SDEV 385 - Homework Week 5

Jarred Glaser

10/04/2020

Write an analysis for this solution to the Barber problem found in your book. Does it work? Are there any issues with the solution? Is it efficient? Support your answers with examples.

```
customer(){
    while(TRUE) {
        customer= nextcustomer();
        p(mutex)
        if(emptyBarberChair < 0)
            {V(mutex); continue;}
            V(mutex);
                     P(chair);
                              p(mutex);
                              emptyBarberChair--;
                              sitInChair(customer);
                     V(mutex);
                     V(waitingCustomer);
               }
}

 theBarber (){
    semaphore mutex =1;
    semaphore chair =N;
    semaphore waitingCustomer =0;
    int emptyBarberChairs = N;
    while (TRUE) {
        P(waitingCustomer);
            P(mutex);
                emptyBarberChair++;
                    customer = acceptCustomer();
        V(mutex);
        V(chair);
        }

    fork(customer,0);
    fork(barber,0);
}
```

In the sleeping barber problem, there is some number of barbers waiitng for customers, some number of waiting customers, and some number of customers arriving to get a haircut. In this problem we want to maximize efficiency by allowing the barber to wait when there are no customers waiting and customers to wait when available barbers are busy. The steps go like this:

1. Barber checks to see if a customer is waiting

2. If no customer is waiting, barber waits
3. When a customer arrives and there is no customer currently getting a haircut, he signals the barber and gets his haircut
4. When a customer arrives and there are no available barbers he checks to see if there are available waiting seats. If there are, he sits in one, if not, he leaves
5. When the barber finishes cutting a customer's hair, the customer leaves, and the barber serves the next customers or waits if there are no more.

It is important that we ensure that we do not land in a situation where the barber is waiting and a customer is waiting (meaning the customer will never get his haircut and the program will be stuck). To do this, we use semaphore and mutex variables. These variables are:

- `Customers` - semaphore
- `Barbers` - semaphore
- `Mutex` - sempahore - mutually exclusive access to FreeSeats variable
- `FreeSeats` - regular integer

Semaphore variables can be used to signal and wait. We decrement the variable when we call `P` and increment the variable when we call `V`. A semaphore can never be below 0, so we block a thread that tries to call `V` on a semaphore that is 0. We use semaphores to control the customers and barbers. Code snippets representing the behavior of wait (P) and signal (V) on a semaphore object can be seen below:

```
// wait or P
wait(S)
{
   while (S<=0); // thread will wait until semaphore is greater than 0
   S--;
}

// signal or S
signal(S)
{
   S++;
}
```

A mutex variable is a special kind of semaphore that can only be locked by one thread at a time and can only be released by the thread that locked it. We can use it to access the FreeSeats variable for incrementing and decrementing so two threads cannot access it at once.

The first change I will make to the code snippet is to move all of the semaphore, mutex, and regular variables to be global, not within the scope of the barber. For me, this makes the variables easier to see and comprehend. I also changed the names of the variables slightly.

```
// Initialize customers and barber at 0
Semaphore Customers = 0; // counting semaphore
Semaphore Barbers = 0; // counting semaphore
Semaphore Mutex = 1; // mutex
int FreeSeats = N;
```

Next, we will first define the barber slightly differently than the example code for the assignment:

```
Barber {
    while (true) {
        P(Customers); // If customer is 0 we wait here, otherwise decrement customers by 1
        P(Mutex); // Aquire the FreeSeats variable in critical section
        FreeSeats++ // increment the free waiting seats
        V(Barber) // barber is free to cut hair
        V(Mutex) // release FreeSeats
        cutHair() // barber is now cutting hair
    }
}
```

Next we define the customer,

```
Customer {
    P(Mutex) // Aquire the FreeSeats variable for critical section
    // Check for free waiting seats, if there aren't enough the customer leaves
    if (FreeSeats > 0) {
        FreeSeats--;
        V(Customers); // notify barber by incrementing customers by 1
        V(Mutex) // Outside critical section - release mutex
        P(Barber) // if barber is 0, wait, if not decrement to 0
        getHaricut() // Get haircut
    } else {
        V(Mutex) // The shop is full, leave
    }
}
```

Other than a few minor code changes that helped make the problem a bit more clear to me, I believe the functionality between my example and the one provided in the homework solution is the same. Both solutions use the mutex to conduct actions on the FreeSeats variable in the critical section so threads do not clash while changing the number of customers who are waiting. This solution is efficient because we can assure that our code will not reach a state where the barber is waiting and the customer is waiting.

References

(2020). Mutex vs. Semaphore. https://www.tutorialspoint.com/mutex-vs-semaphore

(2019). Semaphores in Process Synchronization. Retreived from https://www.geeksforgeeks.org/semaphores-in-process-synchronization/.

(2019). Sleeping Barber Problem in Process Synchronization. Retreieved from https://www.geeksforgeeks.org/sleeping-barber-problem-in-process-synchronization/.