Final Project Report

CSE 536

Name: Vishal Srivastava ID No.:1209824652

Design Phases

Phase I

Assumptions

Under the following assumptions the program works fine:

1. All access rights are available with the required directory structure.
2. Each cell holds no more than one data from a particular file.
3. Each cell block has a size of 1024 bytes and can store value as strings.
4. Each cell is owned by a particular file and thus cannot be accessed or written by another file.
5. The program writes on one cell at a time with its associated file id.
6. The program can only store 100 cells in its temporary storage (state before commit).
7. All blocks/cells belonging to a certain file are committed simultaneously
8. Several cells/block of files need not be concurrent to each other.
9. Currently the system is neither fault tolerant nor is multithreaded

Implementation Procedure

Currently for Phase I, the whole program is divided into 3 main header files namely cellstorage.h, journal_file_manager.h and storageunits.h. A structure defined for storage and is written and read from files. Each text file under the cells folder is a cell block where structure block is written and read from.

Cell Storage

The cell storage contains all the associated functions like IS_OWNER, CHECK_FOR_BLOCK, READ, WRITE, ALLOCATE and DEALLOCATE.

1. IS_OWNER
   Use to check for the ownership for that particular block with the supplied file_id.
   INPUT: int file_id and int cells
   OUTPUT: Returns 1,2,3 for OK, Failure to read and Access violation
2. CHECK_FOR_BLOCK

Use to check if a certain block is available in the cells directory or not.
INPUT: String filename (block no)
OUTPUT: Returns 0,1 for True and False in terms of availability

3. READ
   Use to read data from the block/cell associated with file id.
   INPUT: int file id, int process id (p_id) and int cell no
   OUTPUT: Returns string containing the data stored in the cell
4. WRITE
   Use to write data to the cell block with an associated file id and stores data in the cell.
   INPUT: int file id, int process id (p_id), int cell_no and string value_data
   OUTPUTS: Returns 0,1 for OK, and Error

Journaling File Manager

Uses the implemented storage block for the file system for temporary storage of data. It keeps a state for temporary storage of data before commit and keeps the data on a non-volatile storage. Acts as an abstraction layer over the Cell storage system. It uses the following programs:

1. NEW_ACTION
   Allocates a cell block to a particular file id and creates a process_id
   INPUT: int file_id
   OUTPUT: Returns a integer type generated process_id
2. READ_CURRENT_VALUE
   Reads a value from the cell storage with the associated cell and file id
   INPUT: int file id, int process id, int cell no
   OUTPUT: Returns string of data from that particular cell/block.
3. COMMIT
   Use to commit data associated with a particular file id to the cell storage system. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for data for a particular file_id and process id and valid field. If valid is 1 then the data is committed for the associated file id and process id. After committing make the valid entry to 0.
   INPUT: int file_id, process_id
   OUTPUT: returns 0,1 for OK and failure
4. WRITE_NEW_VALUE
   Used to store the data in the temporary storage before committing. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for new or invalid entry from start of the queue. Enters the data and set the valid field to 1, in the invalid or new storage block and saves it back to temp storage file.

INPUT: int file id, int process id, int cell no, string data
OUTPUT: Returns 0,1 for OK and failure

5. ABORT
   Use to abort all the entry in the temporary storage associated with a particular process id and file id. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for all the possible entries associated with that process id and file id and set their valid field to 0. It again writes back to file with the changes.
   INPUT: int file id and int process id

6. Cleanup
   Used to reset the state of the system by deleting all the state files and temporary storage file.
   INPUT: Nothing
   OUTPUT: Nothing

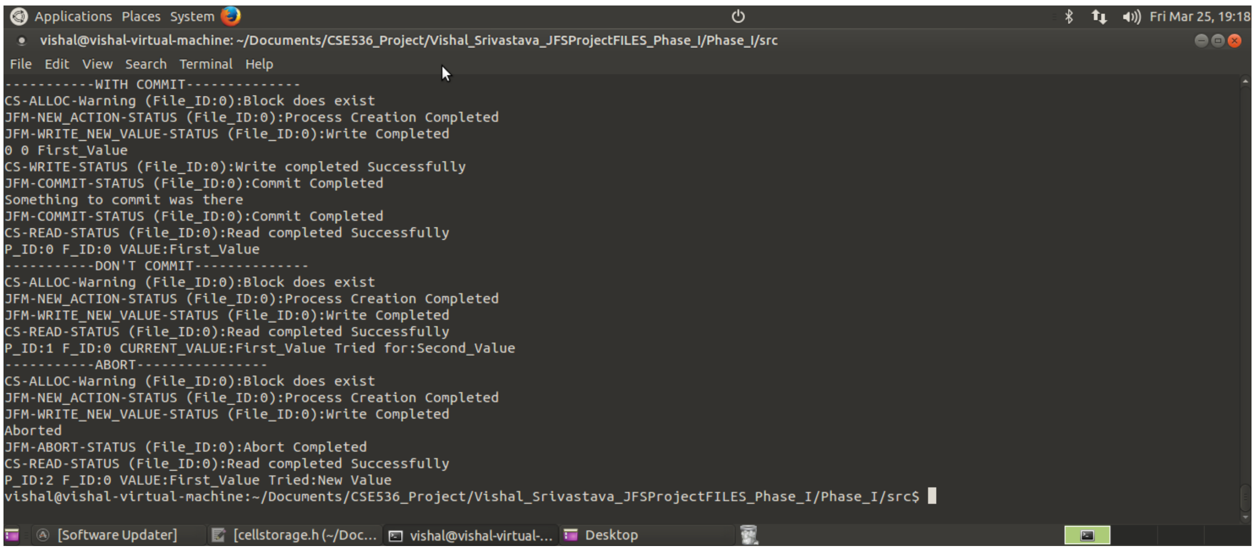
Test cases and reason for the choice

We have chosen the following test cases:

1. Normal read write operation
   In this test case we check for normal read and write functionality of the program with the commit. Here we first allocate space using NEW_ACTION and retrieves the generated process id. Then using process id and field id we write the data to the first block of the temp storage file. Then we store commit the values for the block to cell storage system. After committing we try to read the value using the READ_CURRENT_VALUE and we retrieve back the value from cell storage file and display it.
   This test case checks the normal operation of the file system which is supposed to be the maximum case that can be observed.
2. Read write without commit
   In this test case we did the same as above but just that we didn't commit after writing to the temporary storage block. After writing we again tried reading the values from the cell storage system. As we have not committed the write so the value didn't get stored in the cell storage system. Thus when we tried reading from it it gave the old value that was stored in cell storage system.
   This was done to check the functionality of the commit operation without which the mutation of data is not carried forward to our permanent storage that is cell storage.
3. Read write with abort

We did the same read and write just like then earlier case but after write we put the abort to stop the mutation to be carry forwarded to the cell storage system. Abort flushes down all the entries in temporary storage associated with a valid entry to a file id and process id. Then even after the commit none of the cell blocks will get updated for that file id and process id as none of them are valid now. Then if we tried to read we will get the old data in cell for that particular file id.

Test run screenshot and explanation



The above test cases are tested in the same file. We can see as the blocks/cell were already there from previous run Cell Storage(CS) gave a warning for it. Then New action returned from Journal File Manager (JFM) gave an all clear status for creating the process. Then WRITE_NEW_VALUE successfully wrote the value. As there were some entries left to be committed the JFM searched for those entries and committed those entries. Then the CS returned back the successful commit message and the same message is replicated back to the JFM where it also returns committed successfully. Then a read is done which returned back the value and the value is printed with process Id and file id.

For the second test, is also same but we haven't committed here. So the last line given us the value (First Value) that it retrieved from the cell storage and the value ('Second Value') it tried to write to the storage system.

For the third test, which is also same as second. Here we wrote the values to first temporary storage but then we aborted and thus we removed all the values associated with file id and process id. Then we tried to read value from cell storage and found that the value (New Value) we tried and value (First Value) we got from cell storage were different.

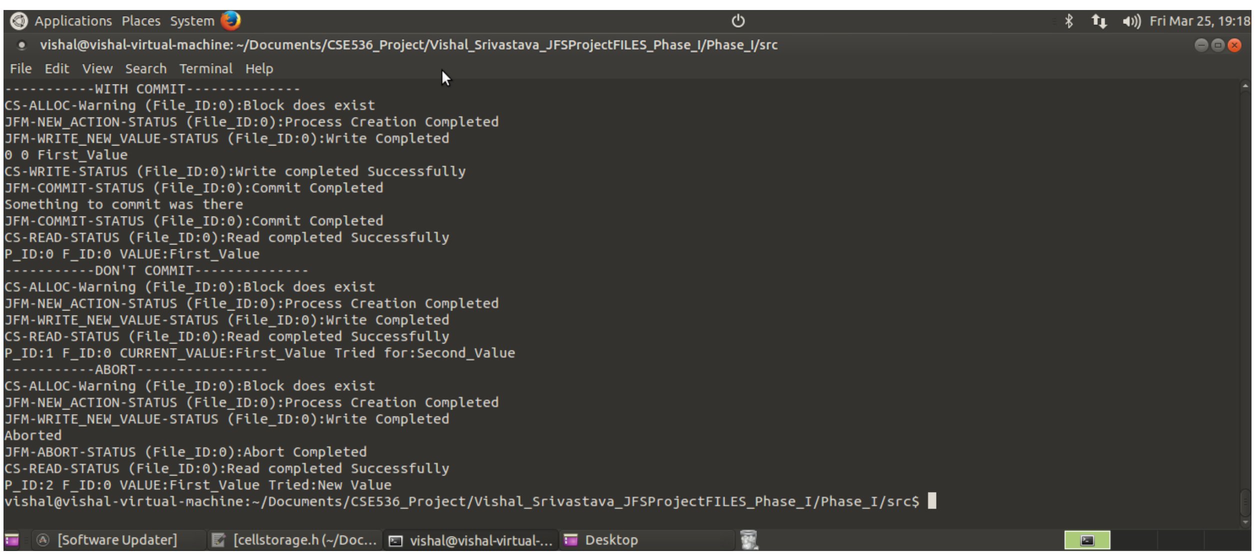
Results

We were able to successfully implement the cell storage and Journaling File system with minor problems. The results we obtained tally with the design. As time was a factor for completing this assignment we had to be fast about it. There could be some problems with the implementation under certain test case as it's still not well tested but works well with the foreseen test cases.

Discussion

As there wasn't enough time so couldn't divide the program but each function is independent to the rest and thus can work effectively in later phases. Also as the whole file system both temporary and cell storage is implemented in the non-volatile storage using files as blocks so in case of failure only a certain segment will be lost but the downside to this implementation is that this system will be slow for read and writes. This kind of system is generally observed in the database management system where failure will be catastrophic. So we went with this type of implementations. Later on maybe we can have cache system to it but for now it works perfectly fine.

Output Screenshots of program running



Phase II

Assumptions

Under the following assumptions the program works fine:

1. All access rights are available with the required directory structure.
2. Each cell holds no more than one data from a particular file.
3. Each cell block has a size of 128 bytes and can store value as strings.
4. Each cell is owned by a particular file and thus cannot be accessed or written by another file.
5. The program writes on one cell at a time with its associated file id.
6. The program can only store 100 cells in its temporary storage (state before commit).
7. All blocks/cells belonging to a certain file are committed simultaneously
8. Several cells/block of files need not be concurrent to each other.
9. Currently the system supports multithreading but not fault tolerant
10. No two runs of test cases and made simultaneously
11. The directory 'cells' is always present but is cleared before each run. Maintained by cleanup ()

Implementation Procedure

Currently for Phase I, the whole program is divided into 3 main header files namely cellstorage.h, journal_file_manager.h and storageunits.h. A structure defined for storage and is written and read from files. Each text file under the cells folder is a cell block where structure block is written and read from. In order to support the multi-threading we have implemented two mutex locks: cell_file_lock and temp_file_lock to ensuring random but sequential access of cell storage and temp storage (as both depends on file I/O). Both locks have only been implemented in Journal File Manager. Certain changes in arguments were also done as each thread can only be given one object to be passed. So we are passing whole storage units here but only in operations related to journal file manager.

