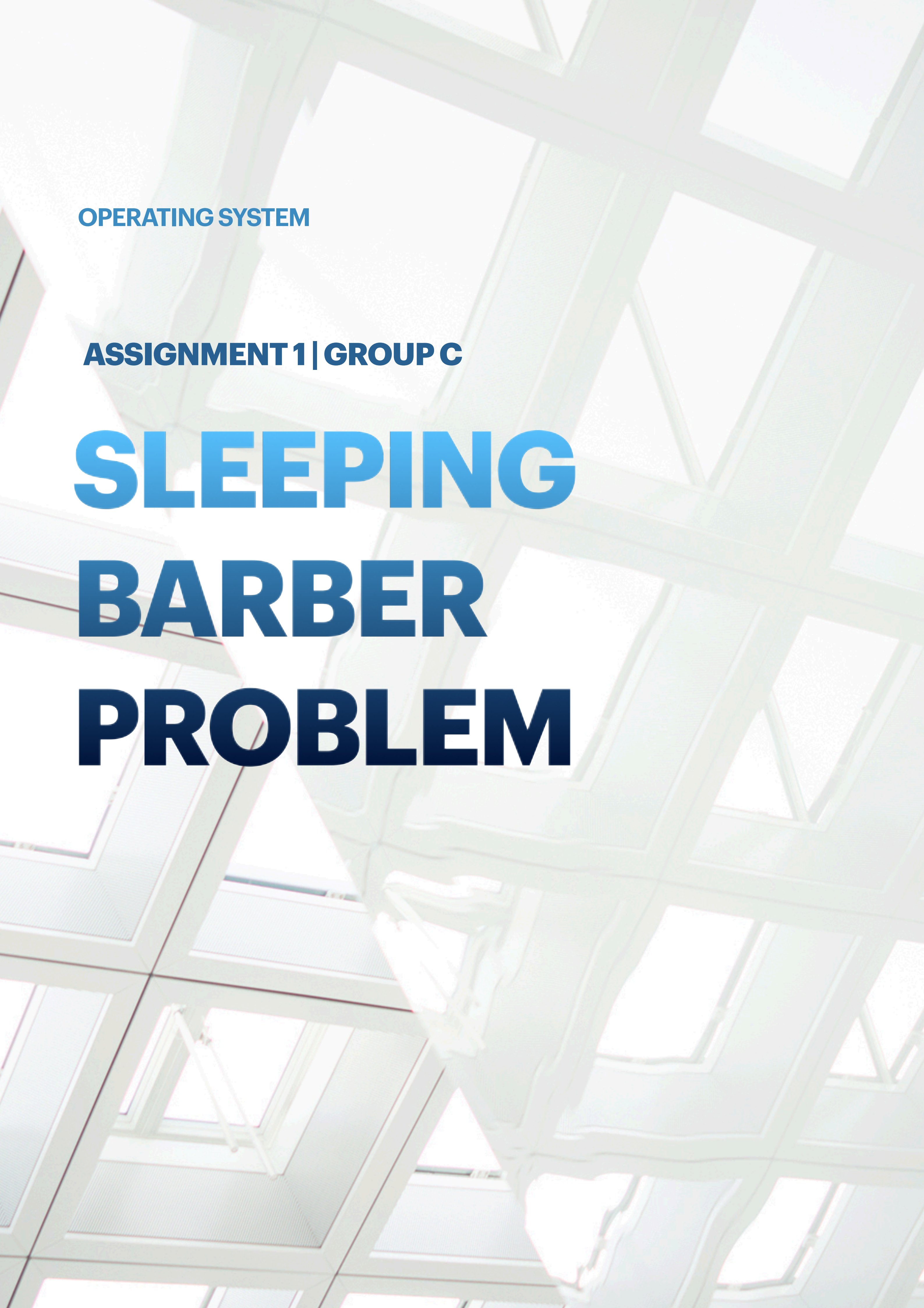


OPERATING SYSTEM

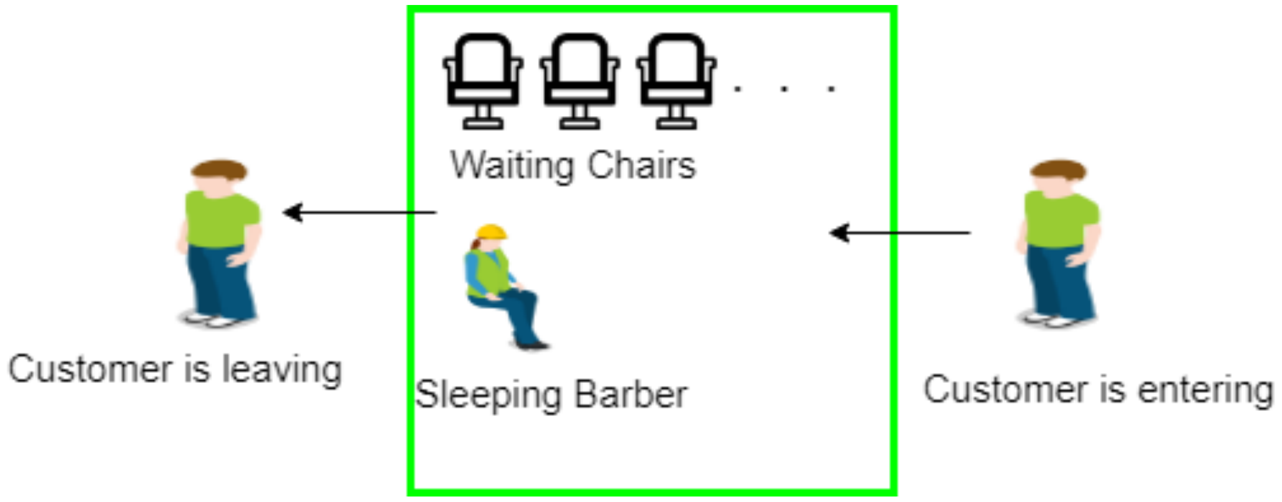
ASSIGNMENT 1 | GROUP C

SLEEPING BARBER PROBLEM

WHAT IS SLEEPING BARBER PROBLEM ?

