CS481 PA06

Ryan Scherbarth & Danny Metcalfe

November 2024

1 Code Explanation

Code Repository: https://github.com/rysc3/PA06

1. Threading plan:

```
// Creating the simulation thread
pthread_create(&sim, NULL, simulation, NULL);

// Creating the car threads
for (int i = 0; i < total_cars; i++)
{
  pthread_create(&cars[i], NULL, car, NULL);
}

// Joining all the threads
pthread_join(sim, NULL);
for (int i = 0; i < total_cars; i++)
{
  pthread_join(cars[i], NULL);
}
```

   We first create a simulation thread that runs the simulation function. This function will mainly update arrivals, waiting people, and handle adding arrivals each minute. It will also broadcast to cars when to start the loading on each minute. We then make a thread for each car. The function car is going to wait on a broadcast to ensure it is synchronized to each minute after we get new arrivals. It will also sleep to simulate loading and ride time as well as subtract from waiting and add to statistics.

2. We have 1 virtual minute synchronized to 1 second in our code.

3. We used two mutex locks. One mutex lock is named lock and this is just used in basic critical sections involving the number of people waiting so there is no conflict between cars and simulation threads regarding how many people are currently waiting. There is one more lock named mutex_clock and this is used when accessing our current time to keep track of time to know when to end the car threads. We have one condition named sync_cond and this is what is used to synchronize to each minute. So that the cars must wait until we have received new arrivals then we can load the cars and run them. We broadcast to the cars once we complete all arrival computations in the simulation thread. We include sleeps to help simulate the time and allow the cars to run.

4. There is race conditions involving both people waiting as well as the current time variable. We avoid these by using two locks as described in number 3. We have one lock named lock and this is used to achieve a critical section on the waiting people variable to avoid any race conditions on this variable between threads. Whenever we read or access this variable we lock the lock in both the simulation

and car threads. There is another race condition involving the current time variable and this is solved in a similar way as explained in question 3. We do a lock whenever we access this variable using mutex_clock lock in both the simulation and car threads.

5. We mainly synchronize by allowing the simulation thread to sleep the needed time and we use a condition variable as explained in question 3 to have the cars wait on the minute. So after our sleep in simulation we get arrivals then allow the cars to continue followed by another sleep. This allows us to keep the cars synchronized with the simulation minutes.

2 Table Output

(N, M)	Total Arrival	Total GoAway	Rejection Ratio	Average Wait Time (mins)	Max Waiting Line
N=2, M=7	19999	10820	54.10%	22	800
N=2, M=9	19999	8417	42.09%	21	800
N=4, M=7	19999	3170	15.85%	15	800
N=4, M=9	19999	947	4.74%	10	800
N=6, M=7	19999	0	0%	4	667
N=6, M=9	19999	0	0%	0	77

