

Sadiq Tijjani Umar

119200113

Abstract

1)Introduction. 2)Methodology. 3)Implementation.   
4)Conclusion.

CMPE 312 Project

The Sleeping Barber Problem

**1)Prepare a report about the synchronization problem you are responsible for.**

**Introduction To The Sleeping Barber Problem**

The sleeping barber problem is a process to process synchronization problem, but what is synchronization? , what is it’s function? . Synchronization involves the co-ordination of one or more processes such that the data the processes hold at the end of execution is identical. Synchronization uses specified tools(such as system clocks) to control access to shared variables to avoid race conditions, these tools are to be used in such way as to prevent a deadlocked state.

The sleeping barber problem was brought forth by the theoretical computer scientist Edsger Dijkstra in his 1965 paper “Co-operating Sequential Processes”, the way the problem is presented Dijkstra’s paper is different, but a variation of the problem is presented as follows , a barber shop has been separated into two rooms, the waiting room has n number of chairs and the working room has only the barber’s chair. If the waiting room is empty , the barber will remain(or go to sleep) in his barber chair, if the waiting room is not empty(meaning there is now a customer), the barber wakes up and the customer has his turn in the barber’s chair(given that the customer is the only customer there). However, if the customer is in the waiting room, and he is not the only customer there, he must take a seat in the waiting room and wait for his turn. The reader may think of the sleeping barber problem as the producer-consumer problem except, customers are produced when they come to the barbershop and barbers consume the customers when they give them haircuts.

In this situations like these, there are many things that could go wrong , firstly a scenario may occur where a customer ends up waiting on for a barber and a barber waiting for a customer which will cause a deadlock, secondly there might be a case where the customer don’t follow an order to cut hair, meaning some customer would not get their hair cut even after they’ve been waiting for extended amounts of time, this is called starvation . The race condition is a key concept to keep in mind when thinking of a solution for this problem. A race condition is a situation where the final result of an operation done by two or more synchronized process on a shared variable returns wrong information, meaning, since the processes can access the shared variable at will, the final value of the shared variable will be incorrect seeing as it is constantly being operated on pre-emptively. There have been various solutions to the problem ever since it was brought forth by Dijkstra in 1965, the most popular involve semaphores(we will see this in the next section).

A popular way to go about solving the sleeping barber problem is with the use of semaphores, but before going further, I want to elucidate to problem to the reader again:

* We have a barbershop with one barber, one barber chair and n number of chairs for waiting customers.
* If there are customers present they may sit on the chairs and wait their turn , if there are no customers present the barber will go to sleep
* When a customer arrives, he wakes up the sleeping barber, if additional customers come they will sit down and wait their turn or leave the shop if there are no free seats.
* The goal is to program the barber and customer functions such that they avoid race conditions.

First of all, since the two processes are to be run con-currently while using a shared variable, we should keep a few things in mind:

* One process may not block other processes from entering in the critical section if it is outside the critical section.(Progress Requirement)
* No two or more running process can be in the critical section at the same time.(Mutual Exclusion)
* Process should not have to wait extended periods of time in order to enter critical section.(Bounded Waiting Requirement)

The most popular solution to the sleeping barber problem is written in the C language, and involves using three semaphores, one which counts the number of waiting customers, another that counts the number of barbers that are sleeping and the number of barbers that are waiting for customers and lastly a mutex to ensure mutual exclusion, below is the rough pseudocode of the usual solution:

