DEADLOCK:

- **Class Definition: Process**

class Process:

This defines a Process class to represent each process in the system.

- **Constructor: __init__**

```
  def __init__(self, pid):
      self.pid = pid
      self.waiting_for = []
```

- pid: The unique process ID.
- waiting_for: A list of Process instances that this process is waiting for (i.e., dependencies in the wait-for graph).

- **Method: request_resource**

```
  def request_resource(self, holders, initiator):
```

- Called by the initiator process to start the deadlock detection.
- holders: The list of processes that this process is waiting for.
- initiator: ID of the process that initiates the detection (important for cycle detection).

```
      if not holders:
          print(f"Process {self.pid} found no holders.")
          return False
```

- If the process is not waiting for any other process, it cannot be in a deadlock.

```
      for h in holders:
          probe = [initiator, self.pid, h.pid]
          print(f"{self.pid} → sending probe {probe} to {h.pid}")
```

- Sends a probe [initiator, sender, receiver] to each process it is waiting for.
- The probe tracks who initiated it, who is sending, and who it's being sent to.

```
    if h.receive_probe(probe):
        print(f"Deadlock detected by process {initiator}!")
        return True
```

- The recipient of the probe (h) calls receive_probe.
- If it returns True, a deadlock is detected.
- Detection halts once a cycle is found.

```
return False
```

- If no cycles found, return False (no deadlock).

🔹 **Method: receive_probe**

```
def receive_probe(self, probe):
    initiator, sender, receiver = probe
    print(f"{self.pid} ← received probe {probe}")
```

- Unpacks the probe message.
- Logs reception.

```
    if self.pid == initiator:
        print(f"Cycle found at {self.pid} → DEADLOCK!")
        return True
```

- If the current process is the initiator, it means the probe has returned → **Cycle detected**.

```
    if not self.waiting_for:
        print(f"{self.pid} waits for no one. Ignoring.")
        return False
```

- If current process is not waiting on anyone, it ends the chain. No deadlock from this path.

```python
    for nxt in self.waiting_for:
        new_probe = [initiator, self.pid, nxt.pid]
        print(f"{self.pid} → forwarding probe {new_probe} to {nxt.pid}")
        if nxt.receive_probe(new_probe):
            return True
```

- Forward the probe to all processes it is waiting for.
- Recursive check continues.
- If any of them finds a cycle, returns True.

```python
    return False
```

- If none leads to a cycle, no deadlock in this path.

🔹 **Function: build_graph**

```python
def build_graph():
    n = int(input("Number of processes: "))
    processes = {i: Process(i) for i in range(1, n + 1)}
```

- Takes user input for number of processes.
- Creates a dictionary of Process instances, keyed by PID.

```python
    print("Enter dependencies (waiting_for), e.g., '1 2' means P1 waits for P2.")
    print("Type 'done' when finished.\n")
```

- Instruction to enter edges of the wait-for graph.

```python
    while True:
        inp = input(">> ")
        if inp == "done":
            break
        try:
            p1, p2 = map(int, inp.split())
```

processes[p1].waiting_for.append(processes[p2])

except:

print("Invalid input. Try again.")

- Continues reading input until "done" is typed.
- Adds an edge p1 → p2, meaning process p1 is waiting for p2.

return processes

- Returns the complete graph of processes.

◆ **Function: run_detection**

def run_detection():

processes = build_graph()

start = int(input("\nEnter initiator process ID: "))

- Builds the graph and asks user for initiator process ID.

if start in processes:

if processes[start].request_resource(processes[start].waiting_for, start):

print("Deadlock confirmed.")

else:

print("No deadlock detected.")

else:

print("Invalid process ID.")

- Starts the deadlock detection from the initiator.
- Calls request_resource to send initial probes.
- Prints result based on return value.

◆ **Main Entry Point**

if __name__ == "__main__":

run_detection()

- Ensures that run_detection() is only executed when script is run directly, not when imported.

🔹 Sample Execution Explained

Input:

Number of processes: 4

>> 1 2

>> 2 3

>> 3 4

>> 4 1

>> done

Enter initiator process ID: 1

- Processes:
  - P1 waits for P2
  - P2 waits for P3
  - P3 waits for P4
  - P4 waits for P1 → **Cycle formed: 1 → 2 → 3 → 4 → 1**

Output:

1 → sending probe [1, 1, 2] to 2

2 ← received probe [1, 1, 2]

2 → forwarding probe [1, 2, 3] to 3

3 ← received probe [1, 2, 3]

3 → forwarding probe [1, 3, 4] to 4

4 ← received probe [1, 3, 4]

4 → forwarding probe [1, 4, 1] to 1

1 ← received probe [1, 4, 1]

Cycle found at 1 → DEADLOCK!

Deadlock detected by process 1!

Deadlock confirmed.

