**LAB : 5**

Token Ring based Mutual Exclusion is a synchronization technique used in distributed systems where multiple processes compete for access to a shared resource. In this scheme, processes are organized in a logical ring structure, and a special token is passed between processes to control access to the critical section, ensuring that only one process can enter the critical section at a time.

The key features of Token Ring based Mutual Exclusion are:

1. **Token Passing**: A token is passed sequentially from one process to another in a predefined order. Only the process holding the token is allowed to enter the critical section, ensuring exclusive access.
2. **Logical Ring Structure**: Processes are arranged in a logical ring topology. Each process is aware of its neighboring processes in the ring, allowing the token to circulate among them.
3. **Mutual Exclusion**: Only one process can hold the token at any given time, ensuring mutual exclusion in accessing the critical section. This prevents concurrent access by multiple processes and avoids data inconsistency.
4. **Imports and Class Declaration**:
   * Imports the **Semaphore** class from the **java.util.concurrent** package.
   * Declares a class named **TokenRingMutualExclusion**.
5. **Constants and Variables**:
   * **NUM\_PROCESSES**: Defines the number of processes in the ring.
   * **token**: Semaphore instance used for token control. It's initialized with a permit count of 1.
   * **currentProcess**: Keeps track of the index of the current process holding the token.
   * **messageBuffer**: Array to store messages for each process.
6. **Main Method**:
   * Creates **NUM\_PROCESSES** instances of the **Process** class, each representing a process in the token ring.
   * Starts each process in a separate thread.
7. **Process Class:**
   * Declared as a nested static class.
   * Represents individual processes in the token ring system.
   * Contains a constructor to initialise a process ID
8. **Run Method in Process Class**:
   * The **run()** method contains the main logic executed by each process.
   * Runs indefinitely in a loop.
   * Simulates non-critical activities with a sleep operation.
   * Enters the critical section by acquiring the token semaphore.
   * Performs critical section operations if the current process holds the token.
   * Implements message passing logic between processes using a message buffer.
   * Exits the critical section and passes the token to the next process.
   * Handles **InterruptedException** by printing the stack trace.
   * uctor to initialize the process ID.
9. **Imports and Class Declaration**:
   * The code imports the **Semaphore** class from the **java.util.concurrent** package.
   * It declares a class named **TokenRingMutualExclusion**.
10. **Constants and Variables**:
    * **NUM\_PROCESSES**: Defines the number of processes in the ring. In this case, it's set to 5.
    * **token**: Semaphore instance used for token control. It's initialized with a permit count of 1, meaning only one process can hold the token at a time.
    * **currentProcess**: Keeps track of the index of the current process holding the token.
    * **messageBuffer**: Array to store messages for each process. Each element corresponds to a process in the ring.
11. **Main Method**:
    * The **main()** method starts the execution of the program.
    * It creates and starts threads for each process in the ring using a loop.
    * Each thread is created with an instance of the **Process** class and started immediately.

Process Class:

Declared as a nested static class within TokenRingMutualExclusion.

Implements the Runnable interface, allowing instances of Process to be executed in separate threads.

Contains a constructor to initialize the processId of each process.

The run() method defines the main logic executed by each process. It runs indefinitely in a loop, simulating continuous execution.

Inside the loop, critical section operations, message passing logic, and exiting the critical section are performed.

In this snippet, we define a nested static class named **Process**, which represents individual processes in the token ring system. Here's how it works with an example:

Suppose we have a token ring-based distributed system with multiple processes, each identified by a unique **processId**.

* **Nested Static Class**: The **Process** class is declared as a nested static class within another class, **TokenRingMutualExclusion** (not shown in this snippet). This organization keeps the code modular and encapsulated.
* **Instance Variable**: The **processId** variable is an instance variable of the **Process** class, representing the identifier of each process in the token ring.
* **Constructor**: The class has a constructor that takes an integer parameter **processId**. When an instance of **Process** is created, the constructor initializes the **processId** of that instance with the provided value.

For example, let's create three **Process** instances:

Process process1 = new Process(0); Process process2 = new Process(1); Process process3 = new Process(2);

In this example:

* **process1** represents the process with **processId** 0.
* **process2** represents the process with **processId** 1.
* **process3** represents the process with **processId** 2.
* **Override Annotation**: Indicates that this method overrides a method in the superclass or interface.
* **run() Method**: This method contains the main logic executed by each process.
* **Infinite Loop**: The **while (true)** loop ensures that the process continues to execute indefinitely.
* **Non-critical Section**: Simulates non-critical activities by sleeping the thread for 1000 milliseconds (1 second). This delay represents the time spent outside the critical section.
* Entering Critical Section: Acquires the token semaphore using the acquire() method. This operation ensures that only one process can enter the critical section at a time.
* **Conditional Check**: Checks if the current process (identified by **processId**) is allowed to enter the critical section. If it is, the process proceeds with critical section operations and message passing logic.
* **Critical Section Entry**: Prints a message indicating that the process has entered the critical section.
* **Critical Section Operations**: This part is a placeholder for critical section operations. Here, we check if the current process has received any messages in its message buffer and process them accordingly.
* **Message Passing Logic**: Determines the recipient process for the message based on the current process's identifier (**processId**). If the recipient's message buffer is empty, the current process sends a message to it.
* **Exiting Critical Section**: Prints a message indicating that the process has exited the critical section. It then updates the **currentProcess** variable to pass the token to the next process in the ring.
* **Release Token Semaphore**: Releases the token semaphore using the **release()** method. This allows other processes to acquire the token and enter the critical section.
* **Exception Handling**: Catches any **InterruptedException** that might occur during thread execution and prints the stack trace. This ensures proper error handling.