1 Methods Used

1.1 Design

1.1.1 Communication Mechanism

The threads communicate with each other by sharing memory. Let's say, thread A wants to tell thread B a message through a shared pointer(global) to a value of int data type. Firstly, a mutex wraps that int pointer so that both writing from A and reading from B are atomic. This mutex prevents data corruption; In this instance, When B is reading that int, A can't modify the int. The other way is the same. Once the atomic operation is guaranteed, the synchronization is achieved through conditional variables. B waits for a conditional variable fired by thread A(one-to-one communication in this example yet one-to-all communication is possible by abusing broadcasting) before B reads the int data. That is to say, B can't read the message until thread A tells B that B can read. Therefore, the synchronization feature(an order of sequence) has been enabled.

1.1.2 key steps in the logic flow of communications

Here, I would highlight some important parts of the design. You could refer to the pseudocodes in the following subsections to get design details or check out the flow diagram(Fig. 3) in the appendix to get an idea of the whole interaction chain.
The customer and barbers have a rather simpler mechanism interacting with the assistant thread. The core and main character is the assistant thread. It coordinates other threads as a man-in-the-middle. In my design, the assistant is only in charge of

- **pairing a customer in the waiting room with an idle barber**
  Customers in the waiting room is paired in a FIFO manner. In addition, I extend this assignment a little further to make sure idle barbers are waiting FIFOly as well instead of letting the OS scheduler to wake up a blocking barber thread randomly. If a customer manages to get a seat from the waiting room queue, it signals to the (if blocking) assistant thread to wake up. Similarly for barber threads. Depending upon a success of a pairing(at this moment, the assistant has dequeued a barber and a customer), the assistant thread broadcasts the id of the paired customer to all customers in the queue. The blocked customer threads are awaken and check if their id matches that of the paired one. Particularly, I let customer threads handle the same way to get the paired barber back to work so that the assistant thread can instantly jump to next pairing procedure. In addition, this way mimics that a customer sit down and tell the barber to start working.

- **initiating the procedure of closing the salon**
  This part is the most crucial as 80% of bugs occurring here though it is only 20% of the implementation(an example of Pareto rule). I use a variable *arrive* to distinguish a normal working flow and a salon closing flow. Once all customers have arrived, the assistant thread checks whether the waiting room is empty. If it is not, the assistant calls available barbers to clear the customer

queue. Then once the waiting room is empty, the assistant checks if all customers have left so as to make sure all barbers are idle(This is guaranteed because the number of arriving matches number of leaving). After this, I add an extra middle layer of communication to check the number of idle barbers is indeed the population of the barbers K and make the barbers aware that they need to wait for an instruction from the assistant then they can leave. Finally, the assistant broadcasts and wakes up barbers to leave the shop.

1.2 Expanded Details

1.2.1 Customers pseudocode

```
arrive
if the waiting room is full
    customer leave
else there's a seat in the waiting room
    queue in the waiting room
    tell the assistant the waiting room is not empty
    wait for assistant calling him
    sit
    tell the barber assigned by the assistant to cut
    wait for the barber getting hair cut done
leave
```

1.2.2 Barbers pseudocode

```
while loop
    queue in the not-working barbers line
    tell the assistant he can work
    wait for the assistant's news
    if the news is that all customers have been served
        tell the assistant he is not working
        wait for the instruction from the assistant to leave the shop
        leave
    else the news is that he has a new customer to cut hair
        wait for the customer assigned to sit
        work
        tell the customer the job is done
```

1.2.3 Assistant pseudocode

```
while loop
    if not all customers have arrived
        wait for the waiting room to be no longer empty
        wait for a barber not working
        pair the first-in customer and the first-in barber
        tell the first-in customer he can sit
    else all customers have arrived
```

```
wait for that all customers have left the shop
tell barbers that all customers have been served
wait for confirmations from all barbers that they have known that
tell barbers they can leave the shop
leave
```

1.3 Implementation

All the following operations are wrapped into a critical section by using mutex.

- customer called by the assistant
  Assistant: dequeues from the customer queue get the thread alias id, stores that id into *nextCustomerID*, dequeues from the barber queue and store *barber_id* into *nextBarberID*. and broadcasts to a conditional variable *customer_pair* to wake up all customers waiting for this conditional variable.
  Customer:

  ```
  pthread_mutex_lock(&pair_lock);
  while (id != nextCustomerID) { //not called
      pthread_cond_wait(&customer_pair, &pair_lock);
  }
  ```

- customer assigned to a barber
  The customer is awaken by the previous calling. At this moment, the lock has been released from the assistant. The customer thread simply accesses *nextBarberID* to know which barber is assigned.

