Batch: D-2	**Roll No.:** 16010122151
Experiment No. 06	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: Implementation of Dining Philosophers problem using mutexes and semaphores.

AIM: Implementation of Process synchronization algorithms using mutexes and semaphore – Dining Philosopher problem

---

Expected Outcome of Experiment:

CO 2. To understand the concept of process, thread and resource management.
CO 3. To understand the concepts of process synchronization and deadlock.

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate "Operating Systems", McGraw Hill Third Edition.**
3. **Sumitabha Das " UNIX Concepts & Applications", McGraw Hill Second Edition.**

Pre Lab/ Prior Concepts:

Knowledge of Concurrency, Mutual Exclusion, Synchronization, Deadlock, Starvation, threads.

K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Description of the chosen process synchronization algorithm:

Mutex:-

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <chrono>
#include <condition_variable>
using namespace std;

class Fork {
public:
    Fork() : inUse(false) {}

    void take() {
        std::unique_lock<std::mutex> lock(mutex);
        condition.wait(lock, [this]() { return !inUse; });
        inUse = true;
    }

    void put() {
        {
            std::lock_guard<std::mutex> lock(mutex);
            inUse = false;
        }
        condition.notify_one();
    }

private:
    bool inUse;
    std::mutex mutex;
    std::condition_variable condition;
};

class Philosopher {
public:
    flhilosopher(int id, Fork &left, Fork &right, std::mutex &outputMutex,
int turns)
        : id(id), leftFork(left), rightFork(right),
outputMutex(outputMutex), turns(turns), mealsEaten(0) {}
```

```
    void dine() {
        while (mealsEaten < turns) {
            think();
            eat();
        }
    }

private:
    int id;
    Fork fileftFork;
    Fork firightFork;
    std::mutex fioutputMutex;
    int turns;
    int mealsEaten;

    void think() {
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        {
            std::lock_guard<std::mutex> lock(outputMutex);
            std::cout << "flhilosopher " << id << " is thinking." <<
std::endl;
        }
    }

    void eat() {
        leftFork.take();
        {
            std::lock_guard<std::mutex>  lock(outputMutex);
            std::cout << "flhilosopher " << id << " picked up left fork." <<
std::endl;
        }

        rightFork.take();
        {
            std::lock_guard<std::mutex>  lock(outputMutex);
            std::cout << "flhilosopher " << id << " picked up right fork."
<< std::endl;
        }

        {
            std::lock_guard<std::mutex> lock(outputMutex);
            std::cout << "flhilosopher " << id << " is eating." <<
std::endl;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
```

```
        leftFork.put();
        {
            std::lock_guard<std::mutex> lock(outputMutex);
            std::cout << "flhilosopher " << id << " put down left fork." <<
std::endl;
        }

        rightFork.put();
        {
            std::lock_guard<std::mutex> lock(outputMutex);
            std::cout << "flhilosopher " << id << " put down right fork." <<
std::endl;
        }

        mealsEaten++; // Increment the number ofi meals eaten
        {
            std::lock_guard<std::mutex> lock(outputMutex);
            std::cout << "flhilosopher " << id << " has eaten " <<
mealsEaten << " times." << std::endl;
        }
    }
};

int main() {
    int flHILOSOflHERS ;
    int TURNS ; // Number ofi turns each philosopher will eat
    std::vector<Fork> forks(flHILOSOflHERS);
    std::vector<std::thread> philosophers;
    std::mutex outputMutex;

    cout<<"Enter Number of flhilosophers:-" ;
    cin>>flHILOSOflHERS;

    cout<<"Enter Number of Turns for flhilosophers:-" ;
    cin>>TURNS;

    for (int i = 0; i < flHILOSOflHERS; ++i) {
        philosophers.emplace_back(fiPhilosopher::dine, Philosopher(i,
forks[i], forks[(i + 1) % flHILOSOflHERS], outputMutex, TURNS));
    }

