Distributed Computing

Experiment – 6

Program to demonstrate Mutual Exclusion Algorithm	
Learning Objective:	To explore mutual exclusion algorithms and deadlock handling in the distributed system.
Learning Outcome:	Ability to demonstrate Mutual Exclusion Algorithm.
Course Outcome:	**CSL801.4**
Program Outcome:	(**PO**1) **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. (**PO**2) **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. **(PO**3) **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations. **(PO6) The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
Bloom's Taxonomy Level:	Analysis, Create
Theory:	**Mutual Exclusion in Distributed Computing:** In distributed systems, mutual exclusion algorithms and deadlock handling are essential for ensuring proper coordination and resource management among multiple concurrent processes running on different nodes. Let's discuss each aspect: **Mutual Exclusion Algorithms:** 1. **Centralized Algorithm**: One node (usually a coordinator) controls access to a shared resource. Processes request permission from this coordinator before accessing the resource. Advantages include simplicity and

straightforward implementation. However, it suffers from a single point of failure and potential bottleneck issues at the coordinator.

2. **Distributed Algorithm**:

   - **Token-Based Algorithm**: A token circulates among nodes, granting the holder exclusive access to the resource. This ensures fairness and decentralization but can suffer from delays if the token holder is far away.
   - **Election Algorithm**: Nodes elect a leader, which grants access to the resource. If the leader fails, a new one is elected. This provides fault tolerance but adds complexity due to leader election mechanisms.

Advantages:

- Ensures that only one process accesses a critical section at a time.
- Facilitates concurrency by allowing multiple processes to execute in parallel while maintaining safety.

Disadvantages:

- Overhead in communication and coordination among distributed nodes.
- Complexity in handling failures and ensuring fairness.
- Potential for bottlenecks or delays, especially in token-based algorithms.

Deadlock Handling:

1. **Detection and Recovery**: Periodic detection of deadlocks followed by recovery mechanisms like aborting processes or rolling back transactions.
2. **Prevention**: Techniques like resource allocation ordering to ensure deadlock never occurs.
3. **Avoidance**: Using algorithms like Banker's Algorithm, where the system ensures that resource allocations do not lead to a circular wait condition.
4. **Timeouts**: Processes waiting for resources are aborted after a certain timeout period to prevent indefinite blocking.

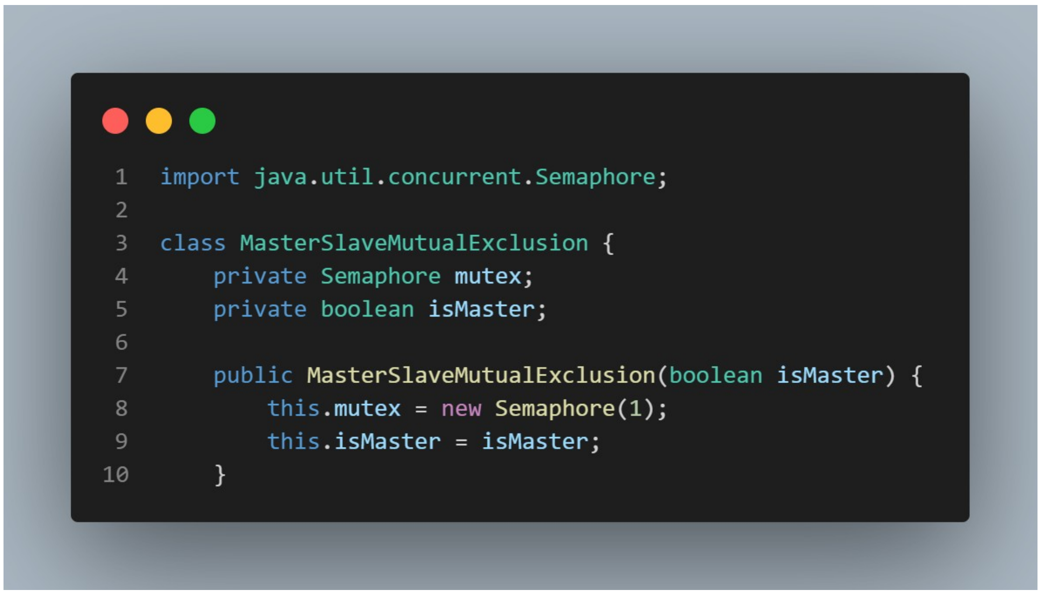
Advantages:

- Prevents system-wide halts due to deadlock situations.
- Enhances system reliability and availability by handling deadlock scenarios gracefully.

