

Project Report Phase II

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 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.

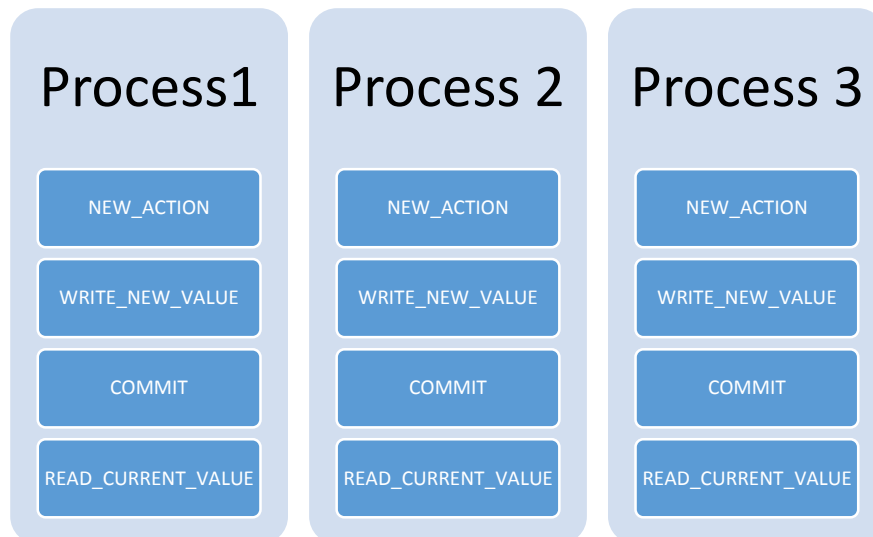


Figure 1: Example for three simultaneous Threads

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 allocated a block (or cell) and then two processes 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 to show before or after atomicity implemented in the journal file system.

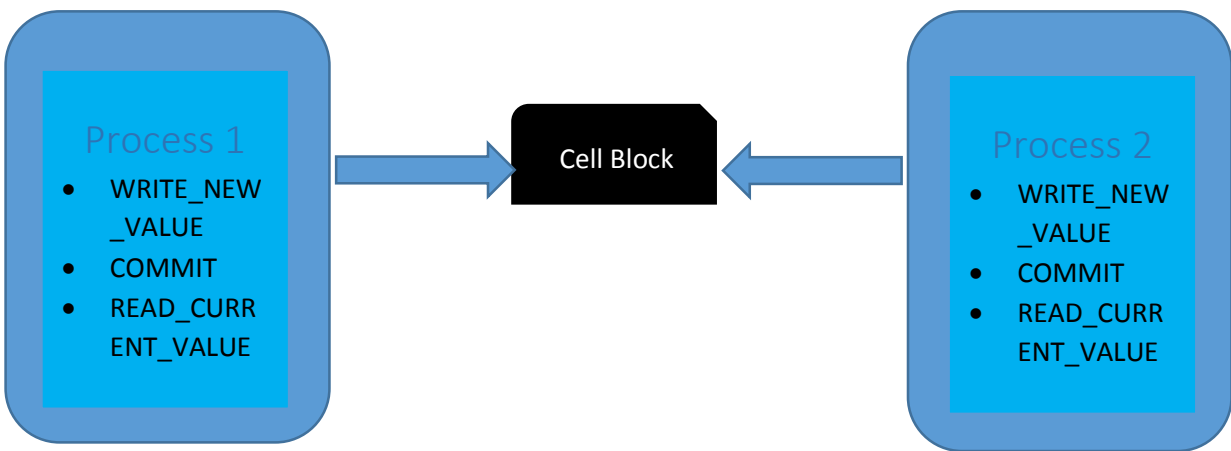
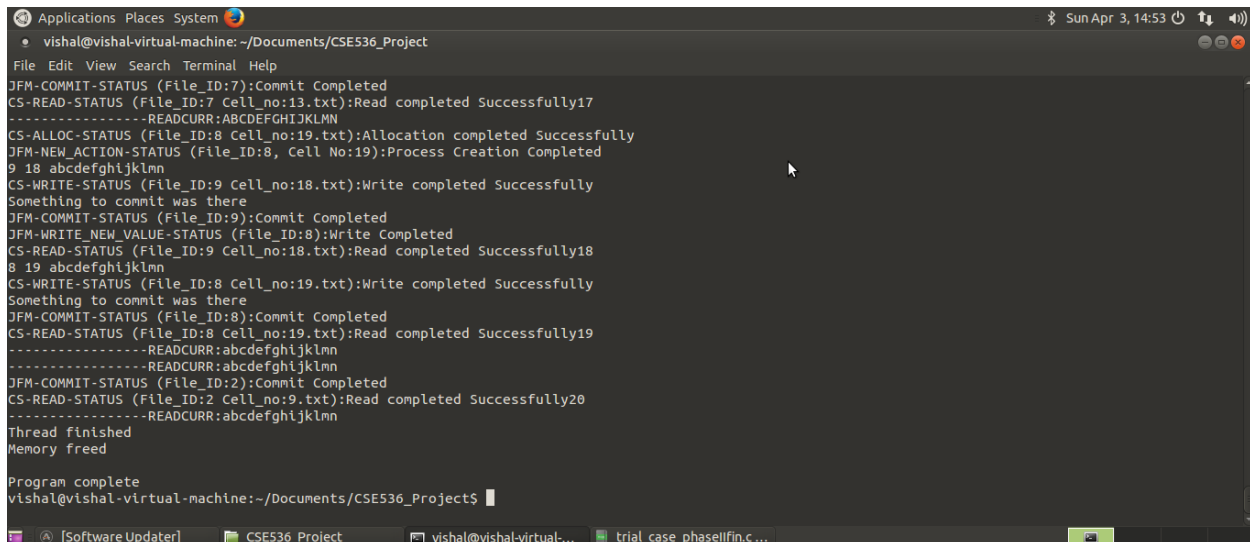


