Codey Sivley

For CS415 Operating Systems

Dr. Lewis, Spring 2022

Barbershop of Evil

Intro

This assignment investigates the Sleeping Barber problem as a model of multithread processing and inter-process communication. Mutex and Control Variables are used to complete these operations.

Following is an overview of my solution.

Mutex & Condition Variables

The main focus of this assignment. As commented in the code:

- Mutex barberReady
  - Used to lock the barber chair. Unlocks when the barber is switching from one client to the next, or when the barber is sleeping.
- Mutex waitChairs
  - Used to lock the waiting room chairs. Regularly accessed by both the barber thread and the customer thread.
- Mutex timeLock
  - Used to lock the barber when he is in the hair cut loop.
  - I found when the barber was not forced to wait, he would cut several customers worth of hair without allowing new customers the chance to come in.
  - Furthermore, the requirement to simulate this for a certain amount of time required a way to quantize both threads to a discrete scale.
  - This mutex is used to lock the barber thread each time he cuts hair, so that the customer thread gets a chance to see if there are any changes to the waiting room or new arrivals. When the customer thread gets a chance to check the roster, it then notifies the barber thread and allows another pass through the loop.
- Condition_variable noCustomers
  - When barber sees no customers, he takes a nap and his thread waits until it is notified by the next customer to walk in the door.
- Condition_variable timeSync
  - This is paired with timeLock, for notifying the barber he can continue cutting hair.

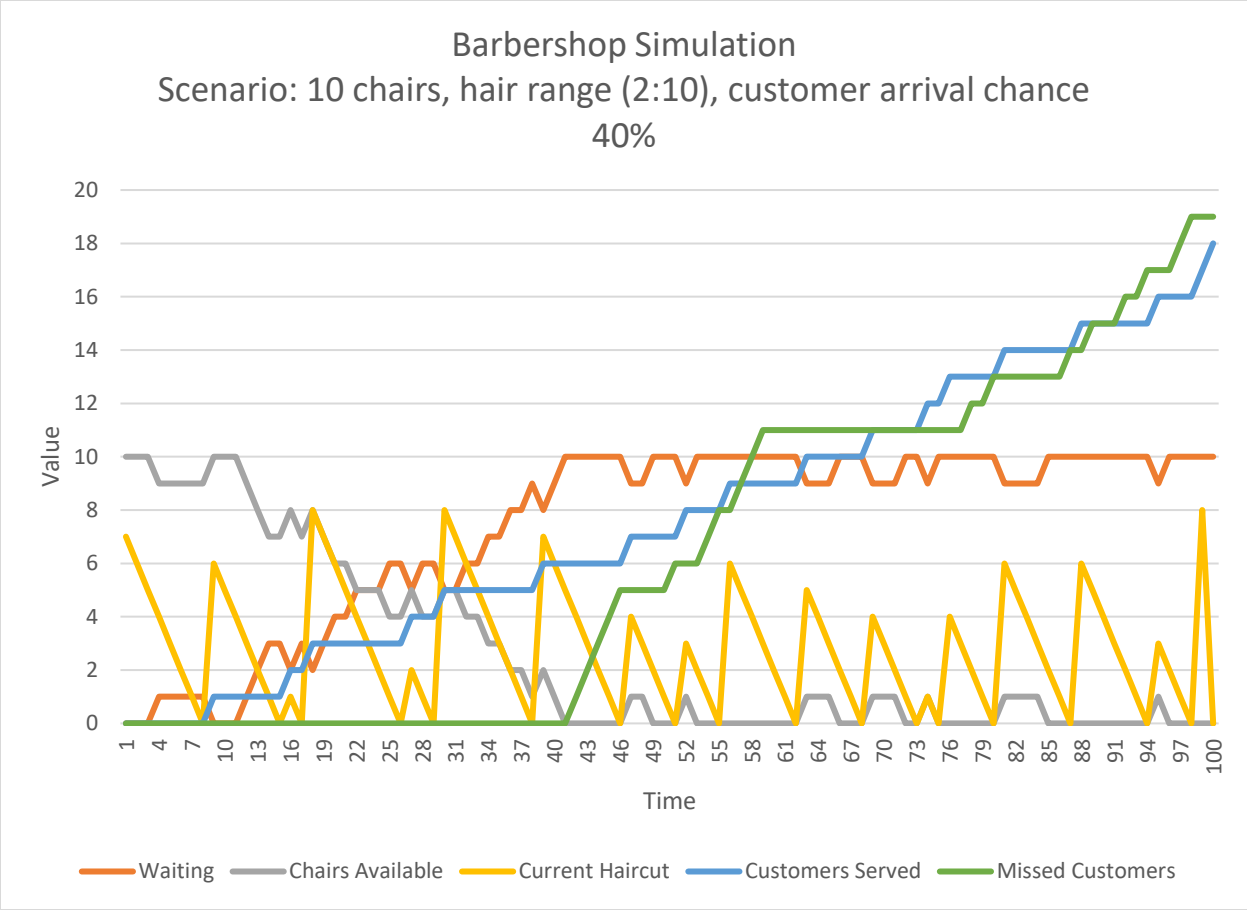
Pitfalls

- Deadlock
  - My original plan was to allow customer to control the time clock. From open to close, the customer thread would send a time update to the barber thread.

