**Aim:** To implement the Sleeping Barber Problem.

**Software used:**

* Atom Text Editor
* GNU Compiler Collection (gcc)
* GTK+

**Theory**:

In computer science, the sleeping barber problem is a classic inter-process communication and synchronization problem between multiple operating system processes. The problem is analogous to that of keeping a barber working when there are customers, resting when there are none, and doing so in an orderly manner.

The analogy is based upon a hypothetical barber shop with one barber. The barber has one barber's chair in a cutting room and a waiting room containing a specific number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting. If there are, he brings one of them back to the chair and cuts their hair. If there are none, he returns to the chair and sleeps in it.

Each customer, when they arrive, looks to see what the barber is doing. If the barber is sleeping, the customer wakes him up and sits in the cutting room chair. If the barber is cutting hair, the customer stays in the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits their turn. If there is no free chair, the customer leaves.

Based on a naïve analysis, the above decisions should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives. In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

The problems are all related to the fact that the actions by both the barber and the customer (checking the waiting room, entering the shop, taking a waiting room chair, etc.) all take an unknown amount of time. For example, a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While they're on their way, the barber finishes their current haircut and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to their chair and sleeps. The barber is now waiting for a customer, but the customer is waiting for the barber. In another example, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

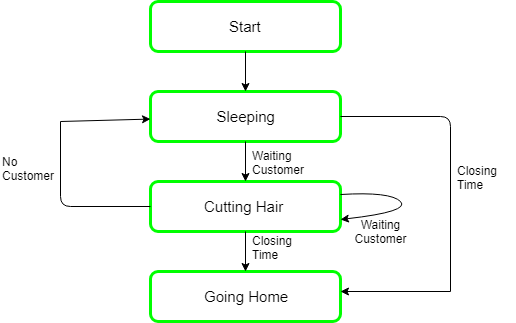
**Procedure/Algorithm:**

Our solution uses three semaphores, customers, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), smfBarber, the number of barbers (0 or 1) who are idle, waiting for customers, and seatMutex, which is used for mutual exclusion. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays, otherwise, he leaves and comes back after a random time and checks the number of waiting customers again.

When a customer arrives, he executes customer, starting by acquiring seatMutex to enter a critical region. If another customer enters shortly thereafter, the second one will no be able to do anything until the first one has released seatMutex. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases mutex and leaves without a haircut.

If there is an available chair, the customer decrements the numberOfFreeSeats. If there are customers in the waiting room, the barber function takes a random time to complete the customers haircut. When the haircut is over the customer exits the barber shop and increases the number of free seats in the waiting room.

Following is the flowchart of the Sleeping Barber Problem:



**Code:**

#include <stdlib.h>

#include <stdio.h>

#include <gtk/gtk.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

//GUI

static GtkWidget \* number1;

static GtkWidget \* number2;

static GtkWidget \* number3;

static GtkWidget \* number4;

static GtkWidget \* result;

static GtkWidget \* lab1;

static GtkWidget \* lab2;

static GtkWidget \* lab3;

static GtkWidget \* lab4;

static GtkWidget \* lab5;

static GtkWidget \* vbox;

static GtkWidget \* text\_view;

static GtkWidget \* window;

static GtkWidget \* grid;

static GtkWidget \* calculate;

GtkTextBuffer \* buffer;

//DEFS

void barber(); // barber thread function

void customer(); //customer thread function

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

sem\_t customers; // Semaphore for customers currently in the waiting room ready to be served

sem\_t smfBarber; // Semaphore for the barber

int runningTime; // Total time in seconds that the main thread will work

int numberOfFreeSeats; // Capacity of the waiting room

int customersCount; // Number of customers in the program that will come to the shop

int customerWait; // Maximum time a customer would wait to come again to the shop

int getHCn; // Number of customers who get haircut

pthread\_t barberThread; // Barber's thread

pthread\_t customersThreads[20]; // An array of customer threads with max size of 20

pthread\_t guithread; // To update GUI constantly

int \* gotHaircut[20];

int gtp = -1;

void addtext(char strr[60]) {

gtk\_widget\_show\_all(window);

buffer = gtk\_text\_view\_get\_buffer(GTK\_TEXT\_VIEW(text\_view));

gtk\_text\_buffer\_insert\_at\_cursor(buffer, "\n", 1);

gtk\_text\_buffer\_insert\_at\_cursor(buffer, strr, strlen(strr));

