**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

events are much more common 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** :

from multiprocessing import Process, Pipe

from os import getpid

from datetime import datetime

def local\_time(counter):

return ' (LAMPORT\_TIME={}, LOCAL\_TIME={})'.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('\nSomething happened in {} !'.format(pid) + local\_time(counter))

return counter

def send\_message(pipe, pid, counter):

counter += 1

pipe.send(('\nEmpty shell', counter))

print('\nMessage sent from ' + str(pid) + local\_time(counter))

return counter

def recv\_message(pipe, pid, counter):

message, timestamp = pipe.recv()

counter = calc\_recv\_timestamp(timestamp, counter)

print('\nMessage received at ' + str(pid) + local\_time(counter))

return counter

def process\_one(pipe12):

pid = getpid()

counter = 0

print("\nProcess 1 Init Counter: "+str(counter))

counter = event(pid, counter)

print("\nProcess 1 Counter: "+str(counter))

counter = send\_message(pipe12, pid, counter)

print("\nProcess 1 Counter: "+str(counter))

counter = event(pid, counter)

print("\nProcess 1 Counter: "+str(counter))

counter = recv\_message(pipe12, pid, counter)

print("\nProcess 1 Counter: "+str(counter))

counter = event(pid, counter)

print("\nProcess 1 Counter: "+str(counter))

def process\_two(pipe21, pipe23):

pid = getpid()

counter = 0

print("\nProcess 2 Init Counter: "+str(counter))

counter = recv\_message(pipe21, pid, counter)

print("\nProcess 2 Counter: "+str(counter))

counter = send\_message(pipe21, pid, counter)

print("\nProcess 2 Counter: "+str(counter))

counter = send\_message(pipe23, pid, counter)

print("\nProcess 2 Counter: "+str(counter))

counter = recv\_message(pipe23, pid, counter)

print("\nProcess 2 Counter: "+str(counter))

def process\_three(pipe32):

pid = getpid()

counter = 0

print("\nProcess 3 Init Counter: "+str(counter))

counter = recv\_message(pipe32, pid, counter)

print("\nProcess 3 Counter: "+str(counter))

counter = send\_message(pipe32, pid, counter)

print("\nProcess 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()

