**Name: Syed Naqi Hussain** **ID: XK39135**

**CMSC621-Advanced Operating Systems**

**Project 2- Report**

**Assignment 1- Distributed system time synchronization using Berkeley Algorithm.**

**Assignment 2- Causally ordered Multicasting and comparison with the FIFO ordering.**

**Assignment 3 - Centrally controlled Distributed Mutual Exclusion.**

**Assignment 1**

1. **Design:**

In the Berkeley time synchronization, one process out of the N processes is designated as a Daemon. The Daemon requests all the processes for their time. Once all the processes send their time, the Daemon calculates the average time of all the processes and sends an offset value to all the slaves requesting them to adjust their clocks. The below figure throws some light on the design.

Daemon

**Send Time**

**Adjustment**

**Machine N logical time**

Machine n

Machine 2

Machine 1

Machine 3

1. **Implementation:**

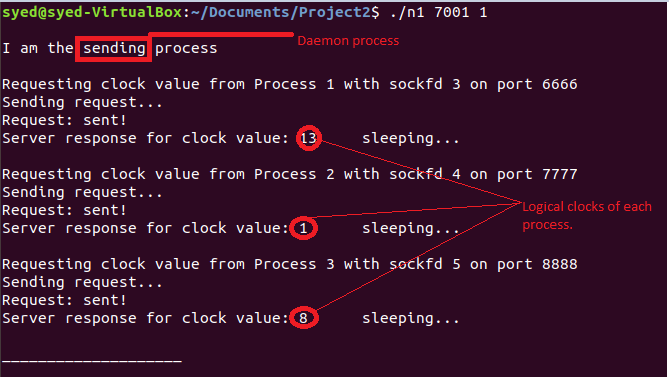
The easiest way to realize Berkeley algorithm is to use a single server-multiple client model. The decision of who can be the leader can be implemented using any of the leader election techniques like Bully algorithms or Ring algorithm. But for simplicity, this implementation choses a random/ user provided process as the daemon process.

The Daemon-Clients communication is established using Sockets. Each process/client waits on a socket to get connected to the Daemon. When all the clients are connected, the Daemon process sends a “Send Time” request to all the clients. The clients respond by sending their respective times to the server. The server then computes the average of all the received clocks and determines the offset to be sent to each of the process.

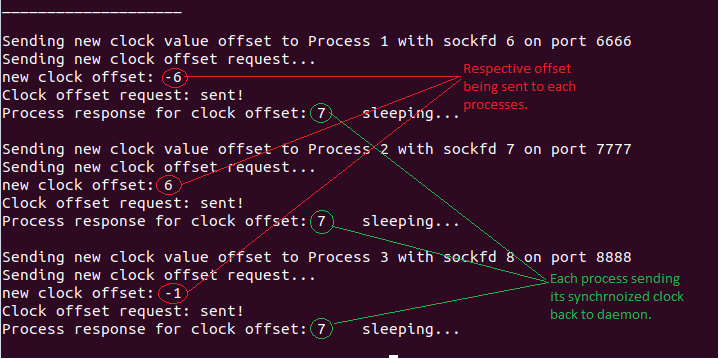
Something worth noticing here is that this implementation deals with logical clocks and not Physical time. A random () function is used at each of the client and the server to generate a random logical number, which is then used as the logical clock of that process.

After determining the offset, the server then broadcasts the offset to all the processes. The processes then go ahead and adjust their times, thus achieving synchronization.

