Header Files

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

➜ stdio.h: For printf, fgets, etc.

➜ stdlib.h: For malloc, free, atoi, etc.

➜ pthread.h: For threading (creating and joining threads).

➜ semaphore.h: For using POSIX semaphores (sem_t, sem_init, etc.).

➜ unistd.h: For sleep, usleep, etc.

➜ string.h: Included, but unused in this code. Can be removed safely.

Global Variables

```
int RED_COUNT = 3;
int BLUE_COUNT = 3;
int TABLES = 2;
int EATING_TIME = 1;
```

➜ Initial default values for number of red/blue customers, tables, and eating time.

```
int red_inside = 0, blue_inside = 0;
int red_served = 0, blue_served = 0;
```

➔ *_inside: Track how many red/blue customers are inside.

➔ *_served: Track how many have finished eating.

```
pthread_mutex_t mutex;
sem_t table_sem;
```

➔ mutex: Used to protect access to shared variables.

➔ table_sem: Semaphore to control number of available tables.

🔴 Red Customer Thread Function

```
void* red_customer(void* arg)
```

Function that runs for each red customer.

```
int id = *(int*)arg;
free(arg);
```

➔ Retrieves customer ID and frees the memory (allocated in main()).

Entry Logic

while (1)

Loop until the customer is allowed to enter.

pthread_mutex_lock(&mutex);

Lock the mutex to safely check/update shared variables.

if (red_inside < blue_inside)

➔ A red can enter if currently fewer reds than blues.

else if (red_inside == 0 && blue_inside == 0)

➔ Allow entry if it's the first customer.

pthread_mutex_unlock(&mutex);

Release mutex.

printf("🔴 Red %d waiting to enter...\n", id);
usleep(100000);

If not allowed, wait a bit and try again.

Eating Logic

```
sem_wait(&table_sem);
```

Wait until a table is available (decrement semaphore).

```
sleep(EATING_TIME);
```

Simulate eating.

```
pthread_mutex_lock(&mutex);
red_inside--;
red_served++;
pthread_mutex_unlock(&mutex);
```

Update stats after eating.

```
sem_post(&table_sem);
```

Release table (increment semaphore).

🔵 Blue Customer Thread Function

```
void* blue_customer(void* arg)
```

Same logic as red customer, just for blue:

➔ Blue enters if blue_inside < red_inside

➔ Same first-customer logic

➔ Waits for a table, eats, then updates counters.

Input Helper Function

int get_positive_integer(const char* prompt)

➔ Prompts the user for input.

➔ Ensures input is a **positive integer**.

Main Function

int main()

Entry point.

Setup

printf("🍰 Bakery Simulation Setup 🍰\n\n");
TABLES = get_positive_integer(...);

➔ **Ask user for simulation settings.**

Thread Creation

pthread_t red[RED_COUNT], blue[BLUE_COUNT];
pthread_mutex_init(&mutex, NULL);
sem_init(&table_sem, 0, TABLES);

- Create arrays to store thread IDs.
- Initialize mutex and semaphore (TABLES available slots).

Start Red Customer Threads

for (int a = 0; a < RED_COUNT; a++)

- Allocate ID
- Create thread
- Wait 100ms before creating next (for nice pacing)

Start Blue Customer Threads

Same as above.

Wait for All Threads

pthread_join(...);

➔ Main waits for all red and blue threads to finish.

Cleanup

pthread_mutex_destroy(&mutex);
sem_destroy(&table_sem);

➔ Free mutex and semaphore resources.

Summary

printf("\n🥕 All customers served. Bakery closed.\n");

...

Final stats after simulation ends.

Summary of Purpose

This program simulates a bakery where:

- ➔ Customers (red and blue) enter in a fair and alternating manner.
- ➔ They wait for tables (limited resource).
- ➔ Eat for a given time.
- ➔ Exit and update statistics.

It uses:

1. **Threads** to simulate multiple customers.
2. **Semaphores** to control table access.
3. **Mutex** to safely read/write shared counters.