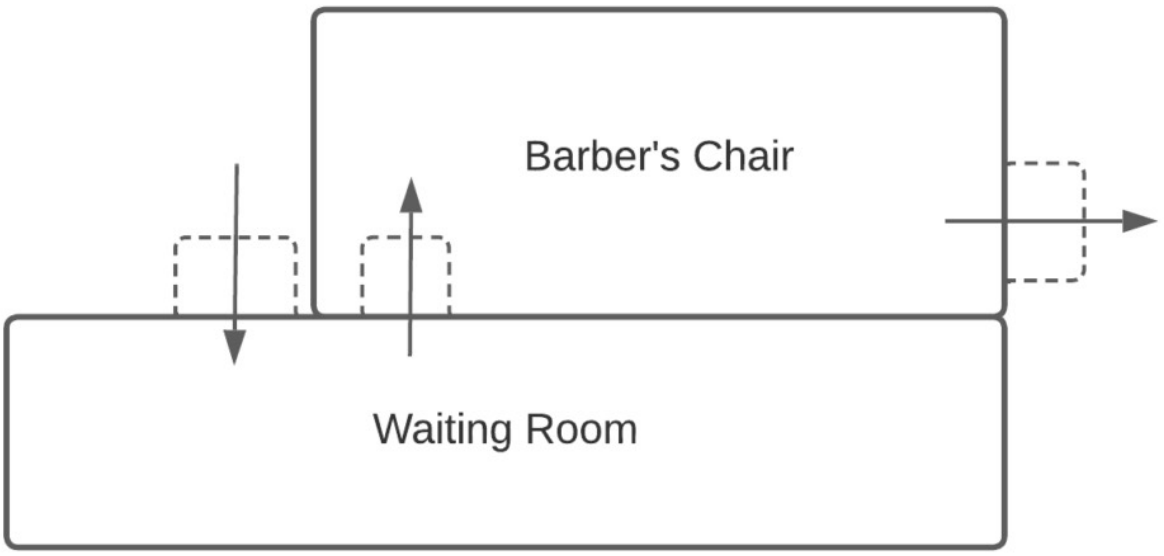
INTRODUCTION

The sleeping barber dilemma was first posed by Dijkstra in 1965. This issue is based on a fictitious situation in which there is a single barber at a barbershop. The waiting area and the workroom are separated in the barbershop. Customers can wait in the waiting area on n seats, however there is only one barber chair in the workroom.



PROBLEM

In this problem, the barbershop has one barber, one barber chair, and a waiting room with chairs for the customers. We can depict the problem by sticking with the original proposal, like in the figure below.

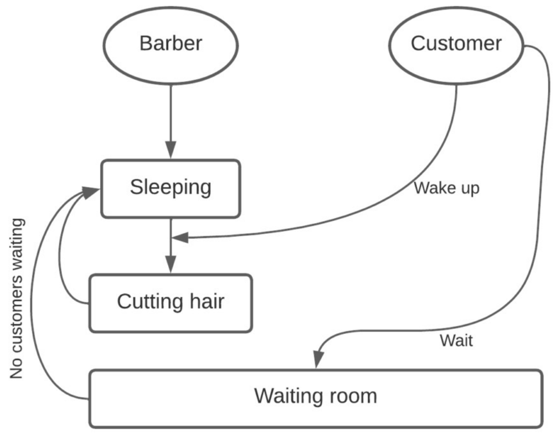


- The barber sleeps when there is no customer in the waiting room. Therefore, when a customer arrives, he should wake up the barber

- As we've seen from the figure, customers can enter the waiting room and should wait for whether the barber is available or not
- If other customers keep coming while the barber is cutting a customer's hair, they sit down in the waiting room
- Customers leave the barbershop if there is no empty chair in the waiting room

WHAT TO DO WE NEED TO DO ?

We need to synchronize the barber and the customers so there wouldn't be any race conditions. Since we have limited resources and waiting tasks, this problem has so many similarities with various queueing situations.



PROBLEM CONCLUSION

The problem arises because the shared resources (the barber chair and the waiting room) must be accessed and modified in a thread-safe manner to avoid race conditions and ensure that customers are served in the correct order. If multiple customers try to access the shared resources simultaneously, they may end up sitting in the wrong chair or waiting in the wrong order.
Therefore, the challenge in the Sleeping Barber Problem is to design a solution that ensures that the shared resources are accessed and modified in a way that is safe and correct, even in the face of multiple threads of execution and potential race conditions.

SOLUTION

Four semaphores are used in the solution to this issue.

- One for customers, which counts the number of waiting customers, excludes the customer in the barber chair since he isn't waiting.
- Another one for the situation of the barber, whether he's waiting or busy.
- And one for the mutual exclusion since there're some critical section that should be thread-safe.
- And the last one for number of free seats in the waiting room

PSEUDOCODE(SOLUTION ALGORITHM)

1. Starting with the semaphores

```
Algorithm 1 : Semaphores & Mutex & Inputs
__________
Mutex Seats = 1;//unlocked binary semaphore
Semaphore Barber_Ready = 0;//sleeping binary semaphore
Semaphore Customer_Ready = 0;//counting semaphore
Int waiting_customers;//variable
Int available_seats;//variable
```

