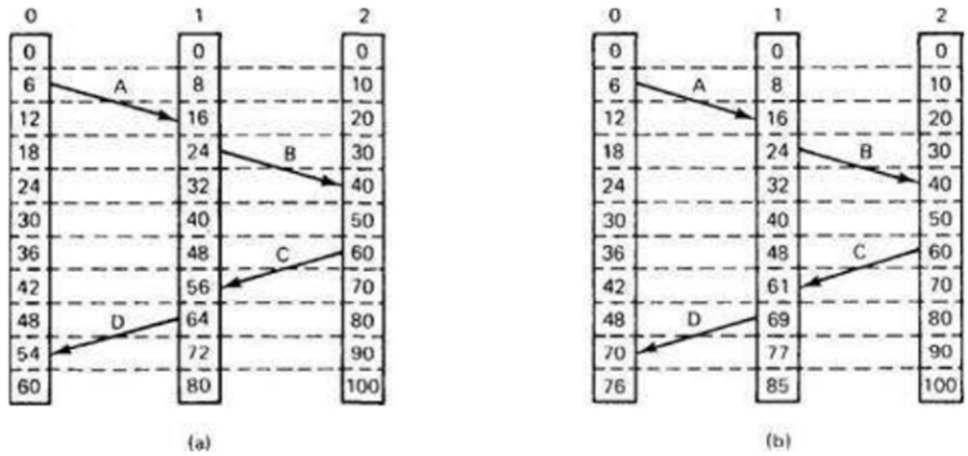
# EXPERIMENT NO. 4

**Aim:** To implement Lamport’s Clock Synchronization Algorithm.

## Theory:

The algorithm of Lamport timestamps is a simple algorithm used to determine the order of events in a distributed computer system. As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead, and conceptually provide a starting point for the more advanced vector clock method. They are named after their creator, Leslie Lamport. Distributed algorithms such as resource synchronization often depend on some method of ordering events to function. For example, consider a system with two processes and a disk. The processes send messages to each other, and also send messages to the disk requesting access. The disk grants access in the order the messages were sent. For example process A sends a message to the disk requesting write access, and then sends a read instruction message to process B. Process B receives the message, and as a result sends its own read request message to the disk. If there is a timing delay causing the disk to receive both messages at the same time, it can determine which message happened- before the other: ( A A happens-before B B if one can get from A A to B B by a sequence of moves of two types: moving forward while remaining in the same process, and following a message from its sending to its reception.) A logical clock algorithm provides a mechanism to determine facts about the order of such events.

21



Lamport invented a simple mechanism by which the happened-before ordering can be captured numerically. A Lamport logical clock is an incrementing software counter maintained in each process. Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender. The above-mentioned vector clock is a generalization of the idea into the context of an arbitrary number of parallel, independent processes. The algorithm follows some simple rules:

1. A process increments its counter before each event in that process;
2. When a process sends a message, it includes its counter value with the message;
3. On receiving a message, the counter of the recipient is updated, if necessary, to the greater of its current counter and the timestamp in the received message. The counter is then incremented by 1 before the message is considered received.

22

## Program:

//Lamport.java

import java.util.\*;

import java.util.HashMap; import java.util.Scanner; import javax.swing.\*; import java.awt.\*;

import java.awt.geom.\*; public class Lamport {

int e[][] = new int[10][10];

int en[][] = new int[10][10]; int ev[] = new int[10];

int i, p, j, k;

HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>(); int xpoints[] = new int[5];

int ypoints[] = new int[5];

class draw extends JFrame { private final int ARR\_SIZE = 4;

void drawArrow(Graphics g1, int x1, int y1, int x2, int y2) { Graphics2D g = (Graphics2D) g1.create();

double dx = x2 - x1, dy = y2 - y1; double angle = Math.*atan2*(dy, dx);

int len = (int) Math.*sqrt*(dx \* dx + dy \* dy);

AffineTransform at = AffineTransform.*getTranslateInstance*(x1, y1); at.concatenate(AffineTransform.*getRotateInstance*(angle)); g.transform(at);

// Draw horizontal arrow starting in (0,0)

g.drawLine(0, 0, len, 0);

g.fillPolygon(new int[]{len, len - ARR\_SIZE, len - ARR\_SIZE, len}, new int[]{0, -ARR\_SIZE, ARR\_SIZE, 0}, 4);

}

public void paintComponent(Graphics g) { for (int x = 15; x < 200; x += 16) {

drawArrow(g, x, x, x, 150); drawArrow(g, 30, 300, 300, 190);

