**Assignment 3**

1. **Problem Statement**
   1. Token-Based Algorithm on Ring Topology.
2. **Source Code**
   1. **>> Main.java**

/\* About Project

Token-Based Mutual Exclusion using Threads

In this system, we implement mutual exclusion using tokens and threads.

Initially, an arbitrary node holds the token.

Nodes pass the token to each other based on requests,

ensuring continuous execution of processes.

Here, distributed systems are represented by nodes that can run independently. \*/

/\* About Main Class

In the Main Class, we retrieve a file from the disk containing the names of the nodes.

The file format consists of either comma-separated node names (nodeName1, nodeName2) or

each node name on a separate line (nodeName1 followed by nodeName2).

We then create Node objects using these names and assign IDs to them before starting the Node threads. \*/

//Author Rahul Biswas, Arijit Mondal

//M.Sc. Computer Science, Semester-2 , University of Calcutta

import java.awt.\*;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

List<String> nodeNames = new ArrayList<>(); // List to store node names from file

int id = 1; // ID of the first node

String filename = openFileDialog(); // Get file name from gui component

if (filename == null) {

System.out.println("No file selected.");

return;

}

try {

//Read node names from the file and set them in nodeNames list

Scanner scanner = new Scanner(new File(filename));

while (scanner.hasNextLine()) {

String line = scanner.nextLine();

String[] tokens = line.split(","); // Split line by commas

nodeNames.addAll(List.of(tokens)); // Store node name

}

} catch (FileNotFoundException e) { // Handle the exception if the file is not found

System.out.println("Error: File not found ");

}

List<Node> nodes = new ArrayList<>(); // list of node objects

for (String nodeName : nodeNames) {

nodes.add(new Node(String.valueOf(id), nodeName));

id++; // Set id according to file

}

Messenger messageSystem = new Messenger(nodes);

Random random = new Random();

int randomIndex = random.nextInt(nodes.size());

nodes.get(randomIndex).setStatus(Status.HAVE\_TOKEN); // Giving token to an arbitrary node

System.out.println();

System.out.println();

System.out.println();

System.out.println("Information Message :: [ "+ nodes.get(randomIndex).getName() +" ] Have Token Initially.");

// Start all threads

for (Node node : nodes) {

node.setMessenger(messageSystem); // Set the messenger object

new Thread(node).start();

}

}

// Open File Dialog box

private static String openFileDialog() {

FileDialog fd = new FileDialog((Frame) null, "Open", FileDialog.LOAD);

fd.setVisible(true);

String filename = fd.getFile();

// Validate file extension

if (filename != null) {

return fd.getDirectory() + filename;

}

return null;

}

}

* 1. **>> Node,java**

/\* About Node

The Node class represents a real-life simulation of distributed processes.

Each node operates independently and lacks knowledge about other processes.

It communicates by sending and receiving tokens through the channel.

\*/

import java.util.LinkedList;

import java.util.Queue;

import java.util.Random;

public class Node implements Runnable {

// Used objects

private Messenger messenger;

private Status status = Status.NONE;

private Random random = new Random();

// Used variables

private String UID; //ID of a node

private String name; // Name of a node

private int criticalStateClock = 0; // Clock while in critical section

private int afterCriticalStateClock = 0; // Clock after critical section

private Boolean isRequested = false; // Is request for token

// Used data structures

private Queue<String> requestQueue = new LinkedList<>(); // Requesting queue for nodes who are requested

public Node(String UID, String name) {

this.UID = UID;

this.name = name;

}

@Override

public void run() {

while (!Thread.currentThread().isInterrupted()) {

if (afterCriticalStateClock == 0) { // Node is come out of a critical state

if (status != Status.REQUEST\_TOKEN && status != Status.HAVE\_TOKEN) {

status = Status.NONE;

}

}

changeStatus(); // change status

checkClock(); // set status according to clock

requestToken(); // request for token

if (status == Status.HAVE\_TOKEN) {

if (criticalStateClock == 0) { // If it can send token

if (!requestQueue.isEmpty()) { // request queue is not empty

sendToken(); // release token

} else {

criticalStateClock += 5; // If request queue is empty then hold the token

}

}

}

try {

Thread.sleep(1000);