Table 1: Table of stats for simulation output

3 Figure Outputs and Descriptions

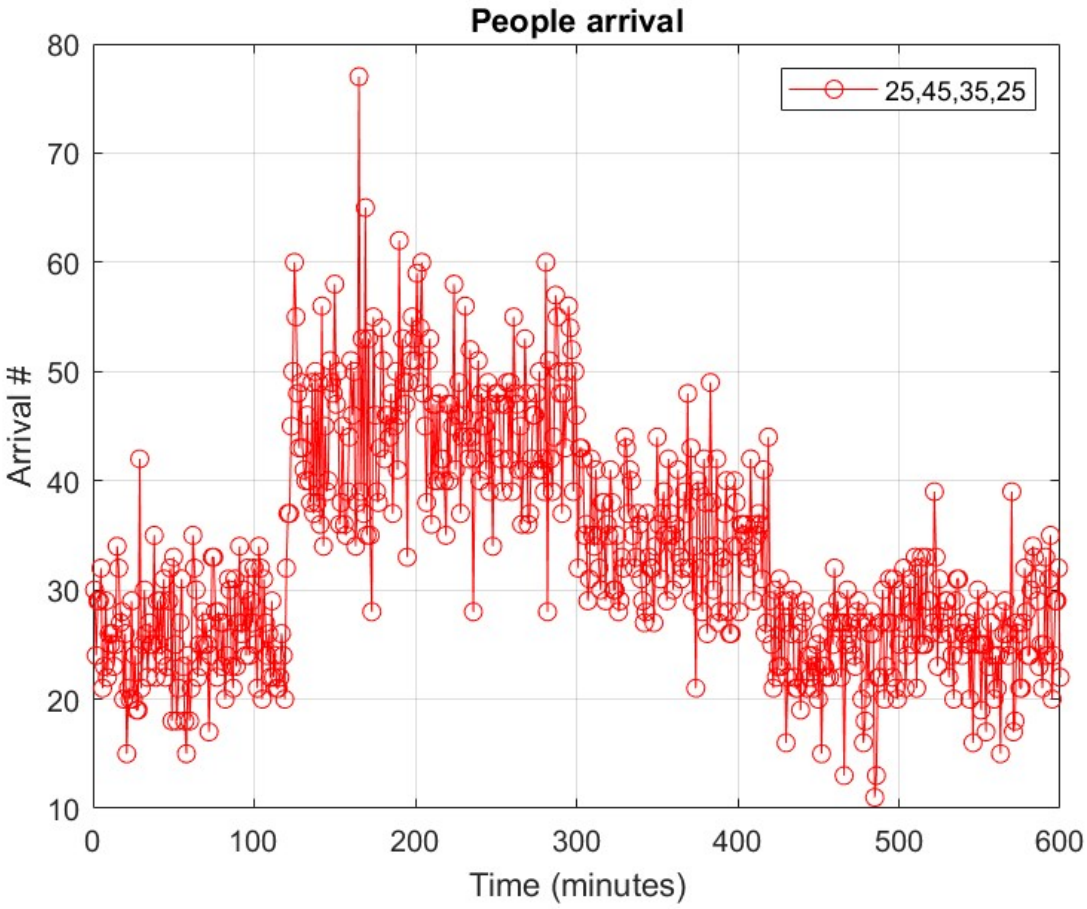

Figure 1: Graph of people arriving vs time

Figure 1 shows the arrivals on every minute of the simulation and we can clearly see the mean times we were given in the figure.