Figure 2: Simultaneous access of a cell by two processes

Test run screenshot and explanation

1. Test case I: Multithreaded read write operation



```
Applications Places System
vishal@vishal-virtual-machine: ~/Documents/CSE536_Project
File Edit View Search Terminal Help
JFM-COMMIT-STATUS (File_ID:7):Commit Completed
CS-READ-STATUS (File_ID:7 Cell_no:13.txt):Read completed Successfully17
-----READCURR:ABCDEFGHIJKLMN
CS-ALLOC-STATUS (File_ID:8 Cell_no:19.txt):Allocation completed Successfully
JFM-NEW_ACTION-STATUS (File_ID:8, Cell No:19):Process Creation Completed
9 18 abcdefghijklmn
CS-WRITE-STATUS (File_ID:9 Cell_no:18.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:9):Commit Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:8):Write Completed
CS-READ-STATUS (File_ID:9 Cell_no:18.txt):Read completed Successfully18
8 19 abcdefghijklmn
CS-WRITE-STATUS (File_ID:8 Cell_no:19.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:8):Commit Completed
CS-READ-STATUS (File_ID:8 Cell_no:19.txt):Read completed Successfully19
-----READCURR:abcdefghijklmn
-----READCURR:abcdefghijklmn
JFM-COMMIT-STATUS (File_ID:2):Commit Completed
CS-READ-STATUS (File_ID:2 Cell_no:9.txt):Read completed Successfully20
-----READCURR:abcdefghijklmn
Thread finished
Memory freed
Program complete
vishal@vishal-virtual-machine:~/Documents/CSE536_Project$
```