- customer told completion by the barber
  Customer:

  ```
  pthread_mutex_lock(&cuts_lock);
  while (cuts[id] != 1) { //not signaled
      pthread_cond_wait(&cut_done, &cuts_lock);
  }
  pthread_mutex_unlock(&cuts_lock);
  ```

  Barber:

  ```
  servingCustomer = nextCustomerID; // mutex wrapping this Op
  pthread_mutex_lock(&cuts_lock);
  cuts[servingCustomer] = 1;
  pthread_cond_broadcast(&cut_done);//notify customer
  pthread_mutex_unlock(&cuts_lock);
  ```

- all barbers told to leave the salon
  Barber:

  ```
  while (id != nextBarberID) {//barber seat empty
      pthread_cond_wait(&barber_pair, &pair_lock);
      if (nextBarberID == -1) {
  ```

```
            pthread_mutex_unlock(&pair_lock);
            pthread_mutex_lock(&count_barbers);
            barber_no_work += 1;
            pthread_cond_signal(&barber_has_no_work);
            while(!to_shut_down) {
                pthread_cond_wait(&shutdown,&count_barbers);
                printf("Barber [%d]: Thanks Assistant. See you tomorrow!\n", id);
                pthread_mutex_unlock(&count_barbers);
                pthread_exit(NULL);
            }
        pthread_mutex_unlock(&count_barbers);
        }
    }
```

Assistant:

2 Test Cases

2.1 Boundary value analysis

There are 7 input parameters for the assignment. But we can merge rate boundary parameters into one condition, that is, on average whether the barbers cut faster than the customers come(analogical to comparing expected consuming rate with expected producing rate). To make life simply, here we assume that all numbers are non-negative and integral(≥ 1), and the upper bounder is simpler greater than or equal to the lower bound. This assumption relies on terminal inputs and there is no error handlers implemented in the source code for cases violating the assumption. For very large input numbers(e.g. non realistic 1000 barbers), the test cases won't cover as well.

2.2 Test corner cases

For you to be able to replicate the test results, I eliminate the randomness of barber working rate and customer arriving rate(simulating average rates). You can check the test results and their corresponding parameters from logs in the code folder. The case where the rates are the same, is trivially covered by either case following. There are 2 possible cases:

- barbers serve faster than customers arrive
  check 'barberfaster.log'. Barbers serve a customer per second; A new customer arrives every 2 seconds.
- barbers serve slower than customers arrive
  check 'barberslower.log'. Barbers serve a customer per 2 seconds; A new customer arrives every second.

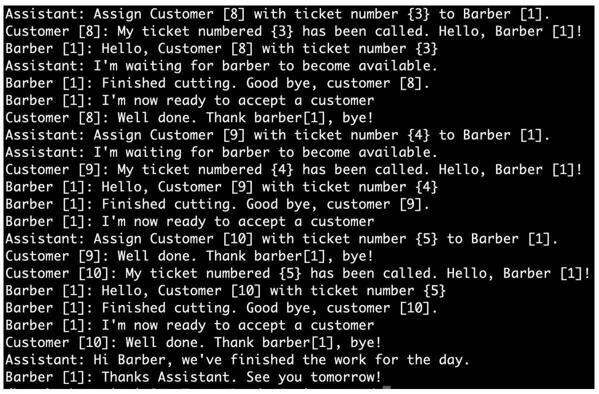
Each log has 15 cases in total from one combined parameters of the following table.

Seat #	Barber #	Customer #s
1	1	1;5;10
5	1	1;5;10
1	5	1;5;10
6	11	1;5;10
11	6	1;5;10

2.3 Important test scenarios

The correctness for most of the test cases above is trivial to comprehend. However, key test cases need more thoughtful validations. Since I use a *customer_arrived* variable to diverge, let's analyze what might happen after all customers have arrived. The moment all customers have arrived, there are two factors to take care of if we want to make sure all customers have left before we close the salon.

- Is there still a customer waiting?
  There are customers to serve still waiting in the room if customers arrive frequently, barbers work slowly and the waiting room is large enough(more waiting seats than barbers). 5 waiting room seats, 1 barber, 10 customers in 'barberslow.log' is an instance for this scenario. We expect that before the salon is closed, the assistant is busy waiting for the barber and assigning customers in the queue to the only barber frequently. And here is the output (Fig. 1)matching the expectation.