**#define MAXCHAIRS 5 // number of chairs**

**typedef int semaphore;**

**semaphore customers = 0; // number of customers waiting for service**

**semaphore barbers = 0; // number of barbers waiting for customers**

**semaphore mutex = 1; // for mutual exclusion**

**int waiting = 0; //waiting customers**

**void barber(void){**

**while (TRUE) {**

**lock(&customers); // go to sleep**

**lock(&mutex); //acquire access**

**waiting = waiting - 1; // decrement number of waiting customers**

**unlock(&barbers); // barber is now ready to cut hair**

**unlock(&mutex); // release access**

**cut\_hair(); // cut hair**

**}**

**}**

**void customer(void){**

**lock(&mutex);**

**if (waiting < MAXCHAIRS) {**

**waiting = waiting + 1; // increment number of waiting customers**

**unlock(&customers); // wake up barber**

**unlock(&mutex); // release access**

**lock(&barbers); // go to sleep if number of free barbers is 0**

**get\_haircut(); // customer is being given a haircut**

**} else {**

**unlock(&mutex); // shop is full; customer should leave**

**}**

**}**

The process runs as follows: When barber arrives, it executes produce barber which will block semaphore customer, barber then goes to sleep, if customer arrives, he executes customer by acquiring the mutex to enter into critical section, if another customer arrives shortly after , they will not be able to enter critical section unless first customer releases mutex. If number of waiting customers is less than the number of chairs customer stays and holds onto mutex if not the customer releases mutex and leaves barbershop(terminates). If there is an available seat, the waiting variable is increased and the customer wakes up the barber. When haircut is finished and customer leaves, barber loops to get the next customer, if there is one, a haircut process is done, if not, the barber goes to sleep!.

The sleeping barber problem is an analogy for a much more general problem, solutions to the problem needn’t apply to just barbers and their barbershops. The solutions could also be applied to a bank and it’s customer application system, a school and it’s student registration system and any other sort of queueing situations.

**Methodology**

I will create a solution of this problem using the C language, the aim is to create an algorithm that satisfies the given critical section condition’s given in the above section. The methodology I am going to use is similar to the popular solution given above in that, we are also going to use three semaphores for handling the various con-currency situations, we are also going to be using the pthreads C library which will allow us to control various overlapping executions. We will create a customer function that handles the entry and exit of the various customers and a barber function that will handle the “execution” of haircuts, inside this barber function, we will also be creating a “make\_haircut” function which simply stalls the barber program for a while before customer leaves.

A few key distinctions between my methodology and with the popular sample given above is that, of course the sequence of locks and unlocks of the semaphores(the reader should please refer the file containing the code about which this report is about) are different, the number of allocated free seats are defined in different ways and the customers and barbers are created using for loops. As has been said in the previous section, of course, the goal is to create a program that avoids race conditions.

The pseudo-code is as follows:

**Semaphore mutex, customer,barber;**

**Int number of customers = 0;**

**Constant int number of chairs = 3;**

**Void customer(){**

**\*customer holds mutex\*;**

**ForLoop(create customers){**

**While(number of customers is less than or equal to number of seats){**

**\*customer has entered waiting room and number of customers is increased\*;**

**\*set customer ready to be served\*;**

**\*is barber ready to serve?\*; //This will all be done with the various semaphores!.**

**\*Barber ready to serve, unlock mutex\*;}**

**Otherwise if number of waiting customers is greater than number of free seats:**

**\*Unlock Mutex and customer will leave\*;**

**Void barber(){**

**ForLoop(create barbers){**

**If(number of waiting customers is 0){**

**\*barber sleeps\*;**

**\*After a while, check for new customers\*;}**

**Else{**

**\*Barber wake up\*;**

**\*customer is set to be served\*;//All this will be done using the respective semaphores!.**

**\*Make\_haircut\*;**

**\*number of customers is reduced\*;}**

**Void make\_haircut(){  
\*Wait arbitrary amount of time and print message\*;}**

**Failed cases:**

In setting up the semaphores in the customer() function, I first set the process as follows:

sem\_post(&customer);

sem\_wait(&barber);

sem\_post(&mutex);

sem\_post(&barber);

The above block of code was to be run given that the number of waiting customers is less than or equal to the number or free seats, but this always resulted in a deadlock because when a customer would come into the barbershop the lock on barber semaphore(sem\_wait(&barber)) would wait for the barber to get ready, but the barber cannot get ready unless the customer entices him to, seeing as there are no ongoing the hair cuts going on at the beginning a deadlock is reached because both barber and customer are waiting on one another. To solve the problem I had to take out “sem\_wait(&barber);” and the code ran perfectly.

I had also considered instead of making a for loop for creating customers, that I create a distinct function the creates customers as follows:

void \*make\_customers() {  
while (1) { int varbl = 0;

 if (varbl < 8) {

pthread\_t customerc;

 int start = 0;

  start = pthread\_create(&customerc, NULL, customer,NULL);

pthread\_join(customer, NULL);

varbl++;

if (start != 0)

perror("No Customers Yet!!!\n"); } }}

This would’ve also worked well but I felt it was excess to the necessary because I could just make a for loop in the customer function to create customers and initialize the pthread in main. Thus I was able to keep the code more short and concise, and maybe even easier to understand. A full run down of the implementation of the code will be done in the section below.

