Week 9: Threads Handling

Assignment 1: The Dining Philosopher Problem

For this problem we had to modify the Dining Philosophers problem which was provided to us. We had to run this for 100000 times without deadlock occurring. The only constraint was that we were not supposed to use sleep for avoiding deadlock.

For this we have used semaphores, we have created a constructor in which we are passing three parameters – the philosopher, the left chopstick semaphore, the right chopstick semaphore. We have also created two objects *'ol'* and *'or'*. We have synchronized our threads on these two objects.

We create a new semaphore for each new philosopher, and pass it to our constructor. Each thread which is spawned foe an individual philosopher goes to the run() function acquires the lock on the *'ol'* object first and then on *'or'*, after acquiring he eats then release locks on both the objects for the next philosopher to eat.

Assignment 2: The Elevator Problem

The problem asked us to build an elevator for an office building, where we have constants on which we had to work with – the number of people, number of floors and a maximum weight constraint. We tried a solution before we came up with our final solution for the problem.

Initial solution:

For this we had a thread pool which represented the number of users waiting to use the elevator. Along with the user threads we had two more threads for the elevator one which handles the upward direction of the elevator and the next thread which handles the downward direction of the elevator. The solution as a whole we thought of building a table in which the number of rows indicate the number of user and have three columns first the floor on which the user is on, the second column indicates the number of floor the user wants to go to, finally the third column we calculate the difference between the floors if the difference is negative we conclude that the user wants to go down, if not then he/she has to go to an upper floor. For example, the below table indicates a snippet of how we tried to implement the solution:

Number of Users – N: 10

Number of Floors – F: 5

Current Floor	**Destination Floor**	**Difference**
4	2	-2 (down)
3	1	-2 (down)
2	4	2 (up)
⋮	⋮	⋮

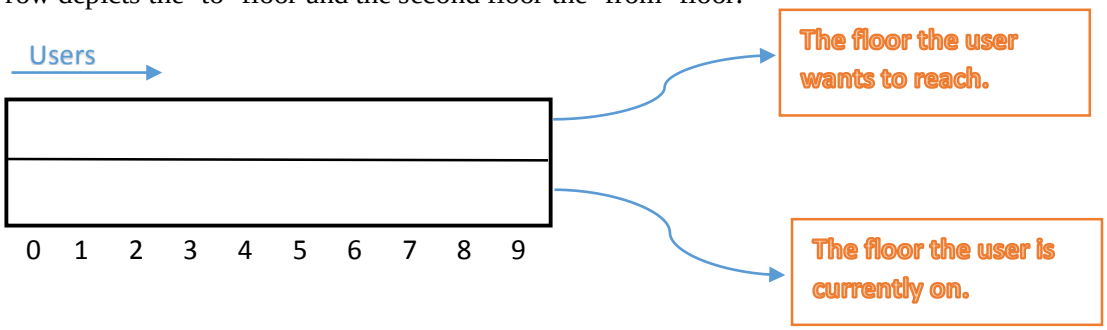
Here if we assume user_1 is no floor number 4 and wants to go to floor 2, we know he/she has to go a couple of floors down, but for the program to understand we gave the indicator that if the difference is negative then down or up. Her both the users, user_1 and user_2 want to go down, but the user_3 has to go up as indicated by the difference.

So the working of user_1 & user_2 would be handled by the thread assigned for when the elevator goes upward and user_3 will be handled by the thread assigned for when the elevator goes down. But when we further tried to conceptualize it, we had a few problems which we thought will not cover the entire problem statement like:

If the two threads handle the working of the same entity there can be an inconsistency in the overall data passed i.e. while checking the weight constraint for the users. There was also a requirement that since we were building it for an office building there will be at least a person at each floor.

Actual Solution:

We took the above solution as the basis and started building our solutions with some improvements to it. Instead of using two separate threads for the upward and downward motion of the elevator we have created two array maps – 2D array – one an up array and the second the down array, where the first row depicts the 'to' floor and the second floor the 'from' floor.



map (2D array) which keeps track of the requests made by each user
Up (2D Array) – Keeps track of the track of the users who want to go upwards.
Down (2D Array) - Keeps track of the track of the users who want to go downwards.
Weight – Stores the weights of individual user.

The up and down arrays are populated using the difference of each value of every row cell. If the difference is less than 0 it is stored in the down array else in the up array.

- We also had to acknowledge the fact that if the final floor request done was 3 then the elevator shouldn't go to the floors above it and should change direction. To get an optimum travelling distance for the elevator we had used a method to find maximum floor till which the elevator has to go. So before starting the elevator in the upward direction, we find the maximum floor. But one problem we faced is that since this problem is based on multithreading, there were times when after creating the arrays, there were some threads which would update the arrays and due to which the consistency of the data was lost. This meant that the max value which was calculated was in actuality no the maximum value and we had values greater than that max value. The solution we came up to solve this issue was to compute the max value just before ending the for loop and using a break condition.
- Additionally, we had to make the person stay on the floor for almost 3 seconds, for this as after the person used to get down the thread used to terminate, we solved this by spawning a new thread for that person to make it wait. So when the person was awake and wanted to go down, there were few threads which were also ready to down, so when the first thread was travelling down the same thread was used to take all the threads down whose wait time was over this was due to such swift processing, this actually saved us from creating threads for each person going down. This in turn optimized our solution in far better way.

- Flow Diagram: