SLEEPING BARBER PROBLEM

DONE BY:

21K-3107	**RAHIMA IRFAN**
21K-3073	**ABDUL ALEEM AHMED**
21K-3063	**KHAJA ABDUL SAMAD**

INTRODUCTION

This code implements a classic computer science problem, known as the Sleeping Barber Problem, using several important concepts in operating systems and concurrent computing. Here are the key components:

Threads: Threads are used to represent barbers and customers. In this case, each barber and customer is running as a separate thread. This allows for concurrent execution, a critical aspect of the problem.

Semaphores: Semaphores are used to control access to the shared resources (the barber chairs and the barber's time) and to synchronize the actions of the barbers and customers. There are three semaphores used in this code:

customer_sem: This semaphore is used to signal the barber when a customer is available. It's incremented each time a customer arrives and takes a seat, and it's decremented each time a barber starts to cut a customer's hair.

barber_sem: This semaphore is used to signal the customer when the barber has finished cutting their hair. It's incremented each time a barber finishes a haircut, and it's decremented each time a customer's hair has been cut.

mutex: This semaphore is used as a mutual exclusion lock to protect the shared data structure (the waiting room queue) from simultaneous access by multiple threads. Without this lock, it's possible that two customers could try to take the same seat at the same time, or a customer could try to take a seat at the same time that a barber is trying to find a customer, leading to race conditions and data inconsistency.

Queue: The queue is a data structure that represents the waiting room chairs. Customers "arrive" by being added to the queue, and barbers "find a customer" by removing them from the queue. The queue operations must be protected by the mutex semaphore to avoid simultaneous access by multiple threads.

Busy Waiting: The barber threads are in a loop where they continually check the customer_sem semaphore to see if a customer is available. This is a form of busy waiting, which can be CPU-intensive but is acceptable here because the barbers have nothing else to do when they're not cutting hair.

Sleep and Wakeup Mechanisms: The sleep() function is used to simulate the time it takes for a barber to cut a customer's hair. The semaphore wait() and post() operations are used to put threads to sleep (when they're waiting for a resource) and wake them up (when the resource becomes available).

In summary, this code demonstrates how to use threads, semaphores, and queues to implement a multi-threaded simulation with shared resources, mutual exclusion, synchronization, and busy waiting.

PROBLEMS:

Yes, starvation can still occur even if you're using semaphores and mutexes without a queue. Semaphores and mutexes are synchronization primitives that are used to ensure that processes or threads don't interfere with each other when accessing shared resources. They don't inherently provide any fairness guarantees or prevent starvation.

For example, consider a scenario where you have multiple customers (threads) and one barber (another thread). You use a semaphore to signal when a customer is ready and another semaphore to signal when the barber is ready. You also use a mutex to protect the shared state (e.g., whether the barber is busy or not).

In this setup, if customers are continuously arriving at a rate faster than the barber can serve them, some customers may never get served. The operating system's scheduling algorithm might continuously schedule the newly arriving customers before the ones that have been waiting. This is a form of starvation.

To prevent this, you need to introduce some form of fairness into the system. This is where a queue can be useful. By having the customers join a queue when they arrive and having the barber serve the customers in the order they joined the queue (First-Come, First-Served), you can ensure that all customers eventually get served, preventing starvation.

In the barber's problem, you can prevent starvation by ensuring that all customers (i.e., threads) get served fairly. A queue is a common and effective data structure for implementing fairness because it naturally enforces a First-Come, First-Served (FCFS) policy. However, it's not the only way to achieve fairness.

POSSIBLE METHOD TO OVERCOME IT THROUGH OS CONCEPTS (WHY DIDN'T WE IMPLEMENT IT):

Here are some other methods that could be used:

Priority scheduling: If the OS supports priority scheduling, you could assign a higher priority to older customers. However, this can be complex to implement and may not be supported in all environments.

Time quantum: Another approach could be to implement a round-robin scheduling policy where each customer gets a fixed amount of time (time quantum) with the barber. However, this doesn't map well to the barber problem because a haircut can't be stopped halfway through.

Condition variables: In some scenarios, condition variables can be used to manage access to the resource. A condition variable allows one thread to signal to other threads that a particular condition (such as the barber being free) now holds. By using condition variables, you can create a system where threads wait for their turn to access the resource. However, this still doesn't inherently guarantee fairness, and you may need to use additional data structures or logic to ensure that all threads get a chance to access the resource.

