**PROGRAM-3**

**Aim:**

Lamport logical clock synchronization between processes with different clocks and update intervals. The processes must exchange messages and correct clocks if required. After each interaction the process must show the clock after updating if required.

**Theory:**

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 in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender (causality).

**Algorithm:**

* All the process counters start with value 0.
* A process increments its counter for each event (internal event, message sending, message receiving) in that process.
* When a process sends a message, it includes its (incremented) counter value with the message.
* 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.

**Code:**

import signal

import sys

import time

import threading

from queue import Queue

initially\_granted\_proc = "A"

procs = {"A", "B", "C"}

resource\_usage\_counts = {"A": 0, "B": 0, "C": 0}

message\_queues = {"A": Queue(), "B": Queue(), "C": Queue()}

class Message(object):

def \_\_init\_\_(self, msg\_type, timestamp, sender, receiver):

self.msg\_type = msg\_type

self.timestamp = timestamp

self.sender = sender

self.receiver = receiver

def \_\_repr\_\_(self):

return "Message {} at {} from {} to {}".format(

self.msg\_type, self.timestamp,

self.sender, self.receiver)

class Process(threading.Thread):

def \_\_init\_\_(self, name, initially\_granted, other\_processes):

super(Process, self).\_\_init\_\_()

self.name = name

self.has\_resource = initially\_granted == name

self.other\_processes = other\_processes

self.lamport\_clock = 0 # tick after each "event"

self.request\_queue = []

self.requested = False

self.request\_queue.append(Message("request",

-1, initially\_granted, initially\_granted))

def remove\_request(self, msg\_type, sender):

index\_of\_req = -1

for i in range(len(self.request\_queue)):

if self.request\_queue[i].msg\_type == msg\_type and \

self.request\_queue[i].sender == sender:

index\_of\_req = i

break

if i == -1:

print("Unable to remove")

else:

del self.request\_queue[i]

def use\_resource(self):

print("Process {} is using resource".format(self.name))

resource\_usage\_counts[self.name] += 1

time.sleep(2)

def process\_message(self, msg):

# Based on msg\_type handle appropriately

if msg.msg\_type == "request":

# Put in our request queue and send an ack

# to the sender

self.request\_queue.append(msg)

for proc in self.other\_processes:

if proc == msg.sender:

message\_queues[proc].put(Message(

"ack", self.lamport\_clock,

self.name, msg.sender))

elif msg.msg\_type == "release":

# Got a release, remove it from our queue

self.remove\_request("request", msg.sender)

elif msg.msg\_type == "ack":

pass

else:

print("Unknown message type")

def run(self):

while True:

if self.has\_resource:

self.use\_resource()

self.remove\_request("request", self.name)

# Tell everyone that we are done

for proc in self.other\_processes:

message\_queues[proc].put(Message(

"release", self.lamport\_clock,

self.name, proc))

self.lamport\_clock += 1

self.has\_resource, self.requested = False, False

continue

# Want to get the resource

if not self.requested:

# Request it

print("Process {} requesting resource".format(

self.name))

self.request\_queue.append(Message(

"request", self.lamport\_clock,

self.name, self.name))

# Broadcast this request

for proc in self.other\_processes:

message\_queues[proc].put(Message(

"request", self.lamport\_clock,

self.name, proc))

self.lamport\_clock += 1

self.requested = True

else:

# Just wait until it is available by processing messages

print("Process {} waiting for message".format(self.name))

msg = message\_queues[self.name].get(block=True)

# Got a message, check if the timestamp

# is greater than our clock, if so advance it

if msg.timestamp >= self.lamport\_clock:

self.lamport\_clock = msg.timestamp + 1

print("Got message {}".format(msg))

self.process\_message(msg)

self.lamport\_clock += 1

# Check after processing if the resource is

# available for me now, if so, grab it.

# We need earliest request to be ours and check that we

# have received an older message from everyone else

if self.check\_available():

print("Resource available for {}".format(self.name))

self.has\_resource = True

print("Process {}: {}".format(self.name, self.request\_queue))

print("Process {} Clock: {}".format(self.name, self.lamport\_clock))

time.sleep(1)

def check\_available(self):

got\_older = {k: False for k in self.other\_processes}

# Get timestamp of our req

our\_req = None

for req in self.request\_queue:

if req.sender == self.name:

our\_req = req

if our\_req is None:

return False

# We found our req make sure it is younger than

# all the others and we have an older one from

# the other guys

for req in self.request\_queue:

if req.sender in got\_older and req.timestamp > our\_req.timestamp:

got\_older[req.sender] = True

if all(got\_older.values()):

return True

return False

t1 = Process("A", initially\_granted\_proc, list(procs - set("A")))

t2 = Process("B", initially\_granted\_proc, list(procs - set("B")))

t3 = Process("C", initially\_granted\_proc, list(procs - set("C")))