Cell Storage

The cell storage contains all the associated functions like IS_OWNER, CHECK_FOR_BLOCK, READ, WRITE, ALLOCATE and DEALLOCATE.

1. IS_OWNER
   Use to check for the ownership for that particular block with the supplied file_id.
   INPUT: int file_id and int cells
   OUTPUT: Returns 1,2,3 for OK, Failure to read and Access violation
2. CHECK_FOR_BLOCK
   Use to check if a certain block is available in the cells directory or not.
   INPUT: String filename (block no)
   OUTPUT: Returns 0,1 for True and False in terms of availability
3. READ
   Use to read data from the block/cell associated with file id.
   INPUT: int file id, int process id (p_id) and int cell no
   OUTPUT: Returns string containing the data stored in the cell
4. WRITE
   Use to write data to the cell block with an associated file id and stores data in the cell.
   INPUT: int file id, int process id (p_id), int cell_no and string value_data
   OUTPUTS: Returns 0,1 for OK, and Error

Journaling File Manager

Uses the implemented storage block for the file system for temporary storage of data. It keeps a state for temporary storage of data before commit and keeps the data on a non-volatile storage. It acts as an abstraction layer over the Cell storage system. For multithreading here, we have implemented two mutex locks, one for temporary storage access (temp_file_lock) and another for the cell storage system(cell_file_lock). It uses the following programs:

1. NEW_ACTION
   Allocates a cell block to a particular file id and creates a process_id. Both locks have been implemented for read and write to temp storage to provide reliable temporary storage access (temp_file_lock) and maintaining consistency on cell_no allocation and updation (by cell_file_lock). Interestingly now cell_no are assigned randomly but correctly to each of the thread so we can observe some later threads (or even file) to acquire starting blocks (or cells) in the cell storage system. But still no unauthorized access is permitted
   INPUT: struct storage*
   OUTPUT: Returns struct storage*
2. READ_CURRENT_VALUE
   Reads a value from the cell storage with the associated cell and file id. Here only cell_file_lock has been implemented for maintaining a single access during read (no mutation possible during read).
   INPUT: struct storage *
   OUTPUT: Returns struct storage*.

3. COMMIT
   Use to commit data associated with a particular file id to the cell storage system. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for data for a particular file_id and process id and valid field. If valid is 1 then the data is committed for the associated file id and process id. After committing make the valid entry to 0. Both locks have been implemented on the block level (i.e. the whole working code) granularity in the function. This is done as we are getting garbage value comited due to I/O bandwidth limitations. Performance do suffer but we had to do it to maintain reliability of commit data when large no of threads are operating simultaneously.
   INPUT: struct storage *
   OUTPUT: returns pointer for 0,1 for OK and failure

4. WRITE_NEW_VALUE
   Used to store the data in the temporary storage before committing. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for new or invalid entry from start of the queue. Enters the data and set the valid field to 1, in the invalid or new storage block and saves it back to temp storage file. For multithreading we have implemented on temp_file_lock as we have only to maintain the reliability of the data in temp storage. The locks are only implemented for read and write of the temp storage.
   INPUT: struct storage*
   OUTPUT: Returns pointer for 0,1 for OK and failure

5. ABORT
   Use to abort all the entry in the temporary storage associated with a particular process id and file id. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for all the possible entries associated with that process id and file id and set their valid field to 0. It again writes back to file with the changes. For multithreading we have implemented on temp_file_lock as we have only to maintain the reliability of the data in temp storage. The locks are only implemented for read and write of the temp storage.
   INPUT: struct storage *
   OUTPUT: Returns pointer for 0,1 for OK and failure

6. Cleanup
   Used to reset the state of the system by deleting all the state files and temporary storage file. As it occurs only after the end of the program no multithreading support is needed here.
   INPUT: Nothing
   OUTPUT: Nothing

Test cases and reason for the choice

We have chosen the following test cases:

1. Multithreaded read write operation
   In this test case we check for multithreaded read and write functionality of the program with the commit. Here we are accessing 10 files each corresponding to a particular process (or file). Each file contains 2 blocks (or cells) of memory. So, total of 20 threads. All processes have NEW_ACTION (to allocate block (or cell) in cell storage and assign and return a process_id), WRITE_NEW_VALUE (to write data on the temporary storage block), COMMIT (to write all the temporary storage blocks to cell storage associated with a certain file_id and process_id) and READ_CURRENT_VALUE (to read and display the most recent update on a particular block (or cell) in the cell storage). This test case checks the normal operation of the file system under multithreaded environment. Though due to low I/O bandwidth of non-volatile storage sometimes the values are not written properly. So under this constraint we are having it limited 20 threads. Though it can support many more (not more than 100 cell access) but reliability of writes is a problem.

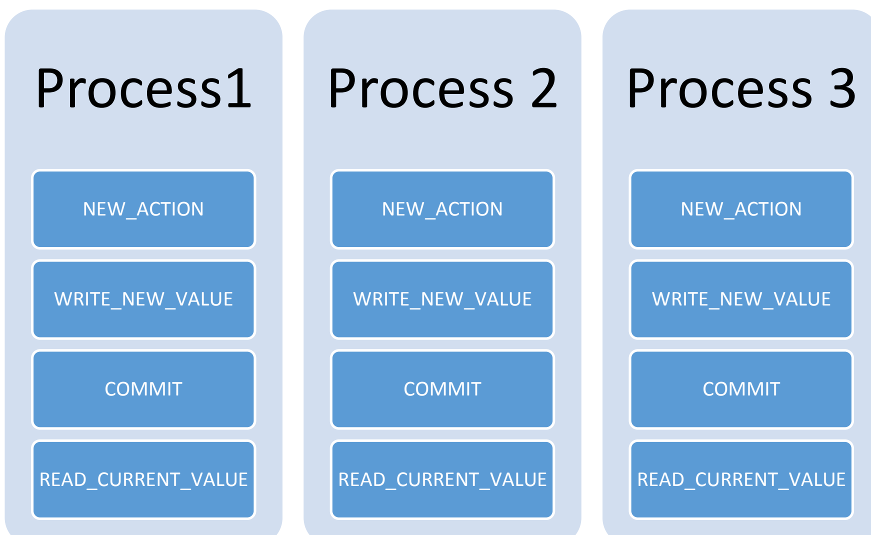


Figure 1:Example for three simultaneous Threads

2. Race condition on simultaneous access of a cell
   In this test case we are trying to introduce a race condition when two processes try to access the same block (or cell) simultaneously. For this we have first allocate a block (or cell) and then two process are trying to run the operation WRITE_NEW_VALUE, COMMIT and READ_CURRENT_VALUE in sequential order. As both threads are assigned the same file id and process id (had to do it manually) they both have access to cell 0. The result of this case is undefined as either of thread can come before or after during

commit but not simultaneously. This is done two show before or after atomicity implemented in the journal file system.

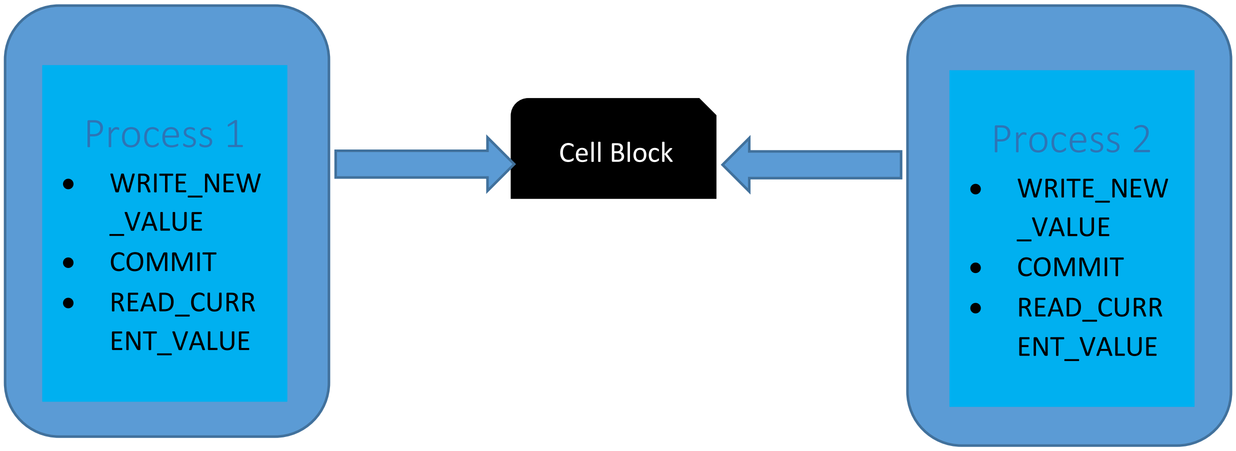
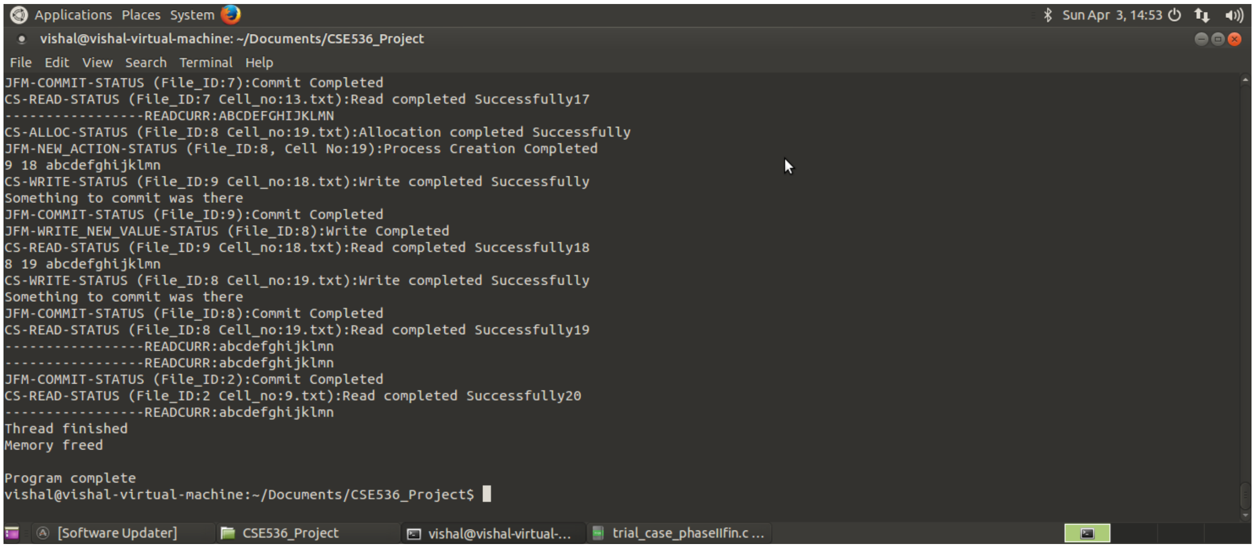


