# Project Report

# CSE 536

Name: Vishal Srivastava ID No.:1209824652

## 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:

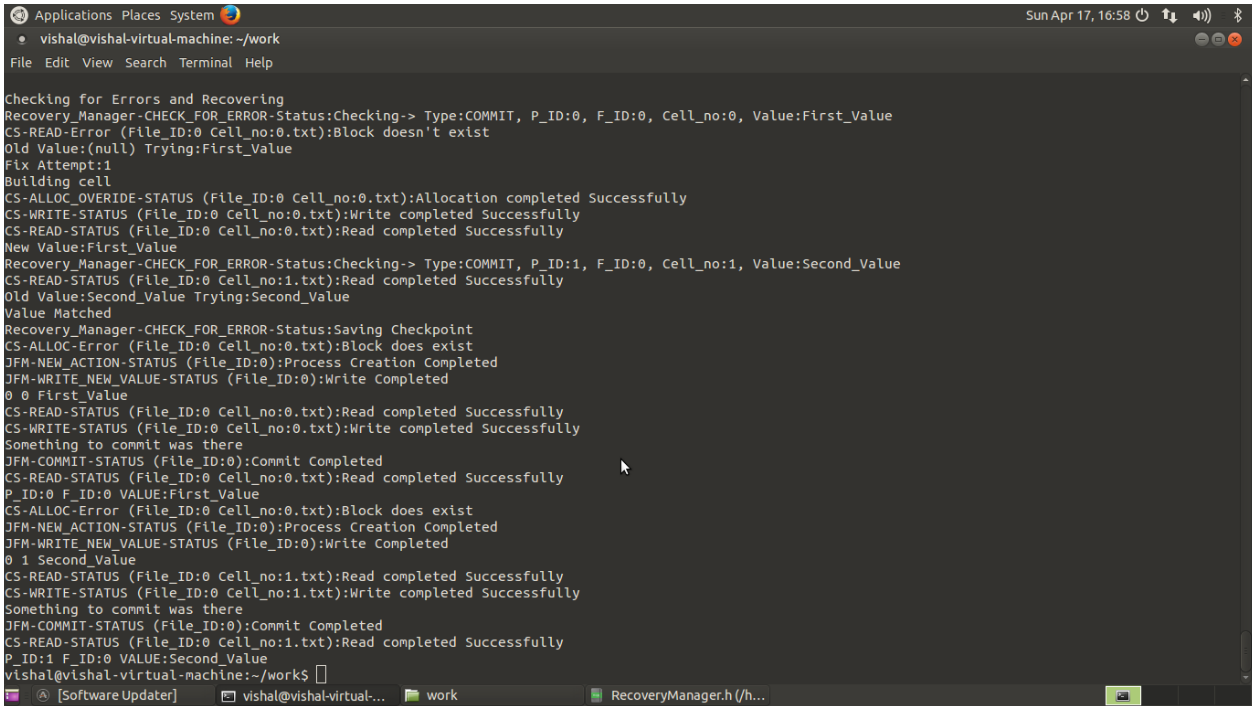
1. Fault tolerance on Soft Faults using ALL mechanism.
   In this case we are introducing some soft faults such as data corruption, I/O errors when writing to a cell storage. We are showcasing how the system deals with the faults using logs and recover till the last successful commit operation from log 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