# Daemonizing threads means that if main thread dies, so do they.

# That way the process will exit if the main thread is killed.

t1.setDaemon(True)

t2.setDaemon(True)

t3.setDaemon(True)

try:

t1.start()

t2.start()

t3.start()

while True:

# Need some arbitrary timeout here, seems a bit hackish.

# If we don't do this then the main thread will just block

# forever waiting for the threads to return and the

# keyboardinterrupt never gets hit. Interestingly regardless of the

# timeout, the keyboard interrupt still occurs immediately

# upon ctrl-c'ing

t1.join(100)

t2.join(100)

t3.join(100)

except KeyboardInterrupt:

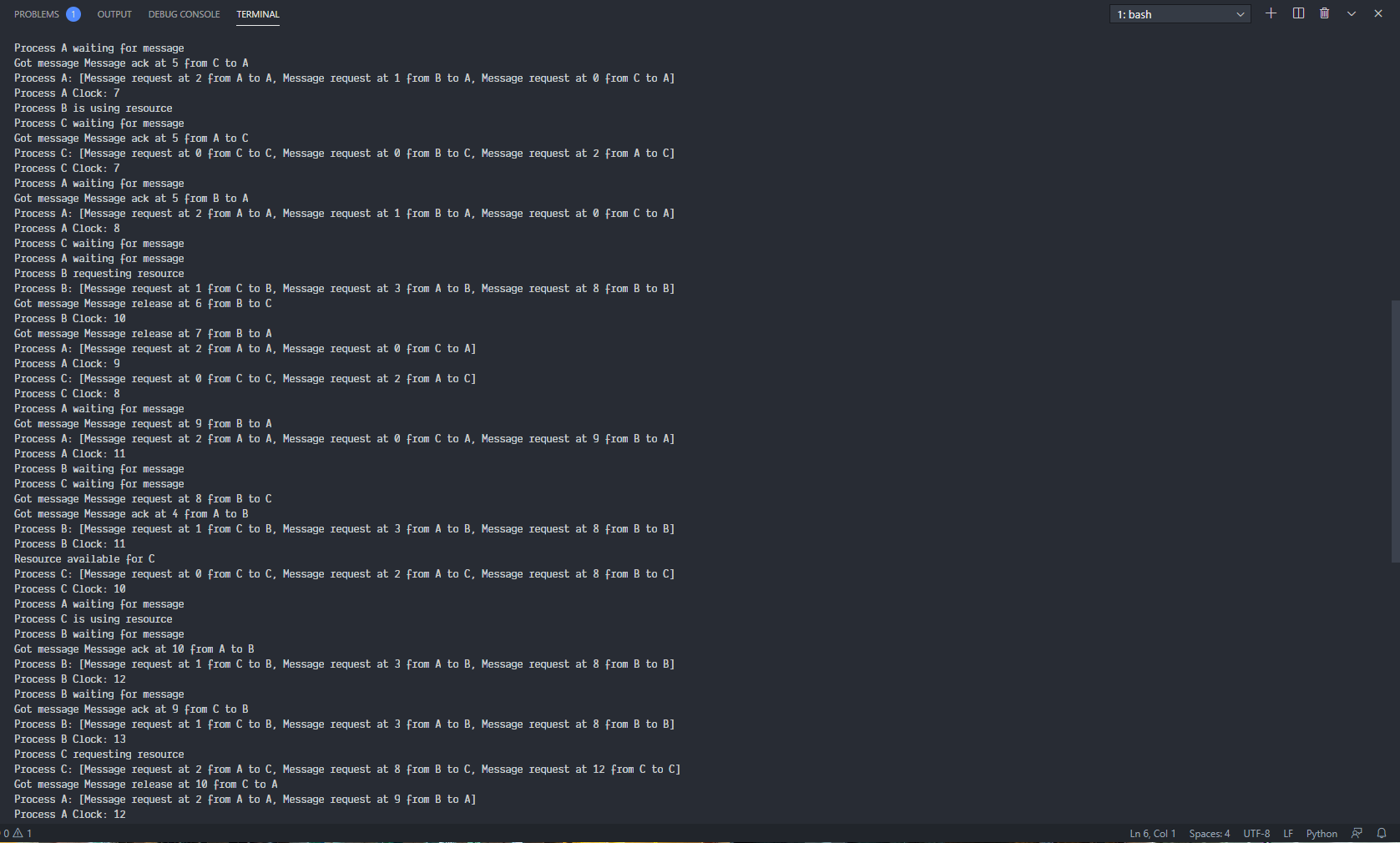
print("Ctrl-c pressed")