Figure 2: Simultaneous access of a cell by two processes

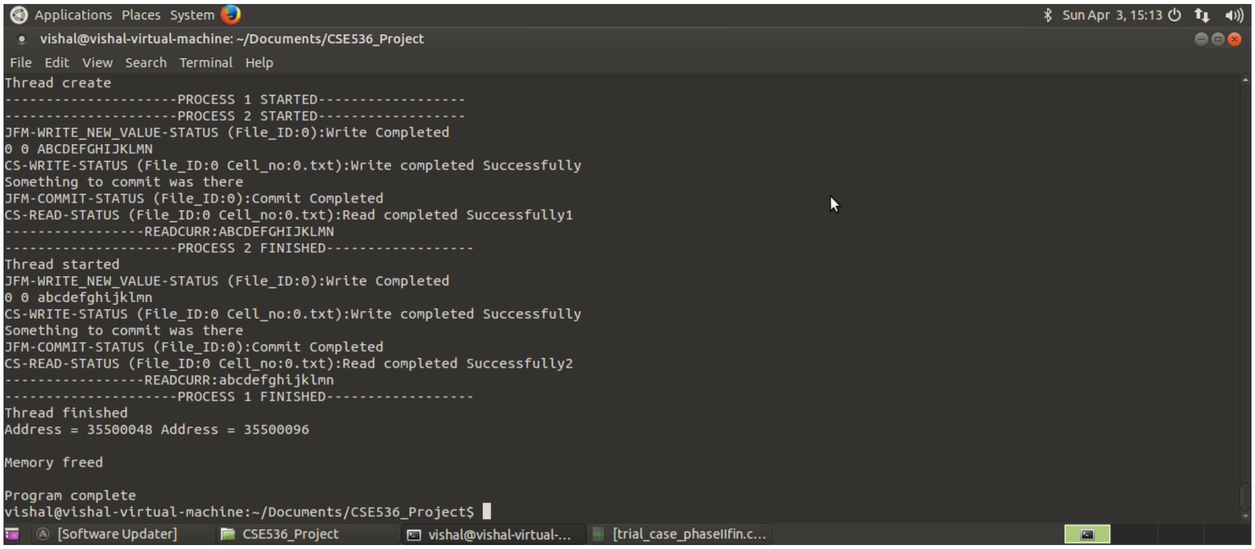
Test run screenshot and explanation

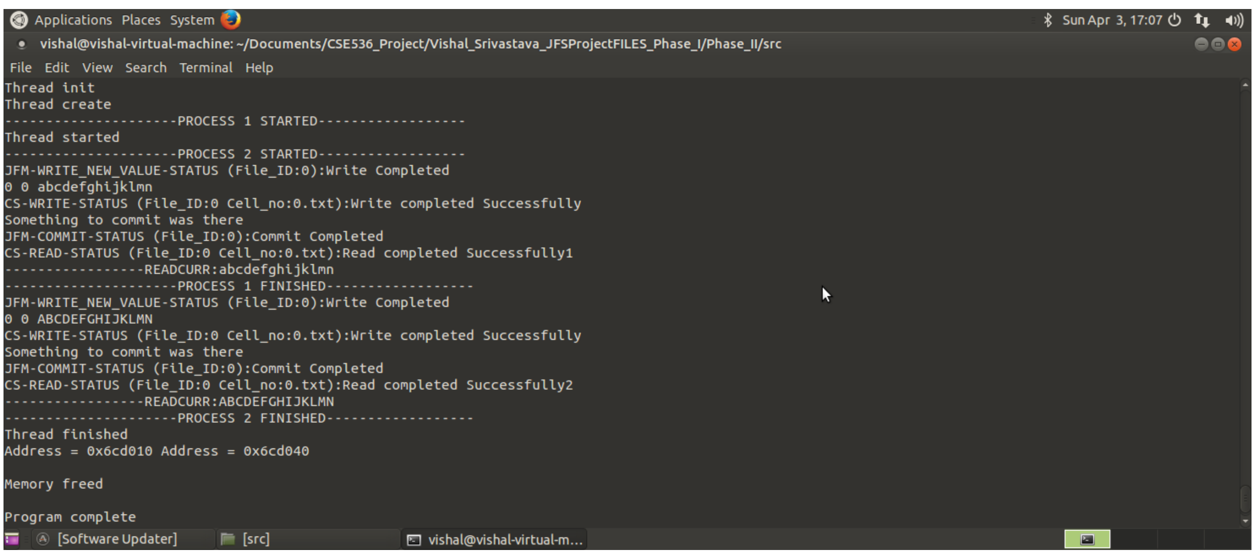
1. Test case I: Multithreaded read write operation



In this test case we are performing simultaneous multithreaded write to multiple cells. Here we are accessing 10 files each with 2 blocks (or cells). So, total of 20 threads. As we can see that first we are allocating blocks using NEW_ACTION and then we are writing to temp blocks, committing them to cell storage blocks and then reading blocks for their value for cell storage. Here we have no introduced the race condition in access to the blocks but we have introduced race condition for allocation of blocks during NEW_ACTION. We can observe that due to multi-threading the allocation of the blocks are done in random order but no two files have been assigned the same block. Also the execution in out of order as it depends on thread scheduler as to who is to be executed. Due to which we observe an out of order execution. Also as the whole process (only the parts within are atomic) is not made atomic, so the execution in showing mixed random order. But within the process the execution is sequential. Also access to temp file storage and cell storage are made random yet sequential so we have no problem in I/O as no two process are assigned the same process_id and file_id.

2. Test Case II: Race condition on simultaneous access of a cell





In this test we also observe the randomness due to thread scheduler. As it was difficult to produce such thing (due to generation of new blocks by NEW_ACTION) in the earlier program we have made this as a certain test case. Here we have two threads competing for the same block in the cell storage. First we have already allocated a block (cell 0) and then we creating 2 threads both performing WRITE_NEW_VALUE, COMMIT, READ_CURRENT_VALUE with different values as 'abcdefghijklmn' and 'ABCDEFGHIJKLMN'. So even if we start both thread simultaneously, it's not certain that a particular thread will end before the other. The two test runs of this test case shows the same effect. In the first figure process 1 finished earlier than process 2 and value 'abcdefghijklmn' was stored finally but it was vice versa in the other and value 'ABCDEFGHIJKLMN' was stored finally. It's a random process and it difficult to predict which one will get completed first.

Results

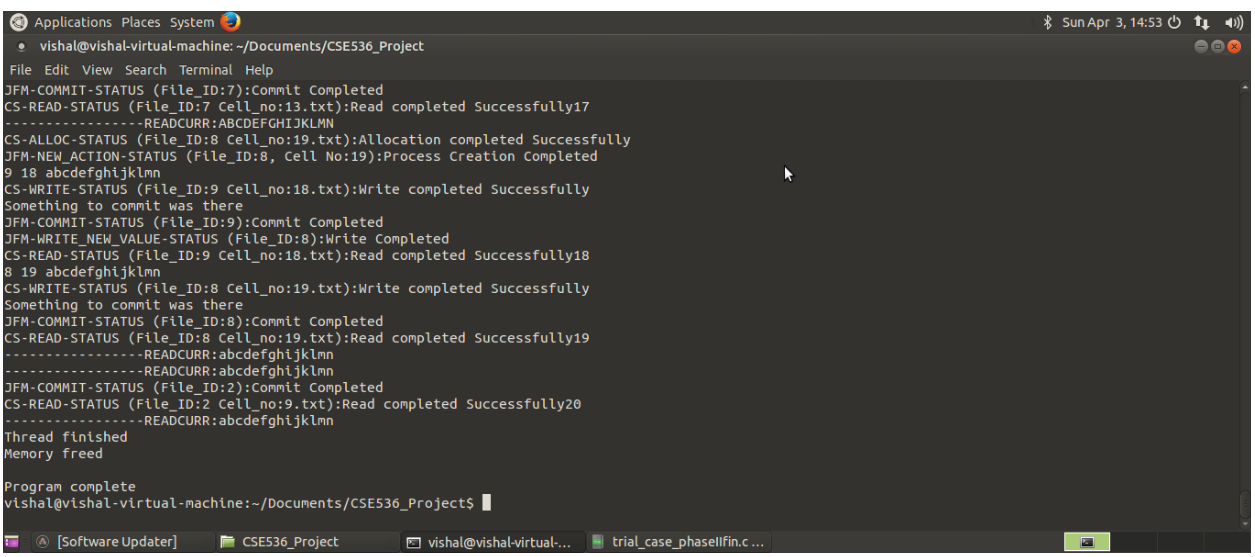
We were able to implement multithreading with before or after atomicity. The implementation was done by mutex locks. Until a process is killed it was ensured that there won't be any deadlocks in the design. Also from the above test cases we have succeed in implementing the before or after atomicity. We also show the performance achieved due to required granularity for different operations.

Discussion

As the whole file system both temporary and cell storage is implemented in the non-volatile storage using files as blocks so in case of failure only a certain segment will be lost but the downside to this implementation is that this system will be slow for read and writes. This kind of system is generally observed in the database management system where failure will be catastrophic. So we went with this type of implementations. While multithreading we also observed certain limitation of the design.

As the temporary storage is on non-volatile storage we do get robustness and fault tolerance as we have to only recover the last mutation during a failure but due to this the I/O performance had decreased. So if we are trying to have multiple threads access the temporary storage we start getting random mutations (garbage values) which we didn't accounted for. To fix this issue we made the COMMIT operation atomic and increased the granularity. We also restricted threads to 20 threads as they gave correct results. Though system should be able to handle more threads but the generation of garbage value is still not well understood. For now, it's seems due to disk I/O operation but we will look into that and try to fix that in later phases of the program.

Output Screenshots of program running



Phase III

Assumptions

Under the following assumptions the program works fine:

1. All access rights are available with the required directory structure.
2. Each cell holds no more than one data from a particular file.
3. Each cell block has a size of 1024 bytes and can store value as strings.
4. Each cell is owned by a particular file and thus cannot be accessed or written by another file.
5. The program writes on one cell at a time with its associated file id.
6. The program can only store 100 cells in its temporary storage (state before commit).
7. All blocks/cells belonging to a certain file are committed simultaneously
8. Several cells/block of files need not be concurrent to each other.
9. Currently the system is fault tolerant but not multithreaded
10. The transactions can be reinitiated.
11. The recovery will be error free
12. Every action is logged and securely and atomically
13. The execution is not hard real time
14. Recovery time overheads is acceptable
15. Can recover only recover from 100 erroneous commits at a time.

Implementation Procedure

Currently for Phase III, the whole program is divided into 5 main header files namely cellstorage.h, journal_file_manager.h, RecoveryManager.h, log.h and storageunits.h. A structure defined for storage and is written and read from files. Each text file under the cells folder is a cell block where structure block is written and read from.

Cell Storage

The cell storage contains all the associated functions like IS_OWNER, CHECK_FOR_BLOCK, READ, WRITE, ALLOCATE, ALLOCATE_OVERIDE and DEALLOCATE.

