CMSC 621: Advanced Operating Systems

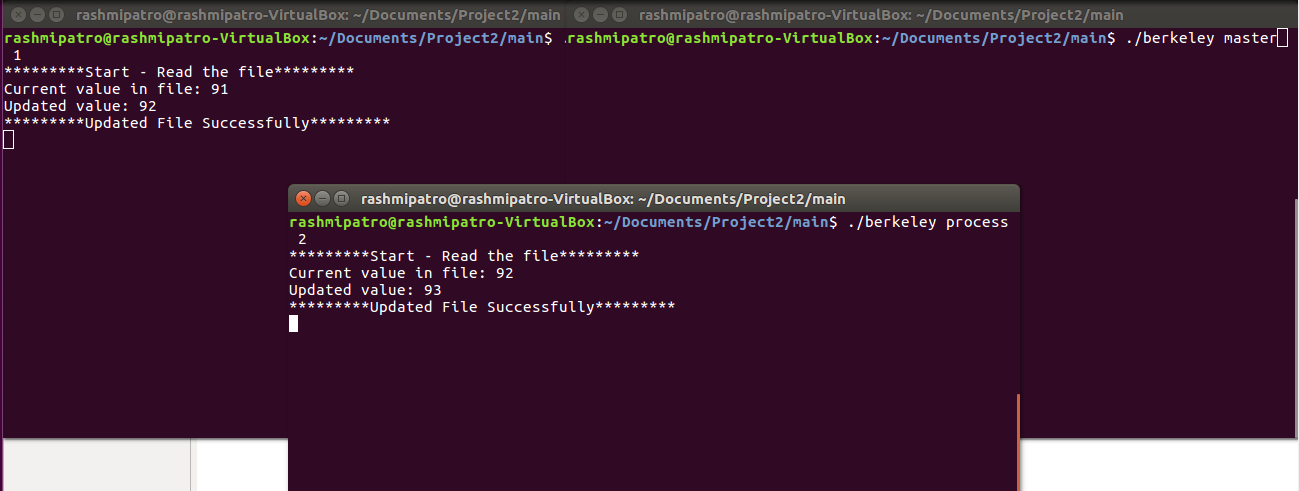
PROJECT 2

1. **Berkeley Algorithm:**

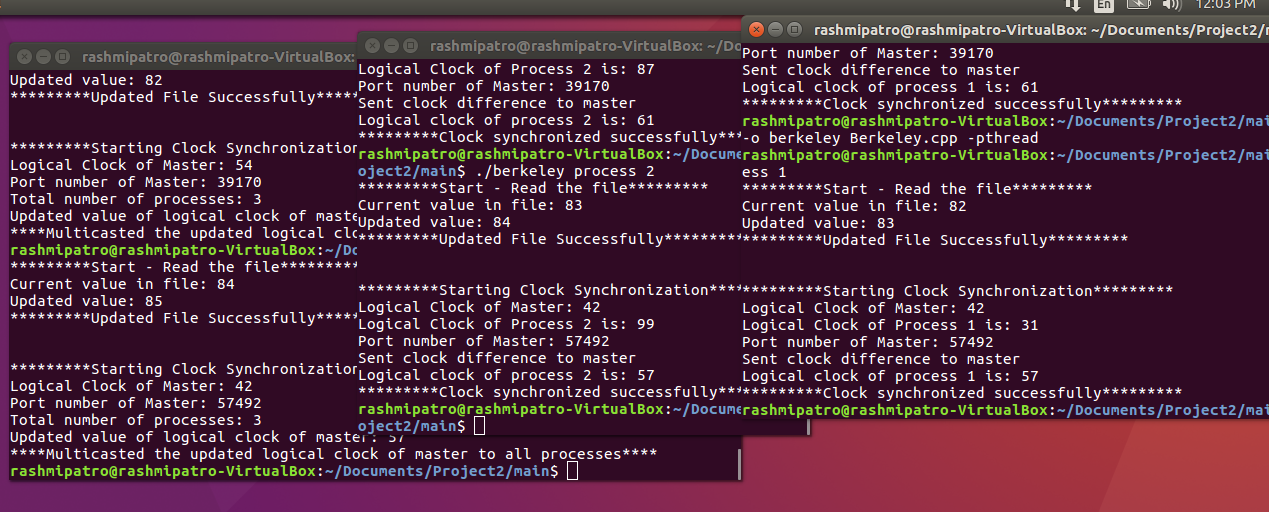
Berkeley.cpp is the program in which “Berkeley Algorithm” has been implemented. I have used sockets to communicate between processes and implemented the multicast concept as well. Multiple instances can be created of the same program. All client processes would wait until the master server starts the program. All processes generate a random clock value for themselves. The master server multicasts its clock value to all the clients and in turn it receives the offset value from each of them.