**2)Write a C Implementation Of The Synchronization Problem You Are Responsible For.**

**Implementation Of Code**

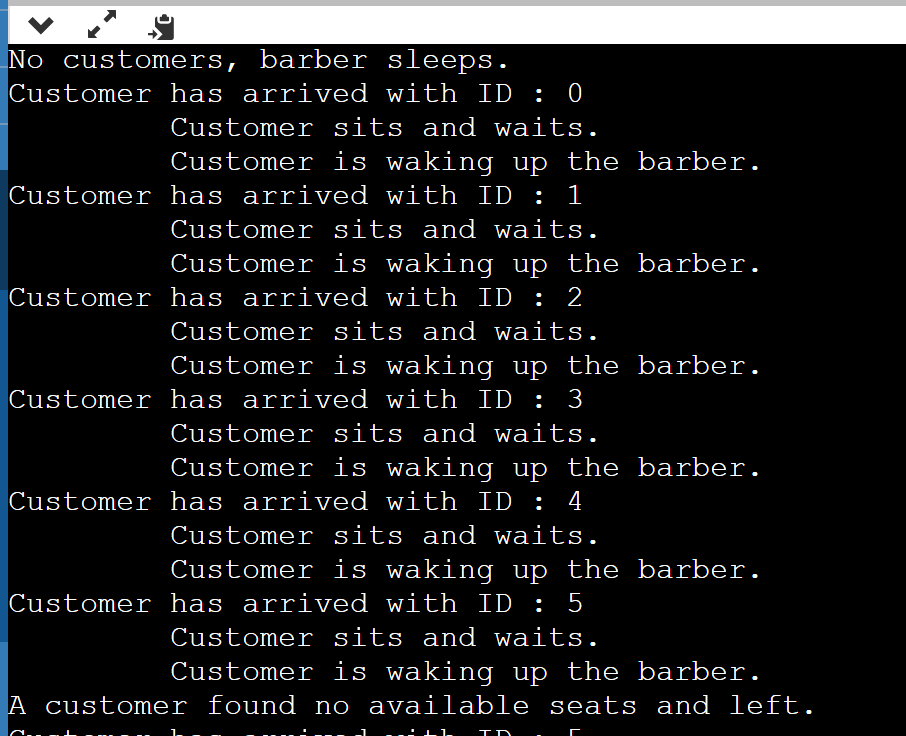
On a successful execution of the code, first it would have to initialize the usual three semaphores(customer, barber and mutex), customer semaphore controls the number of waiting customers, barber semaphore controls whether the barbers are asleep or awake and the mutex semaphore is for mutual exclusion. We then set the number of waiting customers(waiting\_customers) to be zero, meaning that the barbershop is initially empty.

