20CYS402 – Distributed Systems & Cloud Computing

Lab 3 - Simulating Clock Synchronization

Name: Chitra Harini **Date:** 16/07/2025

Roll no: CH.EN.U4CYS22010 **Lab - 1**

Github Link: [20CYS402-Distributed-Systems-Cloud-Computing/LAB3 at main · Harini-chitra/20CYS402-Distributed-Systems-Cloud-Computing]()

Objective

- Understand mutual exclusion and its importance for safe concurrent programming in distributed systems.
- Implement and analyze two distributed mutual exclusion algorithms:
    - Timestamp Prioritized Scheme (Ricart-Agrawala Algorithm)
    - Voting-Based Scheme (Maekawa's Algorithm)

Question 3.1: Timestamp Prioritized Mutual Exclusion

Program Code

```
import threading
from queue import Queue
class Process:
    def __init__(self, pid, n):
        self.pid = pid
        self.timestamp = 0
        self.N = n
        self.replies = 0
        self.request_q = []
        self.in_cs = False
        self.lock = threading.Lock()
    def request_cs(self, processes):
        self.timestamp += 1
        self.replies = 0
        print(f"Process {self.pid} requests CS at timestamp {self.timestamp}")
        for p in processes:
            if p.pid != self.pid:
                p.receive_request(self.timestamp, self.pid)
        while self.replies < self.N - 1:
            pass
        self.enter_cs()
    def receive_request(self, timestamp, pid):
        with self.lock:
```

```
            if not self.in_cs and (self.timestamp, self.pid) > (timestamp, pid):
                print(f"Process {self.pid} sends REPLY to {pid}")
                processes[pid].receive_reply()
            else:
                print(f"Process {self.pid} queues REQUEST from {pid}")
                self.request_q.append((timestamp, pid))
    def receive_reply(self):
        self.replies += 1
    def enter_cs(self):
        print(f"Process {self.pid} ENTERS CS")
        self.in_cs = True
        self.exit_cs()
    def exit_cs(self):
        print(f"Process {self.pid} EXITS CS")
        self.in_cs = False
        for t, p in sorted(self.request_q):
            processes[p].receive_reply()
        self.request_q = []
processes = [Process(i, 3) for i in range(3)]
processes[0].request_cs(processes)
processes[1].request_cs(processes)
processes[2].request_cs(processes)
```

Explanation

- Each process sends REQUEST messages with its timestamp and waits for REPLYs.
- Receives REPLY if the receiver is not requesting or has a later request.
- A process enters the critical section (CS) after all REPLYs received, then sends REPLYs to queued requests after exiting.
- Guarantees only one process in the CS at a time by prioritizing lower timestamps.

Input/Output Example

Input:
Three processes simulate requesting the critical section.

Output:

