University of Washington Bothell
CSS503: System Programming
Program 2: The Sleeping Barbers Problem
Submitted by Snehal Jogdand

Execution Output: `./sleepingBarbers 1 1 10 1000`

```
> ./sleepingBarbers 1 1 10 1000
barber  [0]: sleeps because of no customers.
customer[1]: moves to a service chair[0], # waiting seats available = 1
customer[1]: wait for barber[0] to be done with hair-cut
barber [0]: starts a hair-cut service for customer[1]
customer[2]: takes a waiting chair. # waiting seats available = %d0
barber [0]: says he's done with a hair-cut service for customer[1]
customer[3]: leaves the shop because of no available waiting chairs.
customer[1]: says good-bye to barber[0]
barber  [0]: calls in another customer
customer[2]: moves to a service chair[0], # waiting seats available = 1
customer[2]: wait for barber[0] to be done with hair-cut
barber [0]: starts a hair-cut service for customer[2]
customer[4]: takes a waiting chair. # waiting seats available = %d0
customer[5]: leaves the shop because of no available waiting chairs.
barber [0]: says he's done with a hair-cut service for customer[2]
customer[2]: says good-bye to barber[0]
barber  [0]: calls in another customer
customer[4]: moves to a service chair[0], # waiting seats available = 1
customer[4]: wait for barber[0] to be done with hair-cut
barber [0]: starts a hair-cut service for customer[4]
customer[6]: takes a waiting chair. # waiting seats available = %d0
customer[7]: leaves the shop because of no available waiting chairs.
customer[8]: leaves the shop because of no available waiting chairs.
barber [0]: says he's done with a hair-cut service for customer[4]
customer[4]: says good-bye to barber[0]
barber  [0]: calls in another customer
customer[6]: moves to a service chair[0], # waiting seats available = 1
customer[6]: wait for barber[0] to be done with hair-cut
barber [0]: starts a hair-cut service for customer[6]
customer[9]: takes a waiting chair. # waiting seats available = %d0
barber [0]: says he's done with a hair-cut service for customer[6]
customer[6]: says good-bye to barber[0]
barber  [0]: calls in another customer
customer[9]: moves to a service chair[0], # waiting seats available = 1
barber [0]: starts a hair-cut service for customer[9]
customer[9]: wait for barber[0] to be done with hair-cut
customer[10]: takes a waiting chair. # waiting seats available = %d0
barber [0]: says he's done with a hair-cut service for customer[9]
customer[9]: says good-bye to barber[0]
barber  [0]: calls in another customer
customer[10]: moves to a service chair[0], # waiting seats available = 1
barber [0]: starts a hair-cut service for customer[10]
customer[10]: wait for barber[0] to be done with hair-cut
barber [0]: says he's done with a hair-cut service for customer[10]
customer[10]: says good-bye to barber[0]
barber  [0]: calls in another customer
barber  [0]: sleeps because of no customers.
# customers who didn't receive a service = 4
>
```

