LAB 6

Aim: To implement Lamport algorithm for logical clock synchronization

Lab Outcome:
Implement techniques for clock synchronization.

Theory:
The Lamport algorithm is a logical clock synchronization algorithm used in distributed systems to establish a partial ordering of events. The algorithm is based on the concept of logical clocks, which assign a timestamp to each event in a distributed system.

In Lamport's algorithm, each process maintains a logical clock that ticks whenever an event occurs at that process. The logical clock value assigned to an event is the maximum of the current logical clock value of the process and the timestamp of the incoming message that triggered the event.

The logical clocks in Lamport's algorithm satisfy the following two properties:

- If event A happens before event B, then the logical clock value of A is less than the logical clock value of B.
- If events A and B are concurrent, then the logical clock value of A is not equal to the logical clock value of B.

Using Lamport's algorithm, processes can synchronize their logical clocks so that they agree on the ordering of events. This enables distributed systems to reason about causality, and to detect and resolve conflicts that may arise due to concurrent events.

Lamport Algorithm
Lamport developed a "happens-before" notation to express this: If a and b are events in the same process, and a occurs before b, then a → b is true. If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then a → b. This relationship is transitive i.e. a→b and b→c then a→c. Satisfying conditions for implementing clock: If a→b then c(a)<c(b). Implementation of logical clock:
Condition 1: If a and b are two events within the same process Pi and a occur before b then Ci(a) < Ci (b).
Condition 2: if a is the sending of a message by process Pi and b is the receipt of that message by process Pj then Ci(a) < Cj(b).
Condition 3: A clock Ci associated with a process Pi must always go forward never backward that is correction to the time of clock is done by +ive adding.

Total ordering and Partial ordering
Total ordering is an ordering that defines the exact order of every element in the series.

Partial ordering of elements in a series is an ordering that doesn't specify the exact order of every item, but only defines the order between certain key items that depend on each other.

The meaning of these words is the same in the context of distributed computing. The only significance of distributed computing to these terms is the fact that partial ordering of events is much commoner than total ordering. In a local, single-threaded application, the order in which events happen is ordered, implicitly, since the CPU can only do one thing at a time. In a distributed system, you generally only coordinate a partial ordering of those events that have a dependency on one another and let other events happen in whatever order they happen.

Example, taken from the comments: If you have three events {A, B, C}, then they are ordered if they always have to happen in the order A > B > C. However, if A must happen before C, but you don't care when B happens, then they are partially ordered. In this case we would say that the sequences A > B > C, A > C > B, and B > A > C all satisfy the partial ordering.

(Code & Output:)

```
from multiprocessing import Process, Pipe
from os import getpid
from datetime import datetime

def local_time(counter):
    return ' (LAMPORT_TIME={}, LOCAL_TIME={})\n'.format(counter, datetime.now())

def calc_recv_timestamp(recv_time_stamp, counter):
    return max(recv_time_stamp, counter) + 1

def event(pid, counter):
    counter += 1
    print('Something happened in {} !\n'.format(pid) + local_time(counter))
    return counter

def send_message(pipe, pid, counter):
    counter += 1
    pipe.send(('Empty shell', counter))
    print('Message sent from ' + str(pid) + local_time(counter)+'\n')
    return counter

def recv_message(pipe, pid, counter):
    message, timestamp = pipe.recv()
    counter = calc_recv_timestamp(timestamp, counter) # timestamp - sender's counter,
counter - recv's counter
    print('Message received at ' + str(pid)  + local_time(counter))
    return counter

def process_one(pipe12):
    pid = getpid()
    counter = 0
    print("Process 1 Init Counter: "+str(counter))
    counter = event(pid, counter)
    print("Process 1 Counter: "+str(counter))
    counter = send_message(pipe12, pid, counter)
    print("Process 1 Counter: "+str(counter))
    counter  = event(pid, counter)
    print("Process 1 Counter: "+str(counter))
```

```
    counter = recv_message(pipe12, pid, counter)
    print("Process 1 Counter: "+str(counter))
    counter  = event(pid, counter)
    print("Process 1 Counter: "+str(counter))

def process_two(pipe21, pipe23):
    pid = getpid()
    counter = 0
    print("Process 2 Init Counter: "+str(counter))
    counter = recv_message(pipe21, pid, counter)
    print("Process 2 Counter: "+str(counter))
    counter = send_message(pipe21, pid, counter)
    print("Process 2 Counter: "+str(counter))
    counter = send_message(pipe23, pid, counter)
    print("Process 2 Counter: "+str(counter))
    counter = recv_message(pipe23, pid, counter)
    print("Process 2 Counter: "+str(counter))

def process_three(pipe32):
    pid = getpid()
    counter = 0
    print("Process 3 Init Counter: "+str(counter))
    counter = recv_message(pipe32, pid, counter)
    print("Process 3 Counter: "+str(counter))
    counter = send_message(pipe32, pid, counter)
    print("Process 3 Counter: "+str(counter))

if __name__ == '__main__':
    oneandtwo, twoandone = Pipe()
    twoandthree, threeandtwo = Pipe()

    process1 = Process(target=process_one, args=(oneandtwo,))
    process2 = Process(target=process_two, args=(twoandone, twoandthree))
    process3 = Process(target=process_three, args=(threeandtwo,))

    process1.start()
    process2.start()
    process3.start()

    process1.join()
    process2.join()
    process3.join()
```

