Hanoi University of Science and Technology
The School of Information and Communication Technology

IT3070E – Operating System

Project report
Sleeping barber problem

Instructor: Dr. PhD Do Quoc Huy

Group student:

Bui Khanh Linh	20214910
Tran Le My Linh	20210535
Hoang Tu Quyen	20214929
Dang Kieu Trinh	20214933

Ha Noi, 2023

Table of contents

1. Introduction

In the realm of operating systems, the Sleeping Barber problem serves as a classic illustration of the challenges inherent in concurrent programming and synchronization. This project delves into the intricacies of the Sleeping Barber scenario, aiming to provide a practical demonstration of how such synchronization issues can be addressed in a real-world context.

1.1. Background:

The Sleeping Barber problem is a well-known synchronization problem that models a scenario where a barber shop has a limited number of chairs for customers to wait, and a barber who alternates between cutting hair and taking breaks. The challenge lies in managing the interactions between customers and the barber, ensuring that the system operates smoothly without conflicts or inconsistencies.

1.2. Project Overview:

This project involves the creation of a simulation program to represent the Sleeping Barber problem. The program utilizes Python's threading module and semaphores to mimic the concurrent interactions between customers and the barber in a barber shop setting. Through this simulation, we aim to gain insights into the complexities of managing shared resources and maintaining order in a parallel processing environment.

1.3. Objectives

The primary objectives of this project are as follows:

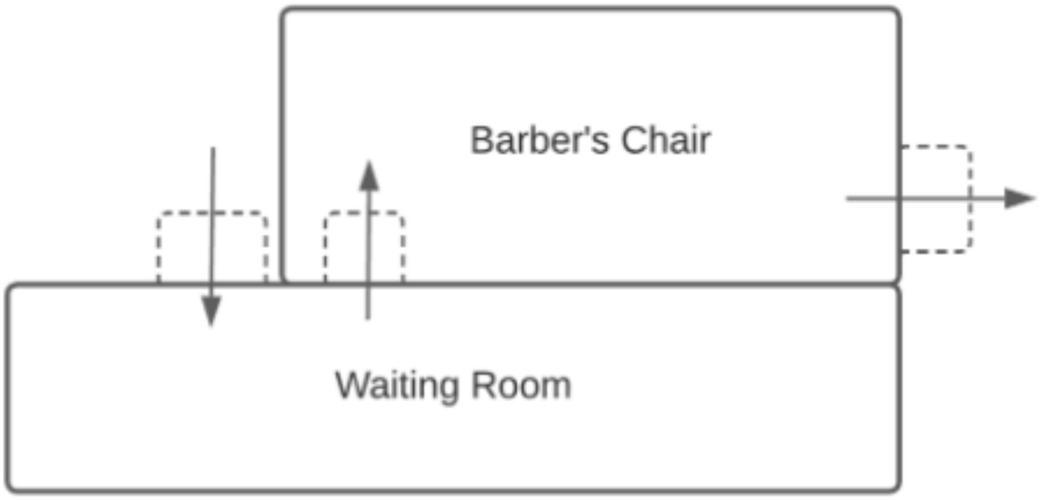
- Develop a clear understanding of the Sleeping Barber problem and its significance in operating systems.
- Implement a simulation program using Python that models the interactions between customers and the barber.
- Showcase the use of synchronization mechanisms, such as semaphores, to address concurrency challenges.
- Analyze the outcomes of the simulation and draw conclusions about the effectiveness of the implemented solution.