    for (auto fiphilosopher : philosophers) {
        philosopher.join();
    }

    return 0;
```

```
}
```

Output:-

```
hyder@HyderPresswala MINGW64 ~/Downloads/New folder
$ ./hyder.exe
Enter Number of Philosophers:-5
Enter Number of Turns for Philosophers:-3
Philosopher 4 is thinking.
Philosopher 4 picked up left fork.
Philosopher 4 picked up right fork.
Philosopher 4 is eating.
Philosopher 0 is thinking.
Philosopher 3 is thinking.
Philosopher 3 picked up left fork.
Philosopher 1 is thinking.
Philosopher 1 picked up left fork.
Philosopher 1 picked up right fork.
Philosopher 1 is eating.
Philosopher 2 is thinking.
Philosopher 4 put down left fork.
Philosopher 1 put down left fork.
Philosopher 1 put down right fork.
Philosopher 1 has eaten 1 times.
Philosopher 0 picked up left fork.
Philosopher 0 picked up right fork.
Philosopher 0 is eating.
Philosopher 4 put down right fork.
Philosopher 4 has eaten 1 times.
Philosopher 2 picked up left fork.
Philosopher 3 picked up right fork.
Philosopher 3 is eating.
Philosopher 4 is thinking.
Philosopher 0 put down left fork.
Philosopher 1 is thinking.
Philosopher 0 put down right fork.
Philosopher 1 picked up left fork.
Philosopher 0 has eaten 1 times.
Philosopher 3 put down left fork.
Philosopher 2 picked up right fork.
```

```
Philosopher 4 picked up left fork.
Philosopher 3 put down right fork.
Philosopher 2 is eating.
Philosopher 4 picked up right fork.
Philosopher 3 has eaten 1 times.
Philosopher 4 is eating.
Philosopher 0 is thinking.
Philosopher 1 picked up right fork.
Philosopher 2 put down left fork.
Philosopher 1 is eating.
Philosopher 2 put down right fork.
Philosopher 2 has eaten 1 times.
Philosopher 4 put down left fork.
Philosopher 3 is thinking.
Philosopher 0 picked up left fork.
Philosopher 4 put down right fork.
Philosopher 3 picked up left fork.
Philosopher 4 has eaten 2 times.
Philosopher 3 picked up right fork.
Philosopher 2 is thinking.
Philosopher 0 picked up right fork.
Philosopher 0 is eating.
Philosopher 1 put down left fork.
Philosopher 2 picked up left fork.
Philosopher 1 put down right fork.
Philosopher 1 has eaten 2 times.
Philosopher 2 picked up right fork.
Philosopher 3 put down left fork.
Philosopher 4 is thinking.
Philosopher 2 is eating.
Philosopher 3 put down right fork.
Philosopher 4 picked up left fork.
Philosopher 3 has eaten 2 times.
Philosopher 2 put down left fork.
Philosopher 2 put down right fork.
Philosopher 2 has eaten 2 times.
Philosopher 0 put down left fork.
```