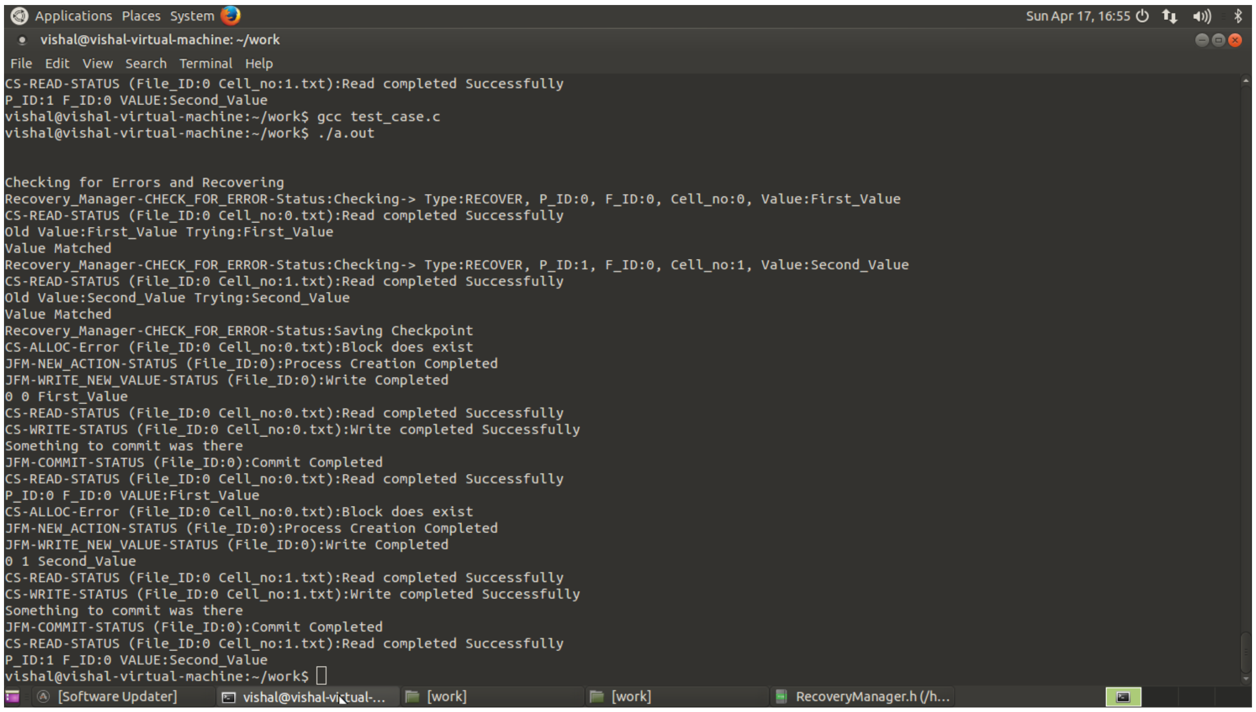
1. Test Case #1: Fault tolerance on Soft Faults using ALL mechanism.


Applications Places System
Sun Apr 17, 16:58
vishal@vishal-virtual-machine: ~/work
File Edit View Search Terminal Help
[Software Updater]
vishal@vishal-virtual-...
work
RecoveryManager.h (/h...

```
Checking for Errors and Recovering
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:0, Value:First_Value
CS-READ-Error (File_ID:0 Cell_no:0.txt):Block doesn't exist
Old Value:(null) Trying:First_Value
Fix Attempt:1
Building cell
CS-ALLOC_OVERIDE-STATUS (File_ID:0 Cell_no:0.txt):Allocation completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:0.txt):Write completed Successfully
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
New Value:First_Value
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:1, F_ID:0, Cell_no:1, Value:Second_Value
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
Old Value:Second_Value Trying:Second_Value
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Saving Checkpoint
CS-ALLOC-Error (File_ID:0 Cell_no:0.txt):Block does exist
JFM-NEW_ACTION-STATUS (File_ID:0):Process Creation Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 0 First_Value
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:0.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:0):Commit Completed
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
P_ID:0 F_ID:0 VALUE:First_Value
CS-ALLOC-Error (File_ID:0 Cell_no:0.txt):Block does exist
JFM-NEW_ACTION-STATUS (File_ID:0):Process Creation Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 1 Second_Value
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:1.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:0):Commit Completed
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
P_ID:1 F_ID:0 VALUE:Second_Value
vishal@vishal-virtual-machine:~/work$
```

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.