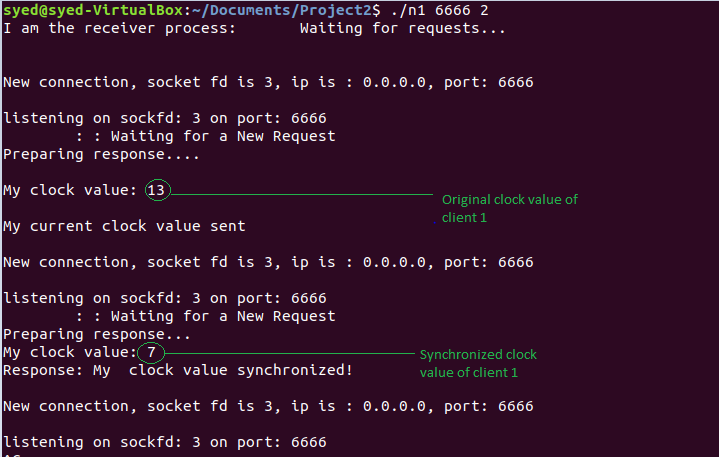
Below screenshots give a clear picture of how the algorithm is implemented:



**Fig 1:** This is the daemon process, connected with three client processes.



**Fig 2**: This is the Daemon process, showing the offset being sent to each process and each client sending the synchronized clock back to the daemon



**Fig 3:** This is the client 1 showing the Original clock value and the final clock value. Similarly, other clients can be shown.

1. **Learnings:**

This project gave me a clear insight into how time synchronization, Berkeley in specific works. I also realized how the introduction of Logical clocks simplifies synchronization to a large extent.

1. **Issues:**

Establishing processes and making sure they communicated to each other was not a trivial task. The socket implementation of the same took time, as I struggled to figure out, how to set up the connections. Assuming a random process to be the Daemon, simplified the task a bit.

-----x-----

**Assignment 2:**

**Design:**

|  |  |
| --- | --- |
| P2  PN  P1  P3 | Machine 5  Machine N  Machine 4  Machine 1 |

**Figure 4: Design for the Causal Multicast**

1. **Implementation:**

In order to realize a true multicast system, it was essential to establish all the nodes/processes in the system such that they can communicate with each other. I used the socket structure for each node in the system, in a way that each server can both connect to other servers and accept connections from other servers. Additionally, each server maintains a vector clock, having the logical clocks of other servers along with its own clock. This mandates that each process/server knows the number of processes in the system.

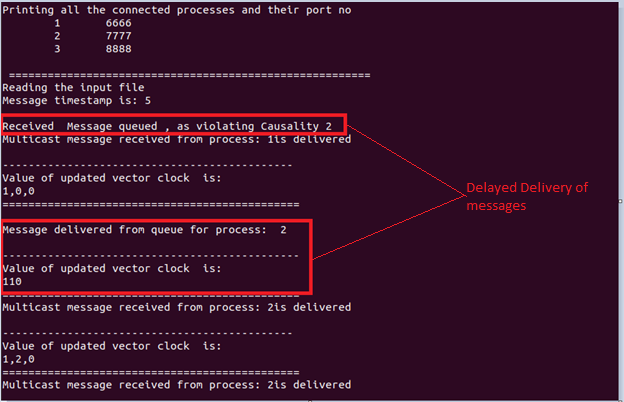
Two versions of this system are implemented: one with causal ordering and other with FIFO ordering.

**Comparison of Causal and FIFO ordering:**

**Causal Ordering:**

In case of causal ordering every message received at a process is first checked for Causality and then either delivered or buffered.

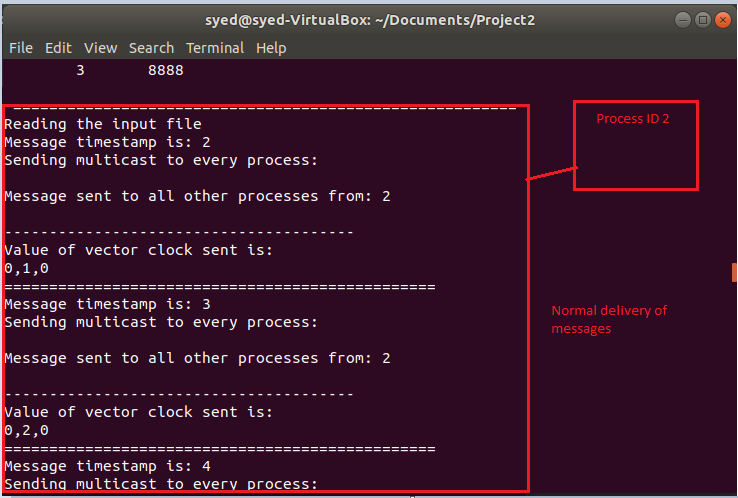
If a message is correctly delivered, then the process checks the buffer to determine if any other messages in the buffer can be delivered.



