7.DEADLOCK

**Code and output**:

class Process:

    def \_\_init\_\_(self, pid):

        self.pid = pid  # Process ID

        self.waiting\_for = []  # List of processes it is waiting for

    def request\_resource(self, resource\_holders, initiator):

        """Initiates a probe if the resource request fails."""

        if not resource\_holders:

            print(f"Process {self.pid} found no resource holders.")

            return False  # No resource holders, no deadlock

        for holder in resource\_holders:

            probe = [initiator, self.pid, holder.pid]  # Probe message

            print(f"Process {self.pid} sending probe {probe} to {holder.pid}")

            if holder.receive\_probe(probe):

                print(f"Deadlock detected involving process {initiator}!")

                return True

        return False

    def receive\_probe(self, probe):

        """Handles received probe messages."""

        initiator, sender, receiver = probe

        print(f"Process {self.pid} received probe {probe}")

        if self.pid == initiator:  # Cycle detected

            print(f"Cycle detected! Process {self.pid} is in a deadlock.")

            return True

        if not self.waiting\_for:  # Not waiting for any process

            print(f"Process {self.pid} is not waiting for any resource. Ignoring probe.")

            return False  # No deadlock

        for next\_holder in self.waiting\_for:

            new\_probe = [initiator, self.pid, next\_holder.pid]  # Modify probe

            print(f"Process {self.pid} forwarding probe {new\_probe} to {next\_holder.pid}")

            if next\_holder.receive\_probe(new\_probe):

                return True

        return False

# Example Usage

print("--- Deadlock Example ---")

p1 = Process(1)

p2 = Process(2)

p3 = Process(3)

p4 = Process(4)

# Defining dependencies (waits-for relationships) - Deadlock case

p1.waiting\_for = [p2]

p2.waiting\_for = [p3]

p3.waiting\_for = [p4]

p4.waiting\_for = [p1]  # Creates a cycle (deadlock)

# Initiate probe from p1

if p1.request\_resource([p2], p1.pid):

    print("Deadlock confirmed.")

else:

    print("No deadlock detected.")

print("\n--- No Deadlock Example ---")

p5 = Process(5)

p6 = Process(6)

p7 = Process(7)

p8 = Process(8)

# Defining dependencies - No cycle

p5.waiting\_for = [p6]

p6.waiting\_for = [p7]

p7.waiting\_for = [p8]

p8.waiting\_for = []  # No cycle

# Initiate probe from p5

if p5.request\_resource([p6], p5.pid):

    print("Deadlock confirmed.")

else:

    print("No deadlock detected.")

5.RING

**Code and output**:

class RingProcess:

    def \_\_init\_\_(self, id):

        self.id = id

        self.coordinator = None

        self.next = None

    def start\_election(self):

        print(f"Process {self.id} starts an election.")

        active\_list = [self.id]

        self.send\_election\_message(active\_list)

    def send\_election\_message(self, active\_list):

        if self.next:

            print(f"Process {self.id} forwards election list {active\_list}.")

            self.next.receive\_election\_message(active\_list)

    def receive\_election\_message(self, active\_list):

        if self.id not in active\_list:

            active\_list.append(self.id)

            self.send\_election\_message(active\_list)

        else:

            new\_coordinator = max(active\_list)

            print(f"Process {self.id} elects {new\_coordinator} as the coordinator.")

            self.send\_coordinator\_message(new\_coordinator)

    def send\_coordinator\_message(self, new\_coordinator):

        if self.next:

            print(f"Process {self.id} informs that {new\_coordinator} is the new coordinator.")

            self.coordinator = new\_coordinator

            self.next.receive\_coordinator\_message(new\_coordinator)

    def receive\_coordinator\_message(self, new\_coordinator):

        if self.coordinator is None:

            self.coordinator = new\_coordinator

            self.send\_coordinator\_message(new\_coordinator)

processes = [RingProcess(i) for i in range(1, 6)]

for i in range(len(processes)):

    processes[i].next = processes[(i + 1) % len(processes)]

processes[1].start\_election()

4.LAMPORTS

**Code and output**:

class LamportClock:

    def \_\_init\_\_(self, process\_id):

        self.process\_id = process\_id

        self.clock = 0  # Initialize Lamport clock

    def send\_request(self):

        """Increment clock and return the timestamp with request"""

        self.clock += 1

        print(f"[Process {self.process\_id}] Sent request with timestamp {self.clock}")

        return self.clock

    def receive\_request(self, timestamp):

        """Update clock on receiving a request"""

        self.clock = max(self.clock, timestamp) + 1

        print(f"[Process {self.process\_id}] Received request (timestamp {timestamp}) → Updated clock to {self.clock}")

    def internal\_event(self):

        """Increment clock for an internal event"""

        self.clock += 1

        print(f"[Process {self.process\_id}] Internal event → Timestamp updated to {self.clock}")

# Simulating interactions between three processes

def simulate():

    process1 = LamportClock(1)

    process2 = LamportClock(2)

    process3 = LamportClock(3)

    print("\n--- Simulation Start ---\n")

    # Process 1: Internal event, then sends requests

    process1.internal\_event()

    timestamp1 = process1.send\_request()  # Sent to Process 2

    timestamp2 = process1.send\_request()  # Sent to Process 3

    # Process 2: Internal event, receives request from P1, sends request to P3

    process2.internal\_event()

    process2.receive\_request(timestamp1)

    timestamp3 = process2.send\_request()  # Sent to Process 3

    # Process 3: Internal event, receives requests from P1 & P2

    process3.internal\_event()

    process3.receive\_request(timestamp2)  # From P1

    process3.receive\_request(timestamp3)  # From P2

    # Process 1 & 2 receive responses from P3

    process1.receive\_request(process3.clock)

    process2.receive\_request(process3.clock)

    print("\n--- Simulation End ---\n")

# Run simulation

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

    simulate()

3.COMMUNICATION

import os

import multiprocessing

def child\_process(pipe\_write):

    os.close(pipe\_write[0])  # Close unused read end

    message = "Hello from child process!"

    os.write(pipe\_write[1], message.encode())  # Write message to pipe

    os.close(pipe\_write[1])

def parent\_process():

    pipe\_read, pipe\_write = os.pipe()  # Create pipe

    # Create child process (pass pipe\_write as a tuple, not a list)

    process = multiprocessing.Process(target=child\_process, args=((pipe\_read, pipe\_write),))

    process.start()

    os.close(pipe\_write)  # Close unused write end

    # Read message from child process

    message = os.read(pipe\_read, 1024).decode()

    print(f"Parent received: {message}")

    os.close(pipe\_read)

    process.join()

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

    parent\_process()

import os

import multiprocessing

def child\_process(pipe\_write):

    os.close(pipe\_write[0])  # Close unused read end

    message = "Hello from child process!"

    os.write(pipe\_write[1], message.encode())  # Write message to pipe

    os.close(pipe\_write[1])

def parent\_process():

    pipe\_read, pipe\_write = os.pipe()  # Create pipe

    # Create child process (pass pipe\_write as a tuple, not a list)

    process = multiprocessing.Process(target=child\_process, args=((pipe\_read, pipe\_write),))

    process.start()

    os.close(pipe\_write)  # Close unused write end

    # Read message from child process

    message = os.read(pipe\_read, 1024).decode()

    print(f"Parent received: {message}")

    os.close(pipe\_read)

    process.join()

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

    parent\_process()

1.CLIENT-SERVER SOCKET PROGRAMMING