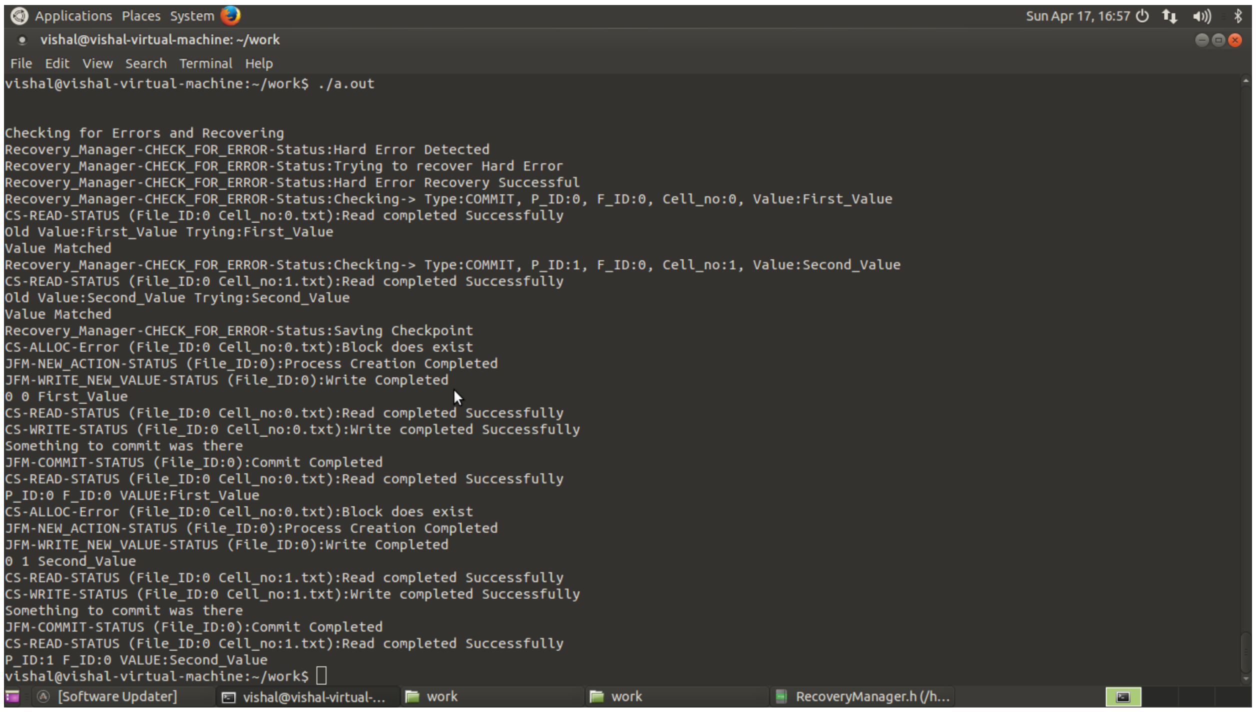

Applications Places System
Sun Apr 17, 16:55
vishal@vishal-virtual-machine: ~/work
File Edit View Search Terminal Help
[Software Updater]
vishal@vishal-virtual-...
[work]
[work]
RecoveryManager.h (/h...

```
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
P_ID:1 F_ID:0 VALUE:Second_Value
vishal@vishal-virtual-machine:~/work$ gcc test_case.c
vishal@vishal-virtual-machine:~/work$ ./a.out


Checking for Errors and Recovering
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:RECOVER, P_ID:0, F_ID:0, Cell_no:0, Value:First_Value
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
Old Value:First_Value Trying:First_Value
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:RECOVER, P_ID:1, F_ID:0, Cell_no:1, Value:Second_Value
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
Old Value:Second_Value Trying:Second_Value
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Saving Checkpoint
CS-ALLOC-Error (File_ID:0 Cell_no:0.txt):Block does exist
JFM-NEW_ACTION-STATUS (File_ID:0):Process Creation Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 0 First_Value
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:0.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:0):Commit Completed
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
P_ID:0 F_ID:0 VALUE:First_Value
CS-ALLOC-Error (File_ID:0 Cell_no:0.txt):Block does exist
JFM-NEW_ACTION-STATUS (File_ID:0):Process Creation Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 1 Second_Value
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:1.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:0):Commit Completed
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
P_ID:1 F_ID:0 VALUE:Second_Value
vishal@vishal-virtual-machine:~/work$
```

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.


Applications Places System
Sun Apr 17, 16:57
vishal@vishal-virtual-machine: ~/work
File Edit View Search Terminal Help
[Software Updater]
vishal@vishal-virtual-...
work
work
RecoveryManager.h (/h...

```
vishal@vishal-virtual-machine:~/work$ ./a.out


Checking for Errors and Recovering
Recovery_Manager-CHECK_FOR_ERROR-Status:Hard Error Detected
Recovery_Manager-CHECK_FOR_ERROR-Status:Trying to recover Hard Error
Recovery_Manager-CHECK_FOR_ERROR-Status:Hard Error Recovery Successful
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:0, Value:First_Value
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
Old Value:First_Value Trying:First_Value
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:1, F_ID:0, Cell_no:1, Value:Second_Value
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
Old Value:Second_Value Trying:Second_Value
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Saving Checkpoint
CS-ALLOC-Error (File_ID:0 Cell_no:0.txt):Block does exist
JFM-NEW_ACTION-STATUS (File_ID:0):Process Creation Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 0 First_Value
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:0.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:0):Commit Completed
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
P_ID:0 F_ID:0 VALUE:First_Value
CS-ALLOC-Error (File_ID:0 Cell_no:0.txt):Block does exist
JFM-NEW_ACTION-STATUS (File_ID:0):Process Creation Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 1 Second_Value
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:1.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:0):Commit Completed
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
P_ID:1 F_ID:0 VALUE:Second_Value
vishal@vishal-virtual-machine:~/work$ 
```

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.