1. IS_OWNER
   Use to check for the ownership for that particular block with the supplied file_id.
   INPUT: int file_id and int cells
   OUTPUT: Returns 1,2,3 for OK, Failure to read and Access violation

2. CHECK_FOR_BLOCK
   Use to check if a certain block is available in the cells directory or not.
   INPUT: String filename (block no)
   OUTPUT: Returns 0,1 for True and False in terms of availability
3. ALLOCATE
   Use to create cell blocks and assign respective file_id. Cell block no is automatic generated and a deallocated cell can't be used.
   INPUT: int file_id
   OUTPUT: int cell_no
4. ALLOCATE_OVERIDE
   Use to create cell blocks and assign respective file_id. Cell block no is manually assigned and a deallocated cell can be used.
   INPUT: int file_id, int cell_no
   OUTPUT: int cell_no
5. READ
   Use to read data from the block/cell associated with file id.
   INPUT: int file id, int process id (p_id) and int cell no
   OUTPUT: Returns string containing the data stored in the cell
6. WRITE
   Use to write data to the cell block with an associated file id and stores data in the cell.
   INPUT: int file id, int process id (p_id), int cell_no and string value_data
   OUTPUTS: Returns 0,1 for OK, and Error

Journaling File Manager

Uses the implemented storage block for the file system for temporary storage of data. It keeps a state for temporary storage of data before commit and keeps the data on a non-volatile storage. Acts as an abstraction layer over the Cell storage system. Here is where logging is implemented for each of its operations. It uses the following programs:

1. NEW_ACTION
   Allocates a cell block to a particular file id and creates a process_id
   INPUT: int file_id
   OUTPUT: Returns a integer type generated process_id
2. READ_CURRENT_VALUE
   Reads a value from the cell storage with the associated cell and file id
   INPUT: int file id, int process id, int cell no
   OUTPUT: Returns string of data from that particular cell/block.
3. COMMIT

Use to commit data associated with a particular file id to the cell storage system. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for data for a particular file_id and process id and valid field. If valid is 1 then the data is committed for the associated file id and process id. After committing make the valid entry to 0.
INPUT: int file_id, process_id
OUTPUT: returns 0,1 for OK and failure

4. WRITE_NEW_VALUE
   Used to store the data in the temporary storage before committing. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for new or invalid entry from start of the queue. Enters the data and set the valid field to 1, in the invalid or new storage block and saves it back to temp storage file.
   INPUT: int file id, int process id, int cell no, string data
   OUTPUT: Returns 0,1 for OK and failure

5. ABORT
   Use to abort all the entry in the temporary storage associated with a particular process id and file id. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for all the possible entries associated with that process id and file id and set their valid field to 0. It again writes back to file with the changes.
   INPUT: int file id and int process id

6. Cleanup
   Used to reset the state of the system by deleting all the state files and temporary storage file.
   INPUT: Nothing
   OUTPUT: Nothing

Recovery Manager

It contains all the required recovery procedure calls. It handles all the required recovery situations such as hard faults (failure of disk etc) and soft faults (data corruption and file i/o failures). It also implements the two policies of recovery in 'ALL OR NOTHING' atomicity. It can either restore the state of a block before the commit operation or after the commit operation. Following are the procedure calls in the file:

1. Search directory

This method is used to check if a directory exists on the system. Use to check for hard faults when the entire disk is replaced and the directory tree is not present
INPUT: String directory name
OUTPUT: Returns 0,1,2 as status codes for present, not present and inaccessible

2. Hard recover
   Use to restore the cells in the cell storage from a remote backup copy that is maintained at checkpoints. Use to copy files from backup to cells directory.
   INPUT: Nothing
   OUTPUT: Status of system call
3. Check and fix
   Checks for consistency b/w the log entry and cell storage and if found inconsistent fixes the value using log entry. Then again checks for the entry. If due to I/O bandwidth the fix has not been completed it tries again. It tries for 3 times before giving up and returning error code.
   INPUT: Stuct entry
   OUTPUT: Returns 0,1 status codes
4. Check for error
   It reads the log file from the last check point entry and processes all commits (or RECOVER, based on policy) and send each log entry to Check and fix for checking for consistency. It also is responsible for checking for hard faults and restoring the state from backup.
   INPUT: int recovery policy
   OUTPUT: Returns status code

Log

It contains all the procedures to maintain the logging system of the program. It is responsible for creation, maintaining, checkpoint saves and reading of logs. Also a hybrid policy of maintenance of logs using checkpoints and logs have been implemented. We took the pros of the both maintenance mechanism and implemented it. Checkpoints are useful when there are hard faults. It contains the image of the cell storage system stored as a remote copy. So when the system fails we can recover the whole storage in a go. Pros are that it can restore system faster than logs and Cons is that it takes the same amount of space as the size for cell storage and cannot be run frequently due to large I/O requirements. Logs on the other hand can be used to restore the state by traversing through it. Pros is that it can keep track of the system to the event of system crash, thus can be used to revert just before crash and also more efficient in recovering soft faults as only faulty processes are rerun again. Cons are that it takes time to recover hard faults as it has to traverse throughout the system and also can become difficult to maintain after a considerable system runtime, will take up more space then check points/cell storage. So, we have implemented the hybrid which at the start of program reads from log (from the last checkpoint) and after consistency creates a checkpoint. After the checkpoint is created we can simply delete the previous log as its not required anymore. This gives us benefit in both hard and soft faults and thus is implemented is many high performance systems. The hybrid logging system contains the following procedure calls:

1. Checkpoint
   It uses to implement a system of checkpoints. It saves an image of the cell storage system as remote copy and clears up the logs. It is run only at the end of the logging system.
   INPUT: Nothing
   OUTPUT: Status codes
2. Logger
   Use to create and maintain an entry in the logging system.
   INPUT: int log_type, int p_id, int f_id, int cell_no, int value
   OUTPUT: Status codes
3. Log Reader
   Used to check the log entries. Used for debugging. Not required in the execution of the program.
   INPUT: Nothing
   OUTPUT: Status Codes

Test cases and reason for the choice

We have chosen the following test cases:

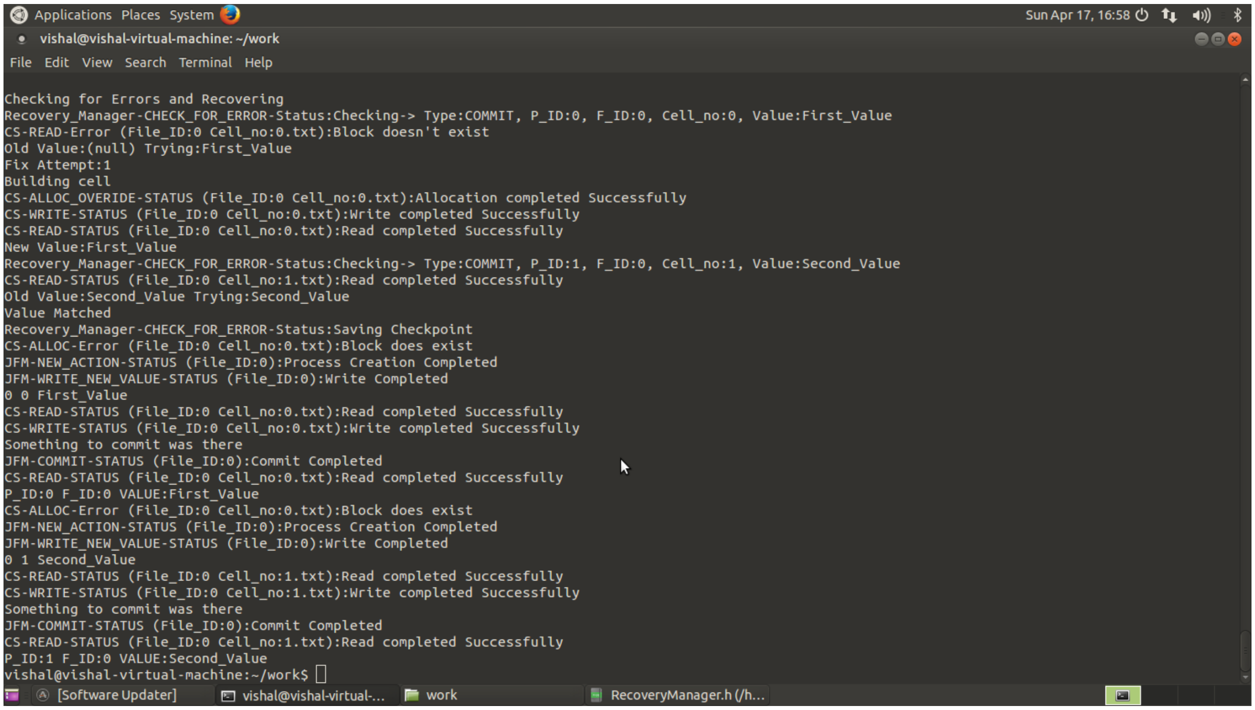
1. Fault tolerance on Soft Faults using ALL mechanism.
   In this case we are introducing some soft faults such as data corruption, I/O errors when writing to a cell storage. We are showcasing how the system deals with the faults using logs and recover till the last successful commit operation from log from the previous system run. In this we are corrupting (or deleting) the value in a cell unit and then we are running the recovery call at the beginning to handle such data corruption and provide us with consistent state of cell storage system. After the recovery we are maintaining a checkpoint and removing all the previous logs. Then we are proceeding on to normal operations on the consistent system.
   This test case checks the ability of the system from recovering from soft errors using log entries.
2. Fault tolerance on Soft Faults using NOTHING mechanism.
   In this case we are introducing some soft faults such as data corruption, I/O errors when writing to a cell storage. We are showcasing how the system deals with the faults using logs and recover till before the last state before the successful commit operations from log from the previous system run. In this we are corrupting (or deleting) the value in a

cell unit and then we are running the recovery call at the beginning to handle such data corruption and provide us with consistent state of cell storage system. After the recovery we are maintaining a checkpoint and removing all the previous logs. Then we are proceeding on to normal operations on the consistent system.
This test case checks the ability of the system to restore and abandon the last commit that could have been the problem of system crash using log entries.

3. Fault tolerance on Hard Error using ALL mechanism.
   In this case we are introducing some hard faults such as disk failures where we have to recover the complete storage system. We are showcasing how the system deals with the faults using logs and checkpoints and recover till the last successful commit operation from log and checkpoints from the previous system run. In this we are deleting the complete cell storage system (imitating a disk failure where we lose all our data) and then we are running the recovery call at the beginning to handle such disk failures and provide us with consistent state of cell storage system from last stored checkpoint and log entries. The storage is recovered from restoring the previous checkpoint and then traversing the log entries and making required changes to the file system. After the recovery we are maintaining a checkpoint and removing all the previous logs. Then we are proceeding on to normal operations on the consistent system.
   This test case checks the ability of the system from recovering from hard faults using checkpoints and log entries.

