Experiment 4

AIM: Write a program to implement Lamport logical clock synchronization between processes with different clocks and update intervals.

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.

ALGORITHM

1. All the process counters start with value 0.
2. A process increments its counter for each event (internal event,message sending, message receiving) in that process.
3. When a process sends a message, it includes its (incremented) counter value with the message.
4. On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one

SOURCE CODE:

```
#include <stdio.h>
int max1(int a, int b) //to find the maximum timestamp between two events
{
 return a > b ? a : b;
}
int main()
{
 int i, j, k, p1[20], p2[20], e1, e2, dep[20][20];
 printf("enter the number of process and events : ");
 scanf("%d %d", &e1, &e2);
 for (i = 0; i < e1; i++)
 p1[i] = i + 1;
```

```
 for (i = 0; i < e2; i++)
 p2[i] = i + 1;
 printf("enter the dependency matrix:\n");
 printf("\t enter 1 if e1->e2 \n\t enter -1, if e2->e1 \n\t else enter 0 \n\n");
 for (i = 0; i < e2; i++)
 printf("\te2%d", i + 1);
 for (i = 0; i < e1; i++)
 {
 printf("\n e1%d \t", i + 1);
 for (j = 0; j < e2; j++)
 scanf("%d", &dep[i][j]);
 }
 for (i = 0; i < e1; i++)
 {
 for (j = 0; j < e2; j++)
 {
 if (dep[i][j] == 1) //change the timestamp if dependency exist
 {
 p2[j] = max1(p2[j], p1[i] + 1);
 for (k = j; k < e2; k++)
 p2[k + 1] = p2[k] + 1;
 }
 if (dep[i][j] == -1) //change the timestamp if dependency exist
 {
 p1[i] = max1(p1[i], p2[j] + 1);
 for (k = i; k < e1; k++)
 p2[k + 1] = p1[k] + 1;
 } }
 }
printf("\nLogical clock value for the above input is as below\n");
 printf("\nP1 : ");
 for (i = 0; i < e1; i++)
 printf("%d  ", p1[i]);
 printf("\nP2 : ");
 for (j = 0; j < e2; j++)
 printf("%d  ", p2[j]);
  printf("\n");
 return 0;
}
```

OUTPUT:

```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ gcc -pthread -o lamport lamport.c
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$ ./lamport
enter the number of process and events : 3 4
enter the dependency matrix:
        enter 1 if e1->e2
        enter -1, if e2->e1
        else enter 0

       e21     e22     e23     e24
 e11   0       0       0       0

 e12   0       0       1       0

 e13   0      -1       0       0

Logical clock value for the above input is as below

P1 : 1  2  3
P2 : 1  2  3  4
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/college/7th Semester/Distributed System/LAb$
```

LEARNING OUTCOMES:

One of the shortcomings of Lamport Timestamps is rooted in the fact that they only partially order events (as opposed to total order). Partial order indicates that not every pair of events need be comparable. If two events can't be compared, we call these events concurrent. The problem with Lamport Timestamps is that they can't tell if events are concurrent or not. This problem is solved by Vector Clocks.