Başak Amasya – 26628

CS307 – HW2 – Report

At the beginning of the run function, a random number is generated for each philosopher to represent the walking time to the table. After thread sleeps for that amount of time, meaning philosopher reaches the table, he puts his plate using the given PutPlate_GUI function. All philosophers must start dining at the same time, that's why barrier implementation comes next. Using the barriers array, philosopher does up (release() function in Java) on every barrier, one for each philosopher. This is like giving permit to every philosopher, since this philosopher has arrived, he is ready to start dining. Then he tries to do down (acquire() function in Java) on the barrier corresponding to his id 5 times, for each up done (each permit given) by the philosophers on his id. If other philosopher(s) hasn't done up on that barrier yet, meaning he hasn't arrived the table, this philosopher cannot do down and gets blocked until the philosopher arrives later does up. This way, all the philosophers will go out of the second for loop at the same time when they managed to do down on their own barriers for the last time and their plates turn black and white, they start dining.

All philosophers' initial state is "THINKING". Dining part consists of thinking, taking the forks, eating, putting the forks back and this should go on forever so that part is implemented inside a while-true loop. Inside this loop, each philosopher's thread sleeps for a random amount of time representing the thinking time. Then take_forks function is called, after its execution finishes, philosopher takes the forks and eats the spaghetti as represented in the GUI, then calls the put_forks function to finish eating.

In take_forks function, first the philosopher tries to down (acquire) the mutex. This mutex protects the critical region. Then state is set to "HUNGRY" and corresponding GUI function is called since the philosopher wants to eat. Then the test function is called to see if left and right forks are available. After the test, philosopher does up on mutex since he is leaving the critical region. If test is successful (meaning up on the semaphore was done in the test function), the philosopher can do a down on its semaphore. But if that's not the case (right and/or left fork was/were not available), it blocks.

For the test function, the philosopher's state should be hungry and right and left philosophers should not be eating, indicating that corresponding left and right forks are available. Then the state is set to "EATING" and up is done on the corresponding semaphore. Taking the forks and eating is visualized in the GUI by calling the function from the run part of the program.

For the put_forks function, the state is set back to "THINKING", StopEating and ForkPut GUI functions are called. Then, we test right and left philosophers, if they are in hungry state, we should unblock them and let them eat by doing up on the semaphore in the test function. Setting the state back to "THINKING", stop eating GUI function, putting the forks down GUI function, testing left and right philosophers are done inside the critical region so they are protected by mutex.

Initial implementation of these functions is taken from lecture slides, modified to Java and GUI functions are added to respective places. This way, we make sure that no philosophers starve, no deadlocks occur and fully utilization (2 philosophers can eat at the same time) is achieved, this can also be observed from the GUI implementation of the program.