

Project Title:

**Sleeping Barber Problem**

Submitted by:

**Qurat Ul Ain 2021-CE-02**

**Noor Fatima 2021-CE-07**

**Syeda Aliya Zahra 2021-CE-16**

**Ansa Aslam 2021-CE-21**

Submitted to:

**Mam Darakshah**

Course:

**Operating System**

Semester:

**5th**

Date of Submission:

**27th Dec, 2023**

**Department of Computer Engineering**

**University of Engineering and Technology, Lahore**

**TABLE OF CONTENTS:**

INTRODUCTION---------------------------------------------------------------3

PROBLEM STATEMENT-----------------------------------------------------3

SOLUTION APPROACH------------------------------------------------------4

FLOWCHART-------------------------------------------------------------------4

FEATURES-----------------------------------------------------------------------4

FUNCTIONAL REQUIREMENTS-------------------------------------------5

WORKING OF THE PROGRAM--------------------------------------------5

APPLICATIONS-----------------------------------------------------------------7

CODING IMPLEMENTATION-----------------------------------------------7

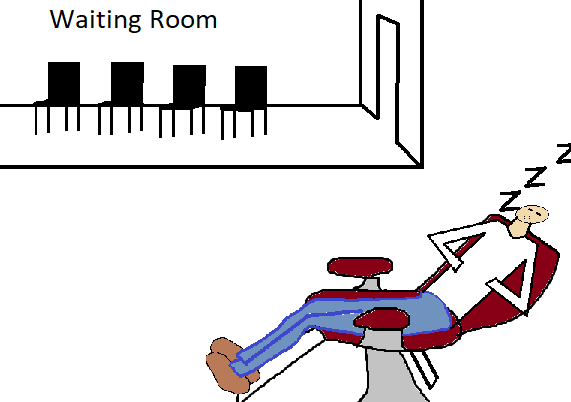
SOFTWARE USED-------------------------------------------------------------12

CONCLUSION------------------------------------------------------------------12

REFERENCES-------------------------------------------------------------------12

**Introduction**

The Sleeping Barber problem is a classic synchronization and concurrency problem often used to illustrate the challenges of resource management in concurrent systems. In this scenario, a barber shop contains a barber and a waiting room with a limited number of seats. Customers enter the shop, take a seat if available, or leave if all seats are occupied. The barber either serves a waiting customer or sleeps if no customers are present. This report aims to explore the implementation of a solution to the Sleeping Barber problem.



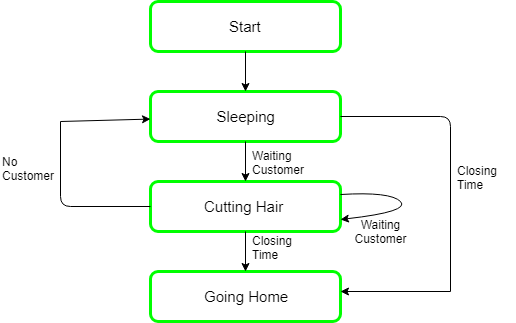
**Problem Statement**

Conceived by Edsger Dijkstra, the problem presents a barbershop with a limited number of chairs and a single barber. The challenge lies in efficiently managing the barber's time and the waiting room capacity, ensuring customers are served without excessive waiting while preventing the barber from remaining idle when there are customers. This scenario is a metaphor for various synchronization issues in operating systems, such as managing threads or processes vying for limited resources.

**Solution Approach**

Our approach involves a multithreaded program in C using the pthreads library. The solution is centered around semaphores, which are used to synchronize access to the barber (a shared resource). The main thread simulates the barbershop's operations, creating threads for customers and the barber. The semaphore mechanism ensures that the barber sleeps when no customers are present and wakes up when a customer arrives, while customers wait or leave based on the availability of chairs.

**Flow Chart**

****

**Features**

* **Dynamic Customer Creation:** The program allows users to input key parameters, including the number of customers, running time, available free seats, and the maximum waiting time for a customer to return.
* **Threaded Implementation:** It utilizes the pthread library to create separate threads for the barber and customers. This mimics a real-world scenario where multiple activities happen concurrently.
* **Semaphore Usage:** The implementation employs semaphores to manage access to shared resources such as the waiting room seats. Semaphores are synchronization primitives that help avoid race conditions.
* **Randomized Wait Times:** To make the simulation more realistic, the program introduces random wait times for both customers and the barber. This randomness reflects the unpredictable nature of customer arrivals and service durations.