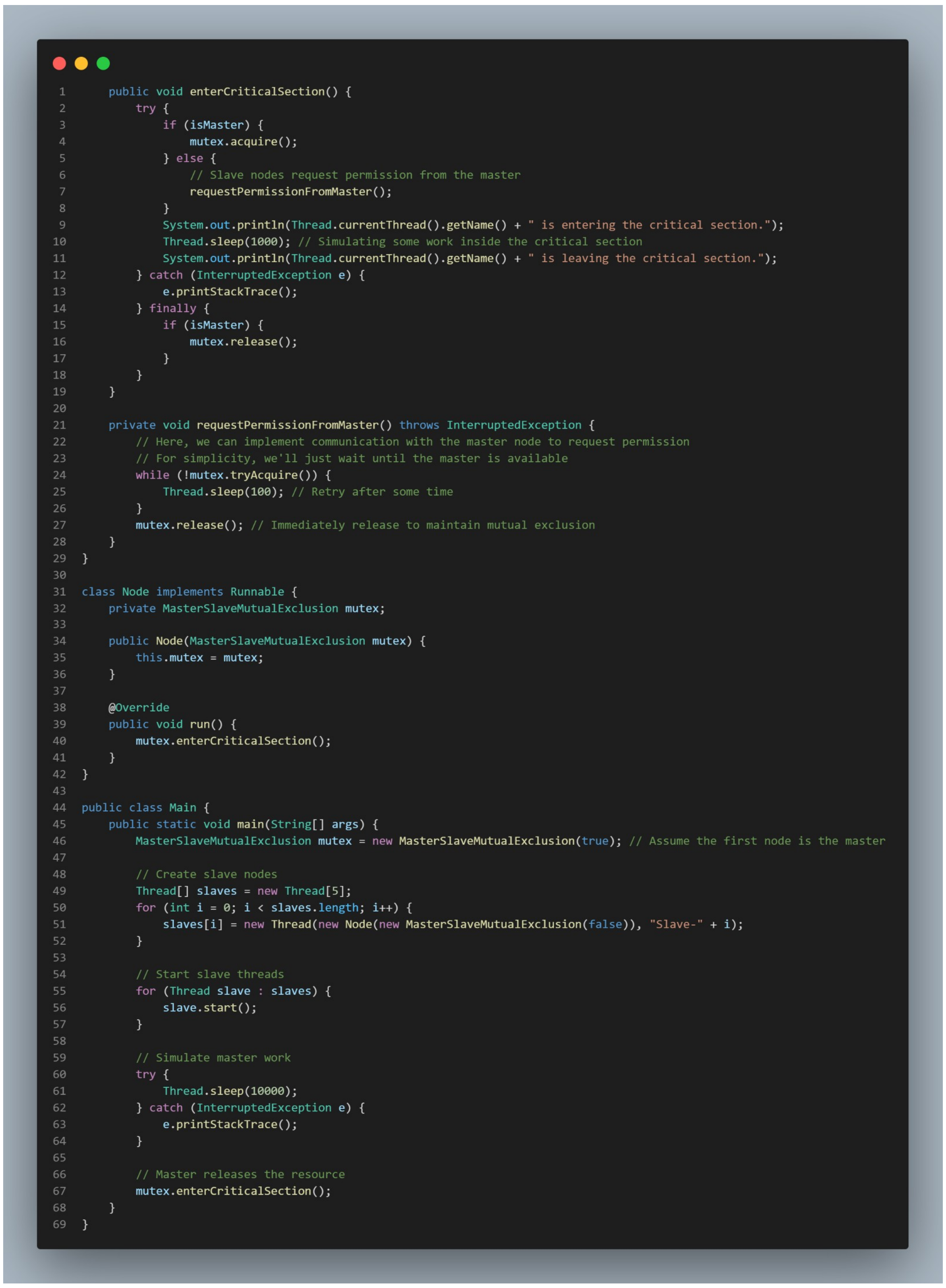
	Disadvantages: • Increased complexity in system design and implementation. • Potential performance overhead due to deadlock detection and recovery mechanisms. • Possibility of false positives or negatives in deadlock detection algorithms. **Application:** One application of mutual exclusion algorithms and deadlock handling in distributed systems is in database management systems (DBMS). In a distributed database environment, multiple users may concurrently access and modify shared data. Mutual exclusion ensures that only one transaction can update a specific data item at a time, preventing data inconsistencies. Deadlock handling mechanisms are crucial to ensure continuous availability of database services, even in the presence of concurrent transactions that might deadlock.