Execution Output: `./sleepingBarbers 3 1 10 1000`

```
> ./sleepingBarbers 3 1 10 1000
barber  [0]: sleeps because of no customers.
barber  [2]: sleeps because of no customers.
barber  [1]: sleeps because of no customers.
customer[1]: moves to a service chair[0], # waiting seats available = 1
customer[1]: wait for barber[0] to be done with hair-cut
barber [0]: starts a hair-cut service for customer[1]
customer[2]: moves to a service chair[2], # waiting seats available = 1
barber [2]: starts a hair-cut service for customer[2]
customer[2]: wait for barber[2] to be done with hair-cut
barber [0]: says he's done with a hair-cut service for customer[1]
customer[1]: says good-bye to barber[0]
barber  [0]: calls in another customer
barber  [0]: sleeps because of no customers.
customer[3]: moves to a service chair[1], # waiting seats available = 1
customer[3]: wait for barber[1] to be done with hair-cut
barber [1]: starts a hair-cut service for customer[3]
barber [2]: says he's done with a hair-cut service for customer[2]
customer[2]: says good-bye to barber[2]
barber  [2]: calls in another customer
barber  [2]: sleeps because of no customers.
customer[4]: moves to a service chair[0], # waiting seats available = 1
customer[4]: wait for barber[0] to be done with hair-cut
barber [0]: starts a hair-cut service for customer[4]
barber [1]: says he's done with a hair-cut service for customer[3]
customer[3]: says good-bye to barber[1]
barber  [1]: calls in another customer
barber  [1]: sleeps because of no customers.
customer[5]: moves to a service chair[2], # waiting seats available = 1
customer[5]: wait for barber[2] to be done with hair-cut
barber [0]: says he's done with a hair-cut service for customer[4]
barber [2]: starts a hair-cut service for customer[5]
customer[4]: says good-bye to barber[0]
customer[6]: moves to a service chair[1], # waiting seats available = 1
customer[6]: wait for barber[1] to be done with hair-cut
barber  [0]: calls in another customer
barber  [0]: sleeps because of no customers.
barber [1]: starts a hair-cut service for customer[6]
customer[7]: moves to a service chair[0], # waiting seats available = 1
customer[7]: wait for barber[0] to be done with hair-cut
barber [0]: starts a hair-cut service for customer[7]
barber [2]: says he's done with a hair-cut service for customer[5]
customer[5]: says good-bye to barber[2]
barber  [2]: calls in another customer
barber  [2]: sleeps because of no customers.
barber [1]: says he's done with a hair-cut service for customer[6]
customer[8]: moves to a service chair[2], # waiting seats available = 1
customer[6]: says good-bye to barber[1]
barber  [1]: calls in another customer
barber [0]: says he's done with a hair-cut service for customer[7]
customer[7]: says good-bye to barber[0]
barber  [0]: calls in another customer
barber  [0]: sleeps because of no customers.
barber [2]: starts a hair-cut service for customer[8]
customer[8]: wait for barber[2] to be done with hair-cut
barber  [1]: sleeps because of no customers.
customer[9]: moves to a service chair[0], # waiting seats available = 1
customer[9]: wait for barber[0] to be done with hair-cut
barber [0]: starts a hair-cut service for customer[9]
customer[10]: moves to a service chair[1], # waiting seats available = 1
customer[10]: wait for barber[1] to be done with hair-cut
barber [1]: starts a hair-cut service for customer[10]
barber [2]: says he's done with a hair-cut service for customer[8]
customer[8]: says good-bye to barber[2]
barber  [2]: calls in another customer
barber  [2]: sleeps because of no customers.
barber [0]: says he's done with a hair-cut service for customer[9]
customer[9]: says good-bye to barber[0]
barber  [0]: calls in another customer
barber  [0]: sleeps because of no customers.
barber [1]: says he's done with a hair-cut service for customer[10]
customer[10]: says good-bye to barber[1]
barber  [1]: calls in another customer
barber  [1]: sleeps because of no customers.
# customers who didn't receive a service = 0
>
```

Documentation

The original Shop_org program included ***service_chair*** and ***money_paid*** boolean values to determine the current state of the customer. There were individual mutex conditions to signal the barber and the customer. But as we extend the program to multiple barbers, I found this way of implementation a bit confusing and not scalable if there are other states we want to extend in future. With increase in number of barbers, the number of those variables would also increase. So, I refactored the code to have defined state for customer and barber. I decided on following 3 states for a customer when he/she is entering the barber shop.

```
enum customerState { ENTER_SHOP, WAITING_ON_CHAIR, MONEY_PAID };
```

The ***ENTER_SHOP*** is the default state for a customer. When no barber is available to service the customer, the customer would have to wait, and the state would be updated to ***WAITING_ON_CHAIR***. Once customer is then serviced by the barber, and exits the shop, we change the state to ***MONEY_PAID***.

To have a structured way of managing a barber and customer, I created following 2 structs that would help us manage individual Barber and the Customer. Having id and currentCustomer/Barber allows us to know which Barber is servicing which Customer at any given point in time easily.

```
// Structure of a Barber to manage Barber state
struct Barber {
   int id;                          // barberId
   pthread_cond_t cond;             // barber condition
   int currentCustomer = -1;        // barber assigned customer id
};

// Structure of a Customer to manage Customer state
struct Customer {
   int id;                          // customerId
   pthread_cond_t cond;             // customer condition
   int currentBarber = -1;          // customer assigned barber id
   customerState state = ENTER_SHOP; // customer current state (default:
ENTER_SHOP)
};
```

The default customer and barber would be set to -1 to signify that there is currently no one assigned for those.

To have a faster look up and update, I decided on 2 MAPS. One for Customer and one for Barbers to help fetch and update given barber and/or customer easily. The key would be the corresponding id value.

```
map<int, Customer> customers;  // map of customers
map<int, Barber> barbers;      // map of barbers
```

Also 2 queues are maintained to keep track of customers who are waiting and barbers who are still available (sleeping).

```
queue<int> waiting_chairs;
```

```
queue<int> sleeping_barbers;
```

Since this is multithreaded application, various race conditions could occur if both barber and customers are allowed to modify same set of data variables. To ensure mutual exclusion, I have used mutex which would ensure that only one of the participants can change state at once. The barber acquires this mutex lock before checking for customers and release it when they begin to sleep or cut hair. A customer acquires it before entering the shop and release it once they are in a waiting room or is getting a haircut. Now since we maintain a state and have mutex condition per Customer/Barber, we don't need the other mutex conditions listed in original program. We can use the state to determine if the customer or barber can enter into critical section.