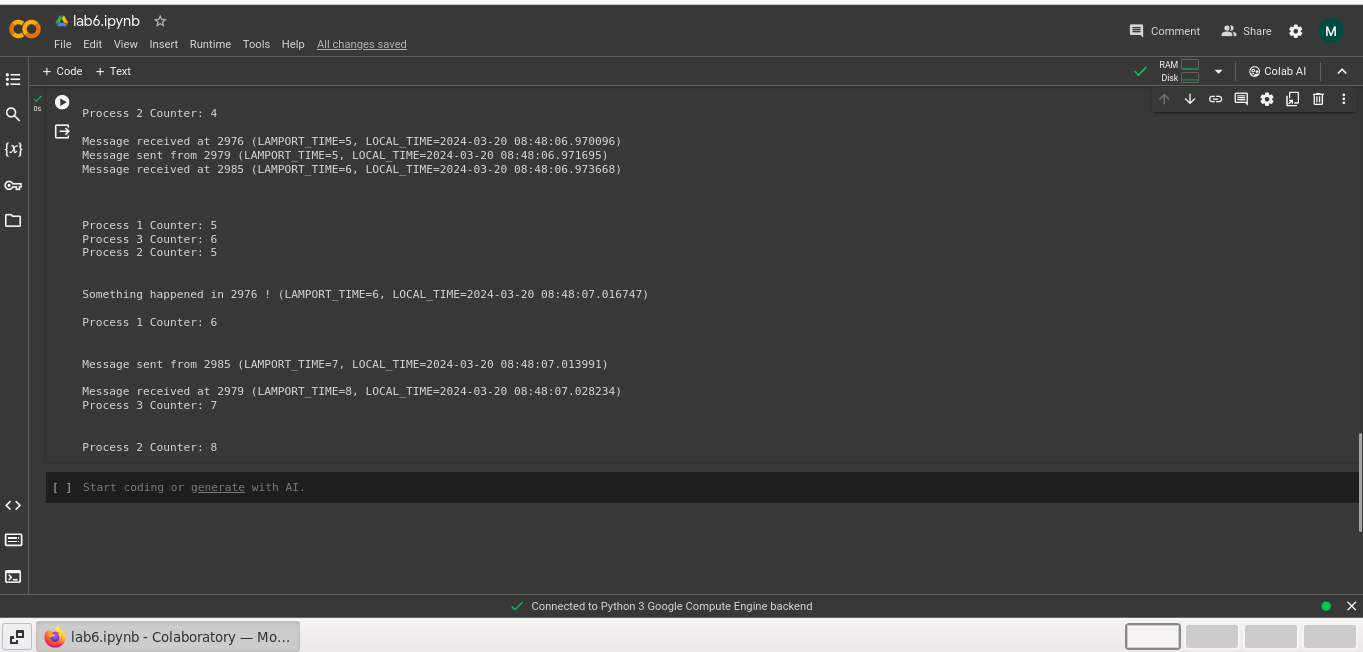
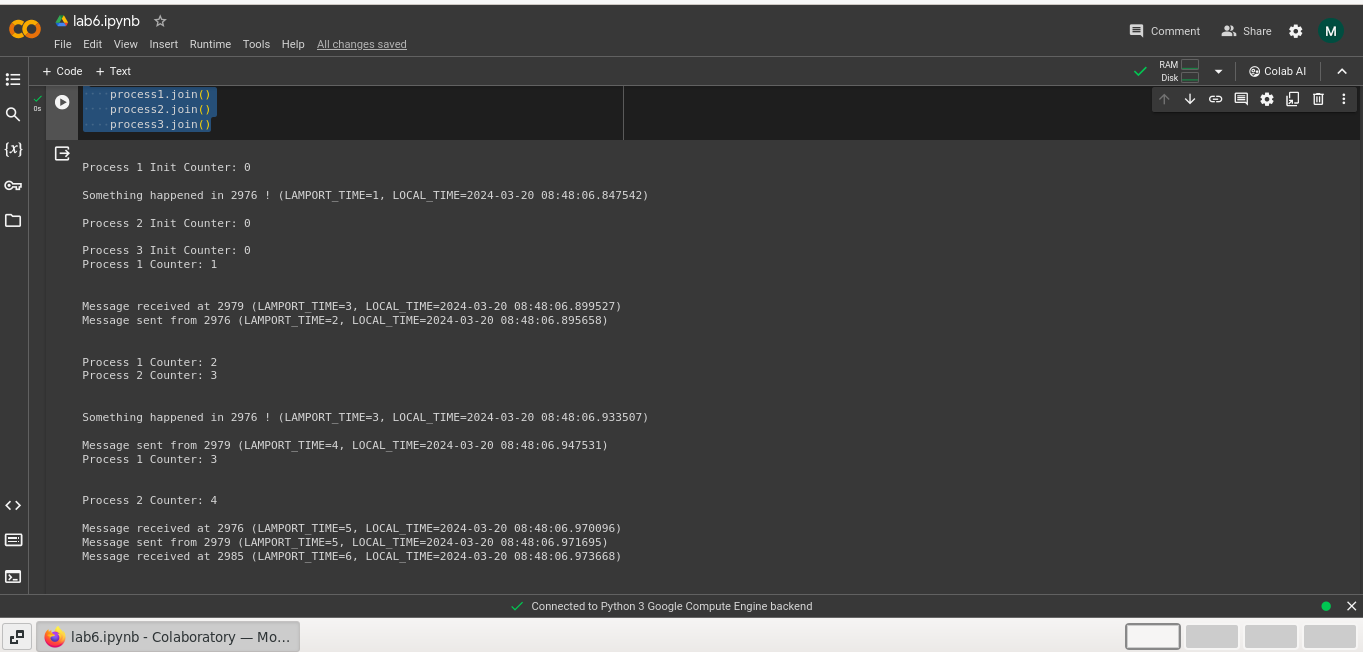
process3.start()

process1.join()

process2.join()

process3.join()

Output:



Working of the code:

This code simulates a distributed system where multiple processes communicate with each other using message passing. Each process runs independently and has its own logical clock (Lamport logical clock) to timestamp events. The processes communicate via pipes.

1. Define Helper Functions:

- "local\_time(counter)": Returns a formatted string containing the Lamport time and local time.

- "calc\_recv\_timestamp(recv\_time\_stamp, counter)": Calculates the timestamp for received messages.

- "event(pid, counter)": Simulates an event happening in a process. Increments the logical clock and prints the event details.

- "send\_message(pipe, pid, counter)": Sends a message through the pipe to another process. Increments the logical clock and prints the sending details.

- "recv\_message(pipe, pid, counter)": Receives a message from another process through the pipe. Adjusts the logical clock according to the received timestamp and prints the receiving details.

2. Define Process Functions:

- "process\_one", "process\_two", and "process\_three" are functions representing different processes. They simulate events, sending and receiving messages.

3. Main Execution:

- Pipes ("oneandtwo", "twoandthree", "threeandtwo") are created for communication between processes.

- Three processes ("process1", "process2", "process3") are created using "Process" from the "multiprocessing" module, each targeting its respective function.

- Processes are started using "start()" method.

- "join()" is called on each process to wait for them to finish execution.

4. Execution Flow:

- Each process executes a series of events, including sending and receiving messages, and incrementing its logical clock accordingly.

- Events and message exchanges are printed with timestamps indicating Lamport time and local time.

