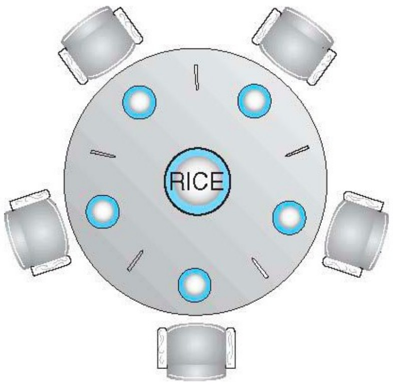
PRABHLEEN KAUR

CONCURRENCY AND PARALLELISM ASSIGNMENT

AIM: Need to implement a dining philosopher problem using synchronization like mutex or semaphores.

DINING PHILOSOPHER PROBLEM:

- It states that there are 5 philosophers sharing a circular table and they eat and think alternatively.
- There is a bowl of rice for each of the philosophers and 5 chopsticks.
- A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.
- It is a classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes



SOLUTION:

- From the problem statement, it is clear that a philosopher can think for an indefinite amount of time, but when a philosopher starts eating, he/ she has to stop at some point of time
- The philosopher is in an endless cycle of thinking and eating
- When a philosopher wants to eat the rice, wait for the chopstick at his left and picks up that chopstick. Then waits for the right chopstick to be available, and then picks it too.
- After eating, puts both the chopsticks down
- Correctness properties it needs to satisfy are :
  - MutualExclusionPrinciple– No two Philosophers can have the two forks simultaneously.
  - FreefromDeadlock–Each philosopher can get the chance to eat in a certain finite time.

   - Free from Starvation –When few Philosophers are waiting then one gets a chance to eat in a while.

- No strict Alternation.
- Proper utilization of time.
- A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

```
semaphore chopstick [5];
```

- Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.
- The idea is to find a solution so that none of the philosophers would starve, i.e. never have the chance to acquire the forks necessary for him to eat.
- The structure of a random philosopher i is given as follows −

```
do {
    wait( chopstick[i] );
    wait( chopstick[ (i+1) % 5] );
    . .
    EATING THE RICE
    signal( chopstick[i] );
    signal( chopstick[ (i+1) % 5] );
    . .
    THINKING
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

CODE:

```
#include <iostream>
#include <mutex>
#include <thread>

using namespace std;

int main()
{
    const int no_of_philosophers = 5;

    struct Chopstics
    {
    public:
        Chopstics(){;}
        mutex mu;
    };

    auto eat = [](Chopstics &left_chopstics, Chopstics& right_chopstics, int philosopher_number) {

        unique_lock<mutex> llock(left_chopstics.mu);
        unique_lock<mutex> rlock(right_chopstics.mu);

        cout << "Philosopher " << philosopher_number << " is eating" << endl;

        chrono::milliseconds timeout(1500);
        this_thread::sleep_for(timeout);

        cout << "Philosopher " << philosopher_number << " has finished eating" << endl;
    };

    //create chopstics
    Chopstics chp[no_of_philosophers];

    //create philosophers
    thread philosopher[no_of_philosophers];

    //Philosophers Start thinking
    cout << "Philosopher " << (0+1) << " is thinking.." << endl;
    philosopher[0] = thread(eat, ref(chp[0]), ref(chp[no_of_philosophers-1]), (0+1));

    for(int i = 1; i < no_of_philosophers; ++i) {
        cout << "Philosopher " << (i+1) << " is thinking.." << endl;
        philosopher[i] = thread(eat, ref(chp[i]), ref(chp[i-1]), (i+1));
    }

    for(auto &ph: philosopher) {
        ph.join();
    }

    return 0;
}
```

OUTPUT:

```
Philosopher 1 is thinking..
Philosopher 2 is thinking..
Philosopher 1 is eating
Philosopher 3 is thinking..
Philosopher 4 is thinking..
Philosopher 5 is thinking..
Philosopher 4 is eating
Philosopher 1 has finished eating
Philosopher 2 is eating
Philosopher 4 has finished eating
Philosopher 5 is eating
Philosopher 2 has finished eating
Philosopher 3 is eating
Philosopher 5 has finished eating
Philosopher 3 has finished eating
```