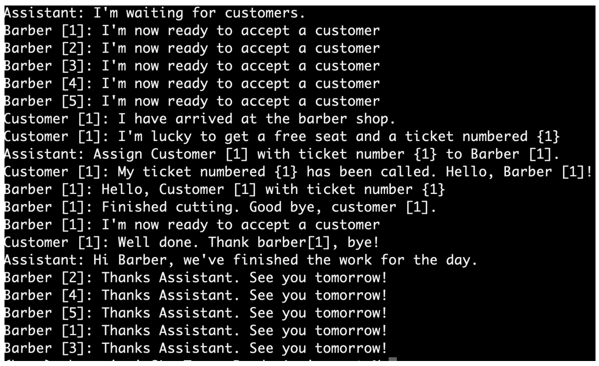


```
Assistant: Assign Customer [8] with ticket number {3} to Barber [1].
Customer [8]: My ticket numbered {3} has been called. Hello, Barber [1]!
Barber [1]: Hello, Customer [8] with ticket number {3}
Assistant: I'm waiting for barber to become available.
Barber [1]: Finished cutting. Good bye, customer [8].
Barber [1]: I'm now ready to accept a customer
Customer [8]: Well done. Thank barber[1], bye!
Assistant: Assign Customer [9] with ticket number {4} to Barber [1].
Assistant: I'm waiting for barber to become available.
Customer [9]: My ticket numbered {4} has been called. Hello, Barber [1]!
Barber [1]: Hello, Customer [9] with ticket number {4}
Barber [1]: Finished cutting. Good bye, customer [9].
Barber [1]: I'm now ready to accept a customer
Assistant: Assign Customer [10] with ticket number {5} to Barber [1].
Customer [9]: Well done. Thank barber[1], bye!
Customer [10]: My ticket numbered {5} has been called. Hello, Barber [1]!
Barber [1]: Hello, Customer [10] with ticket number {5}
Barber [1]: Finished cutting. Good bye, customer [10].
Barber [1]: I'm now ready to accept a customer
Customer [10]: Well done. Thank barber[1], bye!
Assistant: Hi Barber, we've finished the work for the day.
Barber [1]: Thanks Assistant. See you tomorrow!
```

Figure 1: waiting room not empty before closing

- Is there still a barber working?
  The waiting room is empty but some barbers are still working. This typically happens when the waiting room size is small, barbers work much more slowly. An extreme case is 1 waiting seat, 5 barbers, 1 customer and barbers work slowly. We expect that when the only customer served, the shop is instantly closed. Here is the output (Fig. 2) matching the expectation.

Once two conditions above have been satisfied, the assistant is safe to close the shop as all customer threads have terminated and all barber threads are blocked(waiting for the same conditional variable). At this moment, the assistant's broadcasting is guaranteed to synchronize barber threads and wake them to terminate.



```
Assistant: I'm waiting for customers.
Barber [1]: I'm now ready to accept a customer
Barber [2]: I'm now ready to accept a customer
Barber [3]: I'm now ready to accept a customer
Barber [4]: I'm now ready to accept a customer
Barber [5]: I'm now ready to accept a customer
Customer [1]: I have arrived at the barber shop.
Customer [1]: I'm lucky to get a free seat and a ticket numbered {1}
Assistant: Assign Customer [1] with ticket number {1} to Barber [1].
Customer [1]: My ticket numbered {1} has been called. Hello, Barber [1]!
Barber [1]: Hello, Customer [1] with ticket number {1}
Barber [1]: Finished cutting. Good bye, customer [1].
Barber [1]: I'm now ready to accept a customer
Customer [1]: Well done. Thank barber[1], bye!
Assistant: Hi Barber, we've finished the work for the day.
Barber [2]: Thanks Assistant. See you tomorrow!
Barber [4]: Thanks Assistant. See you tomorrow!
Barber [5]: Thanks Assistant. See you tomorrow!
Barber [1]: Thanks Assistant. See you tomorrow!
Barber [3]: Thanks Assistant. See you tomorrow!
```

Figure 2: A barber is working while the waiting room is empty

2.4 Test normal cases

There is no test cases for normal flows appended in the source code folder. Please feel free to(and I encourage you to) introduce randomness and arbitrary people numbers but please make # of customers small, say 20. Otherwise it takes a long time to simulate(100 seconds for 100 customers) even if customers arrive every 1 second.

3 Limitations

- data structure overkill
  The FIFO feature is enabled by a self-implemented queue data structure with APIs. This is convenient as it is a well known data structure. However, the data structure occupies extra memory space, which is a waste. To improve, a modulo operation on an int kept by the assistant should be enough but needs to redesign the enqueue() and dequeue() manually so that FIFO order remains.