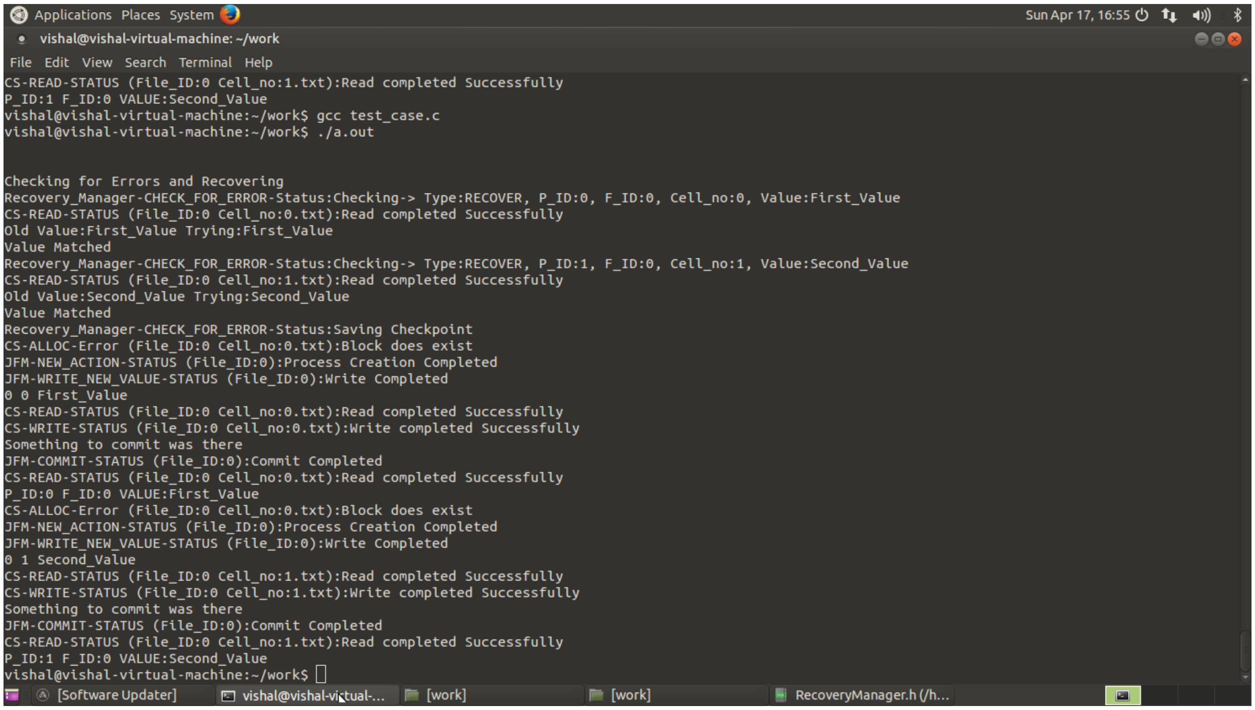
Test run screenshot and explanation

1. Test Case #1: Fault tolerance on Soft Faults using ALL mechanism.



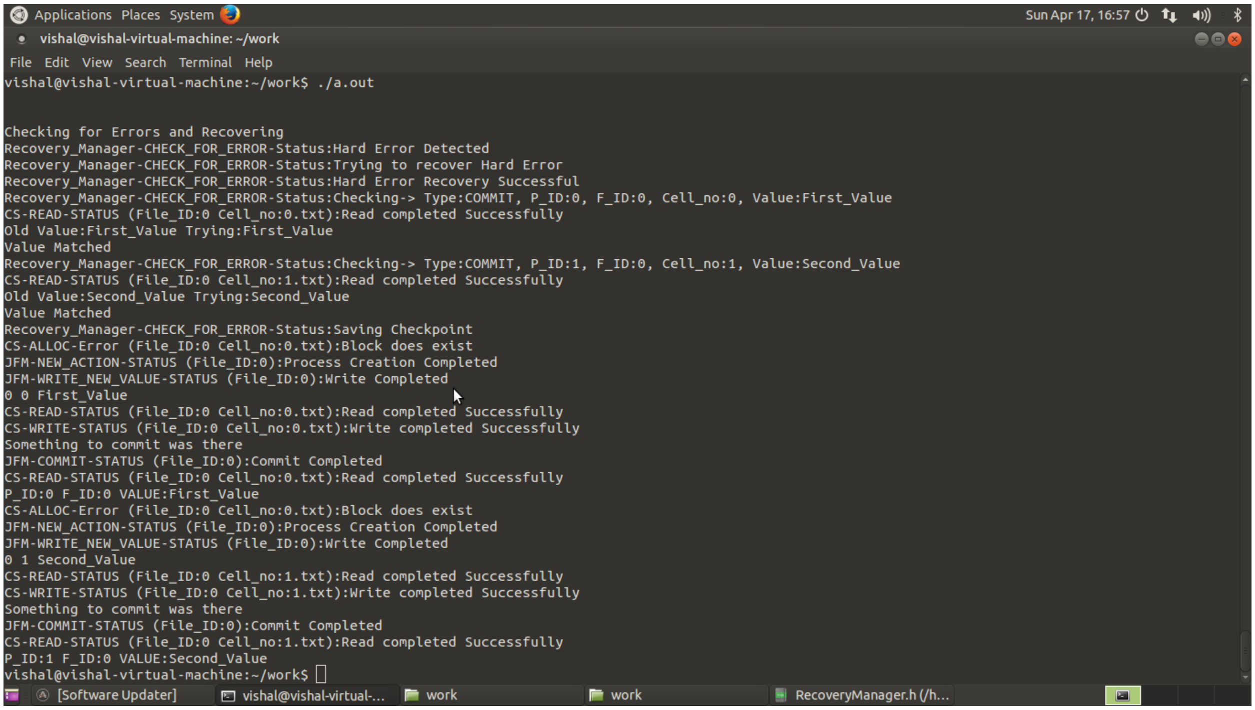
In the above test case as explained earlier we are testing for soft errors and recovering. In this case we are showcasing all mechanism in all-or-nothing atomicity. So first we are recovery and check to check and fix inconsistency b/w logs and cell storage. As in this case we see that values didn't matched (or block doesn't exist) inconsistency was found and fixed using logs (by creating block and storing value of the last commit). After traversing through the logs a storage image is created and checkpoint was created in log and previous log data is removed. We supposing that the operation is atomic and no corruption will happen during this phase.

2. Test Case #2: Fault tolerance on Soft Faults using NOTHING mechanism.



In the above test case as explained earlier we are testing for soft errors and recovering. In this case we are showcasing nothing mechanism in all-or-nothing atomicity. So first we are recovery and check to check and fix inconsistency b/w logs and cell storage. As in this case we see that values matched and no inconsistency was found with the value stored before the commit in the logs. After traversing through the logs a storage image is created and checkpoint was created in log and previous log data is removed. We supposing that the operation is atomic and no corruption will happen during this phase.

3. Test Case #3: Fault tolerance on Hard Faults using ALL mechanism.



In this case we have deleted the complete storage units and then we are trying to recover the cell storage unit using image created in the last checkpoint. So, as we see the system detected a hard fault (cell storage system missing) and recovered from a remote backup copy of the storage. Then it traverses the logs from the respective checkpoint and restores the consistency with the log. In this case values matched as it is already consistent from the previous run. It checks each cell value from the commit value in log. After that saves the checkpoint and resumes the system.

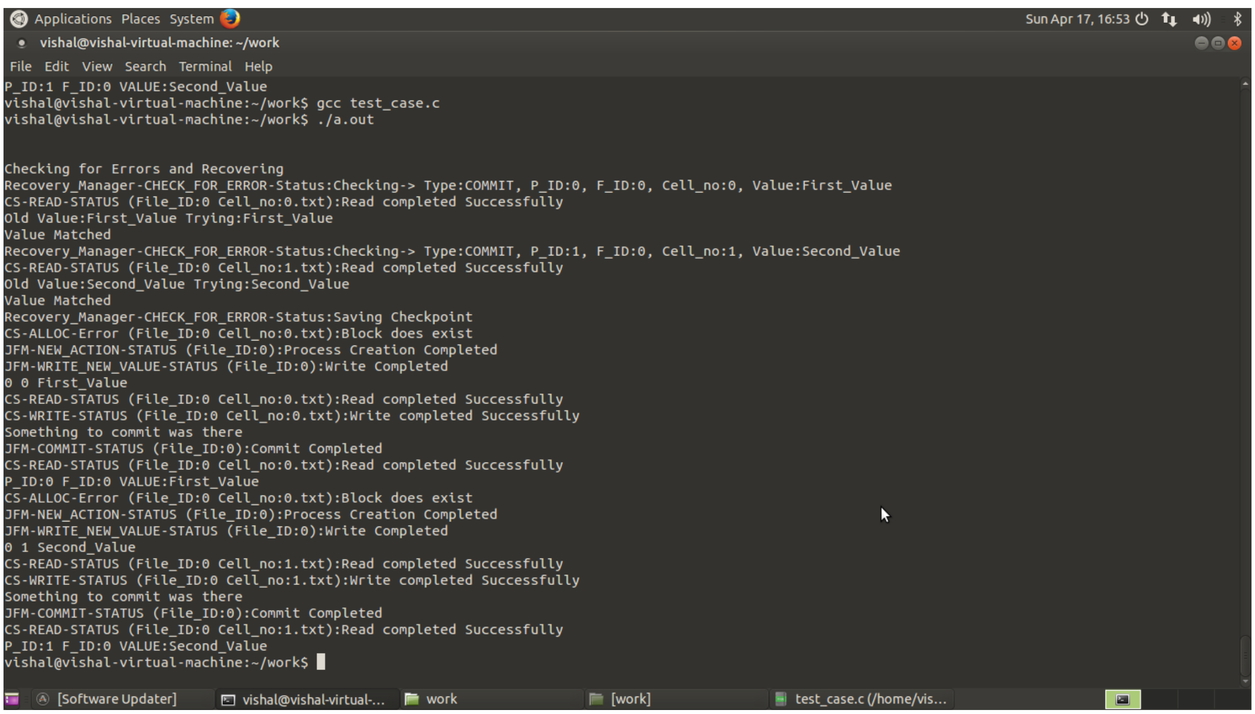
Results

We were able to successfully implement the cell storage and Journaling File system with logs and recovery mechanism through logs and checkpoints. The system was able to handle both hard and soft faults. It is a fail soft system which can more or less handle many kinds of faults such as device failure, I/O corruption, disk degradation, data corruption etc. The recovery mechanism is automatic which checks the system for kinds of fault and implement the recovery mechanism suited for the kind of fault. The recovery policy has to be hardcoded though. SO we can either choose to restore the previous state before the commit or state after the commit. This kind of flexibility helps it suitable for different scenario where a certain policy is more effective.

Discussion

So, the system was able to handle most of the foreseen faults which we noticed when we were having garbage value due to multithreading. The low I/O of storage device was probably the reason for that and using this system we probably can fix this issue. Though this system can increase the startup time of the files system due to recovery mechanism but still it provides us with consistent results which are very crucial in systems like databases. For now the recovery mechanism is slow and single threaded as we cannot guarantee the atomicity and correct result from the recovery mechanism. Also as it recovers from sequential executions of operations from the log, it's difficult to make the recovery mechanism multithreaded. We are giving consistency and correct result more importance than the performance and I think this is a better tradeoff in crucial system where data is more important than the performance of the system.

Output Screenshots of program running



Phase IV

Assumptions

Under the following assumptions the program works fine:

1. All access rights are available with the required directory structure.
2. Each cell holds no more than one data from a particular file.
3. Each cell block has a size of 128 bytes and can store value as strings.
4. Each cell is owned by a particular file and thus cannot be accessed or written by another file.
5. The program writes on one cell at a time with its associated file id.
6. The program can only store 100 cells in its temporary storage (state before commit).
7. All blocks/cells belonging to a certain file are committed simultaneously
8. Several cells/block of files need not be concurrent to each other.
9. Currently the system supports multithreading but now it's also fault tolerant
10. No two runs of test cases and made simultaneously
11. The directory 'cells' is always present but is cleared before each run. Maintained by cleanup ()
12. The transactions can be reinitiated.
13. The recovery will be error free
14. Every action is logged and securely and atomically
15. The execution is not hard real time
16. Recovery time overheads is acceptable
17. Can recover only recover from 100 erroneous commits at a time.

Implementation Procedure

Currently for Phase I, the whole program is divided into 5 main header files namely cellstorage.h, journal_file_manager.h, RecoveryManager.h, log.h and storageunits.h. A structure defined for storage and is written and read from files. Each text file under the cells folder is a cell block where structure block is written and read from. In order to support the multi-threading we have implemented two mutex locks: cell_file_lock and temp_file_lock to ensuring random but sequential access of cell storage and temp storage (as both depends on file I/O). Both locks have only been implemented in Journal File Manager. Certain changes in arguments were also done as each thread can only be given one object to be passed. So we are passing whole storage units here but only in operations related to journal file manager. Also the system checks for any hard or soft faults during the procedure and fixes that using the logs and checkpoints.

Cell Storage

The cell storage contains all the associated functions like IS_OWNER, CHECK_FOR_BLOCK, READ, WRITE, ALLOCATE, ALLOCATE_OVERIDE and DEALLOCATE.