ONLY POSSIBLE METHOD

In general, using a queue is one of the simplest and most effective ways to ensure fairness and prevent starvation in this type of scenario. But the best approach depends on the specific requirements of your system and the capabilities of your operating environment.

CODES:

```
This is the implementation of the simple barber's problem solution
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdbool.h>

sem_t customer_sem;
sem_t barber_sem;
sem_t mutex;

typedef struct {
    int id;
} customer_info;

void *customer(void *arg);
void *barber(void *arg);

int main() {
    int num_customers, num_barbers;

    printf("Enter the number of customers: ");
    scanf("%d", &num_customers);

    printf("Enter the number of barbers: ");
    scanf("%d", &num_barbers);

    pthread_t customer_threads[num_customers];
    pthread_t barber_threads[num_barbers];
    customer_info customer_data[num_customers];

    sem_init(&customer_sem, 0, 0);
```

```
    sem_init(&barber_sem, 0, 0);
    sem_init(&mutex, 0, 1);

    for (int i = 0; i < num_barbers; i++) {
        pthread_create(&barber_threads[i], NULL, barber, NULL);
    }

    for (int i = 0; i < num_customers; i++) {
        customer_data[i].id = i;
        pthread_create(&customer_threads[i], NULL, customer, (void
*)&customer_data[i]);
        sleep(1);
    }

    for (int i = 0; i < num_customers; i++) {
        pthread_join(customer_threads[i], NULL);
    }

    for (int i = 0; i < num_barbers; i++) {
        pthread_cancel(barber_threads[i]);
    }

    sem_destroy(&customer_sem);
    sem_destroy(&barber_sem);
    sem_destroy(&mutex);

    return 0;
}

void *customer(void *arg) {
    customer_info *data = (customer_info *)arg;
    int customer_id = data->id;

    sem_wait(&mutex);
```

```
    printf("Customer %d arrived and is waiting for a haircut\n",
customer_id);
    sem_post(&mutex);

    sem_post(&customer_sem);
    sem_wait(&barber_sem);
    sem_wait(&mutex);
    printf("Customer %d is getting a haircut\n", customer_id);
    sem_post(&mutex);

    return NULL;
}

void *barber(void *arg) {
    while (1) {
        sem_wait(&customer_sem);
        sem_wait(&mutex);
        printf("Barber is cutting hair.\n");
        sem_post(&mutex);
        sleep(3);
        sem_wait(&mutex);
        printf("Barber has finished cutting hair.\n");
        sem_post(&mutex);
        sem_post(&barber_sem);
    }

    return NULL;
}
```

Even though we have used user input for the barber but still in this code, customers arrive and either find an available chair to wait in, or leave if all chairs are occupied. The barbers continuously check if there are customers waiting, and if there are, they give them a haircut.

The potential for starvation arises if there is a high number of customers continuously arriving and a low number of available barbers or chairs. If a customer arrives and all chairs are continuously filled by other customers, they will leave and never receive a haircut. This is a form of starvation, where a thread (in this case a customer) is perpetually denied service.

MODIFIED CODE:

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

sem_t customer_sem;
sem_t barber_sem;
sem_t mutex;

typedef struct {
    int id;
    int num_chairs;
} customer_info;

typedef struct {
    int id;
} barber_info;

int queue[100];
int front = -1;
int rear = -1;

void enqueue(int id);
int dequeue();
bool is_queue_empty();
bool is_queue_full(int num_chairs);

void *customer(void *arg);
void *barber(void *arg);

int main() {

    int num_chairs, num_customers, num_barbers;

    printf("Enter the number of chairs: ");
    scanf("%d", &num_chairs);

    printf("Enter the number of customers: ");
    scanf("%d", &num_customers);

    printf("Enter the number of barbers: ");
    scanf("%d", &num_barbers);
```

```
    pthread_t customer_threads[num_customers];
    pthread_t barber_threads[num_barbers];
    customer_info customer_data[num_customers];
    barber_info barber_data[num_barbers];

    sem_init(&customer_sem, 0, 0);
    sem_init(&barber_sem, 0, 0);
    sem_init(&mutex, 0, 1);

    for (int i = 0; i < num_barbers; i++) {
        barber_data[i].id = i;
        pthread_create(&barber_threads[i], NULL, barber, (void
*)&barber_data[i]);
    }