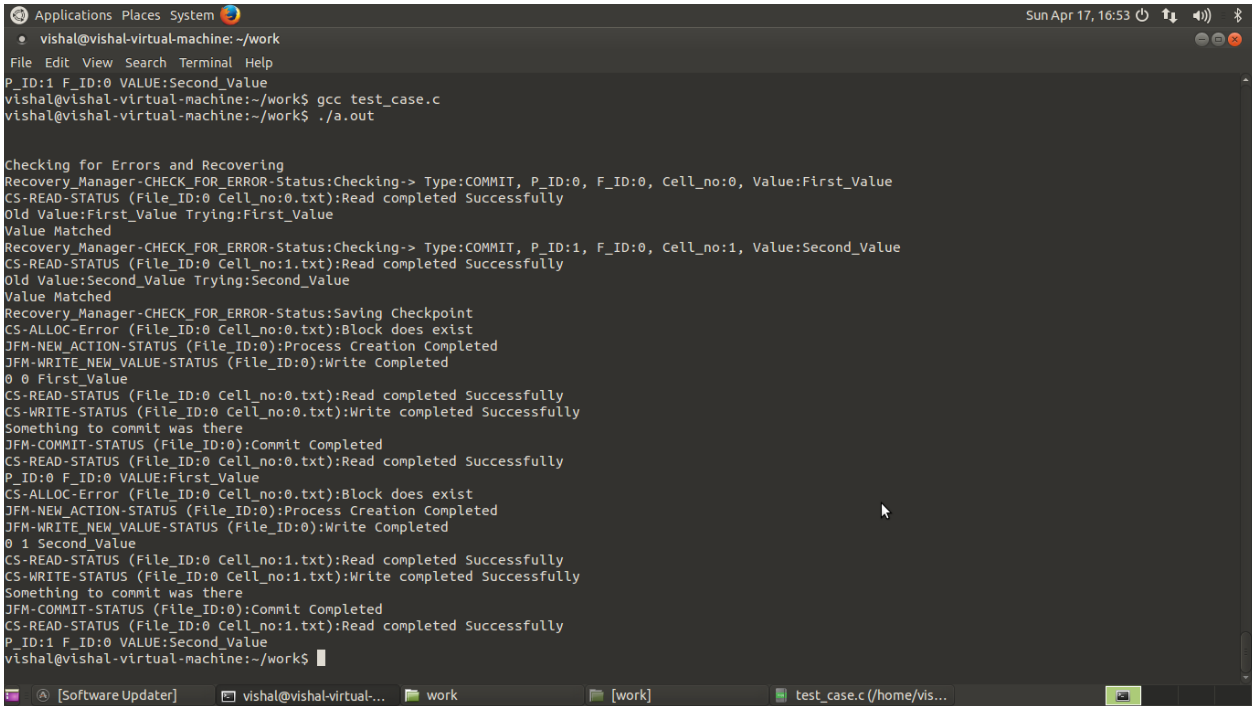
## Results

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


Applications Places System
Sun Apr 17, 16:53
vishal@vishal-virtual-machine: ~/work
File Edit View Search Terminal Help
[Software Updater]
vishal@vishal-virtual-...
work
[work]
test_case.c (/home/vis...

```
P_ID:1 F_ID:0 VALUE:Second_Value
vishal@vishal-virtual-machine:~/work$ gcc test_case.c
vishal@vishal-virtual-machine:~/work$ ./a.out


Checking for Errors and Recovering
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:0, F_ID:0, Cell_no:0, Value:First_Value
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
Old Value:First_Value Trying:First_Value
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Checking-> Type:COMMIT, P_ID:1, F_ID:0, Cell_no:1, Value:Second_Value
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
Old Value:Second_Value Trying:Second_Value
Value Matched
Recovery_Manager-CHECK_FOR_ERROR-Status:Saving Checkpoint
CS-ALLOC-Error (File_ID:0 Cell_no:0.txt):Block does exist
JFM-NEW_ACTION-STATUS (File_ID:0):Process Creation Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 0 First_Value
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:0.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:0):Commit Completed
CS-READ-STATUS (File_ID:0 Cell_no:0.txt):Read completed Successfully
P_ID:0 F_ID:0 VALUE:First_Value
CS-ALLOC-Error (File_ID:0 Cell_no:0.txt):Block does exist
JFM-NEW_ACTION-STATUS (File_ID:0):Process Creation Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 1 Second_Value
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
CS-WRITE-STATUS (File_ID:0 Cell_no:1.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:0):Commit Completed
CS-READ-STATUS (File_ID:0 Cell_no:1.txt):Read completed Successfully
P_ID:1 F_ID:0 VALUE:Second_Value
vishal@vishal-virtual-machine:~/work$ 
```