• Describe your solution to the problem (in words, not in code)

My solution to this problem starting from the beginning of the code (glossing over trivial code) was using malloc to keep dynamic amount of voters and voting booths. Using threads to create the voters in a thread array. I then used semaphores to protect critical sections of globally shared variables such as counts, and stdout. I also used a queue to hold lines of voters. I implemented a counting semaphore(sem_booths) to keep a count of how many voters are allowed in the booths at any one time. Once the voting booths are all full, voters wait on sem_booths to signal.

• How does your solution avoid deadlock?

Interweaving these semaphore along with other critical semaphore (sem_printf, mutex_second_line_count, etc.) could cause deadlock. This could be a problem had I not almost always wait on a semaphore then almost immediately signal it again. For example, when using mutex_second_line_count, I manipulate or view the critical variable second_line_count, then immediately post the semaphore again. When I do use another semaphore with in mutex_second_line_count near the end of voter(void * thread_args), I immediately post the semaphore, eliminating the chance of deadlock.

• How does your solution prevent Republican and Democrat voters from waiting on a voting booth at the same time?

What was truly difficult for this assignment is figuring out how to keep fairness. The voters could not just budge to the front of the line. First, the idea is to keep a queue for all parties. All voters are kept in a queue. Then take the first voter from the queue and look at what party they represent. If the voter is allowed to wait or vote at the voting booths then allow them

to pass, tripping the next voter in the queue. Let's say for this example, the next voter is not allowed to vote yet, there must be away to keep all other voters back while this voter waits. It's simple, don't trip the next voter in the queue until the voter can pass. Sem_repub_dem_wait then is used to keep this voter back. Once the first voter goes through trips the sem_repub_dem_wait to allow any voter waiting on the sem_repub_dem_wait to go through, changing the current party voting. Then triping the next voter in the queue to send the next voter.

• How is your solution "fair"? How are you defining "fairness" for this problem?

My solution is fair, since all voters will be able to vote at some time (no starvation). And whoever gets into the queue first, will be the first to vote, the second voter to get into the queue will vote second, etc. The implemented queue will not allow threads to vote before the thread before that thread goes. Fairness is making sure everyone votes, and no one can budge into the queue.

• What happens if a long line of Democrats start waiting to vote and a Republican shows up late? Should the late Republican be able to vote before all of the Democrats have finished?

No, this is not possible in my solution. Since the queue keeps all voters in line, all voters before the late republican would have to be either voting, done voting, or waiting for a booth. There is no line budging.

• How does your solution prevent starvation? How are you defining "starvation" for this problem?

If I were to implement voters continuously coming back to vote, my current implementation of finicky voter prevents starvation. Since the queue keeps all parties back, but also keeps the First In First Out functionally. When a voters wait in the queue, whoever is

waiting on queue first will be the next to do the check for what type of voter they are. If the voter cannot vote yet, queue does not allow another voter to do the check for what type of voter they are, until the first voter has completed the check. With this implementation, voters will not starve as they will eventually get their chance to vote, while waiting in the queue.