**Figure 5: Buffering of messages**

**Non-Causal ordering:**

In this case messages are delivered as they arrive at a process. No causal property in their message events.



**Figure 6: Normal delivery of messages**

1. **Learnings:**

This project helped me understand in a true sense, how various processes can be set up to act as both a sever and a client, I also learned how processes can be controlled to deliver messages or not by imposing Causal ad Non-Causal ordering. I was able to relate to real time applications of such ordering, like Scoreboards, stocks, social media message service.

1. **Issues:**

It was a non-trivial task to make all the nodes connect and communicate to each other. As the task was to make each node act as both server and a client for other nodes, the logic took some time to implement. The main() program in my code creates a thread for the process to act as a server and accept multiple client requests, and within the main() program, the process also acts as a client and connects to other servers/processes.

The task of making a copy of all the processes and their ID’s available to all the other processes also took some time to implement. This task was accomplished by creating an array of structures, with each structure pointing to process ID, socket descriptor and port number. Once the process details were available it was easy to send multicast messages.

Other issue was to implement delayed messaging, in order to capture Causality violation. This issue was resolved by making each process read an input file given to it from the command line. The format of the input file is as shown below.

*Note: The system takes the number of process in it, through a file named NoofProc.txt*

**Name:** input1.txt 🡪 This file is given to process ID 1 from the command line

**Format:**

12

16

18

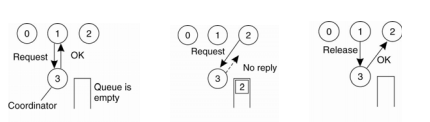
Here the data represents the timestamps, at which the message should be sent.

---x---

**Assignment 3**

1. **Design:**

A centralized algorithm for distributed locking has been implemented using sockets. The design assumes a central server, acting as a coordinator, protecting the shared file from concurrent access. Multiple processes acting as clients can request for the shared file over the sockets. The figure below gives a clear picture of the design.



**Figure 7: Design of Centralized Algorithm for Distributed Locking**

1. **Implementation:**

A server is created to act as a coordinator to manage concurrent access and update to a shared file. Multiple processes can request for access to the shared file. The server spawns a new thread for communicating with respective client.

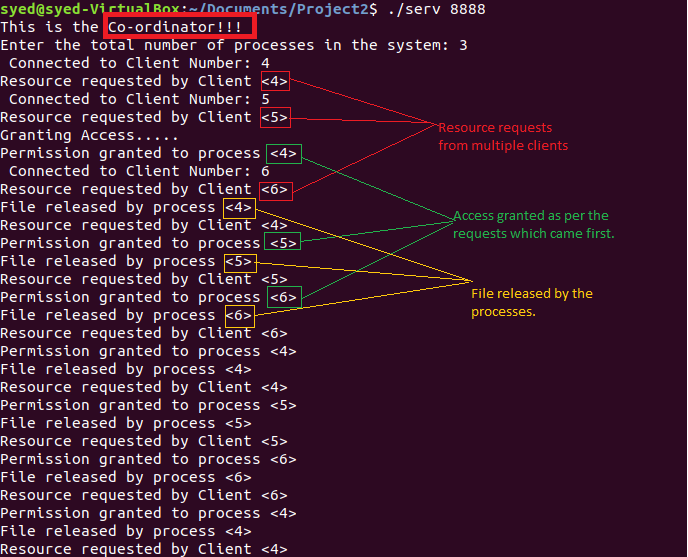
A client when connected to the server, sends a REQUEST message to the server. The corresponding client thread on the server pushes the message onto a Queue, which the server maintains.

