# Design Document

Sydney Macdonald

V00978481

A02

- Each customer will be represented by a thread. Each thread will simulate customer arrival, add the customer to their respective queues, access a clerk when one is available, simulate being serviced, and then release the clerk to be used by another customer. The thread then adds its waiting time to a global customer waiting time and terminates.
- Only customer threads and one main thread are active/created in this program.
- All customer threads are created when the file is read in and immediately wait on a convar until all threads have been created and the simulation is ready to begin. This is to synchronize the arrival times of customers and ensure they don't begin being serviced before all customers have been created. Arrival time is simulated with `usleep(arrival_time * 100000)`.
- Each thread works independently, and there is no controller thread that does the scheduling - the customer threads schedule themselves.
- The main thread initializes all mutexes, convars, and semaphore and starts the simulation time and all customer threads at the same time. It is then idle until each customer thread terminates, and the main thread then joins them all. The main thread then calculates overall waiting times and outputs them before terminating.
- Clerks are simulated by a semaphore with an initial value of 5. Clerks are initialized by the main thread in a queue (linked list).
- Customers add themselves to a queue (linked list) to ensure they are serviced first-come-first-served - though the order customers are added to queues is not guaranteed for customers who arrive at the same time.
- All data structures (the customers queue and clerks queue) are only modified when the calling thread owns the mutex associated with the queues. The same is true for global variables that need to be accessed by multiple threads.
- There are 5 mutexes in this program:
  - `queues_mutex`: guards updates to the customer queues and queue lengths including enqueuing and dequeuing customers

- `clerks_mutex`: guards access to clerks - this mutex is associated with the `clerks_convar` convar that customers in queue will wait on
- `clerk_id_mutex`: guards updates to the clerk queue including enqueuing and dequeuing clerks when they finish servicing or begin servicing a customer, respectively
- `start_synch_mutex`: guards starting the simulation - this mutex is associated with the `start_synch_convar` convar which all customers wait on when they are created, until all threads are created and the simulation can start
- `wait_time_mutex`: guards updates to the global customer wait times - each customer adds their wait time to either the business or economy total wait time once they are finished being serviced

- There are 2 convars in this program:
    - `clerks_convar`: associated with `clerks_mutex`, this convar represents the condition that customers can't leave the queue unless there is a clerk available and they are at the front of the queue (and for economy customers, that there are no business customers to service first). Once `pthread_cond_wait()` has been unblocked, each thread that meets the conditions required to access a clerk unlocks the mutex and accesses a clerk through the `sem_wait()` function.
    - `start_synch_convar`: associated with `start_synch_mutex`, this convar represents the condition that no customer can start simulating arriving at the queue until all customer threads have been created and the simulation is ready to start. Once `pthread_cond_wait()` has been unblocked, each thread unlocks the mutex and starts the simulated arrival time operation.
    - In both cases, the mutexes associated with the convars do not protect an operation or access to shared data like usual. This is due to the way my program was designed and the fact that clerks are simulated using a semaphore. In both cases the mutex & convar are used to block threads from proceeding until a condition is met.
- Overall algorithm:
    - Initialize all mutexes, convars, semaphore, and clerk queue
    - Read in n customers and create a thread for each
    - For each customer i:
        - When customer i arrives, they enter either the business or economy queue based on their type
        - Customer i waits until condition: *there is an available clerk and they are at the front of their queue and there are no business-class customers to service first if they are economy-class*, is true
        - Customer i decrements the number of available clerks and dequeues the next available clerk, j
        - Customer i is serviced by clerk j

        - When customer i is done, they enqueue clerk j and increment the number of available clerks, and wake up all remaining waiting customers to check their condition
        - Customer i adds their waiting time to the total waiting time for their class
        - Customer i terminates
    - Join all customer threads
    - Once all customer threads have terminated, calculate total and average waiting times
    - Destroy mutexes and convars