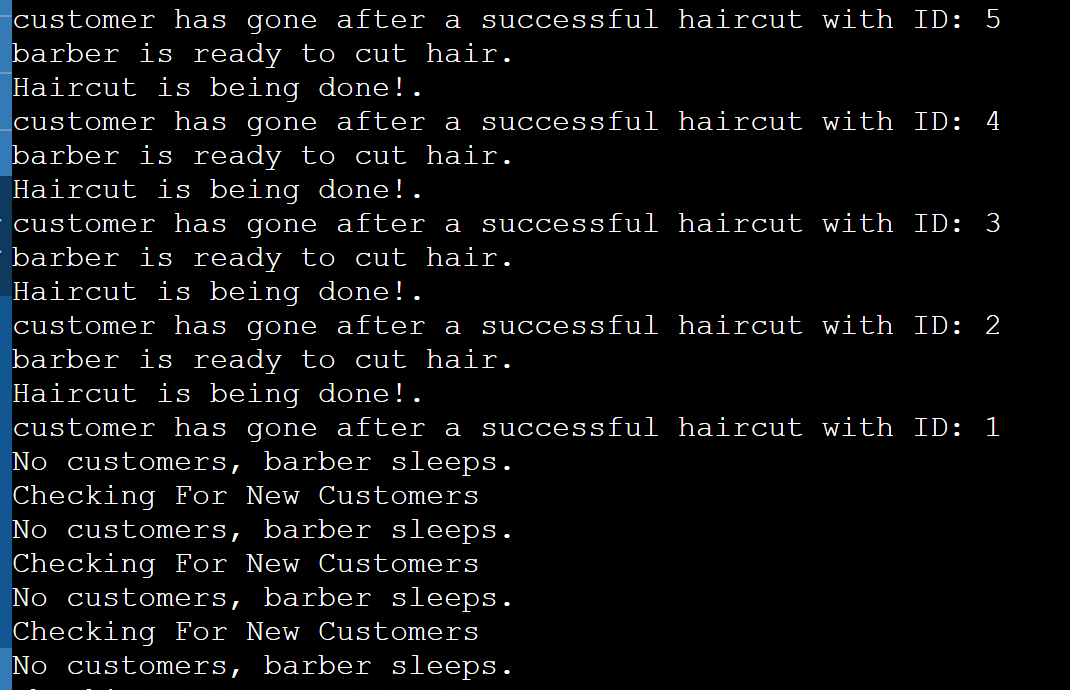
We then assign a constant number of available seats in the barbershop, after which we then enter the Customer function, inside the customer function we first lock the mutex semaphore and then begin creating customers using an assigned for loop, the first customer(with ID = 0) then checks if the number of waiting customers is less than or equal to the number of free chairs, if yes, it enters into the barbershop. The number of waiting customers is increased(“waiting\_customers = waiting + 1”) and the customer proceeds to signal that he is ready and waiting for a haircut using “sem\_post(&customer)”, it will then signal to see of barber is ready to cut hair using “sem\_post(&barber)” and then let go of mutex using “sem\_post(&mutex)” and allows other customers to come in the barbershop and queue. This process will go on repeatedly until “While(waiting\_customers <= free\_seats)” returns false, in which case the customer at that point will have to release the initially held-onto mutex and leave the barbershop as there are no free seats, the number of waiting customers is then reduced by 1.

Going into the barber function, the barbers are created using the assigned for loops as in the customer function(The reader may think of this as,for every instance of the customer,there is a barber,there are not multiple barbers in the barbershop as per the problem), “each” barber then checks if the barbershop is empty i.e if “waiting\_customers” is equal to 0. If so the barber will go to sleep, but after a while it will check to see if there is a customer i.e “sem\_wait(&customer)” . In the case that there are customers In the barbershop, the barber will confirm for customer using “sem\_wait(&customer)”, it will then signal that it is ready to give a haircut using “sem\_post(&barber)”, it will then check for customer is ready to get haircut using “sem\_post(&customer)”, it will then invoke the “make\_haircut” function, this function simply assigns random delay time for a “haircut” using usleep. After a haircut the customer leaves the barbershop and the number of waiting customer is reduced by one.

The content within the main() provides instructions that tell the computer to carry out whatever given tasks the programs above are designed to do. Inside the main we define the pthreads for Customer and Barber function to allow us to run them in such a way that they overlap i.e at the same time. The customer, barber and mutex semaphores are then initialized using the “sem\_init” from the semaphore.h library, the two threads for Customer and Barber function is then formally initialized using “pthread\_create” from the pthread.h library.

After the threads are allowed to run completely, the mutex, customer and barber semaphores are destroyed after. The terminal after a successful execution the terminal should look like this:





**Conclusion**

My solution to the sleeping barber problem is not perfect, there is definitely room for improvement, for example, it is possible of instead of having the number of free seats and waiting customers be defined at the top of the program, we could make it such that the main() ask the user for input on the number of free seats and number of customers, It scans the data from the keyboard sets it to be the number of free seats and the number of waiting customers respectively. Of-course the way in which I sequenced the series locks and unlocks on the various semaphores is not the only way to do so, it can be done in various amounts of ways and still be able to simulate the customer-barber sequence successfully, for example it could be made such that the program ensures that the barber function is FIFO(first in first out) using maybe push/pop and queues or stacks, because this solution doesn’t guarantee that customers are served in the order they come in!.

My given Solution prevents the race condition problem with the shared variable(“waiting\_customers”) because customer and barber function, although they run in an overlapping manner, only gain the shared variable in distinct times in the duration of execution. The solution also does not suffer from the starvation problem as all customers get their haircuts even after waiting for extended amounts of time, it also fulfills the progress requirement because one customer does not prevent any other customer from entering the critical section if it doesn’t need the critical section. Any solution to the sleeping barber problem as has been said in the previous sections, can be seen as an analogy for any sort of queueing process and as such can be very advantageous in the real world also.