} catch (Exception e) {

Thread.currentThread().interrupt();

}

}

}

// Get name

public String getName() {

return this.name;

}

// Get UID

public String getUID() {

return UID;

}

// Get Status

public Status getStatus() {

return this.status;

}

// Set Status

public void setStatus(Status status) {

this.status = status;

}

// Set channel object

public void setMessenger(Messenger messenger) {

this.messenger = messenger;

}

// Add requesting node to the queue

public void setRequest(String recipientID) {

requestQueue.add(recipientID);

}

// Set received queue in own queue

public void setRequestQueue(Queue<String> requestQueue) {

this.requestQueue = requestQueue;

}

// Send token

public void sendToken() {

this.status = Status.AFTER\_CRITICAL\_STATE;

afterCriticalStateClock = 5;

messenger.releaseToken(this.UID, requestQueue); // Release token to channel

}

// Request Token

private void requestToken() {

if (status == Status.REQUEST\_TOKEN && !isRequested) {

MessageType msg = MessageType.REQUESTING;

String message = name + ":" + UID + ":" + msg;

messenger.send(message); // sending token request to the channel

isRequested = true;

}

}

// Receive Token

public void receiveToken(Queue<String> requestQueue) {

setStatus(Status.HAVE\_TOKEN); // Update the status to HAVE\_TOKEN

criticalStateClock = random.nextInt((5 - 1) + 1) + 1; // Stay in critical state for 1 tick to 5 ticks;

System.out.println("Information Message :: [ " + name + " ] is in Critical State.");

setRequestQueue(requestQueue);

}

// Changing status internally

private void changeStatus() {

if (status == Status.NONE) {

int number = random.nextInt(100); // Generate a random number between 0 and 99

if (number < 25) { // 25% probability

this.status = Status.REQUEST\_TOKEN;

}

}

}

// Set status according to the clock

private void checkClock() {

if (afterCriticalStateClock != 0) { // If node is in after Critical state

afterCriticalStateClock--;

}

if (criticalStateClock != 0) { // If node is in critical state

// If still in critical state

status = Status.HAVE\_TOKEN;

criticalStateClock--;

}

if (criticalStateClock == 0 && afterCriticalStateClock != 0) {

// If just came out from critical state but still cannot request

status = Status.AFTER\_CRITICAL\_STATE;

isRequested = false;

}

}

}

* 1. **>> Messenger.java**

/\* About Messenger class

In the Messenger class, we utilize it as a channel representing a logical ring.

We implement this logical ring using a circular queue.

Acting as a channel, it facilitates the delivery of request messages, tokens, and

the request queue where nodes request tokens.

The circular queue or logical ring channel only sends nodes from the sender to the recipient node.

\*/

import java.util.List;

import java.util.Queue;

public class Messenger {

private CircularQueue<Node> nodes; // Logical ring channel of nodes

private String senderName; // Sender's name in the channel

private String recipientName; // Recipient's name in the channel

public Messenger(List<Node> nodeNames) {

nodes = new CircularQueue<>(nodeNames.size() + 1);

for (Node nodeName : nodeNames) {

nodes.enqueue(nodeName); // Set nodes in the channel

}

}

// Token request to the token holder

public synchronized void send(String message) { // Synchronized for thread safety

String[] messageContent = message.split(":");

boolean nodeFound = false; // Requesting Node is found

boolean requestSend = false; // Token requests are sent to the sender.

senderName = null; // Reset senderName

recipientName = null; // Reset recipientName

while (!requestSend) { // While Token request is not sent

for (Node node : nodes) {

if (node.getUID().equals(messageContent[1]) && !nodeFound) {

// Token requesting node found

senderName = node.getName();

nodeFound = true;

}

if (nodeFound && node.getStatus() == Status.HAVE\_TOKEN) { // Token Holder node found

recipientName = node.getName();