**Functional requirements**

**User Input Handling:** The program starts by asking the user for inputs such as the number of customers, the number of available chairs, and the running time of the simulation.

**Dynamic Thread Management:** Threads are created and managed dynamically, representing customers and the barber in the simulation.

**Resource Synchronization:** The program effectively uses semaphores to synchronize access to the shared resources, ensuring orderly operation of the barbershop simulation.

**Working**

The working of the Sleeping Barber program involves the dynamic creation of threads for the barber and customers, along with the synchronization of their actions using semaphores. Here's a detailed breakdown:

**Initialization:**

* The user inputs parameters such as running time, the number of free seats, the number of customers, and the maximum waiting time for a customer to return.
* Semaphores (seatMutex, customers, and smfBarber) are initialized to manage access to shared resources.

**Thread Creation:**

* The main thread creates two types of threads: one for the barber (barberThread) and an array of threads for customers (customersThreads).
* The pthread\_create function is used for thread creation, associating each thread with its respective function (barber or customer).

**Synchronization:**

* Semaphores are crucial for synchronization:
* ***seatMutex:*** Ensures exclusive access to the numberOfFreeSeats variable, preventing race conditions when updating seat counts.
* ***customers:*** Used to signal the barber when a customer is ready to be served.
* ***smfBarber:*** Signals customers that the barber is ready to provide service.
* The sem\_wait and sem\_post functions control access to the critical sections of code, preventing conflicts and ensuring proper sequencing of actions.

**Barber's Routine:**

* The barber thread runs in an infinite loop, representing the continuous operation of the barbershop.
* It waits for a customer (sem\_wait(&customers)) and then acquires the seat mutex (sem\_wait(&seatMutex)) to update the count of free seats.
* The barber serves the customer for a randomly generated time (workingTime), updates the seat count, signals the customer (sem\_post(&smfBarber)), and releases the seat mutex (sem\_post(&seatMutex)).

**Customer's Routine:**

* Each customer thread runs until it successfully receives a haircut.
* The customer acquires the seat mutex and checks if there are available seats.
* If no seats are available, the customer leaves and waits for a random time (waitingTime) before attempting to return.
* If seats are available, the customer decreases the seat count, signals the barber (sem\_post(&customers)), releases the seat mutex, waits for the barber (sem\_wait(&smfBarber)), and receives a haircut.

**Simulation Termination:**

* The main thread sleeps for the specified running time, allowing the simulation to run for designated period.
* After the simulation ends, the program prints the number of customers who received haircuts.

**Applications**

The Sleeping Barber problem serves as an educational tool with broad applications in teaching and understanding concurrent programming concepts. Key applications include:

* **Teaching Concurrency:** The problem illustrates challenges in managing shared resources and synchronizing threads, making it an excellent educational resource for concurrent programming courses.
* **Parallel Computing Concepts**: It provides a practical scenario for learning about parallelism, race conditions, and the use of synchronization mechanisms like semaphores.
* **Real-world Parallelism Analogy:** The barbershop scenario parallels real-world situations where resources need careful management to avoid conflicts, making it applicable to understanding broader concepts in parallel and distributed computing.

**Code Implementation**

The implemented code consists of two main functions:

*barber() and customer().*

*T*he barber() function handles the barber's actions, while customer() represents the behavior of a customer. Semaphores such as seatMutex, customers, and smfBarber are utilized for synchronization. The main function initializes the semaphores, creates the barber thread, and generates customer threads.