    - The problem was that at the end of the day, a customer may be in the chair with a half finished haircut when the customer thread stops sending time updates. Therefore, the barber ceased cutting hair and a customer was left with half a head of hair.
    - My solution was to put a timeout on the control variable that allows it to unlock itself if it waits too long. Source: http://www.gerald-fahrnholz.eu/sw/online_doc_multithreading/html/group___grp_condition_variable_safe_way.html
- Race Conditions
    - It is important to manage the waiting room effectively so that each thread does not make a wrong decision based on the number of customers present.
    - My solution to this was manually declaring, locking and unlocking the mutexes for the barber while allowing scoped resolution for the customer thread. This way, the barber performs each step as he is able, but the customer decision script runs in it's entirety without the barber getting a chance to alter things halfway through.
    - I know it's possible to have done this cleaner, but this is the way that was easiest for me to wrap my head around.
- Starvation
    - Rather than have the barber check if there was a customer standing in the lobby before deciding to sleep, I just decided on the following steps:
    - The customer always sits down first if able. Only then does he check if the barber is sleeping.
    - This way, the barber does not have a special condition for customers when there are none waiting: he simply pulls the next one out of the waiting room if he can.

Results

By altering the bounds of the scenario, different patterns emerge in the resulting data. To make observing the results easier, I output them to .csv as well as terminal. This allowed me to easily make plots based upon the data. If there's one thing I've learned at work, it's that bosses like easy-to-read charts. Below is an example chart of the data generated.