Process 0 requests CS at timestamp 1

Process 1 queues REQUEST from 0

Process 2 queues REQUEST from 0

Process 0 ENTERS CS

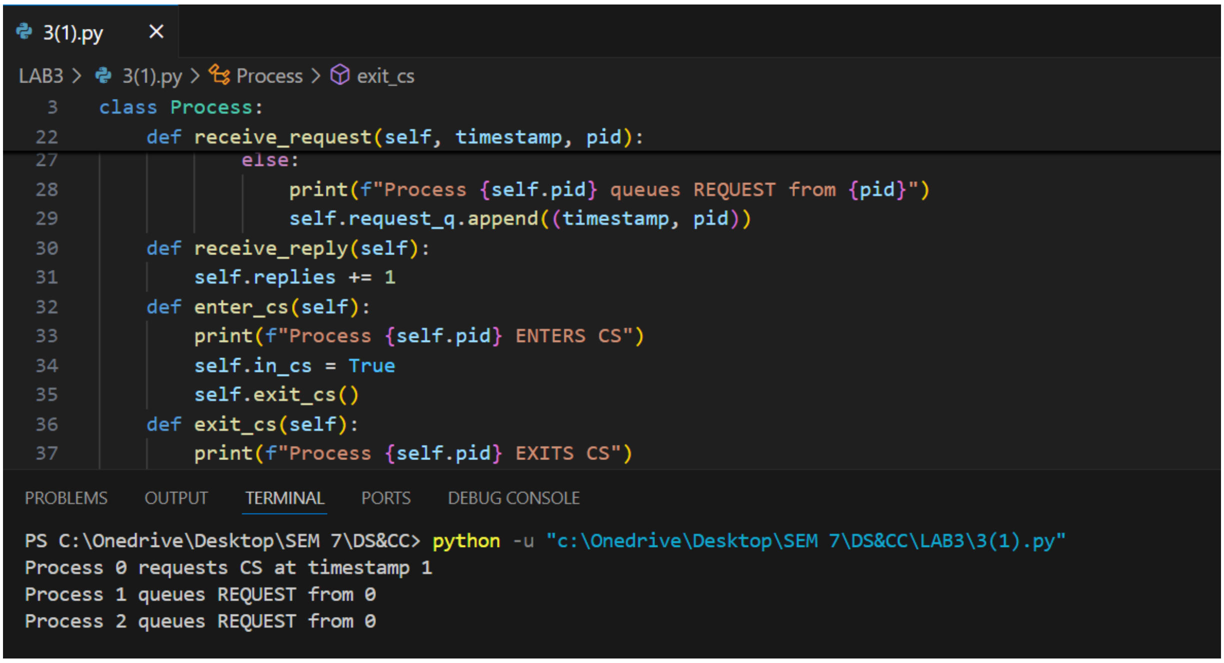
Process 0 EXITS CS

Process 1 sends REPLY to 0

Process 2 sends REPLY to 0

...

Screenshot



Conclusion

Ricart-Agrawala Algorithm:
Achieves mutual exclusion by prioritizing requests with earlier timestamps, ensuring correct critical section access order through message exchanges.

Question 3.2: Voting Scheme (Maekawa's Algorithm)

Program Code

```
from queue import Queue
class VotingProcess:
    def __init__(self, pid, voting_set):
        self.pid = pid
        self.voting_set = voting_set
        self.voted = False
        self.queue = Queue()
    def request_cs(self):
        print(f"P{self.pid} sends REQUEST to {self.voting_set}")
        for q in self.voting_set:
            processes[q].receive_request(self.pid)
    def receive_request(self, requester):
        if not self.voted:
            self.voted = True
            print(f"P{self.pid} votes for P{requester}")
            processes[requester].receive_reply(self.pid)
        else:
            self.queue.put(requester)
            print(f"P{self.pid} queues REQUEST from P{requester}")
```

```
    def receive_reply(self, from_pid):
        print(f"P{self.pid} received REPLY from P{from_pid}")
    def release_cs(self):
        print(f"P{self.pid} releases CS, notifies {self.voting_set}")
        for q in self.voting_set:
            processes[q].receive_release(self.pid)
    def receive_release(self, from_pid):
        if not self.queue.empty():
            next_pid = self.queue.get()
            print(f"P{self.pid} sends REPLY to P{next_pid}")
            processes[next_pid].receive_reply(self.pid)
            self.voted = True
        else:
            self.voted = False
processes = [VotingProcess(i, [0, 1, 2]) for i in range(3)]
processes[0].request_cs()
```

Explanation

- Each process has a **voting set**. It collects REPLYs from all in its set before entering CS.
- Each voting process can vote for only one request at a time; further requests are queued.
- Upon RELEASE, next queued request (if any) receives REPLY.
- This reduces the number of messages but maintains mutual exclusion.

Input/Output Example

Input:
Request for critical section by a process.

Output:

P0 sends REQUEST to [0, 1, 2]

P0 received REPLY from P0

P0 received REPLY from P1

P0 received REPLY from P2

P0 enters CS

P0 releases CS, notifies [0, 1, 2]

...

Screenshot

```
3(1).py    3(2).py
LAB3 > 3(2).py > VotingProcess > request_cs
from queue import Queue
class VotingProcess:
    def __init__(self, pid, voting_set):
        self.pid = pid
        self.voting_set = voting_set
        self.voted = False
        self.queue = Queue()
    def request_cs(self):
        print(f"P{self.pid} sends REQUEST to {self.voting_set}")
        for q in self.voting_set:
            processes[q].receive_request(self.pid)
    def receive_request(self, requester):
        if not self.voted:
```

```
PROBLEMS    OUTPUT    TERMINAL    PORTS    DEBUG CONSOLE

PS C:\Onedrive\Desktop\SEM 7\DS&CC> python -u "c:\Onedrive\Desktop\SEM 7\DS&CC\LAB3\3(2).py"
P0 sends REQUEST to [0, 1, 2]
P0 votes for P0
P0 received REPLY from P0
P1 votes for P0
P0 received REPLY from P1
P2 votes for P0
P0 received REPLY from P2
PS C:\Onedrive\Desktop\SEM 7\DS&CC>
```

Conclusion

Maekawa's Algorithm:
Uses voting sets to reduce message complexity, granting critical section access based on majority approvals while preventing simultaneous entry.