Outcome :	**Programming Code -**
	
	`import java.util.concurrent.Semaphore;`
	
	`class MasterSlaveMutualExclusion {`
	`    private Semaphore mutex;`
	`    private boolean isMaster;`
	
	`    public MasterSlaveMutualExclusion(boolean isMaster) {`
	`        this.mutex = new Semaphore(1);`
	`        this.isMaster = isMaster;`
	`    }`
	`    public void enterCriticalSection() {`
	`        try {`

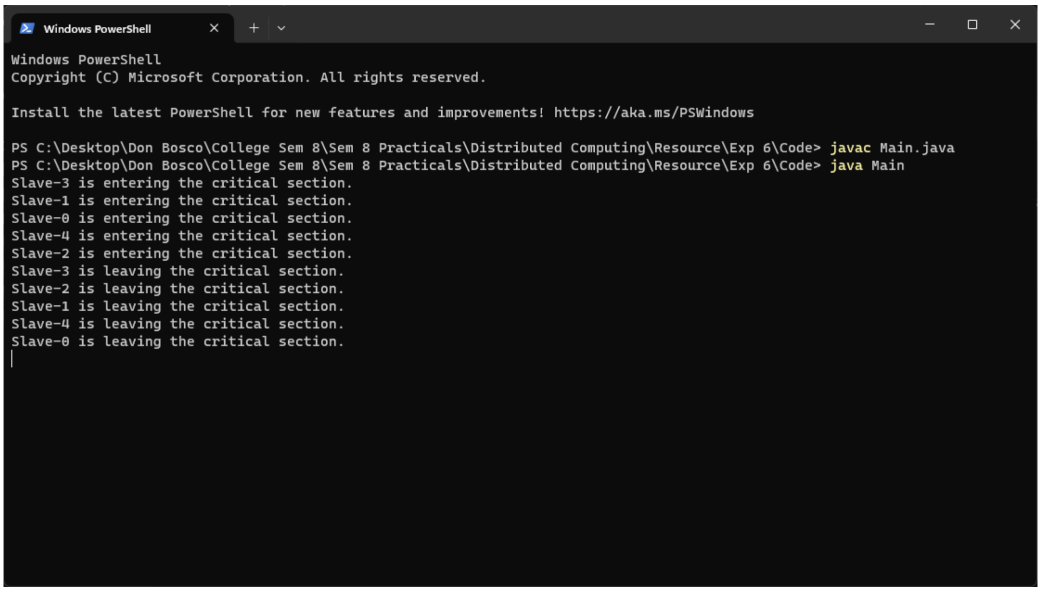
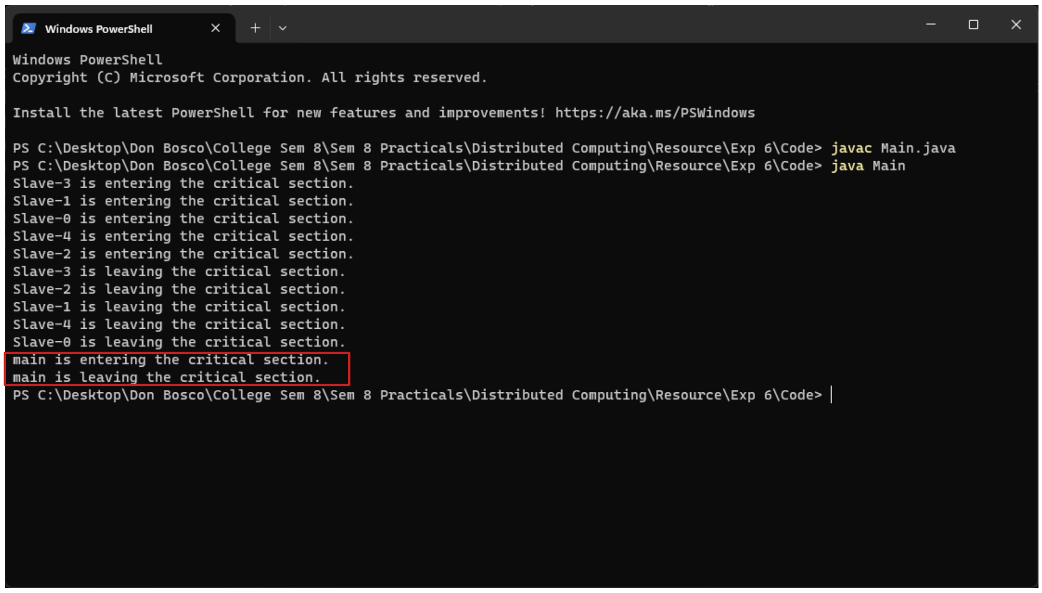
```
            if (isMaster) {
                mutex.acquire();
            } else {
                // Slave nodes request permission from the master
                requestPermissionFromMaster();
            }
            System.out.println(Thread.currentThread().getName() + " is
entering the critical section.");
            Thread.sleep(1000); // Simulating some work inside the critical
section
            System.out.println(Thread.currentThread().getName() + " is
leaving the critical section.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            if (isMaster) {
                mutex.release();
            }
        }
    }
    private void requestPermissionFromMaster() throws
InterruptedException {
        // Here, we can implement communication with the master node to
request permission
        // For simplicity, we'll just wait until the master is available
        while (!mutex.tryAcquire()) {
            Thread.sleep(100); // Retry after some time
        }

        mutex.release(); // Immediately release to maintain mutual exclusion

    }

}

class Node implements Runnable {

    private MasterSlaveMutualExclusion mutex;



    public Node(MasterSlaveMutualExclusion mutex) {

        this.mutex = mutex;

    }

    @Override

    public void run() {

        mutex.enterCriticalSection();

    }

}

public class Main {

    public static void main(String[] args) {

        MasterSlaveMutualExclusion mutex = new MasterSlaveMutualExclusion(true); // Assume the first node is the master



        // Create slave nodes

        Thread[] slaves = new Thread[5];

