# The Sleeping Barber Problem

CMPE 312 Lab Project,

Burak ER,122200065
May, 2025.

# Introduction

The Sleeping Barber Problem is a classic synchronization problem in computer science, often used to illustrate issues related to process synchronization in operating systems. The problem describes a barbershop with one barber, one barber chair, and a fixed number of waiting chairs for customers. If there are no customers, the barber goes to sleep. When a customer arrives, they either wake up the barber if he is sleeping, or sit in a waiting chair if the barber is busy. If all waiting chairs are full, the customer leaves the shop.

This problem directly relates to the process synchronization area of operating systems. It models real-life scenarios where multiple processes compete for limited resources, and it emphasizes the need for mechanisms to avoid issues such as deadlock, race conditions, and starvation.

Typical solutions use semaphores and mutexes to coordinate actions between customers and the barber. For example, semaphores may track waiting customers, ensure mutual exclusion when accessing shared data, and manage the sleep/wake logic of the barber. These synchronization tools are fundamental in ensuring efficient and safe execution in concurrent systems.

# Methodology

To ensure proper coordination between the barber and the customers, we use two semaphores and a few shared variables with one mutex. The following variables are defined:

```
waiting = 0              // number of waiting customers
cut_count = 0            // number of haircuts done
skipped_count = 0        // number of customers who left
CHAIRS = 3               // total waiting chairs

semaphore customers = 0  // counts waiting customers
semaphore barber = 0     // signals when barber finishes
mutex = 1                // binary semaphore(pthread_mutex) for mutual exclusion
```

Algorithm Logic (Pseudocode)

Barber Process

```
barber()
{
  while (true) {
      wait(customers)        // block until a customer arrives

      wait(mutex)          //acquire lock
      if (cut_count + skipped_count >= TOTAL_CUSTOMERS and
waiting == 0) {
            signal(mutex)
            break              // all customers handled, barber exits
      }
      waiting = waiting - 1   // one customer leaves waiting area
      signal(mutex)                //release
      // simulate haircut (sleep)
```

```
        cut hair...
            signal(barber)          // notify customer that haircut is done
    }
    }


customer()) {
    wait(mutex) //acquire lock
    if (waiting < CHAIRS) {
        waiting = waiting + 1
   signal(mutex)             //release lock
        signal(customers)       // notify barber

        wait(barber)            // wait until barber finishes
        wait(mutex)      // acquire lock
        cut_count = cut_count + 1
        signal(mutex)       //release lock
    } else {
        skipped_count = skipped_count + 1
        signal(mutex)                 //release lock
    }
}
```

## Explanation of the Semaphores and mutex

- **`mutex`**: Prevents race conditions by ensuring that only one thread at a time can modify the `waiting` variable.

- **`customers`**: Acts as a counter for how many customers are ready and waiting to be served. Customers signal this when they arrive.
- **`barber`**: Used by the barber to signal to one waiting customer that he is ready to provide a haircut.

## Execution Flow

1. **Initialization**:
   Semaphores and mutex are initialized. The barber thread starts and immediately waits on customer_sem — he sleeps until a customer arrives.
2. **Customer Arrival**:
   Customers arrive one by one at random intervals. Each customer checks for available chairs using a mutex-protected waiting variable.
3. **If Space Available**:
   The customer increments waiting, signals the barber (sem_post(customer_sem)), and waits (sem_wait(barber_sem)) until their haircut is done.
4. **If No Space**:
   The customer leaves immediately, increasing skipped_count.
5. **Haircut Process**:
   When signaled, the barber wakes up, decrements waiting, cuts hair (simulated with sleep), then signals the customer (sem_post(barber_sem)).
6. **Termination**:
   After all customers are handled (cut_count + skipped_count == CUSTOMERS), the barber exits and resources are cleaned up.

# Implementation

The `main()` function initializes the semaphores and the mutex. It then creates the barber thread using the `fn_barber()` function and launches (CUSTOMERS  10) customer threads using the `fn_customer()` function at random intervals (between 1 to 3 seconds).

Then:

```
Barber entered his shop
Customer 1 is waiting. Waiting count: 1
```

- The barber thread is started with the `fn_barber()` function. Inside its infinite loop, it blocks on `sem_wait(customer_sem)`, meaning the barber is asleep and passively waiting for a customer.
- When a `customer` thread (e.g., Customer 1) is created, it first calls `pthread_mutex_lock(&mutex)` before incrementing the `waiting` counter, as no other thread should modify this critical section concurrently. After updating the value, it releases the mutex with `pthread_mutex_unlock(&mutex)`.
- Immediately after modifying the shared variable, the customer signals the `customer_sem` using `sem_post(customer_sem)` to wake up the barber, and then waits for the haircut to finish by calling `sem_wait(barber_sem)`.