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

void barber();

void customer();

sem\_t seatMutex;   // Semaphore for the waiting room

sem\_t customers;   // Semaphore for customers ready to be served

sem\_t smfBarber;   // Semaphore for the barber

int runningTime;

int numberOfFreeSeats;

int customersCount;

int customerWait;

int getHCn;

pthread\_t barberThread;

pthread\_t customersThreads[20];

int main() {

    printf("Enter the running time: ");

    scanf("%d", &runningTime);

    printf("Enter the number of free seats: ");

    scanf("%d", &numberOfFreeSeats);

    printf("Enter the customers count: ");

    scanf("%d", &customersCount);

    printf("Enter the max waiting time for customers: ");

    scanf("%d", &customerWait);

    getHCn = 0;

    printf("\nProgram is beginning\n\n");

    sem\_init(&seatMutex, 0, 1);

    sem\_init(&customers, 0, 0);

    sem\_init(&smfBarber, 0, 0);

    pthread\_create(&barberThread, NULL, barber, NULL);

    printf("Barber has been created.\n");

    for (int i = 1; i <= customersCount; i++) {

        pthread\_create(&customersThreads[i - 1], NULL, customer, (void \*)(long)i);

        printf("Customer %d has been created.\n", i);

    }

    sleep(runningTime);

    printf("\n\nEnd of the day :)\n");

    printf("%d out of %d customers got a haircut.\n", getHCn, customersCount);

    return 0;

}

void barber() {

    int workingTime;

    while (1) {

        sem\_wait(&customers);

        sem\_wait(&seatMutex);

        numberOfFreeSeats += 1;

        workingTime = (rand() % 5) + 1;

        printf("Barber took a new customer. Haircut will take %d seconds.\n", workingTime);

        printf("\tNumber of free seats: %d\n", numberOfFreeSeats);

        sem\_post(&smfBarber);

        sem\_post(&seatMutex);

        sleep(workingTime);

    }

}

void customer(void \*arg) {

int waitingTime;

    int customerID = (int)(long)arg;

    int notEnd = 1;

    while (notEnd == 1) {

        sem\_wait(&seatMutex);

        if (numberOfFreeSeats <= 0) {

            waitingTime = (rand() % customerWait) + 1;

            printf("Customer %d left without a haircut. Will come back after %d seconds.\n", customerID, waitingTime);

            sem\_post(&seatMutex);

            sleep(waitingTime);

        }

else {

            numberOfFreeSeats -= 1;

            printf("Customer %d is waiting.\n", customerID);

            printf("\tNumber of free seats: %d\n", numberOfFreeSeats);

            sem\_post(&customers);

            sem\_post(&seatMutex);

            sem\_wait(&smfBarber);

            printf("Customer %d got a haircut! :)\n", customerID);

            notEnd = 0;

            getHCn += 1;

        }

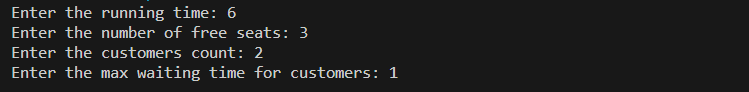
    }

}

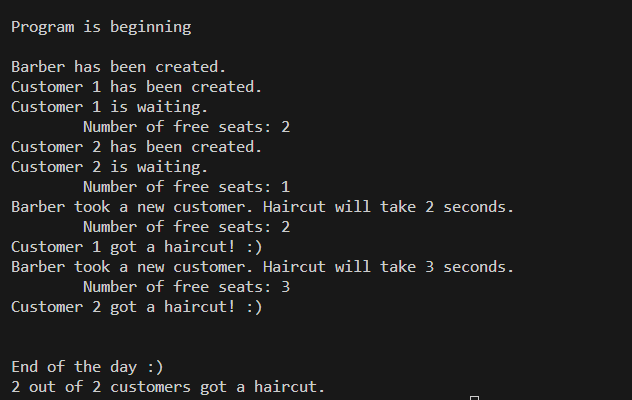
**Input and Output Examples**

**Example 1:**

**Input:**

****

**Output :**

****

**SOFTWARES USED**

**MSYS2 MINGW32:** A software distribution and building platform for Windows that provides a Unix-like environment.

**Visual Studio Code:** An extensible code editor with support for a variety of programming languages, debugging, and version control.

**Conclusion**

The implementation of the Sleeping Barber problem provides a clear and practical example of handling concurrency and synchronization in computing systems. This project not only demonstrates the theoretical aspects of these concepts but also offers a hands-on approach to understanding and resolving complex synchronization challenges in operating systems.

**References**

1. (Downey, A. B., "The Little Book of Semaphores," 2nd ed., Version 2.2.1, Copyright 2016.)