The probe returns to the initiator (1), indicating a **cycle in the wait-for graph**, which is the definition of a deadlock.

🔶 CLASS DEFINITION — LamportClock

class LamportClock:

Defines a class to simulate the **Lamport Logical Clock** for a process in a distributed system. Each object of this class represents **one process**.

🔹 __init__() — Constructor

```
def __init__(self, process_id):
    self.process_id = process_id
    self.clock = 0
```

- process_id: A unique ID to identify this process (like 1, 2, 3).
- self.clock = 0: Initializes the Lamport clock value to 0, meaning the process has not done any events yet.

🔹 send_request()

```
def send_request(self):
    self.clock += 1
```

- Simulates a **send event**.
- Before sending, it increments the clock (+1) because it's a new event.

```
    print(f"[Process {self.process_id}] Sent request with timestamp {self.clock}")
```

- Logs the message send action and the current timestamp.

return self.clock

- Returns the current timestamp, which will be "attached" to the message.
- This value is important when the **receiver** adjusts its clock.

🔹 receive_request(timestamp)

def receive_request(self, timestamp):

- Called when this process receives a message from another process.
- timestamp: The Lamport timestamp that was attached to the received message.

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

- Applies the Lamport Clock rule:
  Update local clock to **max(local clock, received clock) + 1**
  This ensures that the receiving event **happens after** the sending event.

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

- Logs the update, showing how the process's clock is synchronized.

🔹 internal_event()

def internal_event(self):

self.clock += 1

- Simulates a local event inside the process (not related to sending or receiving messages).
- According to Lamport rules, even internal events increment the logical clock.

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

- Logs the internal event and current timestamp.

🔸 HELPER FUNCTION — get_process()

def get_process(processes, pid):

- Retrieves or creates a process by its ID.
- processes: Dictionary of all LamportClock instances.

if pid not in processes:

processes[pid] = LamportClock(pid)

- If the process does not already exist, create it using the constructor.

return processes[pid]

- Return the LamportClock instance for the given process ID.

🔸 MAIN SIMULATION FUNCTION — simulate_with_input()

def simulate_with_input():

- This function runs an **interactive CLI simulation** where users control process events.

🔹 Setup

processes = {}

- Dictionary to store all processes created during simulation.

sent_messages = {}

- Dictionary to map message names (e.g., "msgA") to their sent timestamps.
- Helps track the timestamp that will be used when another process receives the message.

🔹 Display Commands

print("\n--- Lamport Clock Simulation (User Input) ---\n")

print("Commands:")

print(" internal <pid>")

print(" send <sender_pid> <message_name>")

print(" receive <receiver_pid> <sender_pid> <message_name>")

print(" show")

```
print("  exit\n")
```

- Shows instructions to the user so they know how to interact with the simulation.

◆ **Main Loop**

```
while True:
    command = input("Enter command: ").strip().split()
```

- Waits for user input.
- strip() removes any extra spaces; split() breaks input into command parts.

```
    if not command:
        continue
```

- Skip if user presses enter with no command.

◆ **COMMANDS**

◆ **INTERNAL EVENT**

```
    if action == "internal" and len(command) == 2:
        pid = int(command[1])
        proc = get_process(processes, pid)
        proc.internal_event()
```

- Executes an **internal event** for the specified process ID.
- Uses get_process to get or create the process.
- Calls internal_event() on that process.

◆ **SEND MESSAGE**

```
    elif action == "send" and len(command) == 3:
        pid = int(command[1])
```

```
    msg_name = command[2]
    proc = get_process(processes, pid)
    sent_messages[msg_name] = proc.send_request()
```

- Simulates sending a message:
  - msg_name: A unique name like msgA, msgB.
  - The process sends the message and returns the timestamp.
  - That timestamp is stored in sent_messages using msg_name as the key.

◆ **RECEIVE MESSAGE**

```
elif action == "receive" and len(command) == 4:
    receiver_pid = int(command[1])
    sender_pid = int(command[2])
    msg_name = command[3]
```

- Sets up parameters for receiving:
  - receiver_pid: Process receiving the message.
  - sender_pid: Process that originally sent it.
  - msg_name: The message name.

```
    if msg_name not in sent_messages:
        print(f"Error: Message '{msg_name}' not found.")
        continue
```

- Checks if the message actually exists.
- If not found, prints error and skips.

```
    proc = get_process(processes, receiver_pid)
    proc.receive_request(sent_messages[msg_name])
```

- Retrieves or creates the receiver process.
- Calls receive_request() with the stored timestamp for that message.

◆ **SHOW CLOCK VALUES**

```
elif action == "show":
    for pid in sorted(processes):
        print(f"Process {pid}: Clock = {processes[pid].clock}")
```

- Displays current clock values for all processes in sorted order of pid.