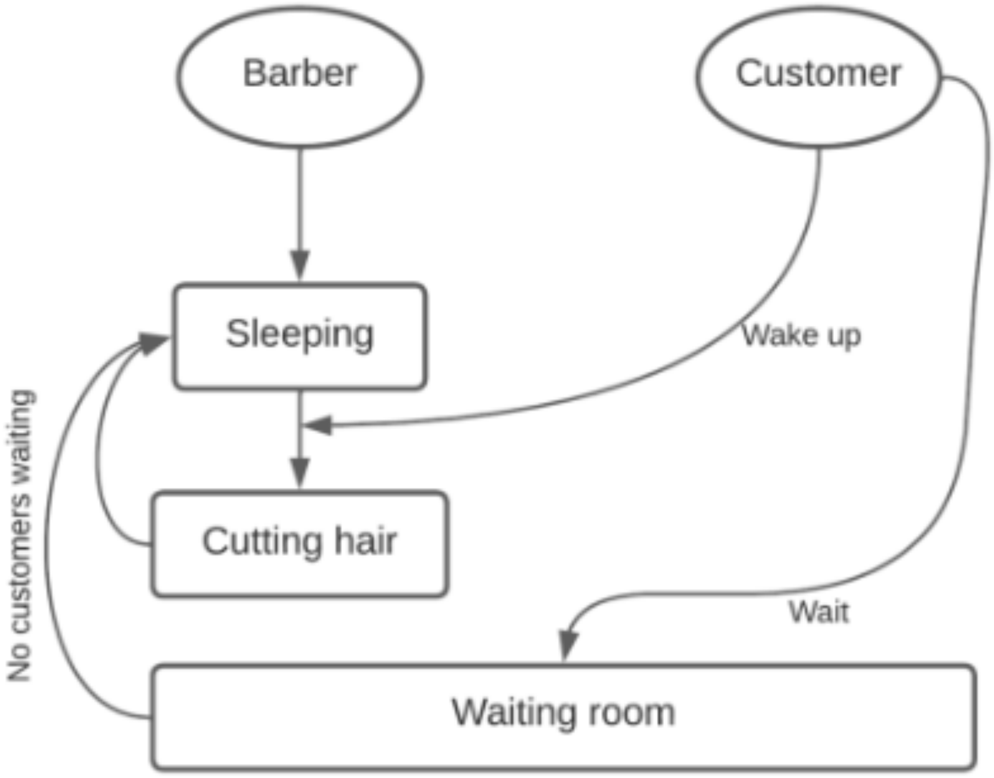
As we proceed, the report will delve into the problem statement, detailing the intricacies of the Sleeping Barber scenario, followed by an exploration of the implemented solution and its underlying design. Through this endeavor, we aim to provide a comprehensive exploration of synchronization challenges in operating systems and shed light on the practical application of theoretical concepts.

2. Problem Statement

2.1. The Sleeping Barber Scenario

The Sleeping Barber problem depicts a barber shop with limited waiting chairs and a single alternating barber. The main challenge is managing interactions between the barber and customers efficiently in a concurrent computing setting to prevent conflicts and inefficiencies.





2.2. Key Challenges

2.2.a. Limited Waiting Room Capacity

A key challenge in the simulated barber shop is the limited capacity of the waiting room. With only a set number of chairs, accommodating customers simultaneously becomes crucial. This constraint raises the risk of customers leaving if the room fills up, highlighting the need for effective resource management.

2.2.b. Barber's Alternating State

In the Sleeping Barber problem, a key challenge is managing the barber's alternating work and rest cycles while handling customer arrivals. We need to ensure the barber stays busy when customers are waiting and create a system that smoothly deals with inactive periods. This involves optimizing the barber's workflow and avoiding potential issues like deadlock - a situation where threads are permanently blocked - during state changes. Additionally, we must address the possibility of customers waiting too long during the barber's breaks, potentially leading to customer starvation, where customers might abandon the shop due to extended waiting times.

2.2.c. Shared Resource Access

Customers and the barber jointly utilize crucial resources—the waiting room and the barber's cutting services. Efficient access management is vital to prevent race conditions, where multiple threads compete for the same resource simultaneously. Beyond this, robust synchronization mechanisms are needed to minimize the risk of deadlock, ensuring that all threads, customers, or the barber, can access shared resources without causing a system-wide standstill. In the context of this problem, deadlock refers to a situation where the interactions between the barber and customers reach an impasse, hindering the progress of the entire system. Furthermore, the solution must adeptly handle the potential for customer starvation, arising from uneven access to shared resources and prolonged waiting times.

3. Solution and Implementation

3.1. Design Overview

To address the challenges posed by the Sleeping Barber problem, we have implemented a simulation using Python's threading module and semaphores. The core components of our solution include a BarberShop class, which encapsulates the barber shop scenario, and methods for the barber and customer threads.