5. Output:

- The output shows the sequence of events, message sends, and receives, along with the respective Lamport timestamps and local times.

Overall, this code demonstrates how distributed processes can interact through message passing while maintaining logical clocks to establish a causal ordering of events.

**Conclusions :**

1. Learned 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).

**Post Lab Questions:**

**1. Distinguish between physical clock and logical clock synchronization**

| **Aspect** | **Physical Clock Synchronization** | **Logical Clock Synchronization** |
| --- | --- | --- |
| **Basis** | **Real time sources** | **Event ordering and causality** |
| **Objective** | **Achieve accurate real-time** | **Establish partial event ordering** |
| **Time Source** | **External (e.g., atomic clocks, GPS)** | **Internal logical rules (e.g., Lamport Clocks)** |
| **Accuracy vs. Precision** | **Focuses on both accuracy and precision** | **Focuses on ordering events causally** |
| **Dependency on Network** | **Often requires network communication** | **Can work independently of network** |
| **Challenges** | **Network latency, clock drift, varying clock speeds** | **Determining causal relationships, handling concurrent events** |
| **Examples** | **Network Time Protocol (NTP), Precision Time Protocol (PTP)** | **Lamport Clocks, Vector Clocks** |

**2. Show the calculation of the time interval between two synchronizations of a physical**

**Clock**

If T1 is 10:00:00 and T2 is 10:00:10, then the time interval between the two synchronization events is:

Time Interval=10:00:10−10:00:00=10 seconds

So, the time interval between the two synchronization events is 10 seconds.

**3. How will you implement Logical clocks by using counters?**

class LogicalClock:

def \_\_init\_\_(self):

self.counter = 0

def increment(self):

self.counter += 1

def update(self, received\_counter):

self.counter = max(self.counter, received\_counter) + 1

def get\_time(self):

return self.counter

def event(process\_clock):

process\_clock.increment()

def send\_message(sender\_clock, receiver\_clock):

sender\_clock.increment()

message = (sender\_clock.get\_time(), "Message data")

return message

def receive\_message(sender\_clock, receiver\_clock, message):

sender\_counter, data = message

receiver\_clock.update(sender\_counter)

print("Received message with data:", data, "at logical time:", receiver\_clock.get\_time())

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

process1\_clock = LogicalClock()

process2\_clock = LogicalClock()

event(process1\_clock)

event(process1\_clock)

event(process2\_clock)

message = send\_message(process1\_clock, process2\_clock)

receive\_message(process1\_clock, process2\_clock, message)

event(process1\_clock)

event(process2\_clock)

message = send\_message(process2\_clock, process1\_clock)

receive\_message(process2\_clock, process1\_clock, message)

In this implementation:

* Each process is associated with its own LogicalClock object.
* Events within each process are recorded by calling the increment() method of the corresponding clock.
* When a message is sent, the sender includes its current logical time (counter value) in the message.
* Upon receiving a message, the receiver updates its logical time using the update() method based on the received counter value.
* The logical time of each process can be obtained using the get\_time() method of the corresponding clock.

**4. Give an example of partial and total ordering of events**

Suppose we have three processes: Process A, Process B, and Process C.

1. Partial Ordering of Events:

In partial ordering, events are ordered based on causality, but concurrent events may not have a defined order relative to each other.

- Event 1: Process A sends a message to Process B.

- Event 2: Process B receives the message from Process A.

- Event 3: Process C sends a message to Process B.

- Event 4: Process B receives the message from Process C.

- Event 5: Process A sends another message to Process B.

In this scenario, Event 1 causally precedes Event 2, Event 3 causally precedes Event 4, but Events 2 and 3 are concurrent. Therefore, there's a partial ordering: ( {Event 1} to {Event 2} ), ( {Event 3} to {Event 4} ), but no ordering between Events 2 and 3.

2. Total Ordering of Events:

In total ordering, all events are ordered relative to each other, even if they occur concurrently.

- Event 1: Process A sends a message to Process B.

- Event 2: Process B receives the message from Process A.

- Event 3: Process C sends a message to Process B.

- Event 4: Process B receives the message from Process C.

- Event 5: Process A sends another message to Process B.

To establish total ordering, we can use Lamport timestamps or vector clocks. Let's assume we use Lamport timestamps.

The Lamport timestamps for these events might be:

- Event 1: Lamport timestamp (A) = 1

- Event 2: Lamport timestamp (B) = 2

- Event 3: Lamport timestamp (C) = 3

- Event 4: Lamport timestamp (B) = 4

- Event 5: Lamport timestamp (A) = 5

In total ordering, events are ordered based on their Lamport timestamps:

[ {Event 1} to {Event 2} to {Event 3} to {Event 4} to {Event 5} ]

Here, even though Events 2 and 3 are concurrent, we establish a total order by comparing their Lamport timestamps.