}

void mainth() {

char buffer[60];

getHCn = 0;

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

addtext("Program started");

// Initializing semaphores:

// zero indicates that this semaphore is not allowed to be shared between processes, 1 is an initial value

sem\_init( & seatMutex, 0, 1); // the initial value is 1 because we need to allow only one thread to access

sem\_init( & customers, 0, 0);

sem\_init( & smfBarber, 0, 0);

// Creating barber thread:

pthread\_create( & barberThread, NULL, barber, NULL); // barber is the function which the thread will start at

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

addtext("Barber thread created");

// Creating customers threads:

for (int i = 0; i < customersCount; i++) {

pthread\_create( & customersThreads[i], NULL, customer, NULL); // customer is the function which the thread will start at

printf < ("Customer %u has been created.\n", customersThreads[i]);

snprintf(buffer, sizeof(buffer), "Customer %u has been created.", customersThreads[i]);

addtext(buffer);

}

// stop the main thread in order to run the other threads

sleep(runningTime);

addtext("\n\n");

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

printf < ("%d out of %d customers get haircut.", getHCn, customersCount);

addtext("End of day");

snprintf(buffer, sizeof(buffer), "%d out of %d customers get haircut.", getHCn, customersCount);

addtext(buffer);

pthread\_cancel(barberThread);

for (int i = 0; i < customersCount; i++) {

pthread\_cancel(customersThreads[i]);

}

for (int i = 0; i <= gtp; i++) {

snprintf(buffer, sizeof(buffer), "Customer %u got haircut", gotHaircut[i]);

addtext(buffer);

}

}

// barber thread function:

void barber() {

char buffer[60];

int workingTime; // Will be generated randomly to indicate the time

// that the barber will take to cut a customer's hair

while (1) {

//===== ENTRY SECTION =====//

// acquire customers to wait for a customer

sem\_wait( & customers);

// acquire seatMutex to access seats count

sem\_wait( & seatMutex);

//===== CRITICAL SECTION =====//

// increase the number of free seats

numberOfFreeSeats += 1;

// generate random time between 1-5 seconds for the period of haircut.

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

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

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

//===== EXIT SECTION =====//

// signal to customer which mean barber is ready

sem\_post( & smfBarber);

// release lock on seat count

sem\_post( & seatMutex);

sleep(workingTime);

}

}

// customer thread function:

void customer() {

char buffer[60];

int waitingTime;

int notEnd = 1;

while (notEnd == 1) {

//===== ENTRY SECTION =====//

// acquire seatMutex to access seats count

sem\_wait( & seatMutex);

// when there are no free seats

if (numberOfFreeSeats <= 0) {

//===== EXIT SECTION =====//

// generate random time for waiting until next try

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

printf("Customer %u left without haircut, and will come back after %d seconds to try again.\n", pthread\_self(), waitingTime);

sem\_post( & seatMutex); // release the semaphore

sleep(waitingTime);

}

// when there are free seats

else {

//===== CRITICAL SECTION =====//

// decrease the number of free seats

numberOfFreeSeats -= 1;

printf("Customer %u is waiting.\n", pthread\_self());

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

//===== EXIT SECTION =====//

// customer is ready

sem\_post( & customers);

// release seatMutex lock on seat count

sem\_post( & seatMutex);

// wait for barber

sem\_wait( & smfBarber);

// get haircut

printf("Customer %u got a haircut\n", pthread\_self());

gotHaircut[++gtp] = pthread\_self();

// the customer thread will end

notEnd = 0;

// increse the number of customers who get haircut

getHCn += 1;

