**Assignment 5 :**

This code implements the Token Ring algorithm for process synchronization in a distributed system with N processes. Let's go through the code step by step:

1. The code begins by importing the necessary Java library (`java.util`) for data structures and utilities.

2. The class `TokenRing` is defined, which contains the `main` method where the execution of the program starts.

3. Three constants are defined:

- `N`: Represents the number of processes in the distributed system.

- `TOKEN`: Represents the value of the token. In this case, `-1` is used.

- `CS\_TIME`: Represents the time (in milliseconds) spent in the critical section.

4. Two boolean arrays are declared:

- `hasToken`: Represents whether process i has the token. It is initialized with `N` elements, and all elements are initially set to `false`.

- `inCS`: Represents whether process i is in the critical section. It is also initialized with `N` elements, and all elements are initially set to `false`.

5. A variable `tokenHolder` is declared and initialized with the value `-1`, indicating that no process currently holds the token.

6. The `process` method is defined, which represents the behavior of each process in the system. It takes an `id` parameter to identify the process.

7. Inside the `process` method, there is an infinite loop that simulates the continuous execution of the process.

8. If the process has the token (`hasToken[id]` is `true`), it enters the critical section:

- It sets `inCS[id]` to `true` to indicate that the process is in the critical section.

- It prints a message indicating that the process is entering the critical section.

- It simulates the execution of the critical section by sleeping for `CS\_TIME` milliseconds.

- It prints a message indicating that the process is exiting the critical section.

- It releases the token by setting `hasToken[id]` to `false`.

- It determines the next process to hold the token by calculating `(id + 1) % N`.

- It assigns the token to the next process by setting `hasToken[nextId]` to `true` and updates the `tokenHolder` variable.

9. If the process does not have the token, it waits for it by sleeping for 100 milliseconds.

10. The `main` method is defined, which is the entry point of the program.

11. The token holder is initialized by setting `hasToken[0]` to `true` and `tokenHolder` to `0`, indicating that process 0 initially holds the token.

12. A list of `Thread` objects is created to store the threads representing each process.

13. A loop is used to create and start a thread for each process:

- The `id` variable is assigned the value of `i`.

- A new `Thread` object is created with a lambda expression representing the process behavior.

- The thread is added to the `threads` list.

- The thread is started.

14. Another loop is used to wait for all the processes to finish by calling `join()` on each thread.

The code simulates a distributed system where each process executes the `process` method. Only the process that holds the token is allowed to enter the critical section. The token is passed from one process to another in a circular manner, ensuring that each process gets a chance to enter the critical section. The program runs until all processes have completed execution.