3.2. Synchronization Mechanisms

3.2.a. Semaphores

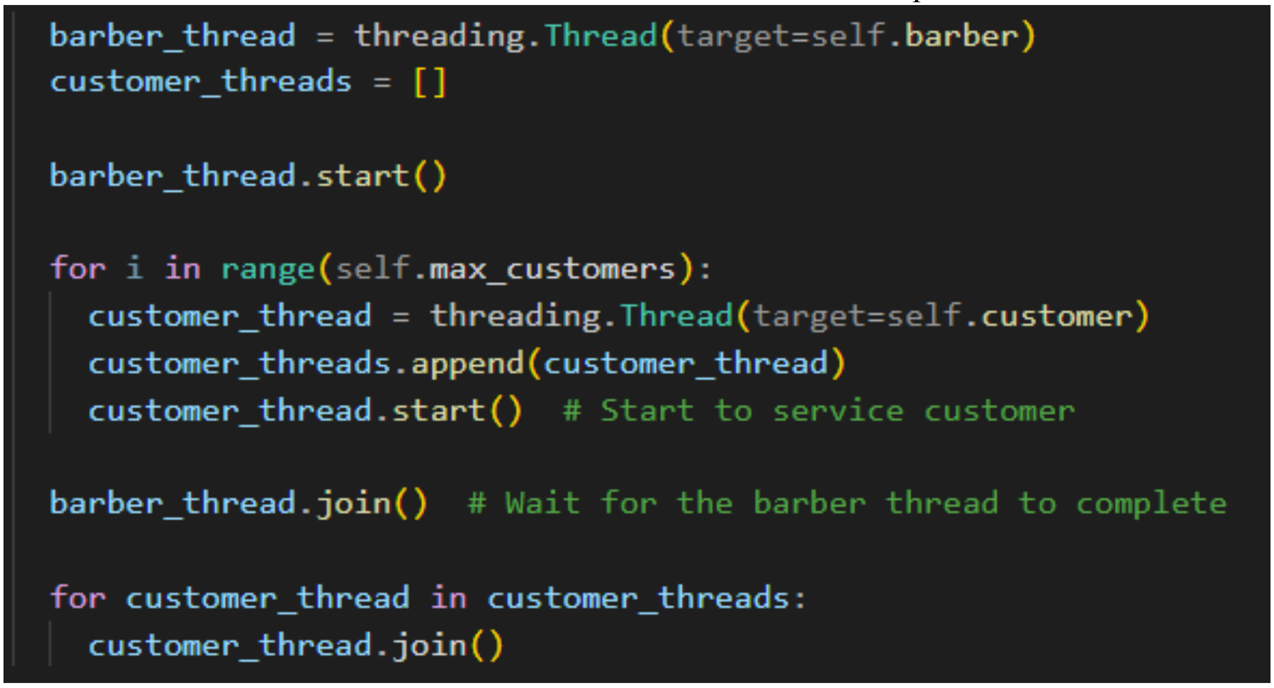
We utilize semaphores to manage synchronization between the barber and customers. The `barber_semaphore` signals the barber to wake up, while the `customer_semaphore` allows customers to wait until the barber finishes cutting their hair. The `mutex` semaphore ensures exclusive access to critical sections, preventing race conditions.

3.2.b. Waiting Room Management

The waiting room, represented by the `waiting_customers` list, is a shared resource protected by the mutex. Customers are added to the list when they decide to wait and removed when the barber serves them. The limited capacity of the waiting room is enforced to emulate real-world constraints.

3.3. Implementation Details

We create threads for the barber and each customer, allowing them to run concurrently. The simulation begins by starting the barber thread, followed by the initiation of multiple customer threads. The simulation concludes when all customer threads have completed.



```python
barber_thread = threading.Thread(target=self.barber)
customer_threads = []

barber_thread.start()

for i in range(self.max_customers):
    customer_thread = threading.Thread(target=self.customer)
    customer_threads.append(customer_thread)
    customer_thread.start()  # Start to service customer

barber_thread.join()  # Wait for the barber thread to complete

for customer_thread in customer_threads:
    customer_thread.join()
```

3.3.a. The BarberShop class

The `BarberShop` class is initialized with parameters defining the maximum number of customers (`MAX_CUSTOMERS`) and the number of chairs in the waiting room (`NUM_CHAIRS`). It uses semaphores (`barber_semaphore, customer_semaphore, and mutex`) to manage access to shared resources.