Master server then calculates the average and sets its own clock and again multicasts the updated value to all the processes. As a result, clock synchronization is established among all the processes.

Below is the snapshot of the initial state of the program, where two client processes are waiting for the master server to connect so that they can proceed.



Below is the snapshot of the final state of the program when the master server connects and performs required actions thereby synchronizing the clocks among all the processes.



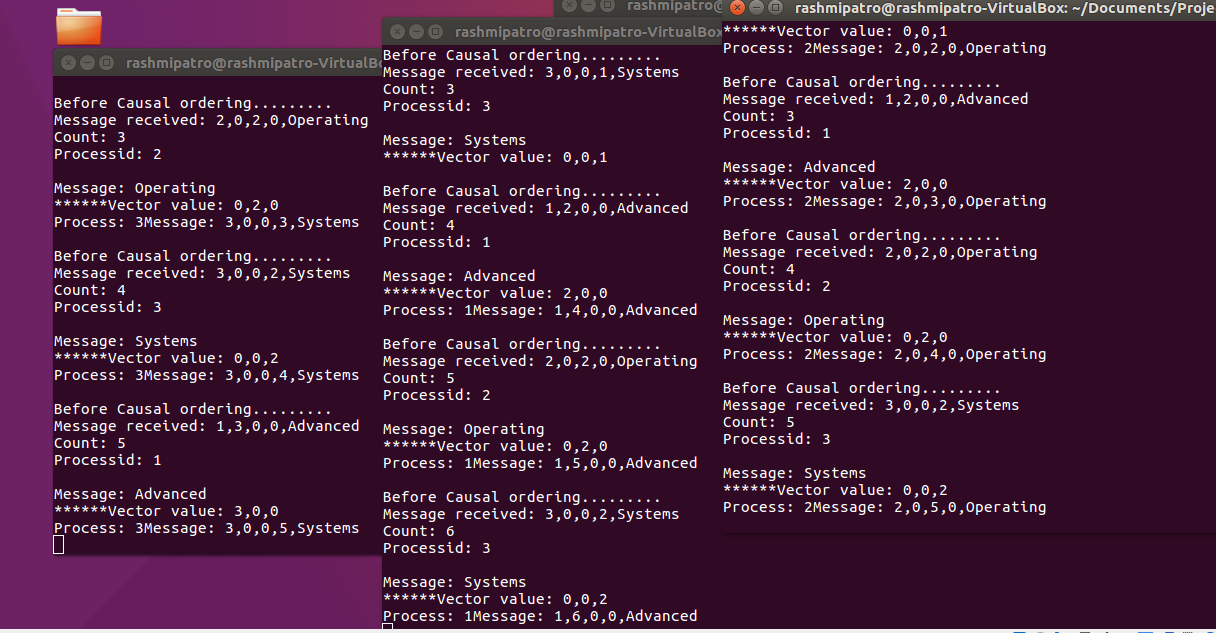
To run the program, we have to provide 3 parameters for client processes and 2 parameters for the master server. All required commands are mentioned in the make file and readme.

