OS Project Two Report

The approach I took to approach a solution to the problem given to us for this project was almost analogous to an OOP one that would’ve been written in C++. I found it to be much more clean, maintainable, and understandable when dealing with the problem that way instead of ending up with spaghetti code. Moreover, this method allowed for more ease in debugging of the multithreaded program since it was much easier to isolate when a problem or a deadlock appeared.

***Actors***

* **Men and Women thread arrays**: these can be considered to be the actual people who want to do their business in the given scenario.
* **PERSON**: This class (technically a struct but they’re pretty much interchangeable in this context) holds an integer id and a Boolean isMan. isMan is pretty self descriptive. The parameter id is just the index of the thread array. Later on, this struct determines which person will enter the bathroom at the given moment by determining which of the above thread arrays to look at and which index through the parameters discussed above.
* **BATHROOM\_QUEUE**: This class is basically a lazily implemented vector with a few parameters to help with synchronization and keeping track of which person is next in line. The size of the queue is initialized in the beginning to n+m where n and m are the total number of men and women. It is implemented this way because practically a real life queue does not have any maximum capacity. Plus this program is used as a proof of concept and has no real life usages so using big numbers that would cause the program to exceed the allowed memory space is not really in the scope of this project.
  + This class holds a PERSON type class array which has its size determined dynamically in the corresponding init function of the class. Using the corresponding enqueue function, a PERSON is initialized and “enqueued” into the internal queue. The parameter size is used to keep track of how many people got enqueued.
  + The bathroom queue also holds a nextInLine\_index parameter which is used to point at the person that would be next in line. Initially initialized to zero and incremented by the corresponding deque function as when we dequeue from the queue the next in line would be nextInLine+=1.
  + Synchronization: I used semaphores here to take care of synchronization. Two semaphores are included in this class.
    - Enqueue\_mutex: Given the nature of how the dispatching process is implemented, two people could’ve gotten enqueued at the same time. So, to mitigate this issue, I used this semaphore to block the critical section of enqueuing a person and then release the lock on exit of the function.
    - Dequeue\_mutex: When the corresponding dequeue method of this class is called, what is basically happening a request to give back a person. To avoid wasting cpu cycles and checking each time if there is a person in the queue first or not, then if there is not loop and check and so on, I used semaphores to implement a non-busy wait. Basically this mutex is initialized to zero then when a person get enqueued, in the enqueue function, the dequeue mutex get incremented (i.e. you can now dequeue someone as someone is already there). The first thing the deque function does is that it waits on the mutex which decrements it if it has a value more than zero, otherwise, it waits until a person X gets enqueued then as soon as that thread is switched back to, person X that just got in, gets dequeued. Just like real life, you stand in front of the restroom and then open the door to go in. Yes, I consider that to be a queue of one.
* **Dispatcher**: This “class” isn’t defined as per the definition of the class practically but abstractly; I came up with some kind of a dispatcher to dispatch men and women to the bathroom queue. There are two threads that take care of this. dispatchMen, and dispatchWomen (can be found in the main function). These two threads are created and run dispatchMen\_fn and dispatchWomen\_fn found in the dispatch\_utils.h file. What these mirrored functions do is enqueue, using the bathroom queue’s corresponding function, a man or a woman, depending on which function we’re talking about, then waits for a random amount of time between 0 and 0.1 seconds then dispatches another person until all men or all women are dispatched then the two threads should join at the end of the main function.
* **BATHROOM**: This is the class where the best of the action happens. It mostly contains of semaphores and to integer values. To best describe this class, I feel like I should dissect its corresponding functions.
  + **Bathroom\_init**: Initializes the values. All but one of the semaphores are initialized to 1. Full\_mutex is initialized to the capacity of the bathroom so that in the future, when someone is at the front of the queue and wants to enter, not only would they have to wait until the other gender is no longer inside, but also even if their own gender is inside, they’d have to wait until the bathroom is no longer full. We’ll see this in the wants\_to\_enter functions below.