1. IS_OWNER
   Use to check for the ownership for that particular block with the supplied file_id.
   INPUT: int file_id and int cells
   OUTPUT: Returns 1,2,3 for OK, Failure to read and Access violation
2. CHECK_FOR_BLOCK
   Use to check if a certain block is available in the cells directory or not.
   INPUT: String filename (block no)
   OUTPUT: Returns 0,1 for True and False in terms of availability
3. ALLOCATE
   Use to create cell blocks and assign respective file_id. Cell block no is automatic generated and a deallocated cell can't be used.
   INPUT: int file_id
   OUTPUT: int cell_no
4. ALLOCATE_OVERIDE
   Use to create cell blocks and assign respective file_id. Cell block no is manually assigned and a deallocated cell can be used.
   INPUT: int file_id, int cell_no
   OUTPUT: int cell_no
5. READ
   Use to read data from the block/cell associated with file id.
   INPUT: int file id, int process id (p_id) and int cell no
   OUTPUT: Returns string containing the data stored in the cell
6. WRITE
   Use to write data to the cell block with an associated file id and stores data in the cell.
   INPUT: int file id, int process id (p_id), int cell_no and string value_data
   OUTPUTS: Returns 0,1 for OK, and Error

Journaling File Manager

Uses the implemented storage block for the file system for temporary storage of data. It keeps a state for temporary storage of data before commit and keeps the data on a non-volatile storage. It acts as an abstraction layer over the Cell storage system. For multithreading here, we have implemented two mutex locks, one for temporary storage access (temp_file_lock) and another for the cell storage system(cell_file_lock). It uses the following programs:

1. NEW_ACTION
   Allocates a cell block to a particular file id and creates a process_id. Both locks have been implemented for read and write to temp storage to provide reliable temporary storage access (temp_file_lock) and maintaining consistency on cell_no allocation and updation (by cell_file_lock). Interestingly now cell_no are assigned randomly but correctly to each of the thread so we can observe some later threads (or even file) to acquire starting blocks (or cells) in the cell storage system. But still no unauthorized access is permitted
   INPUT: struct storage*
   OUTPUT: Returns struct storage*
2. READ_CURRENT_VALUE
   Reads a value from the cell storage with the associated cell and file id. Here only cell_file_lock has been implemented for maintaining a single access during read (no mutation possible during read).
   INPUT: struct storage *
   OUTPUT: Returns struct storage*.
3. COMMIT
   Use to commit data associated with a particular file id to the cell storage system. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for data for a particular file_id and process id and valid field. If valid is 1 then the data is committed for the associated file id and process id. After committing make the valid entry to 0. Both locks have been implemented on the block level (i.e. the whole working code) granularity in the function. This is done as we are getting garbage value comited due to I/O bandwidth limitations. Performance do suffer but we had to do it to maintain reliability of commit data when large no of threads are operating simultaneously.
   INPUT: struct storage *
   OUTPUT: returns pointer for 0,1 for OK and failure

4. WRITE_NEW_VALUE
   Used to store the data in the temporary storage before committing. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for new or invalid entry from start of the queue. Enters the data and set the valid field to 1, in the invalid or new storage block and saves it back to temp storage file. For multithreading we have implemented on temp_file_lock as we have only to maintain the reliability of the data in temp storage. The locks are only implemented for read and write of the temp storage.
   INPUT: struct storage*
   OUTPUT: Returns pointer for 0,1 for OK and failure

5. ABORT

Use to abort all the entry in the temporary storage associated with a particular process id and file id. It retrieves the data from the temp file and retrieves the temporary storage queue. Scans and Searches for all the possible entries associated with that process id and file id and set their valid field to 0. It again writes back to file with the changes. For multithreading we have implemented on temp_file_lock as we have only to maintain the reliability of the data in temp storage. The locks are only implemented for read and write of the temp storage.

INPUT: struct storage *
OUTPUT: Returns pointer for 0,1 for OK and failure

6. Cleanup
   Used to reset the state of the system by deleting all the state files and temporary storage file. As it occurs only after the end of the program no multithreading support is needed here.
   INPUT: Nothing
   OUTPUT: Nothing

Recovery Manager

It contains all the required recovery procedure calls. It handles all the required recovery situations such as hard faults (failure of disk etc) and soft faults (data corruption and file i/o failures). It also implements the two policies of recovery in 'ALL OR NOTHING' atomicity. It can either restore the state of a block before the commit operation or after the commit operation. Following are the procedure calls in the file:

1. Search directory
   This method is used to check if a directory exists on the system. Use to check for hard faults when the entire disk is replaced and the directory tree is not present
   INPUT: String directory name
   OUTPUT: Returns 0,1,2 as status codes for present, not present and inaccessible
2. Hard recover
   Use to restore the cells in the cell storage from a remote backup copy that is maintained at checkpoints. Use to copy files from backup to cells directory.
   INPUT: Nothing
   OUTPUT: Status of system call
3. Check and fix
   Checks for consistency b/w the log entry and cell storage and if found inconsistent fixes the value using log entry. Then again checks for the entry. If due to I/O bandwidth the fix has not been completed it tries again. It tries for 3 times before giving up and returning error code.

INPUT: Stuct entry
OUTPUT: Returns 0,1 status codes

4. Check for error
   It reads the log file from the last check point entry and processes all commits (or RECOVER, based on policy) and send each log entry to Check and fix for checking for consistency. It also is responsible for checking for hard faults and restoring the state from backup.
   INPUT: int recovery policy
   OUTPUT: Returns status code

Logs and Checkpoints

It contains all the procedures to maintain the logging system of the program. It is responsible for creation, maintaining, checkpoint saves and reading of logs. Also a hybrid policy of maintenance of logs using checkpoints and logs have been implemented. We took the pros of the both maintenance mechanism and implemented it. Checkpoints are useful when there are hard faults. It contains the image of the cell storage system stored as a remote copy. So when the system fails we can recover the whole storage in a go. Pros are that it can restore system faster than logs and Cons is that it takes the same amount of space as the size for cell storage and cannot be run frequently due to large I/O requirements. Logs on the other hand can be used to restore the state by traversing through it. Pros is that it can keep track of the system to the event of system crash, thus can be used to revert just before crash and also more efficient in recovering soft faults as only faulty processes are rerun again. Cons are that it takes time to recover hard faults as it has to traverse throughout the system and also can become difficult to maintain after a considerable system runtime, will take up more space then check points/cell storage. So, we have implemented the hybrid which at the start of program reads from log (from the last checkpoint) and after consistency creates a checkpoint. After the checkpoint is created we can simply delete the previous log as its not required anymore. This gives us benefit in both hard and soft faults and thus is implemented is many high performance systems. The hybrid logging system contains the following procedure calls:

1. Checkpoint
   It uses to implement a system of checkpoints. It saves an image of the cell storage system as remote copy and clears up the logs. It is run only at the end of the logging system.
   INPUT: Nothing
   OUTPUT: Status codes
2. Logger
   Use to create and maintain an entry in the logging system.
   INPUT: int log_type, int p_id, int f_id, int cell_no, int value

OUTPUT: Status codes

3. Log Reader

   Used to check the log entries. Used for debugging. Not required in the execution of the program.

   INPUT: Nothing

   OUTPUT: Status Codes

Test cases and reason for the choice

We have chosen the following test cases:

1. Multithreaded read write operation and fault correction by logs

   In this test case we check for multithreaded read and write functionality of the program with the commit and fixing it with logs with ALL mechanism. Here we are accessing 2 files each corresponding to a particular process (or file). Each file contains 20 blocks (or cells) of memory. So, total of 80 threads. All processes have NEW_ACTION (to allocate block (or cell) in cell storage and assign and return a process_id), WRITE_NEW_VALUE (to write data on the temporary storage block), COMMIT (to write all the temporary storage blocks to cell storage associated with a certain file_id and process_id) and READ_CURRENT_VALUE (to read and display the most recent update on a particular block (or cell) in the cell storage). This test case checks the normal operation of the file system under multithreaded environment. Though due to low I/O bandwidth of non-volatile storage sometimes the values are not written properly. So this I/O error has been fixed in this version of program where after it has written the values it checks with the logs and if any entry is mismatch then it fixes with the commit value of the log. Now it can support many threads (not more than 100) for I/O operation.

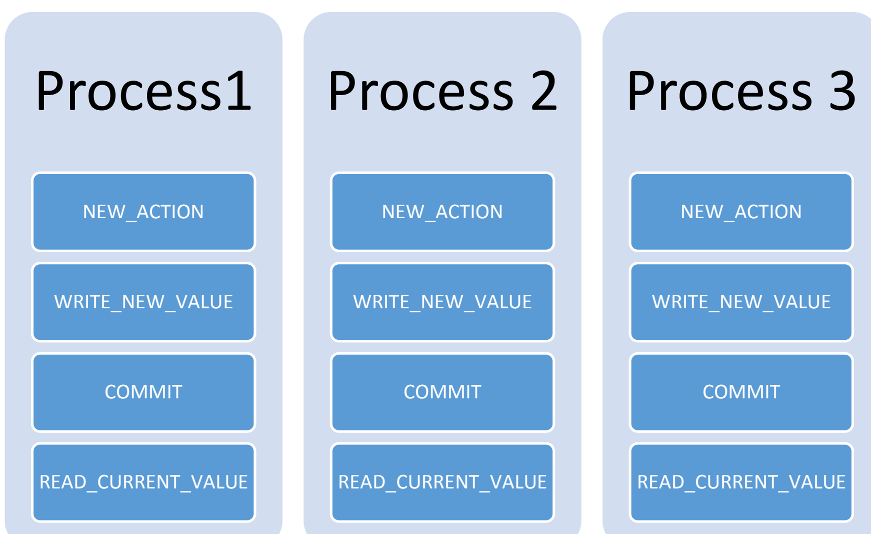


Figure 3:Example for three simultaneous Threads

2. Race condition on simultaneous access of a cell and fault correction by logs
   In this test case we are trying to introduce a race condition when two processes try to access the same block (or cell) simultaneously. For this we have first allocate a block (or cell) and then two process are trying to run the operation WRITE_NEW_VALUE, COMMIT and READ_CURRENT_VALUE in sequential order. As both threads are assigned the same file id and process id (had to do it manually) they both have access to cell 0. The result of this case is undefined as either of thread can come before or after during commit but not simultaneously. This is done two show before or after atomicity implemented in the journal file system. Also a check is made with logs entries which results in mismatch as we have to traverse through a sequence of commit statements. But the final value remains the same with the order of commit execution.