        for (int i = 0; i < slaves.length; i++) {

            slaves[i] = new Thread(new Node(new MasterSlaveMutualExclusion(false)), "Slave-" + i);
        }

        // Start slave threads

        for (Thread slave : slaves) {

            slave.start();

        }

        // Simulate master work

        try {

            Thread.sleep(10000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        // Master releases the resource

        mutex.enterCriticalSection();

    }

}
```



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

class MasterSlaveMutualExclusion {
    private Semaphore mutex;
    private boolean isMaster;

    public MasterSlaveMutualExclusion(boolean isMaster) {
        this.mutex = new Semaphore(1);
        this.isMaster = isMaster;
    }
```



```java
    public void enterCriticalSection() {
        try {
            if (isMaster) {
                mutex.acquire();
            } else {
                // Slave nodes request permission from the master
                requestPermissionFromMaster();
            }
            System.out.println(Thread.currentThread().getName() + " is entering the critical section.");
            Thread.sleep(1000); // Simulating some work inside the critical section
            System.out.println(Thread.currentThread().getName() + " is leaving the critical section.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            if (isMaster) {
                mutex.release();
            }
        }
    }

    private void requestPermissionFromMaster() throws InterruptedException {
        // Here, we can implement communication with the master node to request permission
        // For simplicity, we'll just wait until the master is available
        while (!mutex.tryAcquire()) {
            Thread.sleep(100); // Retry after some time
        }
        mutex.release(); // Immediately release to maintain mutual exclusion
    }
}

class Node implements Runnable {
    private MasterSlaveMutualExclusion mutex;

    public Node(MasterSlaveMutualExclusion mutex) {
        this.mutex = mutex;
    }