  + **X\_wants\_to\_enter:** Both of these functions are mirror functions with inverted logic. What basically happens here is that firstly, we check if no one is entering the bathroom, through waiting for bathroom\_entry\_mutex, at this given moment. This may never happen in my implementation and could’ve been deprecated as a feature, but it doesn’t hurt and if more than one queue was used it would contribute to mimicking the real life logic of two queues and one bathroom (i.e. only one person can enter at a time). It also streamlines the logic of entry and exit. So by waiting for entry\_mutex, we ensure that only one person wants to enter the bathroom now at this given moment, and by doing so similarly in the exit functions, we would have only one person trying to exit at this given moment. So if it happens and one person wants to enter and the other wants to exit at the same given moment, the one who uses the door first (i.e. locks bathroom\_door\_mutex), would get to enter/exit first. This mutex is closest to the critical section so that all checks have been made and finally the person may now enter or leave the bathroom. However, before, the thread may reach for the door, we need also check for the fullness of the bathroom. Thus we wait full\_mutex. If it is zero, then the bathroom is full and we have to wait. Yet, bear in mind that it was inited to 5. Therefore, it would take 5 entries and no exits to reach zero. Also, upon calling wait, the full\_mutex is decremented. However, it is not necessary to include it in the critical section as its main purpose in this function is to check whether for the fullness of the bathroom. The critical section for this function is as follows. We wait on gender Y’s mutex, where gender Y is the other gender and X is the gender using this function, if we were the first of gender X to enter through the if statement present there (i.e. we check if people\_present is zero, if it is, this means that this is the first person to enter after the last of the other gender has exited, or if the bathroom was completely empty before. If this is the case, we want to stop the other gender from entering). Thus, Now it should be clear why we do the wait for X\_mutex; to wait until all the members of the other gender have exited. Then, we increment the people present and unlock bathroom\_door\_mutex and X\_mutex to allow other people to enter or leave the bathroom and so that other X members are not blocked by this an X member too (basically undoing the wait used to check if Y members are present) and finally unlock bathroom\_entry\_mutex.
  + **X\_leaves:** This function implements a similar logic to the enter functions. It first uses the exit\_mutex, so that only one person may be entering at a time. Then the person reaches for the door. So drawing from the above section, only one person may be entering at this moment and now also, only one person may be exiting at this moment. Therefore as above if it happens that both scenarios happen at the same time, we don’t want the integrity of people\_present to get compromised as incrementation is not an atomic operation and also we need to keep the integrity of full\_mutex as we’re more or less treating it as a variable in this context to keep count of people present and signal to the person entering to proceed. Upon entering the critical section we decrement people\_present and if so it happens that this person was the last of gender X to leave (i.e. people\_present == 0), then we post Y\_mutex, to finally allow the other gender to enter, if the person next in line is of the other gender. It basically just sets the mutex back to one. Then we post full\_mutex to signal that the bathroom is no longer full if it was otherwise this is just used to keep track of the fullness and allow people to enter once it is no longer full as mentioned above. Then we unlock the door for other people to use and unlock the exit\_mutex to allow other people to enter.
  + **Destroy:** just destroys semaphores.
* **De-queueing Mechanism and Using the Bathroom:** This is not an actor perse but it is important to highlight how I managed to maintain the integrity of the queue and people leaving and entering from the outside of the BATHROOM class and how the actual threads are used. The for loop in main.c at line 52 is basically all there is to the logic. We repeat it for n+m times as the system contains n+m people that will get their business over with eventually. The person next in line gets dequeued from the bathroom\_queue. As mentioned before, if there is none, then this function would hang until someone gets enqueued from the dispatching threads. If someone is there we signal to the bathroom that someone wants to enter and whether they’re a man or a woman through using the appropriate functions by checking the parameters of the dequeued person object. Now the information about which person just went into the bathroom is printed through the wants\_to\_enter function, in the critical section, so that the printing is accurate. Then, we use the man or woman threads we created at the very beginning to call the goto\_bathroom function.
* **goto\_bathroom:** This function sleeps for a random amount of time (0 to MAX\_BATHROOM\_TIME u-seconds), which represents the amount of time the person gets to do their business in. Then after they finish, naturally, they’d want to get out. So, a request for leaving the bathroom is issued by the thread. Even if it happens and two threads compete for leaving and one gets to go out before the other, it doesn’t compromise the scenario as the extra wait time for the thread that issued the request first can be attributed to extra “business time”. The important part was maintaining the integrity of the queue which was done by serially calling the bathroom\_x\_wants\_to\_enter function in the for loop in the main thread as mentioned in the above section and also through the mechanisms discussed in the BATHROOM section.

***Conclusion***

Through a number of synchronization methods and thread-safe actors, I was able to achieve a simulation of real life queueing for a bathroom and enforce the rules imposed on us by the problem header. The task was easy when a divide and conquer approach was taken as it separated the multi-threaded logic of each entity and allowed for a harmonious flow of data, without the loss of accuracy or integrity. The dispatcher, queue, and bathroom have no problem working simultaneously and therefore, all requirements should’ve been met. The scenarios and possibilities for deadlocks were discussed throughout the paper and upon that, I also discussed and supported the design approaches taken for each part. Enjoy!