◆ **EXIT SIMULATION**

```
elif action == "exit":
    print("\n--- Simulation Ended ---")
    break
```

- Ends the simulation loop.

◆ **HANDLE INVALID COMMANDS**

```
else:
    print("Invalid command. Try again.")
```

- Handles all incorrect or invalid command patterns.

◆ **SCRIPT ENTRY POINT**

```
if __name__ == "__main__":
    simulate_with_input()
```

- Standard Python entry point.
- Ensures simulate_with_input() only runs when the script is executed directly, not when imported.

PIPES:

🔸 **Import the Module**

import multiprocessing

- Imports the multiprocessing module, which allows the creation of multiple processes that can run concurrently.
- Also includes tools like Process, Pipe, Queue, etc.

🔸 **Define the Child Process Function**

def child_process(conn):

- Defines a function to be run in the **child process**.
- conn is one end of a pipe (child_conn), passed to the child for communication.

message = conn.recv() # Receive message from parent

- Waits to receive a message using the pipe.
- recv() is a **blocking call**, meaning it waits until a message arrives.

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

- Displays the received message from the parent.

conn.close()

- Closes the pipe connection from the child's side to free up system resources.

🔸 **Define the Parent Process Logic**

def parent_process():

- This function will create and manage the child process.

🔹 **Create a Pipe**

parent_conn, child_conn = multiprocessing.Pipe()

- Creates a **two-way communication pipe** between parent and child.
- parent_conn is for the parent process.
- child_conn is for the child process.

🔹 Create and Start Child Process

process = multiprocessing.Process(target=child_process, args=(child_conn,))

- Creates a child process that will run child_process() function.
- Passes child_conn to the child so it can receive messages.

process.start()

- Starts the child process.

🔹 Close Child End in Parent

child_conn.close() # Close child end in parent

- Since the **parent only needs to send**, it closes the child's end of the pipe to avoid accidental use.

🔹 Parent Sends Message

message = input("Parent: Enter a message to send to the child: ")

- Takes input from the user in the parent process.

parent_conn.send(message) # Send message to child

- Sends the input message to the child process through the pipe.

parent_conn.close()

- After sending, the parent closes its end of the pipe.

🔹 Wait for Child to Finish

process.join()

- The parent process **waits** for the child process to complete.
- Prevents the parent from exiting before the child finishes its execution.

🔶 **Script Entry Point**

```
if __name__ == "__main__":
  parent_process()
```

- Ensures the parent process logic is only executed when this script is run **directly**, not when imported as a module.

✅ **Sample Output Breakdown**

Parent: Enter a message to send to the child: hello child

Child received: hello child

What happens:

1. You type a message in the parent (e.g., "hello child").
2. The parent sends it through the pipe.
3. The child process receives it via conn.recv() and prints it.

RING ELECTION ALGO:

🔶 **Class Definition**

class Process:

- **Defines a class named Process, representing a node in the ring.**

Constructor

```
  def __init__(self, id):
    self.id = id
```

self.active = True

self.coordinator = None

self.next = None

- id: Unique identifier for the process.
- active: Boolean to track whether the process is up or down.
- coordinator: Stores the ID of the currently known coordinator.
- next: A reference to the next process in the ring.

◆ Starting an Election

```
def start_election(self):
    if not self.active:
        print(f"Process {self.id} is down.")
        return
    print(f"Process {self.id} starts an election.")
    self.pass_election([self.id])
```

- If the process is active, it begins an election by calling pass_election(), passing its own ID in a list.

◆ Passing the Election Message

```
def pass_election(self, ids):
    if self.next.active:
```

- Checks if the next process in the ring is active.

```
        if self.next.id in ids:
```

- Checks for a cycle (i.e., election message came full circle).

```
            leader = max(ids)
            print(f"Process {self.id} elects {leader} as the coordinator.")
```

self.pass_coordinator(leader)

- The process with the highest ID is elected.
- pass_coordinator() informs everyone in the ring about the new coordinator.

else:

ids.append(self.next.id)

print(f"Process {self.id} forwards election list {ids}.")

self.next.pass_election(ids)

- If the next process hasn't seen the election yet, its ID is added, and the list is passed along.

else:

self.next.pass_election(ids)

- If the next process is down, skip to the next one that is up.

🔹 Informing About the Coordinator

def pass_coordinator(self, leader):

self.coordinator = leader

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

- Sets and announces the new coordinator to the current process.

if self.next.coordinator != leader:

self.next.pass_coordinator(leader)

- Passes coordinator info along the ring until all active nodes are updated.

🔸 Setup Ring Topology

def setup_ring(n):

plist = [Process(i) for i in range(1, n + 1)]

- Creates n Process objects with IDs from 1 to n.

```python
    for i in range(n):
        plist[i].next = plist[(i + 1) % n]
```

- Each process points to the next one in a circular fashion (ring).
- The % n ensures the last process connects to the first.

```python
    return plist
```

- Returns the ring of processes.