}

}

}

void startok(GtkWidget \* calculate, gpointer data) {

runningTime = atoi((char \* ) gtk\_entry\_get\_text(GTK\_ENTRY(number1)));

numberOfFreeSeats = atoi((char \* ) gtk\_entry\_get\_text(GTK\_ENTRY(number2)));

customersCount = atoi((char \* ) gtk\_entry\_get\_text(GTK\_ENTRY(number3)));

customerWait = atoi((char \* ) gtk\_entry\_get\_text(GTK\_ENTRY(number4)));

// printf<("%d",runningTime);

/\* Set the default buffer text. \*/

gtk\_text\_buffer\_set\_text(buffer, "................................................Result................................................", -1);

}

void gui() {

int argc;

char \*\* argv;

gtk\_init( & argc, & argv);

window = gtk\_window\_new(GTK\_WINDOW\_TOPLEVEL);

g\_signal\_connect(window, "destroy", G\_CALLBACK(gtk\_main\_quit), NULL);

grid = gtk\_grid\_new();

gtk\_container\_add(GTK\_CONTAINER(window), grid);

calculate = gtk\_button\_new\_with\_label("Start");

g\_signal\_connect(calculate, "clicked", G\_CALLBACK(startok), NULL);

sleep(1);

g\_signal\_connect\_after(calculate, "clicked", G\_CALLBACK(mainth), NULL);

gtk\_grid\_attach(GTK\_GRID(grid), calculate, 1, 6, 1, 1);

lab1 = gtk\_label\_new("Running time of program:");

gtk\_grid\_attach(GTK\_GRID(grid), lab1, 0, 1, 1, 1);

lab2 = gtk\_label\_new("Free Seats:");

gtk\_grid\_attach(GTK\_GRID(grid), lab2, 0, 2, 1, 1);

lab3 = gtk\_label\_new("Customers:");

gtk\_grid\_attach(GTK\_GRID(grid), lab3, 0, 3, 1, 1);

lab4 = gtk\_label\_new("Waiting time:");

gtk\_grid\_attach(GTK\_GRID(grid), lab4, 0, 4, 1, 1);

number1 = gtk\_entry\_new();

gtk\_grid\_attach(GTK\_GRID(grid), number1, 1, 1, 1, 1);

number2 = gtk\_entry\_new();

gtk\_grid\_attach(GTK\_GRID(grid), number2, 1, 2, 1, 1);

number3 = gtk\_entry\_new();

gtk\_grid\_attach(GTK\_GRID(grid), number3, 1, 3, 1, 1);

number4 = gtk\_entry\_new();

gtk\_grid\_attach(GTK\_GRID(grid), number4, 1, 4, 1, 1);

/\* Create a multiline text widget. \*/

text\_view = gtk\_text\_view\_new();

gtk\_grid\_attach(GTK\_GRID(grid), text\_view, 4, 0, 10, 10);

/\* Obtaining the buffer associated with the widget. \*/

buffer = gtk\_text\_view\_get\_buffer(GTK\_TEXT\_VIEW(text\_view));

/\* Set the default buffer text. \*/

gtk\_text\_buffer\_set\_text(buffer, "................................................WATING................................................", -1);

}

// gcc 007\_gtk.c -o 007\_gtk `pkg-config --cflags gtk+-3.0` `pkg-config --libs gtk+-3.0`

int main(int argc, char \*\* argv) {

gui();

gtk\_widget\_show\_all(window);

gtk\_main();

return 0;

}

**Output Screen Shots:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Conclusion:**

In the computing field, there are circumstances wherein one might have one process, in this sleeping barber case embodied in the notion of the barber, and one might have another process, in this case embodied in the customer that’s waiting for the barber. Each might be waiting on the other, not realizing that they are indeed waiting on each other.

At the heart of any solid operating system is the ability to cope with synchronization among multiple processes and trying to ensure that there is proper inter-process communications taking place. If an OS doesn’t have a proper way of dealing with this, the system goes into a deadlock state.