Once, all the clients or processes are connected, the coordinator thread is spawned.

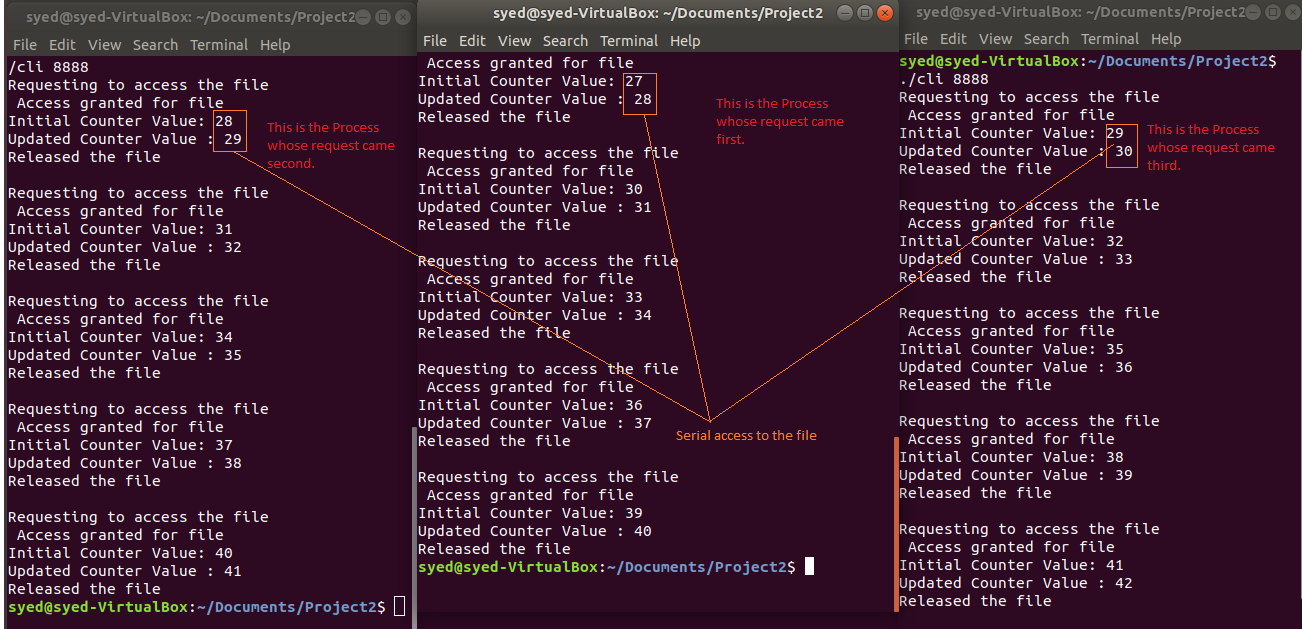
The coordinator thread checks if the no process is accessing the shared file and grants the access to the first process which requested the file, present one the queue. This access is granted by sending an “OK” message to the respective client. On receiving the OK message, the client opens the file, updates the value of the file by 1 and writes back to file. On completion the client process sends a RELEASE message to the coordinator.

The next process on the queue waits on a conditional variable, till the other process releases the share file, and releases the lock and signals on the conditional variable. The waiting process wakes up and gets access to the shared file.

Thus, all the processes acquire the shared file, update the counter value, without any deadlock.



**Figure 8: Coordinator process**

****

**Figure 9: Client processes**

1. **Learnings:**

This assignment gave me a clear picture of how the centralized mutual exclusion works. I learned a great deal about how to utilize locks and conditional variables to protect the critical section and schedule processes to wait and signal.

1. **Issues:**

The idea of using conditional variables to schedule processes was tough to implement.

--x—

**References:**

1. <https://github.com/intekmaster/CLOCKS-MULTICAST-AND-COMMIT/blob/master/README.pdf>
2. <https://stackoverflow.com/questions/23154075/socket-programming-having-a-process-with-both-client-and-server-code>