Batch: A2	**Roll No.: 16010121045**
Experiment No. 06	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: Implementation of dining philosopher problem using threads.

AIM: Implementation of Process synchronization algorithms using threads – 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. **William Stallings, "Operating System Internal & Design Principles", Pearson.**
4. **Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.**

Pre Lab/ Prior Concepts:

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

Description of the chosen process synchronization algorithm:

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.

Solution for the problem:

We use a semaphore to represent a chopstick and this truly acts as a solution of the Dining Philosophers Problem. Wait and Signal operations will be used for the solution of the Dining Philosophers Problem, for picking a chopstick wait operation can be executed while for releasing a chopstick signal semaphore can be executed.

The drawback of the above solution of the dining philosopher problem:

The above solution makes sure that no two neighbouring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.
Some of the ways to avoid deadlock are as follows –

- There should be at most four philosophers on the table
- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Implementation details:

philosopher.java

```
public class philosopher extends Thread {
    public int number;
    public Fork leftFork;
    public Fork rightFork;

    philosopher(int num, Fork left, Fork right) {
        number = num;
        leftFork = left;
        rightFork = right;
    }

    public void run() {
        think();
        eat();
    }

    void think() {
```

```
        System.out.println("Philosopher " + number + " Thinking");
        Log.Delay(1000);
    }

    public void take_fork() {
        leftFork.getFork();
        System.out.println("Philosopher " + number + ": Take left fork");
        rightFork.getFork();
        System.out.println("Philosopher " + number + ": Take right
fork");
    }

    public void put_fork() {
        leftFork.putFork();
        System.out.println("Philosopher " + number + ": Put left fork");
        rightFork.putFork();
        System.out.println("Philosopher " + number + ": Put right fork");
    }

    public void eat() {
        if (!leftFork.isUsed()) {
            if (!rightFork.isUsed()) {
                take_fork();
                System.out.println("Philosopher " + number + " Eating");
                Log.Delay(1000);
                put_fork();
            }
        }
    }
}

class Log {
    public static void msg(String msg) {
        System.out.println(msg);
    }

    public static void Delay(int ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException ex) {
        }
    }
}
```

Fork.java

```
import java.util.concurrent.Semaphore;
```

```
public class Fork {
    private boolean inuse;
    Semaphore sem;

    public Fork() {
        inuse = false;
        sem = new Semaphore(1);
    }

    public void getFork() {
        try {
            while (inuse) {
                try {
                    sem.acquire();
                } catch (InterruptedException e) {
                }
            }
            inuse = true;
        } catch (Exception e) {
        }
    }

    public void putFork() {
        try {
            inuse = false;
            sem.release();
        } catch (Exception e) {
        }
    }

    public boolean isUsed() {
        return inuse;
    }
}
```

Main.java

```
import java.util.Scanner;

public class Main {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        int philosophersNumber;
        System.out.print("Enter the number of philosophers: ");
        philosophersNumber = sc.nextInt();
```

```
        final philosopher philosophers[] = new
philosopher[philosophersNumber];
        Fork forks[] = new Fork[philosophersNumber];
        for (int i = 0; i < philosophersNumber; i++) {
            forks[i] = new Fork();
        }
        for (int i = 0; i < philosophersNumber; i++) {
            Fork leftfork = forks[i];
            Fork rightfork = forks[(i + 1) % philosophersNumber];
            if (i == philosophers.length - 1) {
                // The last philosopher picks up the right fork first
                philosophers[i] = new philosopher(i, rightfork,
leftfork);
            } else {
                philosophers[i] = new philosopher(i, leftfork,
rightfork);
            }
            Thread t = new Thread(philosophers[i], "Philosopher " + (i +
1));
            t.start();
            sc.close();
        }
    }
}
```

Output:

```
PS D:\SEM 5\OSSS\Lab\Expt 6>  & 'c:\Users\dhair\.vscode\extensions\vscjava.vscode-java-debug-0.36.0\scripts\launcher.bat' 'C:\P
gram Files\Java\jdk-16.0.2\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-Dfile.encoding=UTF-8' '-cp' 'C:\Users\dhai
AppData\Roaming\Code\User\workspaceStorage\55971eb8dc44e1f499ed1c5e483592b6\redhat.java\jdt_ws\Expt 6_4b491ba2\bin' 'Main'
Enter the number of philosophers: 5
Philosopher 0 Thinking
Philosopher 2 Thinking
Philosopher 1 Thinking
Philosopher 4 Thinking
Philosopher 3 Thinking
Philosopher 3: Take left fork
Philosopher 0: Take left fork
Philosopher 1: Take left fork
Philosopher 3: Take right fork
Philosopher 3 Eating
Philosopher 1: Take right fork
Philosopher 1 Eating
Philosopher 3: Put left fork
Philosopher 0: Take right fork
Philosopher 1: Put left fork
Philosopher 0 Eating
Philosopher 3: Put right fork
Philosopher 1: Put right fork
Philosopher 0: Put left fork
Philosopher 0: Put right fork
PS D:\SEM 5\OSSS\Lab\Expt 6>
```

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?

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.

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.

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. Define clearly the dining-philosophers problem?

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

A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus none ever stop waiting. It is a logical error that can occur when programming with threads. A deadlock happens when two threads wait on each other.

Such deadlock would occur if:

a. all philosophers would pick up their left forks first or
b. b. all philosophers pick up their right forks first

Thus, in dining philosopher's problem, a deadlock occurs when all the philosophers are waiting for a resource held by some other philosopher

Date: _____ **Signature of faculty in-charge**