    for (int i = 0; i < num_customers; i++) {
        customer_data[i].id = i;
        customer_data[i].num_chairs = num_chairs;
        pthread_create(&customer_threads[i], NULL, customer, (void
*)&customer_data[i]);
        sleep(1);
    }

    for (int i = 0; i < num_customers; i++) {
        pthread_join(customer_threads[i], NULL);
    }

    for (int i = 0; i < num_barbers; i++) {
        pthread_cancel(barber_threads[i]);
    }

    sem_destroy(&customer_sem);
    sem_destroy(&barber_sem);
    sem_destroy(&mutex);

    return 0;
}

void *customer(void *arg) {
    customer_info *data = (customer_info *)arg;
    int customer_id = data->id;
    int num_chairs = data->num_chairs;

    sem_wait(&mutex);
    if (!is_queue_full(num_chairs)) {
        enqueue(customer_id);
        printf("Customer %d is waiting. (Total waiting customers:
%d)\n", customer_id, rear - front);
        sem_post(&customer_sem);
        sem_post(&mutex);

        sem_wait(&barber_sem);
        sem_wait(&mutex);
        dequeue();
        printf("Customer %d is getting a haircut. (Total waiting
customers: %d)\n", customer_id, rear - front);
        sem_post(&mutex);
    } else {
```

```
        printf("No chairs available. Customer %d leaves.\n",
customer_id);
        sem_post(&mutex);
    }

    return NULL;
}

void *barber(void *arg) {
    barber_info *data = (barber_info *)arg;
    int barber_id = data->id;

    while (1) {
        sem_wait(&customer_sem);
        sem_post(&barber_sem);
        printf("Barber %d is cutting hair.\n", barber_id);
        sleep(3);
        printf("Barber %d has finished cutting hair.\n", barber_id);
    }

    return NULL;
}

void enqueue(int id) {
    if (rear == -1) {
        front = rear = 0;
        queue[rear] = id;
    } else {
        rear++;
        queue[rear] = id;
    }
}

int dequeue() {
    if (is_queue_empty()) {
        return -1;
    }

    int data = queue[front];
    front++;

    if (front > rear) {
        front = rear = -1;
    }

    return data;
}

bool is_queue_empty() {
    return front == -1;
}

bool is_queue_full(int num_chairs) {
    return rear == num_chairs - 1;
}
```

DIFFERENCE OF CODE AND WHY?

This code implements a classic computer science problem, known as the Sleeping Barber Problem, using several important concepts in operating systems and concurrent computing. Here are the key components:

Threads: Threads are used to represent barbers and customers. In this case, each barber and customer is running as a separate thread. This allows for concurrent execution, a critical aspect of the problem.

Semaphores: Semaphores are used to control access to the shared resources (the barber chairs and the barber's time) and to synchronize the actions of the barbers and customers. There are three semaphores used in this code:

customer_sem: This semaphore is used to signal the barber when a customer is available. It's incremented each time a customer arrives and takes a seat, and it's decremented each time a barber starts to cut a customer's hair.

barber_sem: This semaphore is used to signal the customer when the barber has finished cutting their hair. It's incremented each time a barber finishes a haircut, and it's decremented each time a customer's hair has been cut.

mutex: This semaphore is used as a mutual exclusion lock to protect the shared data structure (the waiting room queue) from simultaneous access by multiple threads. Without this lock, it's possible that two customers could try to take the same seat at the same time, or a customer could try to take a seat at the same

time that a barber is trying to find a customer, leading to race conditions and data inconsistency.

Queue: The queue is a data structure that represents the waiting room chairs. Customers "arrive" by being added to the queue, and barbers "find a customer" by removing them from the queue. The queue operations must be protected by the mutex semaphore to avoid simultaneous access by multiple threads.

Busy Waiting: The barber threads are in a loop where they continually check the customer_sem semaphore to see if a customer is available. This is a form of busy waiting, which can be CPU-intensive but is acceptable here because the barbers have nothing else to do when they're not cutting hair.

Sleep and Wakeup Mechanisms: The sleep() function is used to simulate the time it takes for a barber to cut a customer's hair. The semaphore wait() and post() operations are used to put threads to sleep (when they're waiting for a resource) and wake them up (when the resource becomes available).

In summary, this code demonstrates how to use threads, semaphores, and queues to implement a multi-threaded simulation with shared resources, mutual exclusion, synchronization, and busy waiting.