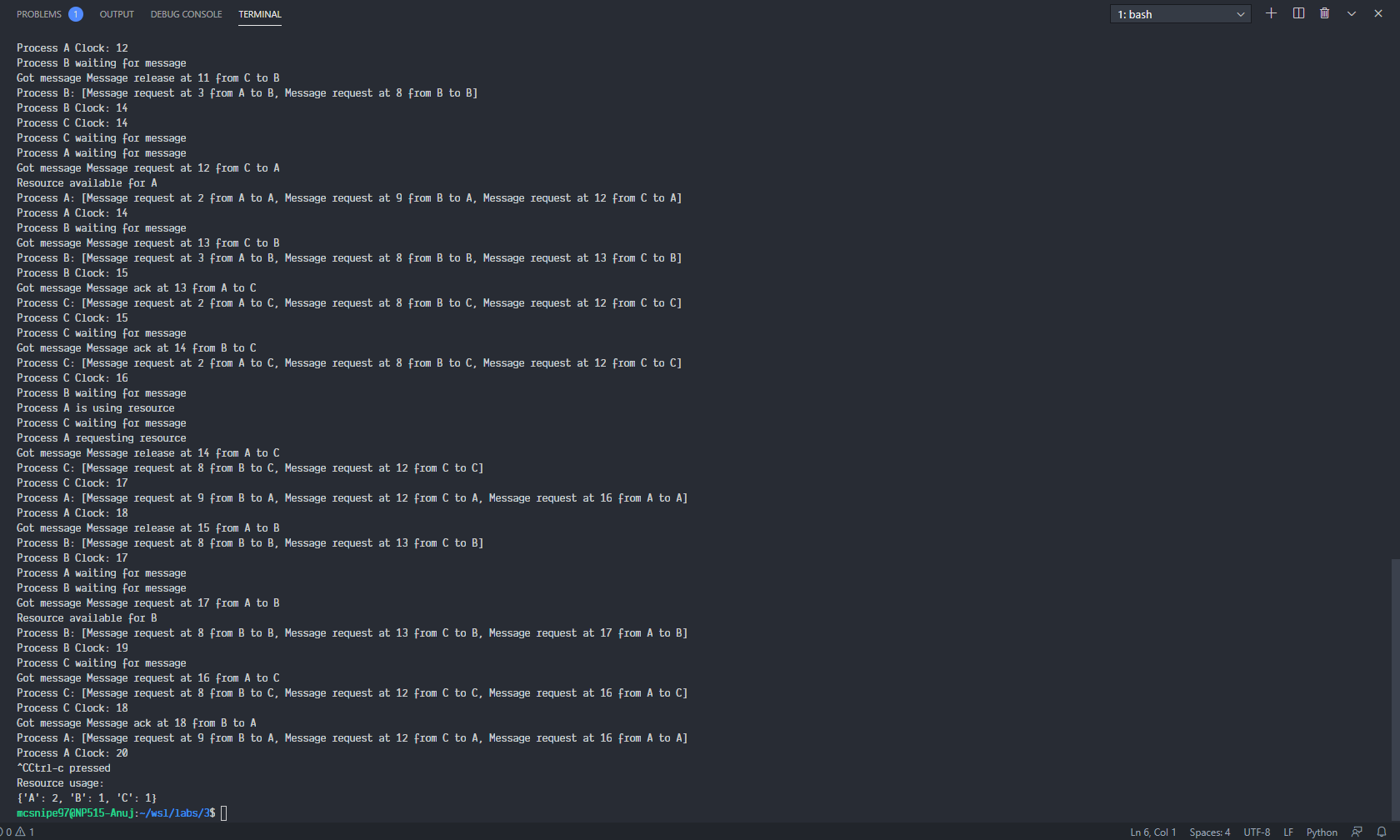
print("Resource usage:")

print(resource\_usage\_counts)

sys.exit(1)

**Output:**

****

****

**Conclusion:**

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. We successfully implemented Lamport Clock.