Experiment 08 – Design an program for non-token algorithm & Mutual Exclusion Algorithm

Learning Objective:

Students should be able to understand and implement non-token-based and mutual exclusion algorithms, ensuring synchronization and consistency in distributed systems.

Aim:

a) To design a program that demonstrates non-token-based algorithms for achieving mutual exclusion in distributed systems.

b) To develop a program that illustrates mutual exclusion algorithms for process synchronization.

Tools: NodeJS, VSCode.

Theory:

Theory:

1. Introduction to Non-Token-Based Algorithms
   - Non-token-based algorithms manage mutual exclusion by exchanging messages between processes rather than passing a token.
   - Examples include Ricart-Agrawala and Maekawa algorithms.

2. Key Features of Non-Token-Based Algorithms
   - Message Passing: Processes request and release critical section access through message exchange.
   - Logical Clocks: Maintain the order of events using Lamport or vector clocks.
   - Decentralization: No single point of failure due to distributed decision-making.

3. Introduction to Mutual Exclusion Algorithms
   - Mutual exclusion ensures that only one process accesses the critical section at a time to prevent inconsistencies.
   - Types include token-based and non-token-based approaches.

4. Implementation of Non-Token-Based Algorithms
   - Request: Processes send requests for accessing the critical section.
   - Grant: Access is granted when conditions like time-order or quorum are satisfied.
   - Release: Notify all processes after exiting the critical section.

5. Implementation of Mutual Exclusion
   - Token-Based: A unique token circulates among processes, granting critical section access to the holder.
   - Non-Token-Based: Processes communicate directly to coordinate critical section access.

6. Applications
   - Distributed Systems: Ensuring data consistency across nodes.
   - Databases: Managing concurrent access to shared resources.
   - Real-Time Systems: Synchronizing tasks to prevent race conditions.

7. Challenges
   - Communication Overhead: Increased messages in non-token-based algorithms.
   - Fault Tolerance: Handling failures in token-based systems without a central controller.
   - Scalability: Maintaining performance with increasing processes.

Implementation:

Non-Token Algorithm:

```
class LamportClock {
  constructor(pid, maxRequests) {
    this.pid = pid;
    this.clock = 0;
    this.replies = 0;
    this.requests = [];
    this.maxRequests = maxRequests; // Maximum number of critical section requests
    this.requestCount = 0; // Count of critical section requests made by this process
  }

  increment() {
    this.clock += 1;
    return this.clock;
  }

  requestCriticalSection() {
    if (this.requestCount >= this.maxRequests) {
      console.log(`Process ${this.pid} has stopped making requests.`);
      return; // Stop making requests after maxRequests
    }

    this.increment();
    console.log(
      `Process ${this.pid} requesting critical section at ${this.clock}`
    );
    this.requestCount += 1;

    // Send request to all other processes
    processes.forEach((process) => {
      if (process.pid !== this.pid) {
        process.receiveRequest(this.pid, this.clock);
      }
    });
  }

  receiveRequest(senderPid, senderClock) {
    this.increment();
    console.log(
      `Process ${this.pid} received request from ${senderPid} at ${this.clock}`
    );
    // Compare timestamps and send reply if necessary
    if (
      this.clock > senderClock ||
      (this.clock === senderClock && this.pid > senderPid)
    ) {
      this.sendReply(senderPid);
    }
  }
```

Multi Exclusion Algorithm:

```
class LamportMutex {
  constructor(numProcesses) {
    this.numProcesses = numProcesses;
    this.timestamps = new Array(numProcesses).fill(0);
    this.requests = new Array(numProcesses).fill(false);
    this.queue = [];
    this.processing = false;
  }

  requestCS(processId) {
    this.timestamps[processId] =
      Math.max(this.timestamps[processId], this.getMaxTimestamp()) + 1;
    this.requests[processId] = true;
    this.queue.push({ processId, timestamp: this.timestamps[processId] });

    this.checkQueue(processId);
  }

  checkQueue(processId) {
    this.queue.sort((a, b) => a.timestamp - b.timestamp);

    const currentTurn = this.queue[0];

    if (
      currentTurn &&
      currentTurn.processId === processId &&
      !this.processing
    ) {
      console.log(`Process ${processId} entering critical section.`);
      this.processing = true;
      this.queue.shift();
      setTimeout(() => this.releaseCS(processId), 100); // Release after 100ms
    } else {
      console.log(`Process ${processId} waiting for critical section.`);
      setTimeout(() => this.checkQueue(processId), 100); // Check again after 100ms
    }
  }

  releaseCS(processId) {
    console.log(`Process ${processId} exited critical section.`);
    this.processing = false;
    this.requests[processId] = false;
  }

  getMaxTimestamp() {
    return Math.max(...this.timestamps);
  }
}
```

Output:

Non-Token Algorithm Output:

```
~/Desktop/PRACS/DC/8th
     07:26:02 PM
> node A/server.js
Process 2 requesting critical section at 1
Process 0 received request from 2 at 1
Process 1 received request from 2 at 1
Process 1 requesting critical section at 2
Process 0 received request from 1 at 2
Process 2 received request from 1 at 2
Process 2 replying to 1
Process 2 requesting critical section at 3
Process 0 received request from 2 at 3
Process 1 received request from 2 at 3
Process 0 requesting critical section at 4
Process 1 received request from 0 at 4
Process 1 replying to 0
Process 2 received request from 0 at 4
Process 2 replying to 0
Process 2 entering critical section
Process 2 exiting critical section
```

Multi Exclusion Algorithm Output:

```
~/Desktop/PRACS/DC/8th
> node B/server.js
Process 0 entering critical section.
Process 1 waiting for critical section.
Process 0 exited critical section.
Process 2 waiting for critical section.
Process 1 entering critical section.
Process 0 exited critical section.
Process 2 entering critical section.
Process 1 exited critical section.
Process 1 exited critical section.
Process 2 exited critical section.
Process 2 exited critical section.
```

Learning Outcomes:

1. LO1: Understand the principles of non-token-based and mutual exclusion algorithms.
2. LO2: Implement message-passing techniques to achieve mutual exclusion.
3. LO3: Apply synchronization concepts using logical clocks or token-based mechanisms.
4. LO4: Design scalable and fault-tolerant algorithms for distributed systems.
5. LO5: Demonstrate the practical application of these algorithms in managing critical sections in real-world scenarios.

.

Course Outcomes:

1. Design and implement non-token-based algorithms for mutual exclusion in distributed systems.
2. Understand the principles of process synchronization and consistency in critical section management.

Conclusions:

Non-token-based and mutual exclusion algorithms are vital for achieving synchronization and consistency in distributed systems. By implementing these algorithms, students learn to manage access to critical sections efficiently while addressing challenges like communication overhead and fault tolerance.

Viva Questions:

1. What is a non-token-based algorithm, and how does it achieve mutual exclusion?
2. Explain the difference between token-based and non-token-based algorithms.
3. How does the Ricart-Agrawala algorithm work?
4. What is the role of logical clocks in non-token-based algorithms?

For Faculty Use:

Correction Parameter s	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				