}

23

}

public void paint(Graphics g) { int h1, h11, h12;

Graphics2D go = (Graphics2D) g; go.setPaint(Color.*black*);

for (i = 1; i <= p; i++) {

go.drawLine(50, 100 \* i, 450, 100 \* i);

}

for (i = 1; i <= p; i++) {

for (j = 1; j <= ev[i]; j++) { k = i \* 10 + j; go.setPaint(Color.*blue*);

go.fillOval(50 \* j, 100 \* i - 3, 5, 5);

go.drawString("e" + i + j + "(" + en[i][j] + ")", 50 \* j, 100 \* i - 5); h1 = hm.get(k);

if (h1 != 0) { h11 = h1 / 10;

h12 = h1 % 10;

go.setPaint(Color.*red*);

drawArrow(go, 50 \* h12 + 2, 100 \* h11, 50 \* j + 2, 100 \* i);

}

}

}

}

}

public void calc() {

Scanner sc = new Scanner(System.*in*);

System.*out*.println("Enter the number of process:"); p = sc.nextInt();

System.*out*.println("Enter the no of events per process:"); for (i = 1; i <= p; i++) {

ev[i] = sc.nextInt();

}

System.*out*.println("Enter the relationship:");

24

for (i = 1; i <= p; i++) { System.*out*.println("For process:" + i); for (j = 1; j <= ev[i]; j++) {

System.*out*.println("For event:" + (j)); int input = sc.nextInt();

k = i \* 10 + j; hm.put(k, input); if (j == 1) {

en[i][j] = 1;

}

}

}

for (i = 1; i <= p; i++) {

for (j = 2; j <= ev[i]; j++) { k = i \* 10 + j;

if (hm.get(k) == 0) { en[i][j] = en[i][j - 1] + 1;

} else {

int a = hm.get(k); int p1 = a / 10; int e1 = a % 10;

if (en[p1][e1] > en[i][j - 1]) {

en[i][j] = en[p1][e1] + 1;

} else {

en[i][j] = en[i][j - 1] + 1;

}

}

}

}

for (i = 1; i <= p; i++) {

for (j = 1; j <= ev[i]; j++) {

System.*out*.println(en[i][j]);

}

}

JFrame jf = new draw(); jf.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*); jf.setSize(500,500);

jf.setVisible(true);

}

public static void main(String[] args) {

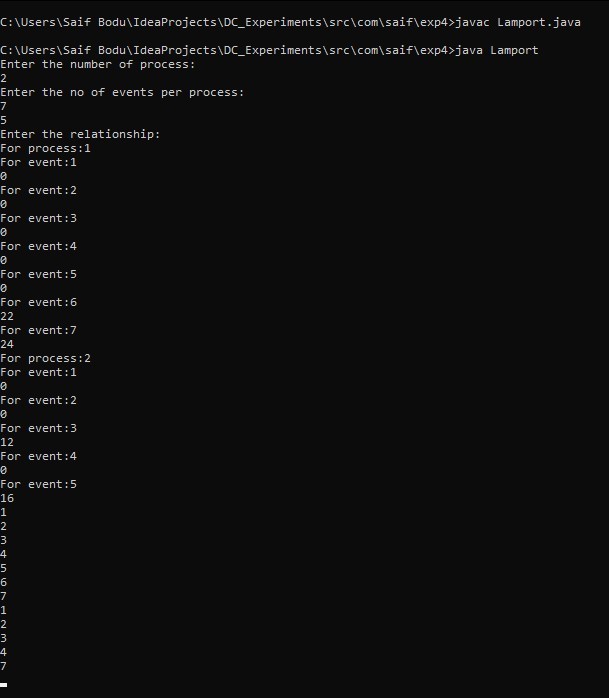
25

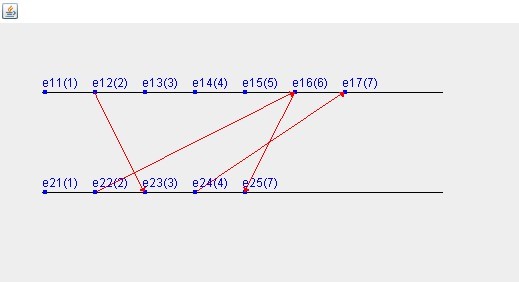
Lamport lam = new Lamport(); lam.calc();

}

}

## Output:

****



27