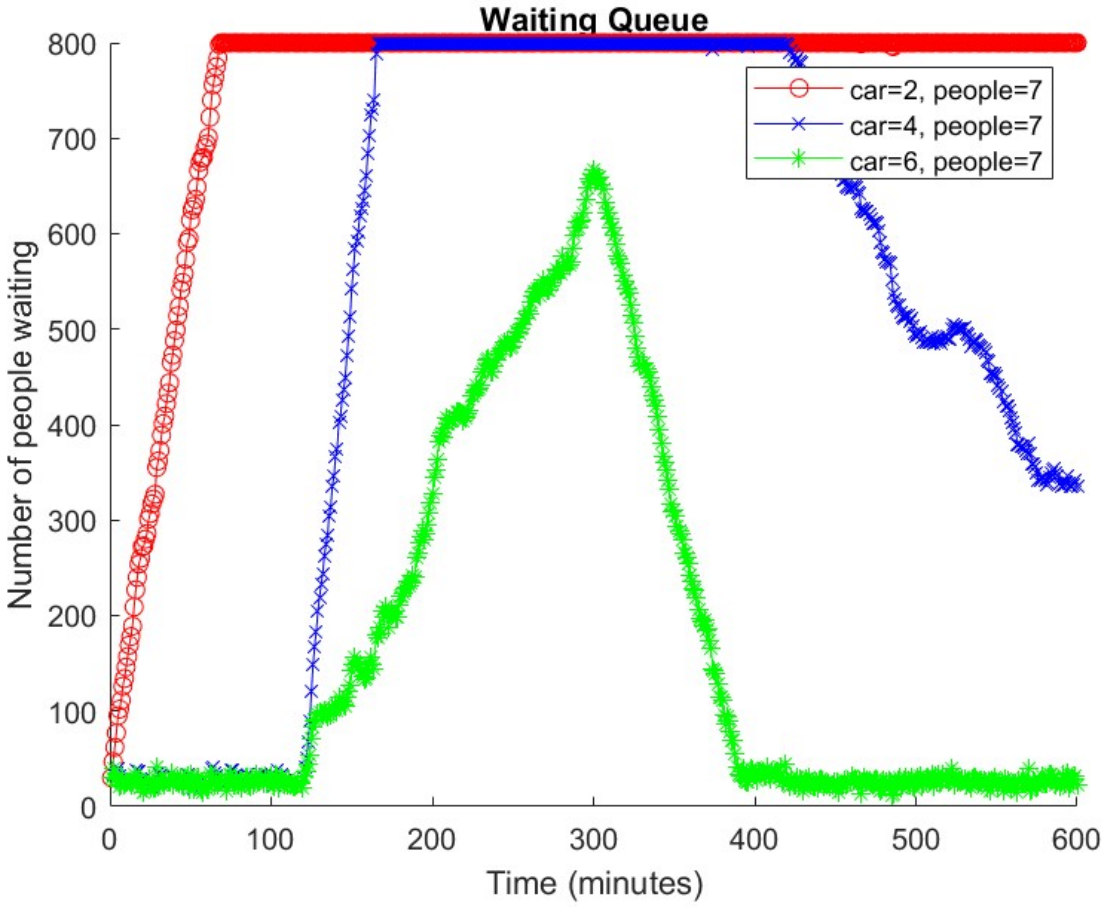

Figure 2: Graph of people waiting vs time for different car amounts

Figure 2 shows the waiting queue for different car numbers all with 7 people as their capacity. We can see as we increase the number of cars there are overall less people waiting especially during the peak times with 45 and 35 mean arrivals.

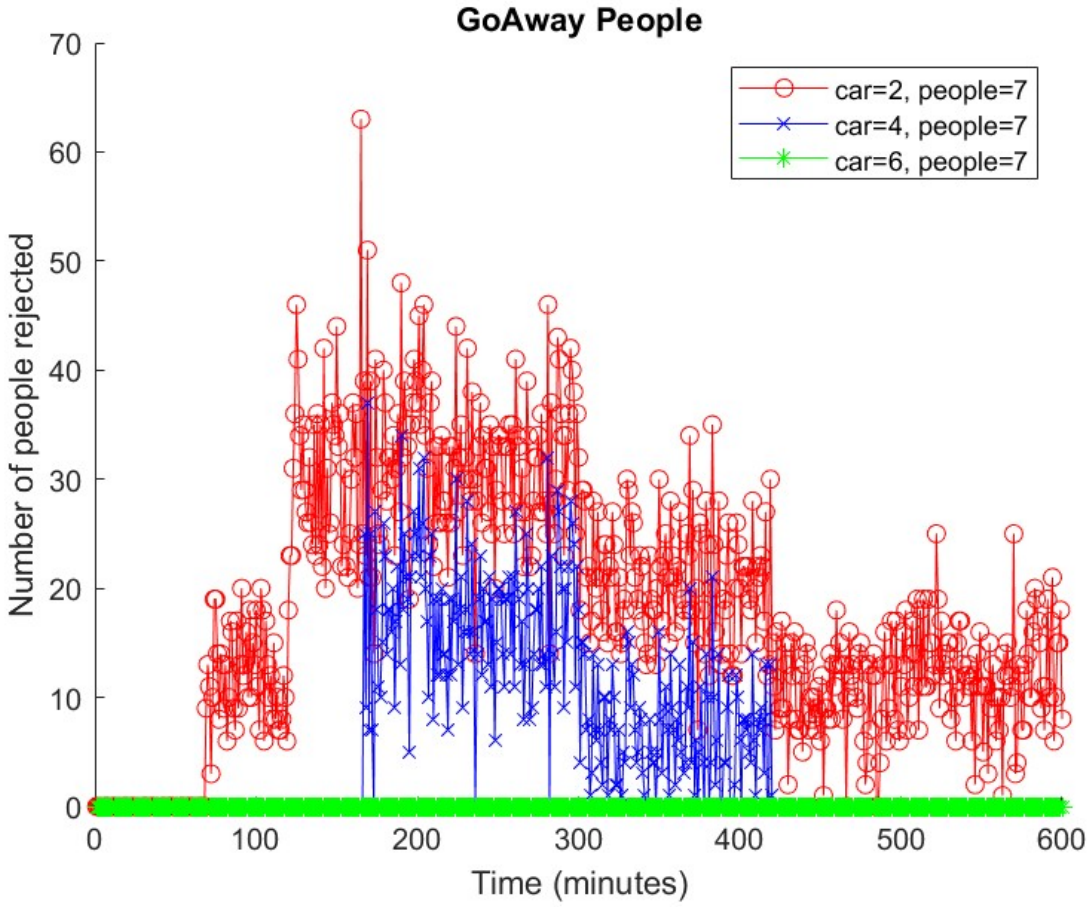

Figure 3: Graph of rejections vs time

Figure 3 shows the number of rejections for 2, 4, and 6 cars all with 7 capacity. It is clear that rejections decrease as the number of cars increase. With 6 cars there are no visible rejections in this graph where 2 cars has a lot of clear rejections.