2. When the barber comes to the barber shop he executes the usual routine because there is no customers he needs to wait until a customer arrives. That's why he goes to sleep and stays asleep until the first customer arrives.by using the *customer's* semaphore we wait for the barber and provide one of the important synchronization.

```
Algorithm 2 : Barber Routine

void Barber(){
while (1){
/*Barber sleeps if there are no customers*/
|   sem_wait(Customer_Ready);
/*Enters critical section */
|   sem_wait(seats);
/* Barber brings one customer to the barber chair*/
|   waiting_customers--;
/* barber is now ready to work*/
|   sem_post(Barber_Ready);
/*leves critical region*/
|   sem_post(seats);
|   cut_hair();}}
```

3. *So, we've synchronized the barber routine, but we still need to arrange the customer routine to provide a flawless concurrent system for the barbershop. When a customer arrives, he executes the customer's routine. The routine starts by acquiring the mutex to enter the critical region. First, when he arrives at the barbershop, he needs to check if there is an available chair for him. If there isn't, he leaves the shop. Otherwise, he could wait for the barber, and when the barber is available, he can get a haircut:*

```
Algorithm 3 : Customer Routine

void Customer(){
/*Enters critical region*/
|   sem_wait(Seats);
/*Customers waits if theres an empty waiting chair*/
if (waiting_customers < available_seats){
/*Occupy a seat*/
|   waiting_customers++;
/*Barber gets a wakeup */
|   sem_post(Customer_Ready);
/*leaves critical region */
|   sem_post(seats);
/*waits to be called by the barber*/
|   sem_post(Barber_Ready);
|    get_haircut();
/*Customer leaves if barbershop is full*/
Else{
/* leaves critical section*/
|   sem_post(seats);
|    levae_barbe();
}
```

WHY USE THE MUTEX IN OUR SOLUTION

In the Sleeping Barber Problem, a solution without mutex is not feasible as it would not provide a thread-safe way to access and modify the shared resources (the waiting room and the barber chair). Without a synchronization mechanism such as mutex, multiple threads could access and modify the shared resources simultaneously, leading to race conditions, data corruption, and other synchronization issues. Therefore, a solution without mutex cannot ensure that the customers are served in a fair and orderly manner, and may result in incorrect or inconsistent behavior of the system.

ANALYSIS

When the barber starts his shift, the barber procedure is carried out, and he checks to see whether any clients are ready or not. Pick up the client for a haircut and block the client's semaphore if anybody is available. If nobody requests a haircut, the barber goes to sleep.

The customer releases the mutex, the barber wakes up, and the waiting counter is increased if a chair becomes available. The barber then enters the crucial portion, obtains the mutex, and begins the haircut.

The client departs when the haircut is finished. Now the barber looks to see if there are any other clients waiting for a haircut or not in the waiting area. The barber is going to bed if it is not.

EXAMPLE ON SLEEPING BARBER PROBLEM

In a typical database management system, multiple client applications can access the same database simultaneously. The database is usually protected by a set of locks to prevent multiple clients from accessing the same data simultaneously, which can lead to inconsistencies and race conditions.

The Sleeping Barber Problem can be applied in this scenario by considering the locks as the shared resource, and the client applications as the customers waiting for access to the database. When a client application requests access to a specific piece of data in the database, it must acquire a lock on that data before it can read or modify it. If the requested lock is already held by another client, the requesting client must wait until the lock is released before it can proceed.

To implement this scenario, the client applications represent the "customers," while the locks represent the "barber's chairs." The database management system can use semaphores to control access to the locks and ensure that the client applications access the database in a mutually exclusive manner.

When a client application requests access to a piece of data, it acquires a lock on that data. If the lock is already held by another client, the requesting client must wait until the lock is released before it can proceed. Once the client application finishes accessing the data, it releases the lock, allowing other client applications to access the same data.

This scenario illustrates how the Sleeping Barber Problem can be applied in a database management system to ensure that the shared resource, in this case, the database locks, are accessed in a mutually exclusive manner, preventing race conditions and ensuring data consistency.

RESOURCES

- javapoint.com
- geeksforgeeks.com
- https://mycareerwise.com
- https://www.studytonight.com