# Assignment 2 Design

## 1. How many threads are you going to use? Specify the task that you intend each thread to perform.

I am going to use 4 threads including the main initial thread.

There will be two clerk threads. These threads are the "clerks" and will grab customers from the queue and service them. The critical section for the clerk threads will be grabbing new customers from the queue. After a customer has been grabbed, the lock can be released and the clerk can service the customer. After the customer is processed, the total service time global variable will be incremented.

The third thread I will use is a thread for managing the customers arrival time. All this thread will do is wait for a customer to arrive and add them to the queue when they have arrived.

The final thread will be the initial main thread. This thread will create the mutexes and condition variables, start the clerk and customer threads, wait until they are finished, and then print out the average service time.

## 2. Do the threads work independently? Or, is there an overall "controller" thread?

There is no controller thread in the sense that a single thread is managing the others. Each thread will run independently from the other threads. The only "communication" is unlocking of mutexes or signaling of condition variables.

The clerk threads do depend on the customer thread in order for them to receive customers, though. What is nice about the condition variable and mutex design is that there is no direct communication between threads. A thread will just wait on some resource until it can access it safely. When a clerk thread has no work to do it will just wait on a signal letting it know a customer has been added to the queue. There is no polling or wasted CPU time. The operating system efficiently manages the locks and condition variables.

## 3. How many mutexes are you going to use? Specify the operation that each mutex will guard.

I will use two mutexes.

The first mutex will guard the 4 customer queues. Since the length of each queue will need to be compared to each other, I figured a single lock for all of the queues would suffice.

The second mutex will guard the total service time variable. This will ensure the average service time for all the customers is calculated properly.

## 4. Will the main thread be idle? if not, what will it be doing?

The main thread will start the customer and clerk threads. After that it will be idle and will wait for each thread created to finish.

## 5. How are you going to represent customers? What type of data structure will you use?

I will represent the customers with a typed struct data structure.

```
typedef struct _Customer Customer;
struct _Customer {
  int user_id;
  int service_time;
  int arrival_time;
};
```

I will then use a linked list for storing the list of customers in each queue, as well as the list of customers that have not yet arrived.

```
typedef struct _CustomerNode CustomerNode;
struct _CustomerNode {
  struct _Customer *customer;
  struct _CustomerNode *next;
};
```

The advantage of using a linked list is that removing and inserting customers does not require copying or shrinking arrays. It only requires the manipulation of pointers.

## 6. How are you going to ensure that data structures in your program will not be modified concurrently?

There will be 5 variables which will be modified and read by multiple threads. These are the 4 customer queues and the total service time variable. I will ensure these data structures will not be modified concurrently by using mutex locks when writing to them. The `queue_lock` will prevent concurrent access to the 4 customer queues. The `avg_lock` will prevent concurrent access to the total service time variable.

## 7. How many convars are you going to use? For each convar

- (a) Describe the condition that the convar will represent

- (b) Which mutex is associated with the convar? Why?
- (c) What operation should be performed once pthread cond wait() has been unblocked and re-acquired the mutex?

I will only use a single convar.

- (a) The condition it represents is a customer being added to some queue
- (b) The queue lock mutex is associated with it. This is because when the condition variable is signaled, the thread who receives the signal and wakes up will be accessing the customer queues. Therefore it will need to acquire a lock to access them. When there is no work for the clerk thread to do, it should also unlock the queue lock until it receives a signal that there is work to do.
- (c) When the clerk thread has been unblocked by pthread cond wait, it means that there is a customer in a queue that needs servicing. So when the clerk thread is unblocked by the condition variable, it should grab the next customer from a queue, unlock the mutex, and then process the customer.

## 8. Briefly sketch the overall algorithm you will use.

The pseudocode for my algorithm is as follows.

### Clerk Threads

```
void clerk() {
  while(true) {
    pthread_mutex_lock(&queue_lock);

    /* Critical Section */
    if (customers_remaining() == 0) {
      // Signal to the other clerk thread if it is waiting
      pthread_cond_signal(&queue_cond);
      pthread_mutex_unlock($queue_lock);
      pthread_exit(NULL);
      return;
    }

    customer = customer_from_queue();
    if (customer == NULL) {
      // Wait for customer to be added to queue
      pthread_cond_wait(&queue_cond, &queue_lock);
      customer = customer_from_queue();
    }
    pthread_cond_signal(&queue_cond);
    /* End Critical Section */

    pthread_mutex_unlock($queue_lock);
    process_customer(customer);
  }
}

void process_customer(customer) {
  time = get_current_time();
  printf("A clerk starts serving a customer: start time %.2f,
          the customer ID %2d, the clerk ID %1d. \n");
  usleep(customer.service_time);
  printf("A clerk finishes serving a customer: end time %.2f,
          the customer ID %2d, the clerk ID %1d. \n");

  pthread_mutex_lock(&avg_lock);

  /* Critical Section */
  total_service_time += get_current_time() - time;
  /* End Critical Section */

  pthread_mutex_unlock(&avg_lock);
}
```

**Customer Thread**

```
void customers() {
  sort_by_arrival_time(&customers);

  while(customers_remaining() > 0) {
    next_customer = customers[0];
    usleep(get_time() - next_customer.arrival_time);
    pthread_mutex_lock(&queue_lock);

    /* Critical Section */
    add_customer_to_queue(next_customer);
    pthread_cond_signal(&queue_cond);
    /* End Critical Section */

    pthread_mutex_unlock(&queue_lock);
  }

  pthread_exit(NULL);
}
```

**Main Thread**

```
int main() {
  create_queue_mutex();
  create_avg_mutex();
  create_queue_convar();

  create_cleark_threads();
  create_customer_thread();

  wait_for_threads();

  printf("The average waiting time for all
          customers in the system is: %.2f seconds. \n");

  pthread_mutex_destroy(&queue_lock);
  pthread_mutex_destroy(&avg_lock);
  pthread_cond_destroy(&queue_cond);

  return 0;
}
```