3.3.b. Barber Thread Function

The `barber_thread_function` represents the behavior of the barber. It runs indefinitely, waiting for the `barber_semaphore` to be signaled. Once signaled, the barber acquires the mutex, checks for waiting customers, and proceeds to cut hair if there are any. The barber then releases the mutex, simulates cutting hair, and signals the `customer_semaphore` upon completion. [1]

```
Algorithm 2: Barber Routine
while $True$ do
    $sem\_wait(customers)$
    $sem\_wait(mutex)$
    $waiting \leftarrow waiting - 1$
    $sem\_post(barbers)$
    $sem\_post(mutex)$
    $cut\_hair()$
end
```

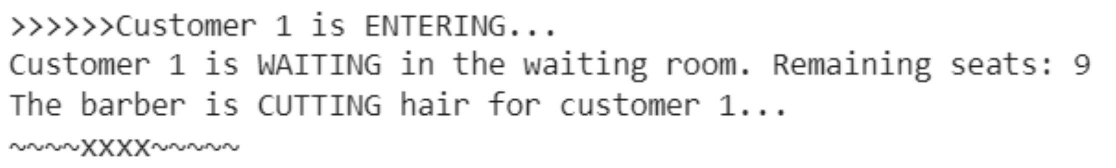
3.3.c. Customer Thread Function

The `customer_thread_function` simulates the behavior of a customer. Each customer thread, upon initialization, waits for a random time before attempting to enter the barber shop. Upon entering, the customer acquires the mutex, checks for available chairs in the waiting room, and either waits or leaves depending on the availability of chairs. If the customer decides to wait, they release the mutex, signal the barber using `barber_semaphore`, and wait for their turn using `customer_semaphore`. [1]

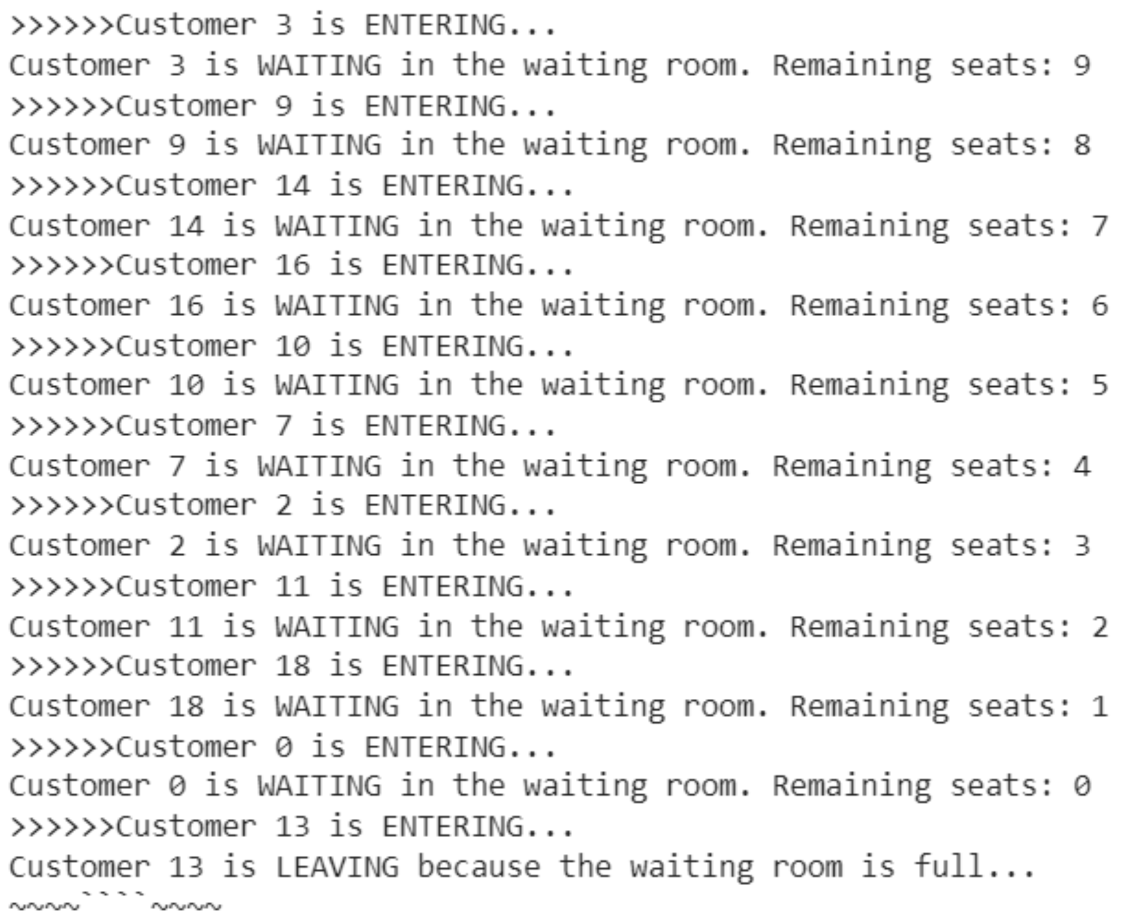
```
Algorithm 3: Customer Routine
$sem\_wait(mutex)$
if $waiting < CHAIRS$ then
    $waiting \leftarrow waiting + 1$
    $sem\_post(customers)$
    $sem\_post(mutex)$
    $sem\_wait(barbers)$
    $get\_haircut()$
end
$sem\_post(mutex)$
```