Source Code:

```
// BarbershopOfEVIL.cpp : This file contains the 'main' function. Program execution begins and ends there.
//by Codey Sivley
//for CS415 Operating Systems by Dr. Lewis
//makeup work for Incomplete grade of Spring 2022
//

#include <iostream>
#include <fstream>
#include <random>
#include <thread>
#include <chrono>
#include <mutex>

//mutex things
std::mutex barberReady; //locks when barber is busy cutting hair.
std::mutex waitChairs; //locks when barber or a customer is checking waiting room.
std::mutex timeLock; //used as a barrier: only allows barber one "cut" per time click.
std::condition_variable noCustomers; //wakes up sleeping barber when invoked.
std::condition_variable timeSync; //paired with timeLock for making barber take turns.

//prototypes
```

```
void barber();
void customer();
int randomHair(int lb, int ub); //returns mers.twis. randomized hair length for cutting.
bool customerArrival(int gate, int ub); //returns mers.twis. randomized chance for
customer to arrive.
void statusReport(); //writes data to console and file.

//Running Variables
int readyCustomers = 0; //current number of waiting customers
int currentClient = 0; //time remaining on current customer's haircut
bool barberChairAvailable = true; //for checking if the barber is asleep
int customersServed = 0; //tally of happy customers
int missedCustomers = 0; //tally of snubbed walk-ins
int quittingTime = 100; //length of day in time ticks
int currentTime = 0; //current time of day

//Parameters: adjust these to see how to day changes.
int MAXCHAIRS = 10; //number of chairs in the waiting room.
int CUSTOMER_CHANCE = 6; //number to beat on a d10 to get a customer each tick
int HAIR_MIN = 2; //minimum amount of time to cut hair
int HAIR_MAX = 10; //maximum amount of time to cut hair

void barber() {
    //create locks, delay locking until we need them
    std::unique_lock<std::mutex> barberChecksWaiting(waitChairs, std::defer_lock);
    std::unique_lock<std::mutex> barberWorking(barberReady, std::defer_lock);
    std::unique_lock<std::mutex> barberTime(timeLock, std::defer_lock);

    while (quittingTime > currentTime) {
        //check for customers, move to chair if available
        barberChecksWaiting.lock();
        if (readyCustomers) {
            --readyCustomers;
            currentClient = randomHair(HAIR_MIN, HAIR_MAX);
            barberChairAvailable = false;
        }
        barberChecksWaiting.unlock();

        //if there's someone in the chair, get busy cutting!
        barberWorking.lock();
        barberTime.lock();
        if (currentClient) {
            while (currentClient) { //actual haircut performed here
                --currentClient;
                timeSync.wait_for(barberTime, std::chrono::milliseconds(500));
                //pause so customer thread gets a turn. Wait for allows us to
                //finish our last customer's haircut once the customer thread
                //stops notifying timeSync after the end of the day.
            }
            ++customersServed;
        }
        barberWorking.unlock();
        barberTime.unlock();

        //haircut done, let's check the wait again
        barberChecksWaiting.lock();
        if (!readyCustomers) { //if no customers
            barberChairAvailable = true;
```

```
            noCustomers.wait(barberChecksWaiting);  //snooze until one wakes us up.
        }
        //if we are here, barber was either woken by a new customer or has one waiting,
        //so release barberChecks and loop.
        barberChecksWaiting.unlock();
    }
    return;
}

void customer() {
    while (quittingTime > currentTime) {
        //decide if a customer shows up.
        if (customerArrival(CUSTOMER_CHANCE, 10)) {
            //check if anyone is waiting
            {
                std::lock_guard<std::mutex> customerChecksWaiting(waitChairs);
                if (readyCustomers == MAXCHAIRS) { //if the seats are full
                    ++missedCustomers; //we lost one!
                }
                else if (readyCustomers) { //if anyone else is here
                    ++readyCustomers; //we sit in a chair
                }
                else { //we're the first customer in line.
                    ++readyCustomers;
                    //if there's no other customers, wake up the barber!
                    if (barberChairAvailable) { noCustomers.notify_one(); }
                    //otherwise, that ends our job and we wait for the current haircut to
end.
                }
            }
        } //end customer arrival

        ++currentTime; //no matter what, add a time tick
        statusReport();
        timeSync.notify_all(); //allow barber to progress if he is in haircut loop.
    }
    return;
}

int randomHair(int lb, int ub) {
    //returns integer for random hair length
    //args: int lower bound, int upper bound
    std::random_device rd{}; //init random seed
    std::mt19937 gen{ rd() }; //mersenne twister from seed
    std::uniform_int_distribution<int> uD(lb, ub);
    int hairOut = uD(gen);
    return hairOut;
}

bool customerArrival(int gate, int ub) {
    //returns integer for random hair length
    //args: int score to beat, int upper bound
    std::random_device rd{}; //init random seed
    std::mt19937 gen{ rd() }; //mersenne twister from seed
    std::uniform_int_distribution<int> uD(1, ub);
    bool result = (gate < uD(gen));
    //if roll beats gate, new customer arrives.
    return result;
```

```
}

void statusReport() {
    std::ofstream fileOut;
    fileOut.open("BarberShop.csv", std::fstream::app);
    if (!fileOut) {
        std::cerr << "File Issues, fix your files, man." << std::endl;
        exit(1);
    }
    std::cout << "Current Time: " << currentTime << std::endl;
    std::cout << "Customers Waiting: " << readyCustomers << std::endl;
    std::cout << "Chairs available: " << (MAXCHAIRS - readyCustomers) << std::endl;
    std::cout << "Current Haircut: " << currentClient << std::endl;
    std::cout << "Customers Served: " << customersServed << std::endl;
    std::cout << "Missed Customers: " << missedCustomers << std::endl;
    std::cout << "\n----------\n";
    fileOut << currentTime << "," << readyCustomers << "," << (MAXCHAIRS -
readyCustomers) <<
        "," << currentClient << "," << customersServed << "," << missedCustomers << "\n";
    fileOut.close();
}

int main()
{
    std::cout << "BarberShop\n";
    //file management
    std::ofstream fileOut;
    fileOut.open("BarberShop.csv");
    if (!fileOut) {
        std::cerr << "File Issues, fix your files, man." << std::endl;
        exit(1);
    }
    fileOut << "Current Time,Waiting,Chairs Available,Current Haircut,Customers
Served,Missed Customers\n";
    fileOut.close();

    //threads start
    std::thread barberThread(&barber);
    std::thread customerThread(&customer);
    barberThread.join();
    customerThread.join();
    std::cout << "---------------All threads joined.--------\nFinal Stats:\n";
    //threads end

    statusReport();

    return 0;
}
```

Sources Referenced

http://www.gerald-fahrnholz.eu/sw/online_doc_multithreading/html/group___grp_condition_variable_safe_way.html

https://cplusplus.com/reference/condition_variable/condition_variable/wait/

https://en.cppreference.com/w/cpp/thread/condition_variable

https://en.cppreference.com/w/cpp/thread/lock

https://www.softwaretestinghelp.com/cpp-sleep/