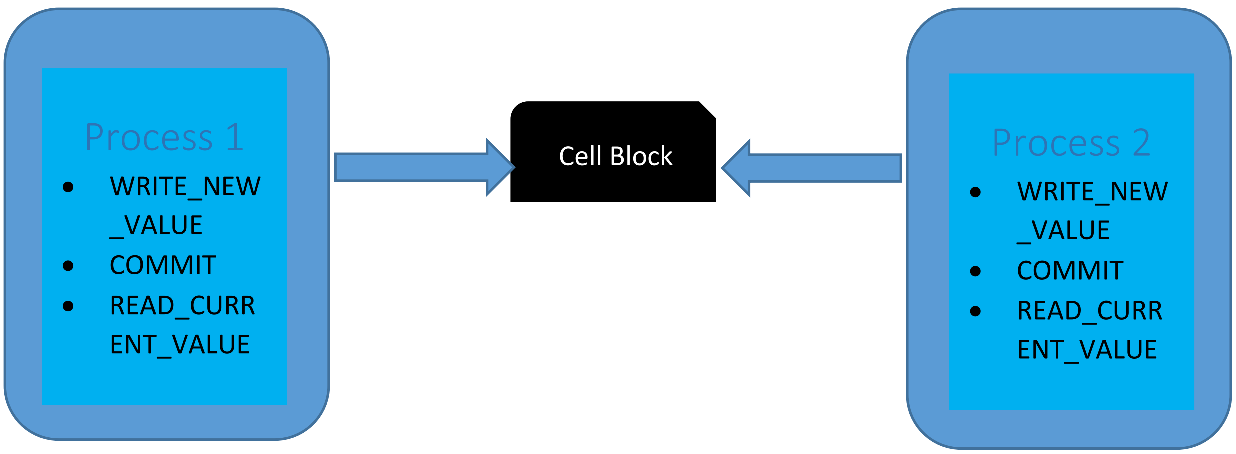
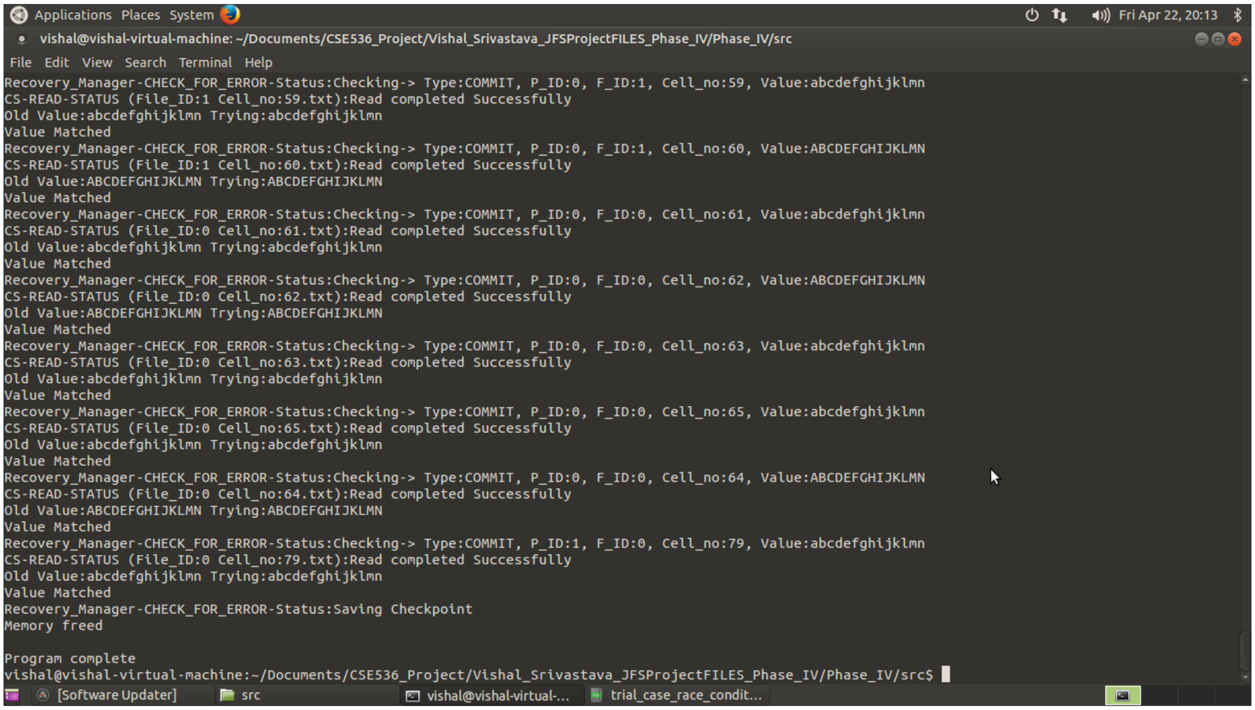


Figure 4: Simultaneous access of a cell by two processes

Test run screenshot and explanation

1. Test case I: Multithreaded read write operation with fault correction

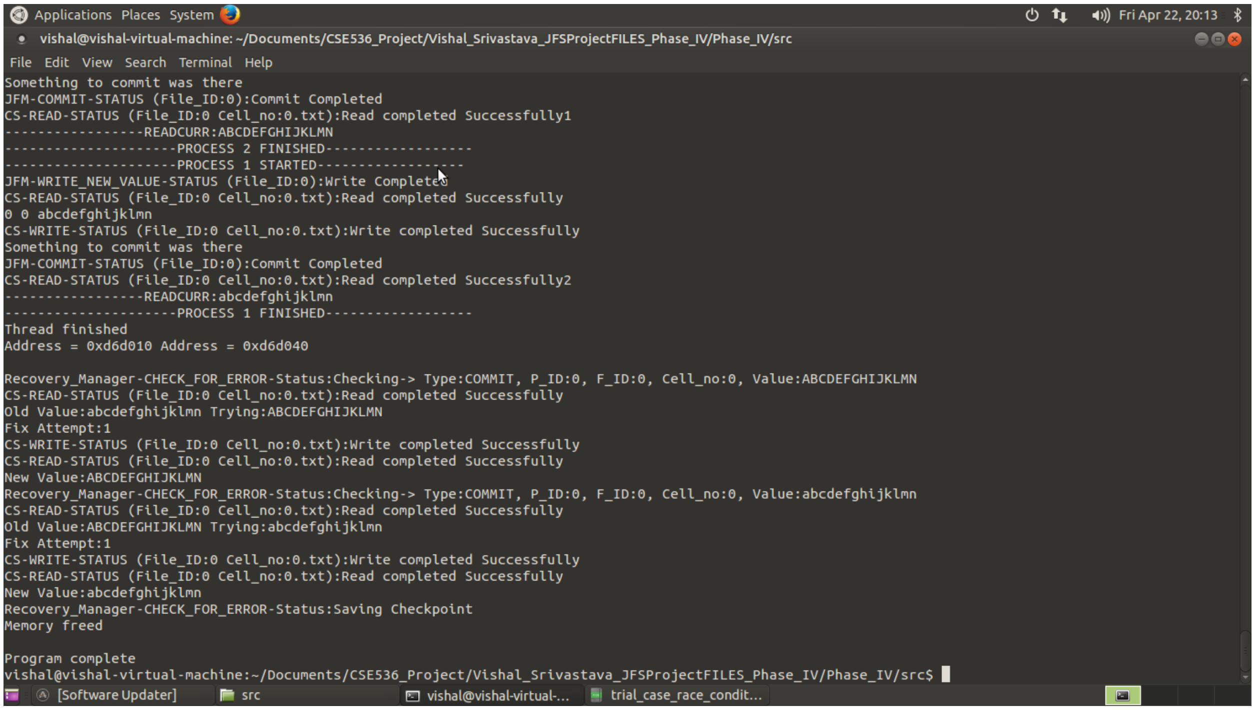


```
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:1, Cell_no:59, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:1 Cell_no:59.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:1, Cell_no:60, Value:ABCDEFGHIJKLMN
CS-READ-STATUS (File_ID:1 Cell_no:60.txt):Read completed Successfully
Old Value:ABCDEFGHIJKLMN Trying:ABCDEFGHIJKLMN
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:61, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:0 Cell_no:61.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:62, Value:ABCDEFGHIJKLMN
CS-READ-STATUS (File_ID:0 Cell_no:62.txt):Read completed Successfully
Old Value:ABCDEFGHIJKLMN Trying:ABCDEFGHIJKLMN
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:63, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:0 Cell_no:63.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:65, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:0 Cell_no:65.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:64, Value:ABCDEFGHIJKLMN
CS-READ-STATUS (File_ID:0 Cell_no:64.txt):Read completed Successfully
Old Value:ABCDEFGHIJKLMN Trying:ABCDEFGHIJKLMN
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:1, F_ID:0, Cell_no:79, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:0 Cell_no:79.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Saving Checkpoint
Memory freed

Program complete
vishal@vishal-virtual-machine:~/Documents/CSE536_Project/Vishal_Srivastava_JFSProjectFILES_Phase_IV/Phase_IV/src$
```

In this test case we are performing simultaneous multithreaded write to multiple cells. Here we are accessing 2 files each with 40 blocks (or cells). So, total of 80 threads. In this first we are allocating blocks using NEW_ACTION and then we are writing to temp blocks, committing them to cell storage blocks and then reading blocks for their value for cell storage. Here we have no introduced the race condition in access to the blocks but we have introduced race condition for allocation of blocks during NEW_ACTION. We can observe that due to multi-threading the allocation of the blocks are done in random order but no two files have been assigned the same block. Also the execution in out of order as it depends on thread scheduler as to who is to be executed. Due to which we observe an out of order execution. Also as the whole process (only the parts within are atomic) is not made atomic, so the execution in showing mixed random order. But within the process the execution is sequential. Also access to temp file storage and cell storage are made random yet sequential so we have no problem in I/O as no two process are assigned the same process_id and file_id. Later after completion of execution of each of the threads we are doing a consistency check (as shown here) with the logs and if any error found hard or soft the log and checkpoint (in case of hard fault) will fix these errors (tries four times) and the problem of inconsistent commits in Phase II was fixed due to this.

2. Test Case II: Race condition on simultaneous access of a cell and fault correction with logs



In this test we also observe the randomness due to thread scheduler. As it was difficult to produce such thing (due to generation of new blocks by NEW_ACTION) in the earlier program we have made this as a certain test case. Here we have two threads competing for the same block in the cell storage. First we have already allocated a block (cell 0) and then we creating 2 threads both performing WRITE_NEW_VALUE, COMMIT, READ_CURRENT_VALUE with different values as ‘abcdefghijklmn’ and ‘ABCDEFGHIJKLMN’. So even if we start both thread simultaneously, it’s not certain that a particular thread will end before the other. The two test runs of this test case shows the same effect. In the first figure process 1 finished earlier than process 2 and value ‘abcdefghijklmn’ was stored finally but it was vice versa in the other and value ‘ABCDEFGHIJKLMN’ was stored finally. It’s a random process and it difficult to predict which one will get completed first. Later on the we do a check for consistency with the logs and apply the same operations in order as of commits in the logs and we reach a consistent state with the logs

Results

We were able to implement multithreading with before or after atomicity. The implementation was done by mutex locks. Until a process is killed it was ensured that there won't be any deadlocks in the design. Also from the above test cases we have succeed in implementing the before or after atomicity. We also show the performance achieved due to required granularity for different operations. Also due to locks now we can support any no. of threads/files simultaneously and due to fault tolerant mechanism we are assured to get the correct result.

Discussion

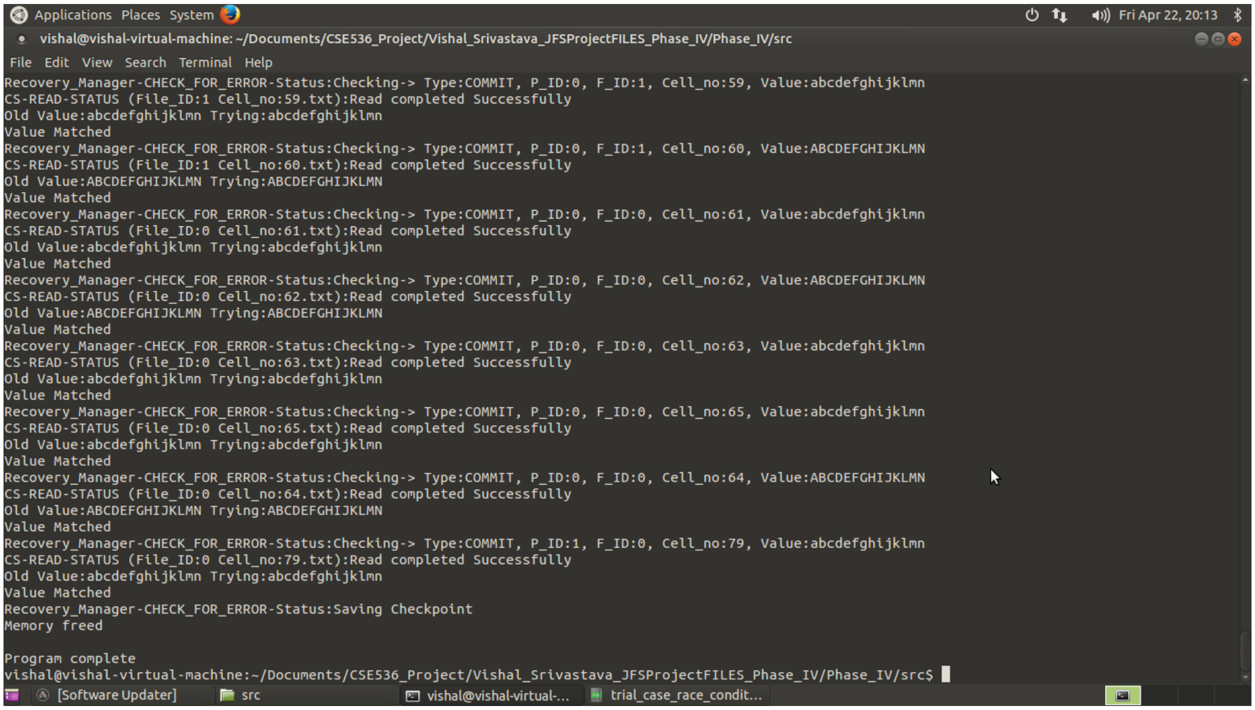
As the whole file system both temporary and cell storage is implemented in the non-volatile storage using files as blocks so in case of failure only a certain segment will be lost but the downside to this implementation is that this system will be slow for read and writes. This kind of system is generally observed in the database management system where failure will be catastrophic. So we went with this type of implementations. While multithreading we also observed certain limitation of the design.