When a customer visits the shop, the customer gets first added to waiting_chairs queue. In the queue, the customer would wait until there is a barber available to service. We ensure this by waiting on customer condition and wait till there is a barber assigned (current Barber != -1):

```
while (customers[customerId].currentBarber == -1)
pthread_cond_wait(&customers[customerId].cond, &mutex);
```

Similarly, when a customer is getting serviced, we wait for the customer condition to be resolved until the current barber assigned to customer value gets resets to -1 (currentBarber == -1)

```
while (customers[customerId].currentBarber != -1)
pthread_cond_wait(&customers[customerId].cond, &mutex);
```

Within helloCustomer(), the barber would wait until a customer state is updated to WAITING_ON_CHAIR. When this condition is resolved, it knows that the customer is now ready to be serviced.

```
while (customers[barbers[barberId].currentCustomer].state !=
WAITING_ON_CHAIR) pthread_cond_wait(&(barbers[barberId].cond), &mutex);
```

Within byeCustomer(), the barber would continue to service the customer until the state of the customer is updated to MONEY_PAID. This is done by using following while loop.

```
while (customers[barbers[barberId].currentCustomer].state != MONEY_PAID)
pthread_cond_wait(&(barbers[barberId].cond), &mutex);
```

When a customer visits the shop, we wake up the assigned barber's by using signal

```
pthread_cond_signal(&(barbers[barberId].cond));
```

Similarly, when a customer leaves the shop, we signal the barber to allow to pick up next customer from waiting list.

```
pthread_cond_signal(&(barbers[barberId].cond));
```

More documentation is added in the code to explain this logic more.

Discussions

Limitations and possible extensions of the program?
Answer:
The current program implementation, a customer waits for a particular barber to service the haircut. Example in results we see, customer[5] is waiting for barber[2]. This isn't optimized. If barber[3] is free earlier, he/she could service that same customer. This would avoid cases where barber ends up in sleep state when a customer is still waiting. Extending this program to uncouple barbers and customers and allow any barber to service any customer would be a good further optimization.

The current program provides a convenient and effective mechanism for process synchronization but is prone to timing errors which is difficult to detect. Also, since the Shop class itself is managing the entire synchronization of the different operations, it is error prone if any updates are needed. Having a separate monitor which could allow us to synchronize the data in more abstracted manner would be the next helpful extension to this program.

Also, based on the run results, there are additional optimization that could be done to ensure we service more customers. The results show that for a smaller chairs size, the number of customers that are not serviced would increase. Possibly having a dynamic size for chairs or computing it based on available cores and number of customers and barbers, instead of fixed size would ensure we service more customers.

Question:

Run your program with

```
./sleepingBarbers 1 chair 200 1000
```

where *chairs* should be 1 ~ 60.

Approximately how many waiting chairs would be necessary for all 200 customers to be served by 1 barber? *Note that depending on the number of cores and clock skews, it may very well require over 60 chairs to accomplish the task.*

Answer:

I tested my program on UW Linux labs, and the results show that it would need approximately 95 chairs to serve all customers with 1 barber. With ~60 chairs, it still shows around 30 customers who didn't receive a service.

For a smaller chair size, we observe that the number of customers that are not serviced is very high. For a larger chair size, that number decreases. The reason is that as chair size increases, we have more queue capacity for customers to wait. So, once barbers are available, they can service these customers from queue. Also one thing to note is if we have more CPU cores available, we can have more threads doing the processing and even with same chairs size, we would see more number of customers being serviced.

Question:

Run your program with

```
./sleepingBarbers barbers 0 200 1000
```

Where *barbers* should be 1 ~ 3.

Approximately how many barbers would be necessary for all 200 customers to be served without waiting?

Answer:

I tested this on UW Linux labs, and the results show that it would need 3 barbers to server all 200 customers.

For fewer number of barbers, if the number of customers is high, the waiting queue won't be enough. So, more customers would end up returning not serviced. As we increase the number of barbers, we have more threads available for processing the customers. So, we would see an increase in number of customers being serviced. Here too, the number would still depend on CPU cores available. For fewer CPU cores, we won't get enough benefit of multithreading and the customer serviced number then may not differ as much for 1 vs 3 barbers.