- number of mutex
  If the previous one is improved, I can use fewer locks as least I can merge the lock for customers' queue and the one for barbers' queue into a single one.

- change brainless one-to-all communication to targeted one-to-one communications
  6 broadcasts for different conditional variables are used. Those awaken blocked threads acquire the lock again at least for a moment. If this tiny amount of time accumulates, it's still a waste of running time. However, this requires increasing the number of conditional variables. Therefore I am not sure what kind of trade-off makes the system optimized.

- code structure
  In the assistant routine, the same chunks of code for waiting available barbers are used. Perhaps I should have wrapped that chunk as a single function and call it when I need to?

Appendix

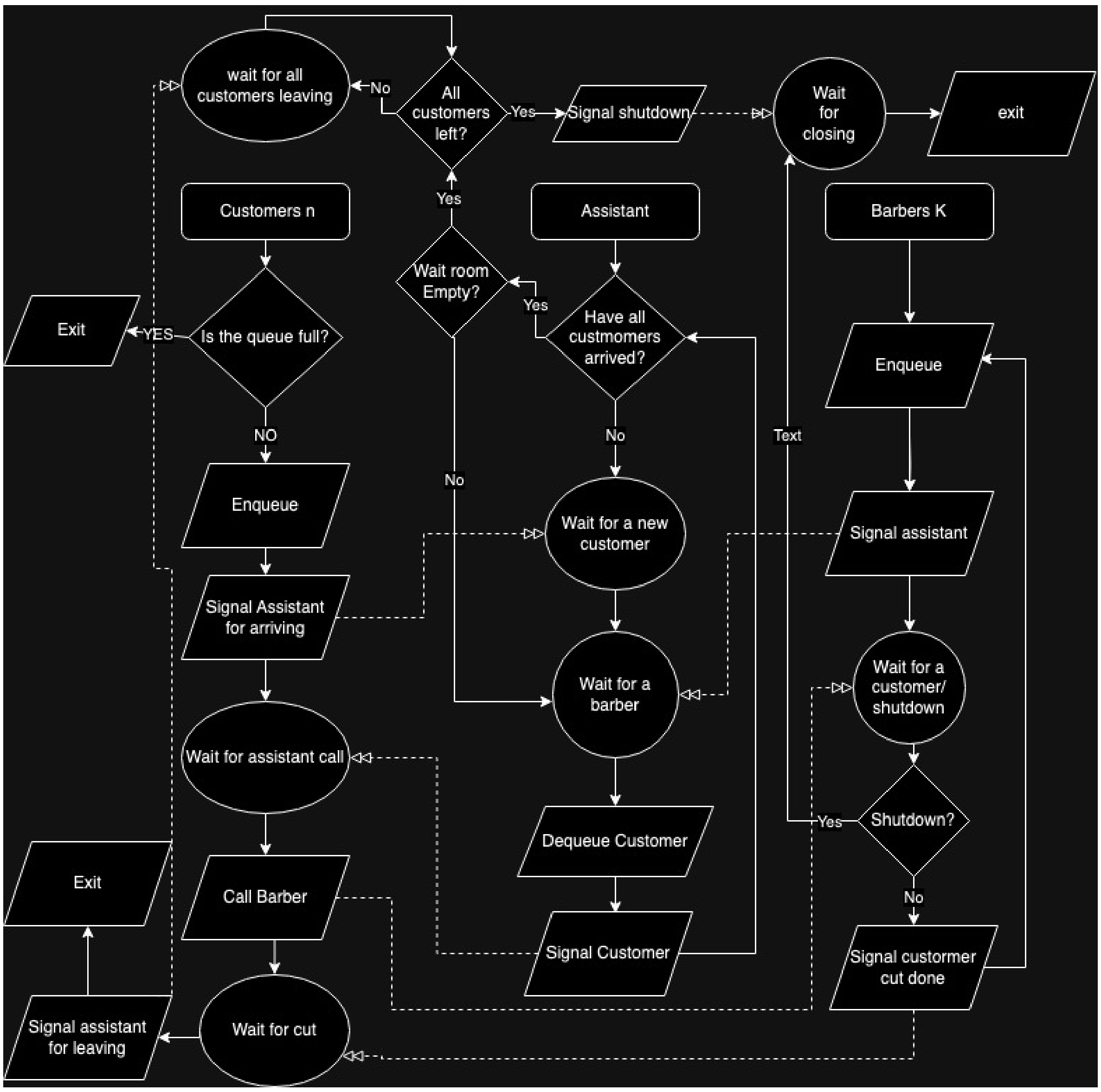


Figure 3: The overall interaction mechanism