As the temporary storage was on non-volatile storage we do get robustness and fault tolerance as we have to only recover the last mutation during a failure but due to this the I/O performance had decreased. So if we are trying to have multiple threads access the temporary storage we were getting random mutations (garbage values) which we didn't accounted for. Earlier to fix this issue we made the COMMIT operation atomic and increased the granularity. We also restricted threads to 20 threads as they gave correct results. Though system should be able to handle more threads but the generation of garbage value was not well understood. It seemed due to disk I/O operation.

As of now system is fault tolerant so the above problem we have encountered is now fixed and thus we have shown the fault tolerant mechanism on 80 threads in comparison to 20 threads. The mechanism can support even more threads but as temp storage is limited to 100 blocks (for faster I/Os) we are going with 80 threads right now.

We have assumed that fault checks at beginning and end of program are acceptable as it is in large database system. These fault checks are necessary and needs to be done by in order scanning of logs. So any kind of performance boost is difficult to implement without adding any cache mechanism to it.

Output Screenshots of program running


```
Applications Places System                                    Fri Apr 22, 20:13
vishal@vishal-virtual-machine: ~/Documents/CSE536_Project/Vishal_Srivastava_JFSProjectFILES_Phase_IV/Phase_IV/src
File Edit View Search Terminal Help
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:1, Cell_no:59, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:1 Cell_no:59.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:1, Cell_no:60, Value:ABCDEFGHIJKLMN
CS-READ-STATUS (File_ID:1 Cell_no:60.txt):Read completed Successfully
Old Value:ABCDEFGHIJKLMN Trying:ABCDEFGHIJKLMN
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:61, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:0 Cell_no:61.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:62, Value:ABCDEFGHIJKLMN
CS-READ-STATUS (File_ID:0 Cell_no:62.txt):Read completed Successfully
Old Value:ABCDEFGHIJKLMN Trying:ABCDEFGHIJKLMN
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:63, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:0 Cell_no:63.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:65, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:0 Cell_no:65.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:64, Value:ABCDEFGHIJKLMN
CS-READ-STATUS (File_ID:0 Cell_no:64.txt):Read completed Successfully
Old Value:ABCDEFGHIJKLMN Trying:ABCDEFGHIJKLMN
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:1, F_ID:0, Cell_no:79, Value:abcdefghijklmn
CS-READ-STATUS (File_ID:0 Cell_no:79.txt):Read completed Successfully
Old Value:abcdefghijklmn Trying:abcdefghijklmn
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Saving Checkpoint
Memory freed

Program complete
vishal@vishal-virtual-machine:~/Documents/CSE536_Project/Vishal_Srivastava_JFSProjectFILES_Phase_IV/Phase_IV/src$
[Software Updater]   src   vishal@vishal-virtual-...   trial_case_race_condit...
```

Design Flow

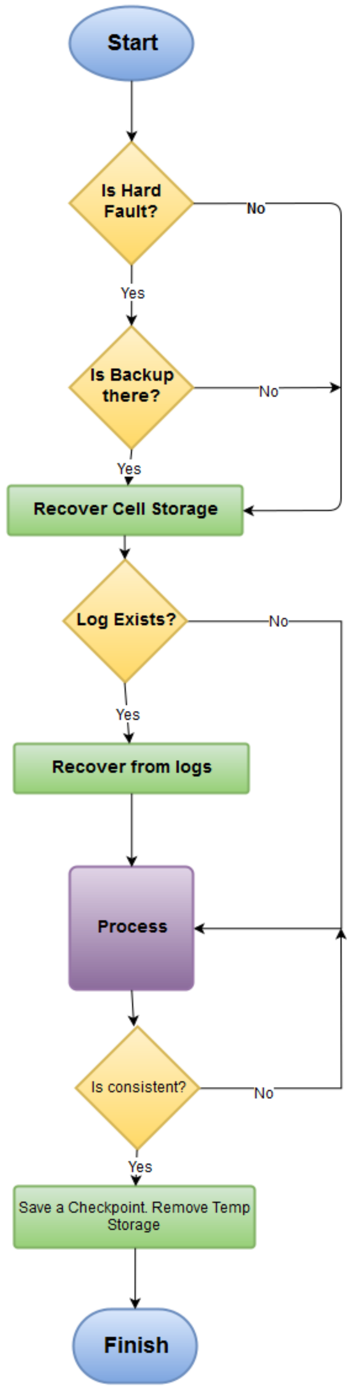


Figure 5:Program Flow

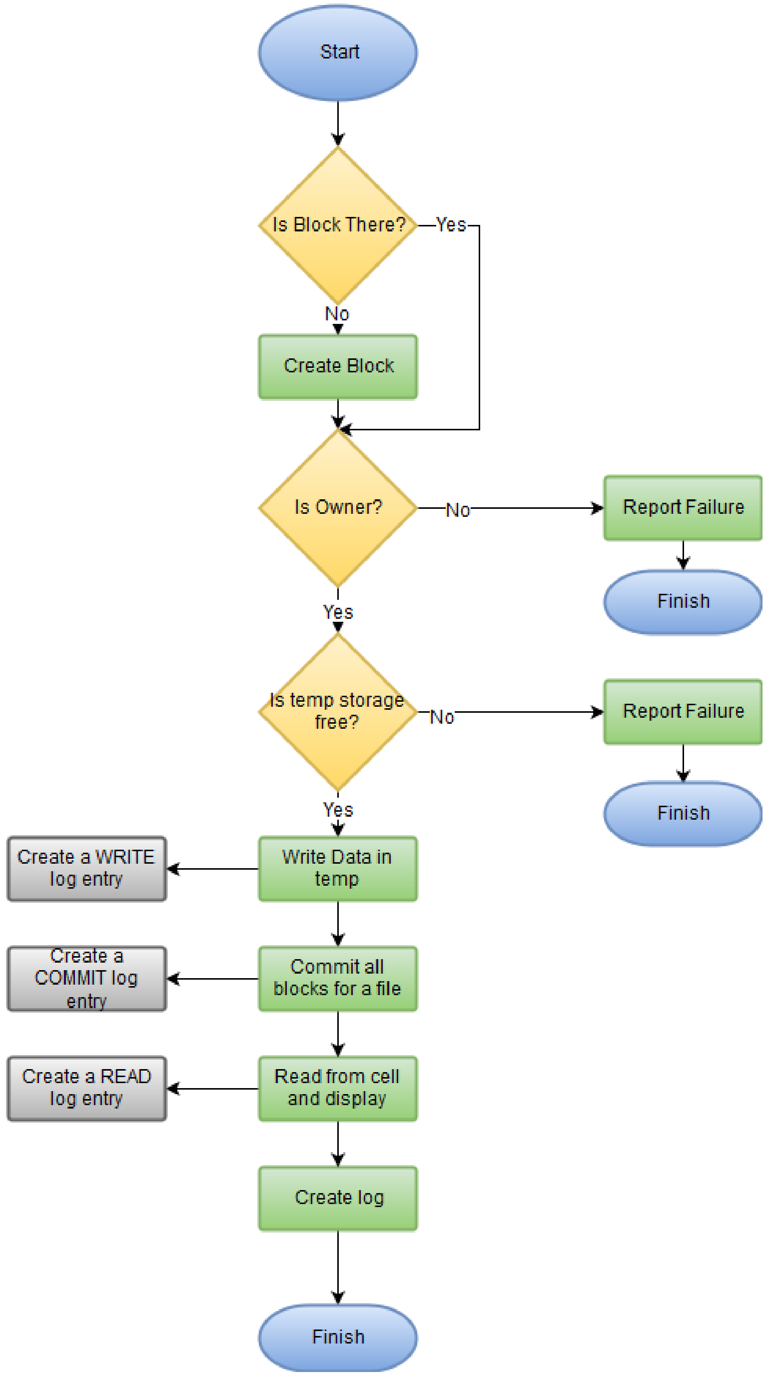


Figure 6: Process Flow

Summary

The Journaling File system is one of the high performance fault tolerant file system on which many of the latest filesystem such as ext4 is based on. This kind of file storage mechanism can be used in handling many of the disks simultaneously making it scalable. As this in memory kind of system are specifically useful in many of the designs where keeping the data in the volatile storage is not feasible. There is always a tradeoff in this kind of system where reliability and scalability are given higher precedence over performance. In large enterprises where these systems are generally used, reliability is very crucial as a system going offline can turn into huge losses for the company. So redundant copies, Logs are a small price to pay for reliability that it offers. The downside in performance are generally met using multiple server to offload some independent parts of the work. The system is evolving day by day and probably this system might get outdated soon but the design considerations will always be present in upcoming file systems.

Conclusion

As we completed though the above phases in our program. We went from basic Cell storage and Journal File manager to a Performance intensive multithreaded implementation of Journal File Manager to Fault Tolerant Journal File Manager to now Multithreaded Journal File Manager with Fault tolerance. We tried to have a design as close to a high performance distributed database system and we were able to achieve it to an extent. Only the volatile caching mechanism was not implemented due to complexity and time involved in designing it. Still on the performance side we have multithreading in it which can do multiple commits simultaneously. If suppose the cell blocks and distributed on different disks, we can achieve a state of parallelism during commits to hard disks which could give us more high performance. Also with fault tolerant mechanism we have achieved state of consistency that may have got violated due to I/O errors, Hard faults and many other similar situations. The cost of reading and maintaining a log and checkpoint is supposed to be minimal as they are maintained on a different disk altogether so no performance bottleneck should be visible. Also due to being a hybrid log and checkpoint mechanism we have dealt with hard faults too that are often taken care in high performance systems. So overall system can be used to implement a fault tolerant database system on top of it.

Lessons Learned

We went through many of the design of databases and studied how they manage to keep performance and also be fault tolerant. We understood some of the design principles of atomicity such as ALL or Nothing and Before or after atomicity that have been used to implement fault tolerant and synchronism mechanisms respectively. We went through some of the design principle of implementing on ALL or Nothing such as circular linklist based commits and log based commits.

We also went through a paper which discusses the rollback recover concepts where we learned on different checkpoint based recovery mechanisms such as independent checkpoint, coordinated checkpoint and communication-induced checkpoints. The paper also discussed various log based recovery patterns such as optimistic logging, pessimistic logging and causal logging. We also tried to understand different contexts of applications and where will a certain design is most suitable. We also went with the problems associated with each kind of designs such as distributed designs, embedded systems design and other kinds of designs and problems associated with the designs in different types of application and their consequences on the system. It also discusses some of the solutions for each of the designs in different contexts. The hybrid logging is a part of the causal logging technique discussed.

References

[1] Titos Saridakis. Design Patterns for Log-Based Rollback Recovery, 2nd Nordic Conference on Pattern Languages of Programs (VikingPLoP), September 15, 2003.

[2] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[3] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.