**Name - Manthan Patil.**

**Topic - Concurrency and Parallelism Assignment.**

**Que** :- Implement Dining Philosopher problem using synchronization.

There are 5 philosophers sitting around a circular table eating noodles and discussing philosophy. The problem is that each philosopher needs 2 chopsticks to eat, and there are only 5 chopsticks, one between each 2 philosophers. So we need to make sure that every philosopher eats until the bowl of noodles is empty and no-one starves.

**Solution :-**

* Philosophers(User) have 3 states - THINKING,HUNGRY,EATING.
* The user should follow the sequence of states as - 1.THINKING

2.HUNGRY.

3.EATING.

* User should check :-

1. User should pick up the left chopstick if it is available.

2. User should check if the right chopstick is available and pick it up.

3. If the right chopstick is not available then user should keep left chopstick down on

the table and go in a HUNGRY state until the chopstick is available.

3. If the user has both left and right chopstick he should start eating till the allotted

Time. Once done he should keep both the chopsticks down on the table and go in

THINKING state.

* At a time 2 users can eat and 2 should go in THINKING state, and 1 in HUNGRY state.
* User in the HUNGRY state should go in EATING state when both the chopsticks are available.
* For this purpose threads are created to maintain philosopher state.
* Locks should be applied to chopsticks and released when the user goes in EATING state.

**Algorithm for process:**

1. Start.

2. Declare and initialize the thread variables (philosophers) as required.

3. Declare and initialize the mutex variables (chopsticks) as required.

4. Create the threads representing philosophers.

5. Wait until the threads finish execution.

6. Stop.

**Algorithm for thread (philosopher i) function:**

1. Start.

2. Philosopher (i) is thinking.

3. Philosopher (i) is hungry.

4. Lock the left fork spoon.

5. Lock the right fork spoon.

6. Philosopher (i) is eating.

7. Philosopher (i) finished eating.

8. Release the left fork spoon.

9. Release the right fork spoon.

10. Philosopher i Finished eating.

11. Stop.

* Creating thread and locks :-

pthread\_cond\_phcond[5];

pthread\_mutex\_condLock;

* Checking the state of user(CHECK IF USER IN HUNGRY STATE)

void test(int i)

{

if (state[(i + 1) % 5] != HUNGRY && state[(i + 4) % 5] != EATING && state[i] ==

HUNGRY)

{

state[i] = HUNGRY;

cout << "Philosopher “<<i<<"is Hungry"<<endl

pthread\_cond\_signal(&phcond[i]);

}

}

* To take the fork when philosopher is eating(GOES IN EATING STATE) :-

void take\_fork(int i)

{

pthread\_mutex\_lock(&condLock);

state[i] = HUNGRY;

test(i);

if (state[i] != EATING)

{

pthread\_cond\_wait(&phcond[i], &condLock);

}

cout << "Philosopher “<<i<<"is Eating"<<endl;

pthread\_mutex\_unlock(&condLock);

}

* To put the fork when philosopher is done eating (GOES TO THINKING STATE):-

void put\_fork(int i)

{

pthread\_mutex\_lock(&condLock);

state[i] = THINKING;

test(RIGHT);

test(LEFT);

cout << "Philosopher “<<i<<"is thinking"<<endl

pthread\_mutex\_unlock(&condLock);

}

* Assigning task to thread to completed the eating

monitor()

{

for (int i = 0; i <n; i++)

{

state[i] = THINKING;

}

for (int i = 0; i < n; i++)

{

pthread\_cond\_init(&phcond[i]);

}

pthread\_mutex\_init(&condLock);

}

* Destroy thread when completed eating and goto to end.

~ monitor()

{

for (int i = 0; i <n; i++)

{

pthread\_cond\_destroy(&phcond[i]);

}

pthread\_mutex\_destroy(&condLock);

cout << "Eating completed"<<endl;

}