    @Override
    public void run() {
        mutex.enterCriticalSection();
    }
}

public class Main {
    public static void main(String[] args) {
        MasterSlaveMutualExclusion mutex = new MasterSlaveMutualExclusion(true); // Assume the first node is the master

        // Create slave nodes
        Thread[] slaves = new Thread[5];
        for (int i = 0; i < slaves.length; i++) {
            slaves[i] = new Thread(new Node(new MasterSlaveMutualExclusion(false)), "Slave-" + i);
        }

        // Start slave threads
        for (Thread slave : slaves) {
            slave.start();
        }

        // Simulate master work
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Master releases the resource
        mutex.enterCriticalSection();
    }
}
```

	Output - Slaves performing operation -  (terminal output below) Master Starts performing Operations after 10 seconds delay -  (terminal output below)
Conclusion :	Thus, we have studied mutual exclusion algorithms and deadlock handling in distributed systems, uncovering pivotal strategies for managing concurrent access to shared resources and mitigating the risk of deadlock scenarios. Through practical exploration and implementation, we've gained valuable insights into synchronization mechanisms crucial for maintaining system integrity and fostering robust distributed computing environments.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Desktop\Don Bosco\College Sem 8\Sem 8 Practicals\Distributed Computing\Resource\Exp 6\Code> javac Main.java
PS C:\Desktop\Don Bosco\College Sem 8\Sem 8 Practicals\Distributed Computing\Resource\Exp 6\Code> java Main
Slave-3 is entering the critical section.
Slave-1 is entering the critical section.
Slave-0 is entering the critical section.
Slave-4 is entering the critical section.
Slave-2 is entering the critical section.
Slave-3 is leaving the critical section.
Slave-2 is leaving the critical section.
Slave-1 is leaving the critical section.
Slave-4 is leaving the critical section.
Slave-0 is leaving the critical section.
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Desktop\Don Bosco\College Sem 8\Sem 8 Practicals\Distributed Computing\Resource\Exp 6\Code> javac Main.java
PS C:\Desktop\Don Bosco\College Sem 8\Sem 8 Practicals\Distributed Computing\Resource\Exp 6\Code> java Main
Slave-3 is entering the critical section.
Slave-1 is entering the critical section.
Slave-0 is entering the critical section.
Slave-4 is entering the critical section.
Slave-2 is entering the critical section.
Slave-3 is leaving the critical section.
Slave-2 is leaving the critical section.
Slave-1 is leaving the critical section.
Slave-4 is leaving the critical section.
Slave-0 is leaving the critical section.
main is entering the critical section.
main is leaving the critical section.
PS C:\Desktop\Don Bosco\College Sem 8\Sem 8 Practicals\Distributed Computing\Resource\Exp 6\Code>
```

References:	https://www.geeksforgeeks.org/mutual-exclusion-in-distributed-system/ https://www.tutorialspoint.com/mutual-exclusion-in-a-distributed-system https://en.wikipedia.org/wiki/Mutual_exclusion

Rubrics for Assessment

Timely Sub-mission	Late	Submitted just after deadline	On time Submission
	0 points	***1 points***	***2 points***
Under-stand-ing	Student is confused about the concept	Students has justifiably understood the concept	Students is very clear about the concepts
	0 points	***2 points***	***3 points***
Per-form-ance	Students has not per-formed the Experiment	Student has performed with help	Student has independently performed the experiment
	0 points	***2 points***	***3 points***
Devel-op-ment	Student struggles to write code	Student can write code with the requirement stated	Student can write exceptional code with his own ideas
	0 points	***2 points***	***3 points***
Overall Discip-line in Lab	No Discipline maintained	Follows Rules	Sincerely performs the prac-tical is always on time
	0 points	***1 points***	***4 points***