1. **Causal Ordering:**

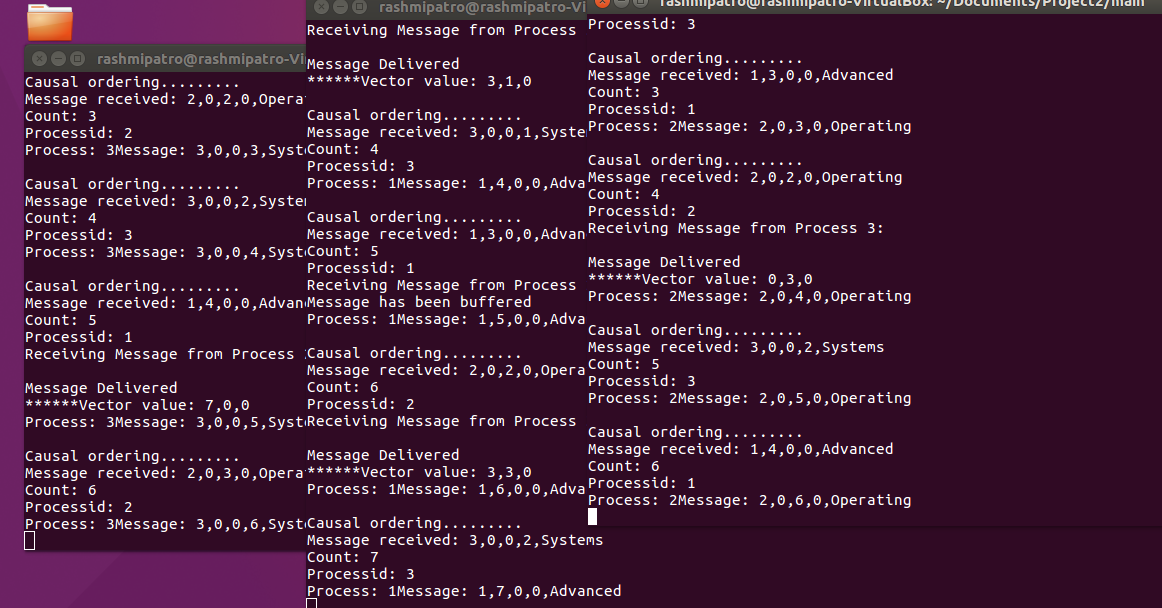
There are two programs, one with causal ordering and other with non-causal ordering. In order to start the program we have to compile CausalOrdering.cpp and NonCausalOrdering.cpp files. Both of them are separate programs and multiple sessions can be created for those programs. We have to provide 3 parameters in the command line. The output filename, rank of the process and message that needs to be multicasted. This works as below:

1. There are two threads created for sending and receiving the messages to/from all the processes.
2. When a process sends a message, it associates a vector value with the message and sends that as well.
3. While receiving, the values present in the vector are compared with the respective values present in each processes.
4. There are two programs:
   1. In CausalOrdering.cpp each process checks the vector value of its own and other processes before delivering the messages. If the values is less than or equal to the current value, then the message is delivered else the messages are buffered and are kept in buffer until the messages in sequence are received.
   2. In NonCausalOrdering.cpp each process sends the message along with its vector value. While receiving, processes does not check the vector values and they do not prioritize the messages. All messages are delivered in the way they reach the processes.

Below is the snapshot of the non-causal ordering program, which shows that all the messages are delivered irrespective of their vector values.



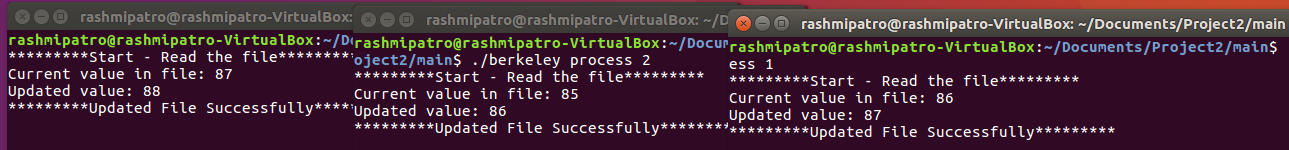
Below is the snapshot of the causal ordering program, which shows that before the messages are delivered, the vector values are checked and are buffered if those values are not in the order.



1. **Distributed Lock:**

Part I and Part III are in one program which is Berkeley.cpp. To implement distributed locking I have used mutex which on a shared file. All the processes try to connect and exhibits a mutex lock on the file. The file contains a value which needs to be read and updated by each process. ­­­Only after they read and update the value in the file, mutex is unlocked.

Below is the snapshot of the performance of 3 processes sharing the same file and having distributed lock.­



Through this project I gained knowledge on the implementation of Berkeley algorithm and causal ordering. I learnt how to multicast messages among a group of processes. In addition to that, I learnt about distributed locking.

There were many challenges, such as:

* Establishing the connection among different processes in order to multicast the message
* Implementing Causal Ordering was really tough, because we had to calculate the vector values associated with each message and compare it with the current values.