System.out.println("Request Message :: [ " + senderName + " ] Requesting Token From [ " + recipientName + " ].");

node.setRequest(messageContent[1]); // Set node name in the requesting queue of token holder

requestSend = true; // Token request is sent

break; // Exit the loop after sending the request

}

}

}

}

// Sending token to the next token holder (first element in the requesting queue)

public synchronized void releaseToken(String senderUID, Queue<String> requestQueue) { // Synchronized for thread safety

boolean nodeFound = false;

String priorityRecipientUID = requestQueue.poll(); // Getting the first node from the queue

for (Node node : nodes) {

if (node.getUID().equals(senderUID) && !nodeFound) {

// Token Holder node found

senderName = node.getName();

nodeFound = true;

} else if (nodeFound && node.getUID().equals(priorityRecipientUID)) {

// Next Token holder node found

recipientName = node.getName();

System.out.println("Send Message :: [ " + senderName + " ] Sending Token To [ " + recipientName + " ].");

node.receiveToken(requestQueue); // Releasing token to the next token holder

break;

}

}

}

}

* 1. **>> Circular Queue.java**

/\* This is Circular Queue implementation using Queue \*/

import java.util.Iterator;

import java.util.LinkedList;

import java.util.Queue;

public class CircularQueue<T> implements Iterable<T> {

private Queue<T> queue;

private int maxSize;

public CircularQueue(int maxSize) {

this.maxSize = maxSize;

this.queue = new LinkedList<>();

}

public void enqueue(T item) {

if (queue.size() == maxSize) {

queue.poll(); // Remove the oldest element if the queue is full

}

queue.offer(item);

}

@Override

public Iterator<T> iterator() {

return new CircularIterator();

}

private class CircularIterator implements Iterator<T> {

private Iterator<T> iterator;

public CircularIterator() {

this.iterator = queue.iterator();

}

@Override

public boolean hasNext() {

return !queue.isEmpty();

}

@Override

public T next() {

if (!iterator.hasNext()) {

iterator = queue.iterator(); // Reset iterator when reaching the end

}

return iterator.next();

}

@Override

public void remove() {

iterator.remove();

}

}

}

* 1. **>> Status.java**

public enum Status {

REQUEST\_TOKEN,

HAVE\_TOKEN,

AFTER\_CRITICAL\_STATE,

NONE

}

* 1. **>> MessageType.java**

public enum MessageType {

REQUESTING

}

1. **Pre-requisites & Assumptions**
   1. **Message Handling:**

The Messenger class handles message passing between nodes in logical ring topology.

* 1. **Node Behavior:**

Each Node simulates behavior in a distributed system, including requesting, entering, and exiting critical states using the token-based algorithm for ring topology.

* 1. **Randomized Status Changes:**

The Node class randomly changes its status to simulate requests for entering the critical section. Each node 15% chance for requesting state.

* 1. **File Structure**

Ensure you have a text file containing the node names separated by commas. The file should look like this:

node1, node2, node3, node4

The file can contain multiple lines if you have many nodes:

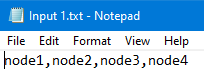
node1, node2

node3, node4

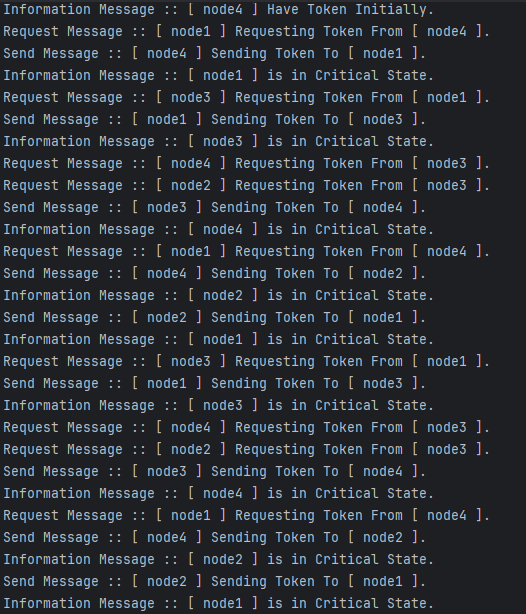
* 1. **Awt requirement**

You need Java AWT as the program includes some AWT components for selecting files.

