer 2 arrived, waiting to enter.
[Wait ] blue customer 2 waiting outside (Red: 0, Blue: 1, Inside: 1/8 capacity).
[Queue] blue customer 3 arrived, waiting to enter.
[Wait ] blue customer 3 waiting outside (Red: 0, Blue: 1, Inside: 1/8 capacity).
[Leave] blue customer 1 finished eating, leaving table (3 sec).
[Exit ] blue customer 1 LEFT bakery. (Red: 0, Blue: 0, Inside: 0/8 capacity).
[Enter] blue customer 3 ENTERED bakery. (Red: 0, Blue: 1, Inside: 1/8 capacity).
[Wait ] blue customer 2 waiting outside (Red: 0, Blue: 1, Inside: 1/8 capacity).
[Queue] red customer 4 arrived, waiting to enter.
[Enter] red customer 4 ENTERED bakery. (Red: 1, Blue: 1, Inside: 2/8 capacity).
[Leave] blue customer 3 finished eating, leaving table (3 sec).
[Exit ] blue customer 3 LEFT bakery. (Red: 1, Blue: 0, Inside: 1/8 capacity).
[Enter] blue customer 2 ENTERED bakery. (Red: 1, Blue: 1, Inside: 2/8 capacity).
[Leave] red customer 4 finished eating, leaving table (3 sec).
[Exit ] red customer 4 LEFT bakery. (Red: 0, Blue: 1, Inside: 1/8 capacity).
[Leave] blue customer 2 finished eating, leaving table (3 sec).
[Exit ] blue customer 2 LEFT bakery. (Red: 0, Blue: 0, Inside: 0/8 capacity).

=== Simulation Complete! All customers have visited Sweet Harmony. ===
```

## Observations

1. Balance rule enforced: No instance of imbalance inside the bakery.
2. No deadlocks: All customer threads entered, waited, ate, and exited as expected.
3. Efficient resource use: All available tables were properly allocated and released.

## Conclusion

This project demonstrates effective thread synchronization and coordination in a concurrent environment using mutexes, condition variables, and semaphores. The simulation ensures that customers enter and exit the bakery without conflicts or race conditions. Red and blue customers alternate fairly, and critical sections are protected to maintain consistency—giving each thread the illusion of independent operation. The result is a realistic and balanced model of controlled access, reflecting key concurrency principles in a practical, real-world scenario.