🔸 Main Simulation Function

```python
def run():
    n = int(input("Enter number of processes: "))
    ring = setup_ring(n)
```

- Takes user input for number of processes and initializes the ring.

🔹 Command Input Loop

```python
    while True:
        cmd = input(">> ").strip().split()
        if not cmd: continue
```

- Continuously waits for user commands. Skips if nothing entered.

🔹 Start Election

```python
        if cmd[0] == "start":
            ring[int(cmd[1]) - 1].start_election()
```

- Starts an election from the given process (adjusted to 0-based index).

🔹 Bring Down a Process

```python
        elif cmd[0] == "down":
```

```
        ring[int(cmd[1]) - 1].active = False
        print(f"Process {cmd[1]} is now down.")
```

- Sets the specified process as inactive.

◆ **Bring a Process Back Up**

```
    elif cmd[0] == "up":
        p = ring[int(cmd[1]) - 1]
        p.active, p.coordinator = True, None
        print(f"Process {cmd[1]} is back up.")
```

- Marks a process as active again and clears old coordinator data.

◆ **Show Status of All Processes**

```
    elif cmd[0] == "status":
        for p in ring:
            state = "UP" if p.active else "DOWN"
            coord = f" | Coordinator: {p.coordinator}" if p.coordinator else ""
            print(f"Process {p.id} is {state}{coord}")
```

- Prints whether each process is UP/DOWN, and who they know as the coordinator.

◆ **Exit the Simulation**

```
    elif cmd[0] == "exit":
        print("Exiting simulation.")
        break
```

- Exits the simulation loop.

◆ **Entry Point**

if __name__ == "__main__":

run()

- **Ensures run() is only called when the script is run directly.**

✅ **Sample Output Explanation**

>> down 5

Process 5 is now down.

- **Process 5 is marked as inactive.**

>> start 3

Process 3 starts an election.

Process 3 forwards election list [3, 4].

Process 5 forwards election list [3, 4, 1].

Process 1 forwards election list [3, 4, 1, 2].

Process 2 elects 4 as the coordinator.

- **Process 3 initiates election → goes through the ring → Process 2 sees it completed and elects 4 (highest ID).**

CLIENT AND SERVER:

✅ MyClientUser.java — Client Side Code

import java.io.*;

import java.net.*;

import java.util.Scanner;

- Imports:
    - java.io.*: For input/output operations (DataOutputStream, etc.).
    - java.net.*: For networking classes (Socket).

- java.util.Scanner: To read user input from the keyboard.

public class MyClientUser {

- Declares a public class named MyClientUser.

public static void main(String[] args) {

- The main method: starting point of the program.

try {

- Begin try block to handle exceptions related to I/O or networking.

Socket s = new Socket("localhost", 6666);

- Creates a socket connection to the server running on localhost (same computer) and port 6666.
- If server isn't running, this line will throw an exception.

DataOutputStream dout = new DataOutputStream(s.getOutputStream());

- Creates a stream to send data to the server using the socket's output stream.

Scanner scanner = new Scanner(System.in);

- Initializes a Scanner to read input from the keyboard.

System.out.print("Enter your message: ");

- Prompts the user to enter a message.

String message = scanner.nextLine();

- Reads the entire line input from the user.

dout.writeUTF(message);

- Sends the message to the server in UTF format.

dout.flush();

- Ensures all buffered data is sent out immediately.

dout.close();

s.close();

scanner.close();

- Closes the data stream, socket, and scanner to free up resources.

```
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

- Catches and prints any exception that occurs during connection or data transfer.

✅ MyServerUser.java — Server Side Code

```
import java.io.*;
import java.net.*;
```

- Imports:
  - For networking (ServerSocket, Socket) and I/O (DataInputStream).

```
public class MyServerUser {
```

- Declares the MyServerUser class.

```
  public static void main(String[] args){
```

- Main method to run the server.

```
    try{
```

- Begin try block to catch exceptions.

```
      ServerSocket ss = new ServerSocket(6666);
```

- Creates a ServerSocket bound to port 6666 to listen for client connections.

```
      Socket s = ss.accept(); //establishes connection
```

- Waits for a client to connect. Once a client connects, it returns a Socket object s.

```
      DataInputStream dis = new DataInputStream(s.getInputStream());
```

- Sets up an input stream to receive data from the client.

```
      String str = (String)dis.readUTF();
```

- Reads a UTF-encoded string sent by the client.

```
System.out.println("message= " + str);
```

- Prints the received message to the console.

```
ss.close();
```

- Closes the server socket.

```
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
```

- Catches and prints any exceptions during server operation.

✅ Working Summary

1. First, run MyServerUser. It waits for a client.
2. Then run MyClientUser, input a message, and it sends to the server.
3. Server displays the received message.