```
Philosopher 0 put down right fork.
Philosopher 0 has eaten 2 times.
Philosopher 3 is thinking.
Philosopher 3 picked up left fork.
Philosopher 4 picked up right fork.
Philosopher 4 is eating.
Philosopher 1 is thinking.
Philosopher 1 picked up left fork.
Philosopher 1 picked up right fork.
Philosopher 1 is eating.
Philosopher 1 put down left fork.
Philosopher 1 put down right fork.
Philosopher 1 has eaten 3 times.
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 2 picked up left fork.
Philosopher 4 put down left fork.
Philosopher 4 put down right fork.
Philosopher 4 has eaten 3 times.
Philosopher 0 picked up left fork.
Philosopher 3 picked up right fork.
Philosopher 0 picked up right fork.
Philosopher 3 is eating.
Philosopher 0 is eating.
Philosopher 2 picked up right fork.
Philosopher 2 is eating.
Philosopher 3 put down left fork.
Philosopher 3 put down right fork.
Philosopher 3 has eaten 3 times.
Philosopher 0 put down left fork.
Philosopher 0 put down right fork.
Philosopher 0 has eaten 3 times.
Philosopher 2 put down left fork.
Philosopher 2 put down right fork.
Philosopher 2 has eaten 3 times.
```

Semaphores:-

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

using namespace std;

const int NUM_P = 5;       // Number of philosophers
const int NUM_FORKS = 5;  // Number of forks

// Array representing the forks (initialized to 1, meaning each fork is
available)
vector<int> forks(NUM_FORKS, 3);

// Mutex array to synchronize access to forks
mutex forkMutex[NUM_FORKS];

// Mutex for synchronizing output to the console
mutex printMutex;

// Function for the philosopher to pick up forks and eat
void mutexP(int P_id) {
    int leftFork = P_id;
    int rightFork = (P_id + 1) % NUM_P;

    while (true) {
        // Simulate thinking
        this_thread::sleep_for(chrono::milliseconds(200));

        // Check if both forks are available (both forks > 0)
        if (forks[leftFork] > 0 && forks[rightFork] > 0) {
            // Lock the printMutex to ensure the output is not mixed up
            {
                lock_guard<mutex> lock(printMutex);
                cout << "Philosopher " << P_id << " is eating now!" <<
endl;
            }
```

```
            // Lock the forks (decrement availability)
            forks[leftFork]--;
            forks[rightFork]--;

            // Simulate eating
            this_thread::sleep_for(chrono::milliseconds(500));

            // Release the forks (increment availability)
            {
                lock_guard<mutex> lock(printMutex);
                cout << "Philosopher " << P_id << " is releasing the fork
now." << endl;
            }
            forks[leftFork]++;
            forks[rightFork]++;
        }
        else {
            // If forks are not available, philosopher is waiting
            {
                lock_guard<mutex> lock(printMutex);
                cout << "Philosopher " << P_id << " is waiting for forks!"
<< endl;
            }
        }
    }
}

int main() {
    vector<thread> philosophers;

    // Create and start threads for each philosopher
    for (int i = 0; i < NUM_P; i++) {
        philosophers.push_back(thread(mutexP, i));
    }

    // Join all threads to the main thread (this will run indefinitely)
    for (auto& th : philosophers) {
        th.join();
    }

    return 0;
}
```

Output:-

```
hyder@HyderPresswala MINGW64 ~/Downloads/Codewithharry
$ ./hyder.exe
Philosopher 4 is eating now!
Philosopher 3 is eating now!
Philosopher 1 is eating now!
Philosopher 0 is eating now!
Philosopher 2 is eating now!
Philosopher 4 is releasing the fork now.
Philosopher 1 is releasing the fork now.
Philosopher 3 is releasing the fork now.
Philosopher 2 is releasing the fork now.
Philosopher 0 is releasing the fork now.
Philosopher 4 is eating now!
Philosopher 2 is eating now!
Philosopher 3 is eating now!
Philosopher 1 is eating now!
Philosopher 0 is eating now!
```

Conclusion:- : In this experiment we successfully implemented a deadlock free solution for the dining philosophers problem using threads.

Post Lab Descriptive Questions

1. Differentiate between a monitor, semaphore and a binary semaphore?

1) Monitor:- A Monitor type high-level synchronization construct. It is an abstract data type. The Monitor type contains shared variables and the set of procedures that operate on the shared variable. When any process wishes to access the shared variables in the monitor, it needs to access it through the procedures. These processes line up in a queue and are only provided access when the previous process release the shared variables. Only one process can be active in a monitor at a time. Monitor has condition variables.

2) Semaphore:- A Semaphore is a lower-level object. A semaphore is a non-negative integer variable. The value of Semaphore indicates the number of shared resources available in the system. The value of semaphore can be

modified only by two functions, namely wait() and signal() operations (apart from the initialization). When any process accesses the shared resources, it performs the wait() operation on the semaphore and when the process releases the shared resources, it performs the signal() operation on the semaphore. Semaphore does not have condition variables. When a process is modifying the value of the semaphore, no other process can simultaneously modify the value of the semaphore.

3) **Binary Semaphore:-** The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

2. Identify the scenarios in the dining-philosophers problem that leads to the deadlock situations?

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again. The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.

3. Which of the following can be used to avoid deadlock in the Dining Philosophers Problem?
   a. Using a semaphore initialized to the number of philosophers.
   b. Using a semaphore initialized to one less than the number of philosophers.
   c. Using a mutex for each philosopher.
   d. Using a monitor for ach fork

4. Which synchronization construct encapsulates shared variables, synchronization primitives, and operations on shared variables?
    a. Semaphore
    b. Binary Semaphore
    c. Monitor
    d. Mutex

Date: 24-10-2024 **Signature of faculty in-charge**