Output:

```
PS C:\Users\krisc\Documents\.Notes\.codes> & C:/Python312/python.exe c:/Users/krisc/Documents/.Notes/.codes/dc/6/lamport.py
Process 1 Init Counter: 0
Something happened in 21788 !
 (LAMPORT_TIME=1, LOCAL_TIME=2024-05-03 00:10:59.379867)

Process 1 Counter: 1
Message sent from 21788 (LAMPORT_TIME=2, LOCAL_TIME=2024-05-03 00:10:59.379867)


Process 1 Counter: 2
Something happened in 21788 !
 (LAMPORT_TIME=3, LOCAL_TIME=2024-05-03 00:10:59.379867)

Process 1 Counter: 3
Process 2 Init Counter: 0
Message received at 23768 (LAMPORT_TIME=3, LOCAL_TIME=2024-05-03 00:10:59.381860)

Process 2 Counter: 3
Message sent from 23768 (LAMPORT_TIME=4, LOCAL_TIME=2024-05-03 00:10:59.381860)

Message received at 21788 (LAMPORT_TIME=5, LOCAL_TIME=2024-05-03 00:10:59.381860)


Process 1 Counter: 5
Process 2 Counter: 4
Something happened in 21788 !
 (LAMPORT_TIME=6, LOCAL_TIME=2024-05-03 00:10:59.382441)
Message sent from 23768 (LAMPORT_TIME=5, LOCAL_TIME=2024-05-03 00:10:59.382441)



Process 1 Counter: 6
Process 2 Counter: 5
Process 3 Init Counter: 0
Message received at 9728 (LAMPORT_TIME=6, LOCAL_TIME=2024-05-03 00:10:59.389585)

Process 3 Counter: 6
Message sent from 9728 (LAMPORT_TIME=7, LOCAL_TIME=2024-05-03 00:10:59.390582)

Message received at 23768 (LAMPORT_TIME=8, LOCAL_TIME=2024-05-03 00:10:59.390582)


Process 3 Counter: 7
Process 2 Counter: 8
PS C:\Users\krisc\Documents\.Notes\.codes>
```

Conclusions :

1. Learnt the functioning of Lamport's Algorithm for logical clock synchronization.
2. Understood the terminologies such as Partial ordering and Total ordering
3. In conclusion, a Lamport logical clock is an incrementing counter maintained in each process. Conceptually, this logical clock can be thought of as a clock that only has meaning concerning messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender (causality).

Postlab Questions:

1. Distinguish between physical clock and logical clock synchronization
2. Show the calculation of the time interval between two synchronizations of a physical clock
3. How will you implement Logical clocks by using counters?
4. Give an example of partial and total ordering of events