3.4. Demonstration

The program outputs detailed information about the state of the barber shop, including when customers enter, wait, get their hair cut, or leave. The random sleep intervals for both customers and the barber add variability to the simulation, allowing us to observe different scenarios and potential synchronization issues.



```
>>>>>>Customer 1 is ENTERING...
Customer 1 is WAITING in the waiting room. Remaining seats: 9
The barber is CUTTING hair for customer 1...
~~~~XXXX~~~~~
```



```
>>>>>>Customer 3 is ENTERING...
Customer 3 is WAITING in the waiting room. Remaining seats: 9
>>>>>>Customer 9 is ENTERING...
Customer 9 is WAITING in the waiting room. Remaining seats: 8
>>>>>>Customer 14 is ENTERING...
Customer 14 is WAITING in the waiting room. Remaining seats: 7
>>>>>>Customer 16 is ENTERING...
Customer 16 is WAITING in the waiting room. Remaining seats: 6
>>>>>>Customer 10 is ENTERING...
Customer 10 is WAITING in the waiting room. Remaining seats: 5
>>>>>>Customer 7 is ENTERING...
Customer 7 is WAITING in the waiting room. Remaining seats: 4
>>>>>>Customer 2 is ENTERING...
Customer 2 is WAITING in the waiting room. Remaining seats: 3
>>>>>>Customer 11 is ENTERING...
Customer 11 is WAITING in the waiting room. Remaining seats: 2
>>>>>>Customer 18 is ENTERING...
Customer 18 is WAITING in the waiting room. Remaining seats: 1
>>>>>>Customer 0 is ENTERING...
Customer 0 is WAITING in the waiting room. Remaining seats: 0
>>>>>>Customer 13 is ENTERING...
Customer 13 is LEAVING because the waiting room is full...
~~~~````~~~~
```

4. Conclusion

The simulation of the Sleeping Barber problem has provided valuable insights into the complexities of concurrent programming and synchronization in operating systems. Through the implementation of a Python program utilizing semaphores and threading, we have modeled a scenario where a barber and customers interact within a constrained environment, mirroring real-world challenges faced in shared-resource systems.

The insights gained from this simulation are directly applicable to real-world scenarios where synchronization is crucial. The principles demonstrated here can be extended to various systems, including those in fields such as process scheduling, resource allocation, and parallel computing.

5. Reference

[1] S. Sari, "The Sleeping Barber Problem," [Online]. Available: https://www.baeldung.com/cs/sleeping-barber-problem.

[2] Wikipedia, "Sleeping barber problem," [Online]. Available: https://en.wikipedia.org/wiki/Sleeping_barber_problem.

[3] kunaljoshi1, "Sleeping Barber problem in Process Synchronization," [Online]. Available: https://www.geeksforgeeks.org/sleeping-barber-problem-in-process-synchronization/.