Operating Systems

Practical 6 :- Write a program to implement the Dining Philosopher's Problem.

Dining Philosophers Problem in OS is a classical synchronization problem in the operating system. With the presence of more than one process and limited resources in the system the synchronization problem arises. If one resource is shared between more than one process at the same time then it can lead to data inconsistency.
Consider two processes P1 and P2 executing simultaneously, while trying to access the same resource R1, this raises the question of who will get the resource and when. This problem is solved using process synchronization.
Each philosopher does only two things:

- **Think**: The philosopher is thinking (not using any resources).
- **Eat**: The philosopher eats using two resources, a fork in each hand.

There are five forks on the table, one between each pair of philosophers. The challenge arises because each philosopher needs two forks to eat: one from their left and one from their right. The problem arises in coordinating these philosophers so that:

- No philosopher starves (gets stuck thinking forever).
- There are no deadlocks (where all philosophers are holding one fork and waiting for the other).
- No philosopher gets stuck indefinitely, waiting for the fork held by another philosopher.

The Solution of the Dining Philosophers Problem

The solution to the process synchronization problem is Semaphores, A semaphore is an integer used in solving critical sections.

The critical section is a segment of the program that allows you to access the shared variables or resources. In a critical section, an atomic action (independently running process) is needed, which means that only a single process can run in that section at a time.

Semaphore has 2 atomic operations: wait() and signal(). If the value of its input S is positive, the wait() operation decrements, it is used to acquire resources while entry. No operation is done if S is negative or zero. The value of the

signal() operation's parameter S is increased, it is used to release the resource once the critical section is executed at exit.

CODE :- #include <iostream>

#include <thread>

#include <mutex>

#define NUM_PHILOSOPHERS 5

std::mutex forks[NUM_PHILOSOPHERS];

void philosopher(int id) {

std::lock(forks[id], forks[(id + 1) % NUM_PHILOSOPHERS]);

std::lock_guard<std::mutex> lock1(forks[id], std::adopt_lock);

std::lock_guard<std::mutex> lock2(forks[(id + 1) % NUM_PHILOSOPHERS], std::adopt_lock);

std::cout << "Philosopher " << id << " is eating.\n";

}int main() {

std::thread threads[NUM_PHILOSOPHERS];

for (int i = 0; i < NUM_PHILOSOPHERS; i++) {

threads[i] = std::thread(philosopher, i);

} for (int i = 0; i < NUM_PHILOSOPHERS; i++) {

threads[i].join();

} return 0;

}

Output :-

Philosopher 0 is eating.

Philosopher 4 is eating.

Philosopher 2 is eating.

Philosopher 3 is eating.

Philosopher 1 is eating.

Course Code: CS3CO36
Course : Operating System

Enrollment: EN23CS301105
Name: Akshat Sharma