--------------------------- 15CS01011_OSLAB5_RESONING ----------------------------

DINING PHILOSPHER'S PROBLEM -

Monitor Solution using Semaphores :

Let the philosophers be numbered philosopher 1, philosopher 2, philosopher 3, philosopher 4, philosopher 5.

1. Philosopher 1 uses fork 1 and fork 5.
2. Philosopher 2 uses fork 1 and fork 2.
3. Philosopher 3 uses fork 2 and fork 3.
4. Philosopher 4 uses fork 3 and fork 4.
5. Philosopher 5 uses fork 4 and fork 5.

Philosophers who have finished eating can release other philosophers who are waiting to eat.

1. Philosopher 1 can release Philosopher 2 or Philosopher 5.
2. Philosopher 2 can release Philosopher 3 or Philosopher 1.
3. Philosopher 3 can release Philosopher 4 or Philosopher 2.
4. Philosopher 4 can release Philosopher 5 or Philosopher 3.
5. Philosopher 5 can release Philosopher 1 or Philosopher 4.

-> 5 threads were created to schedule all of them without a race for forks, one for each philosopher that simulates eating and thinking in a `while(1)`loop.

-> The **pick_up(i)** function is being executed once at a time, for one philosopher, i.e no other philosopher can enter into this function (CS) untill it signals the mutex. This is guaranteed by the mutex semaphore whose value becomes 0 once one philosophers enters into, and only releases after the critical section and signals other philosophers. In the critical section, it changes the state[i] to HUNGRY and tests if he/she can eat using the test function. IT releases the mutex lock so that now other threads(philosophers) can execute this function. If the semaphore s[i] is not signalled in the test function then the philosopher(i) waits for that semaphore s[i] to be signalled infinitely.

-> The **put_down(i)** function also has the mutex wait and signal in the beginning and end of the function respectively. Here, it changes the state[i] back to THINKING and signals other philosophers who are waiting that the forks are put down. And test(LEFT) and test(RIGHT) are called so that, if they are waiting for eating, either of them can proceed to eat as philosopher i has put down the forks. If both are hungry, then preference is given to the LEFT philosopher.

->The **test(i)** function is used to check if the both forks are free and if the philosopher i is hungry. If yes, then it canges the state to EATING and signals the semaphore i , so that the semaphore i which is waiting in the pick_up(i) can now proceed to put the forks down.

->Created Monitor class which contains the functions and variables and the initialization_code.

DEADLOCK:

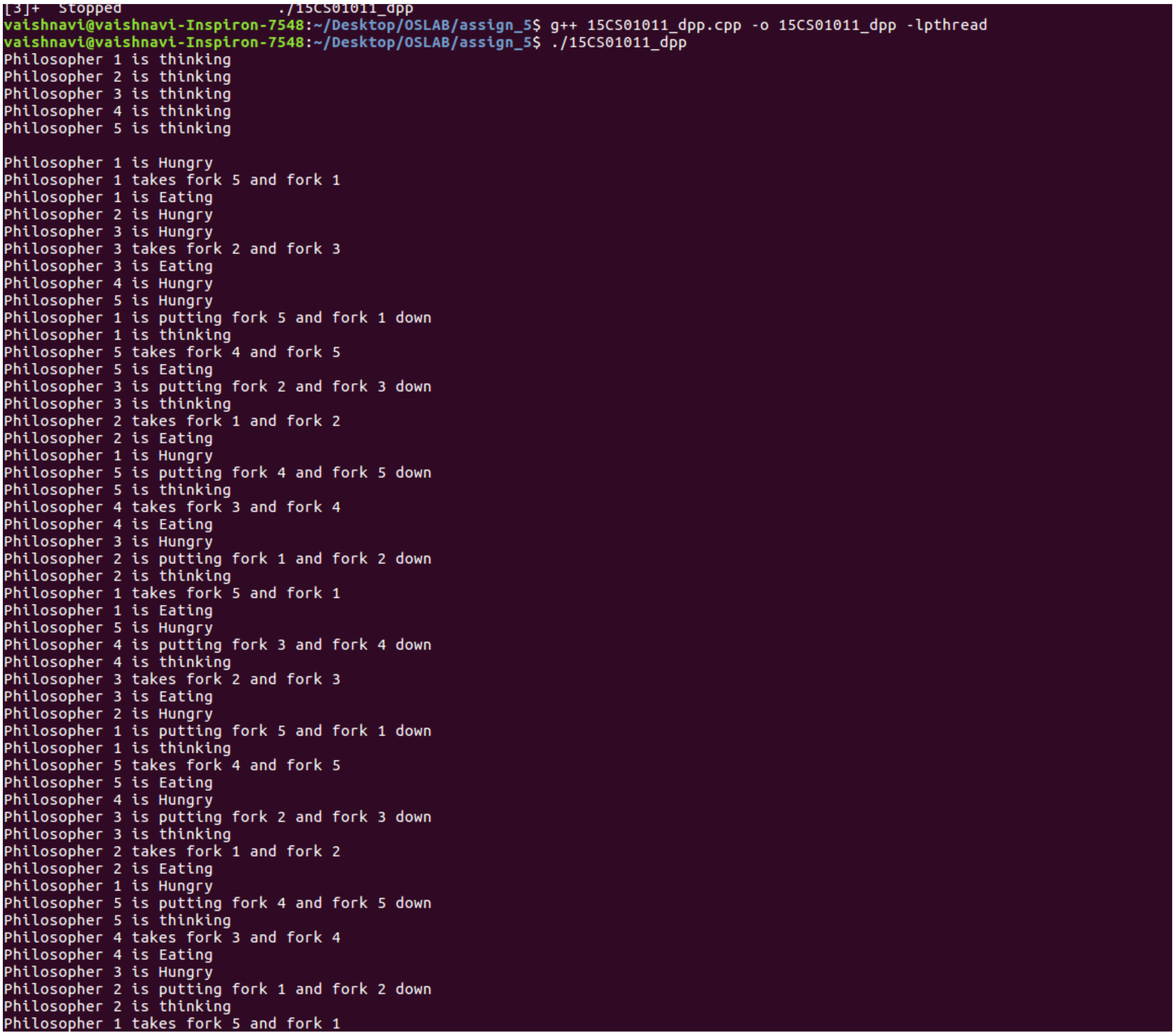
No Deadlock is there. The semaphores ensure that no deadlock happens. And it can be seen in the terminal when the program is run that the program which is infinite loop never stops it goes on executing.

SYNCHRONIZATION:

Synch is maintained by the sem_t s[5]. s[i] is initialized to 0. As, the wait of the s[i] can only be cleared only after the signalling of it in the test(i) function. AS a result, synch is there.

MUTUAL EXCLUSION:

Only one philosopher at a time can enter the pick_up function. Same for the pick_down function. Therefore, Mutual exclusion exists.



```
[3]+  Stopped                 ./15CS01011_dpp
vaishnavi@vaishnavi-Inspiron-7548:~/Desktop/OSLAB/assign_5$ g++ 15CS01011_dpp.cpp -o 15CS01011_dpp -lpthread
vaishnavi@vaishnavi-Inspiron-7548:~/Desktop/OSLAB/assign_5$ ./15CS01011_dpp
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking

Philosopher 1 is Hungry
Philosopher 1 takes fork 5 and fork 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and fork 3
Philosopher 3 is Eating
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 1 is putting fork 5 and fork 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and fork 5
Philosopher 5 is Eating
Philosopher 3 is putting fork 2 and fork 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and fork 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 5 is putting fork 4 and fork 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and fork 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 2 is putting fork 1 and fork 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and fork 1
Philosopher 1 is Eating
Philosopher 5 is Hungry
Philosopher 4 is putting fork 3 and fork 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and fork 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 is putting fork 5 and fork 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and fork 5
Philosopher 5 is Eating
Philosopher 4 is Hungry
Philosopher 3 is putting fork 2 and fork 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and fork 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 5 is putting fork 4 and fork 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and fork 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 2 is putting fork 1 and fork 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and fork 1
```

Compilation: g++ 15CS01011_dpp.cpp -o 15CS01011_dpp -lpthread

Execution: ./dpp