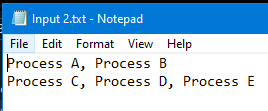
1. **Output**
   1. **Input & Output 1**
      * 1. **Input File:**

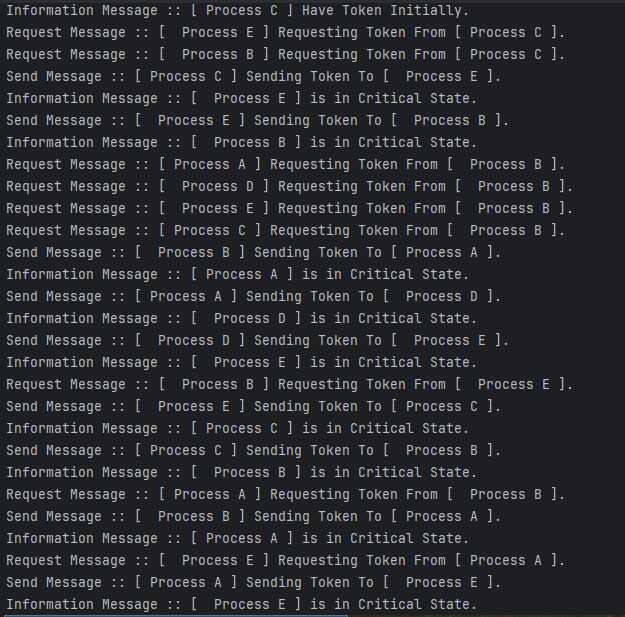


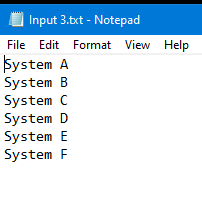
* + - 1. **Output File:**

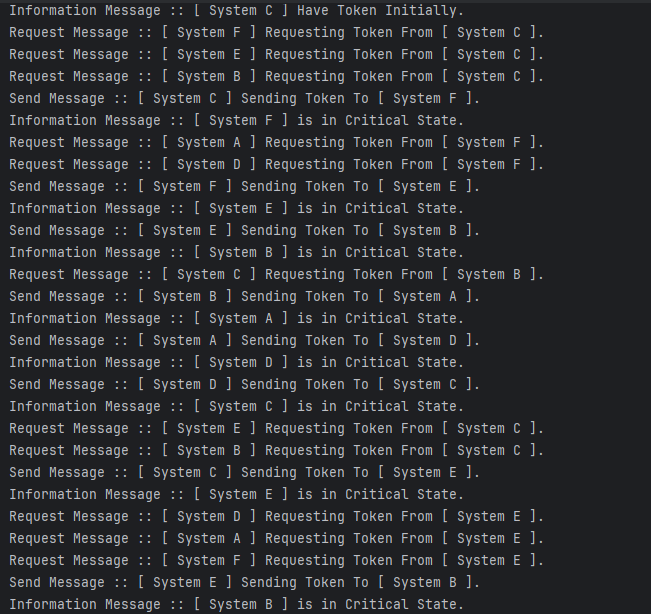
****

* 1. **Input & Output 2**
     + 1. **Input File:**



* + - 1. **Output File:**
  1. **Input & Output 3**
     + 1. **Input File:**



* + - 1. **Output File:**

1. **Remarks**
   1. **Thread Safety:**

Synchronized critical sections in the Messenger class to ensure safe concurrent access and modifications.

* 1. **Clear State Management:**

Reset sender and recipient names at the beginning of the send method to avoid incorrect state persistence.

* 1. **Probability Correction:**

Adjusted the probability logic in the Node class to accurately reflect a 15% chance for entering the requesting state.

* 1. **Robust Logging:**

Improved logging to track token requests and transfers clearly, ensuring no message loss and better traceability.

* 1. **Acknowledge Message Implementation:**

Acknowledge messages can be easily implemented using the MessageType enum.

* 1. **Testing Specific Conditions:**

Nodes can be tested for specific conditions by setting specific statuses when starting.