Then:

```
Barber started cutting
```

- The barber, inside `fn_barber()`, wakes up when `sem_wait(customer_sem)` is triggered, and locks the mutex using `pthread_mutex_lock(&mutex)`.
- After decrementing the `waiting` counter, he releases the mutex with `pthread_mutex_unlock(&mutex)`.
- He then begins cutting the customer's hair, simulated with a random delay between 2 to 5 seconds

.Then:

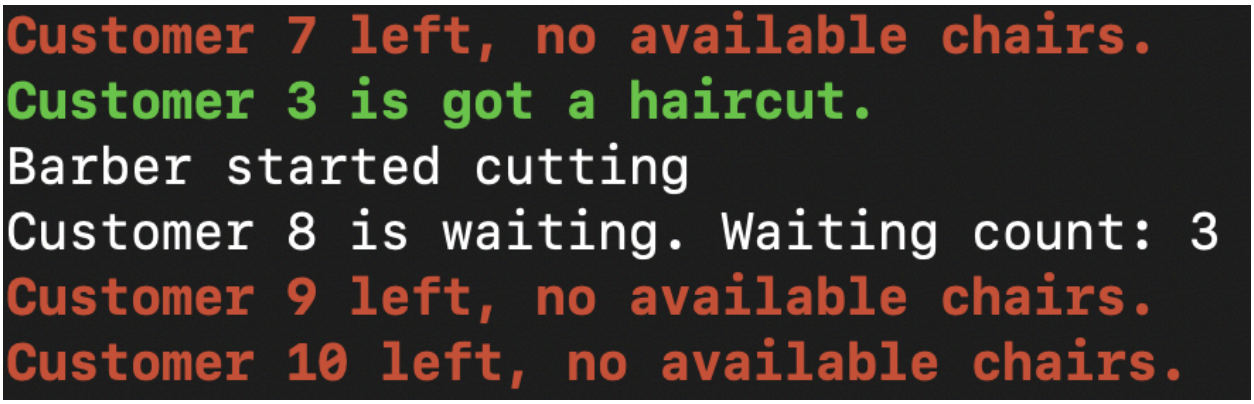
```
Customer 2 is waiting. Waiting count: 1
Customer 3 is waiting. Waiting count: 2
Customer 1 is got a haircut.
Barber started cutting
Customer 4 is waiting. Waiting count: 2
Customer 2 is got a haircut.
Barber started cutting
Customer 5 is waiting. Waiting count: 2
Customer 6 is waiting. Waiting count: 3
```

Customer threads continue to be created at random time intervals using `pthread_create()` along with the `fn_customer()` function.

- Once the barber finishes a haircut, the `fn_barber()` function signals the semaphore with `sem_post(barber_sem)`.
- The corresponding fn_customer() thread, which has been waiting on sem_wait(barber_sem), resumes, acknowledges that its haircut is complete, and

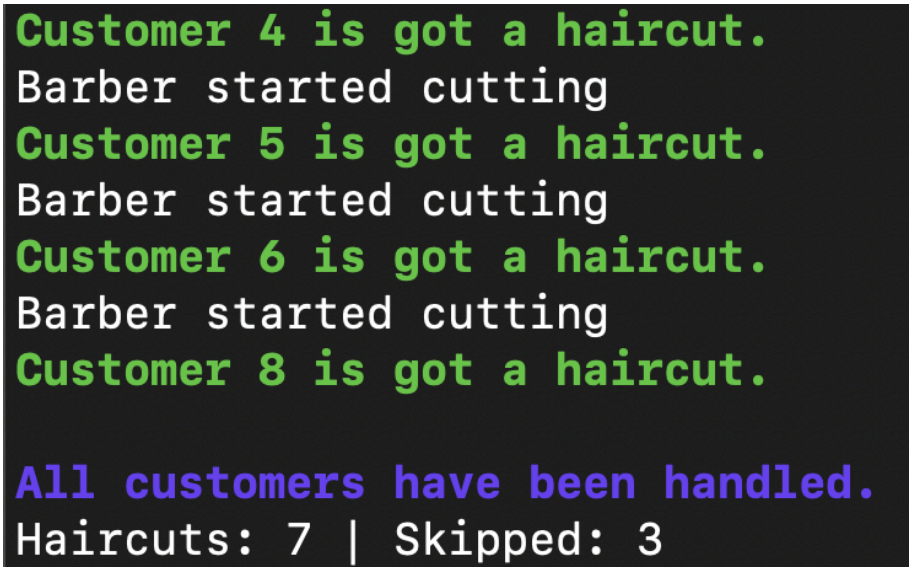
locks the mutex to increment the cut_count variable. It then releases the mutex. (The purpose of the cut_count variable will be explained later.)

Then:




- The `fn_customer()` thread, after acquiring the mutex, first checks whether the `waiting` count is less than the total number of available chairs in the shop (`CHAIRS = 3`).
- If this condition is not met — meaning the waiting area is full — the thread increments the `skipped_count`, releases the mutex, and exits, indicating that the customer has left the barbershop.
- Meanwhile, the haircuts and waiting process continue as previously described.

Then:




- After that, each customer continues to get their haircut in turn. Once there are no more customers left to signal the `customer_sem`, the barber goes back to sleep, waiting again on `sem_wait(customer_sem)`.
- Once the `main()` function ensures that all customer threads have completed using `pthread_join()`, it performs one final `sem_post(customer_sem)` to wake up the still-running `fn_barber()` thread.
- The purpose of this is to allow the barber to reacquire the mutex and verify the condition `cut_count + skipped_count >= CUSTOMERS`. Once this check passes, the barber releases the lock and exits the infinite loop.
- This mechanism ensures that the simulation terminates properly with a finite number of customers, making the program testable and suitable for controlled evaluation.

# Summary

The Sleeping Barber Problem effectively demonstrates key concepts of process synchronization using semaphores and mutexes. It models a scenario where limited resources (chairs and barber) must be managed efficiently among multiple competing processes (customers). This implementation uses semaphores to handle customer arrivals and service, and a mutex to safely manage shared variables.

Through this simulation, issues like race conditions and starvation are avoided by ensuring mutual exclusion and proper signaling. The barber sleeps when idle and only wakes when a customer is ready, while customers either wait, get serviced, or leave depending on resource availability. Once all customers are processed, the barber exits gracefully.

`