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 is 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 is 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

```
Applications Places System
vishal@vishal-virtual-machine: ~/Documents/CSE536_Project
File Edit View Search Terminal Help
Thread create
-----PROCESS 1 STARTED-----
-----PROCESS 2 STARTED-----
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 0 ABCDEFGHIJKLMN
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 Successfully1
-----READCURR:ABCDEFGHIJKLMN
-----PROCESS 2 FINISHED-----
Thread started
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 0 abcdefghijklmn
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 Successfully2
-----READCURR:abcdefghijklmn
-----PROCESS 1 FINISHED-----
Thread finished
Address = 35500048 Address = 35500096
Memory freed
Program complete
vishal@vishal-virtual-machine:~/Documents/CSE536_Project$
```

```
Applications Places System
vishal@vishal-virtual-machine: ~/Documents/CSE536_Project/Vishal_Srivastava_JFSProjectFILES_Phase_I/Phase_II/src
File Edit View Search Terminal Help
Thread init
Thread create
-----PROCESS 1 STARTED-----
Thread started
-----PROCESS 2 STARTED-----
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 0 abcdefghijklmn
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 Successfully1
-----READCURR:abcdefghijklmn
-----PROCESS 1 FINISHED-----
JFM-WRITE_NEW_VALUE-STATUS (File_ID:0):Write Completed
0 0 ABCDEFGHIJKLMN
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 Successfully2
-----READCURR:ABCDEFGHIJKLMN
-----PROCESS 2 FINISHED-----
Thread finished
Address = 0x6cd010 Address = 0x6cd040
Memory freed
Program complete
```

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

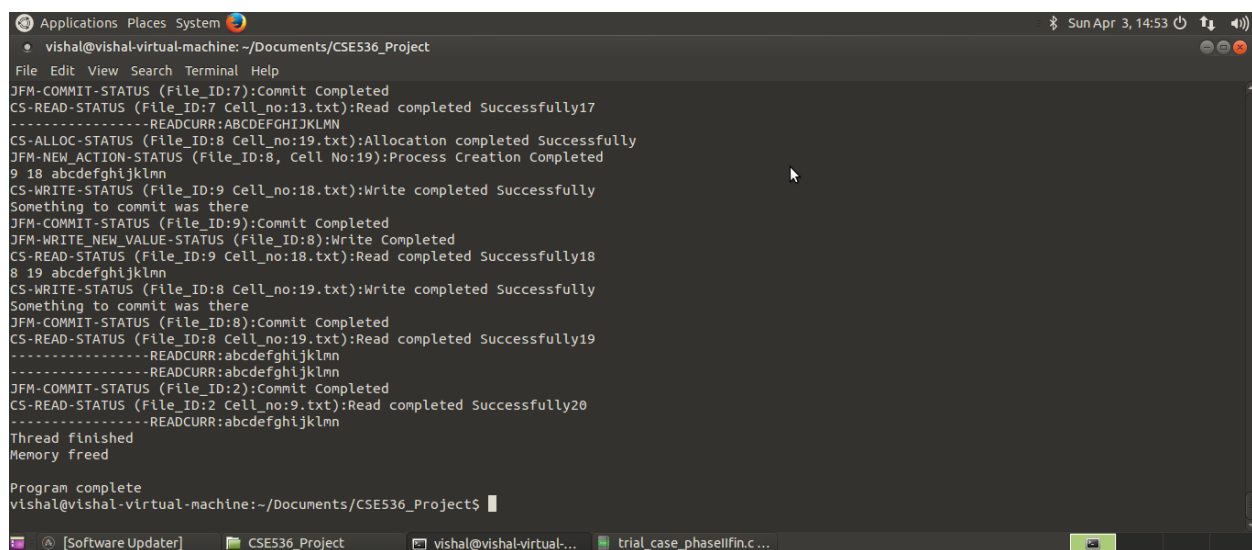
We were able to implement multithreading with before or after atomicity. The implementation was done by mutex locks. Until a process is killed it was ensured that there won't be any deadlocks in the design. Also from the above test cases we have succeed in implementing the before or after atomicity.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

A screenshot of a terminal window titled 'Applications Places System' with a date and time of 'Sun Apr 3, 14:53'. The terminal shows the output of a program running in a directory named 'CSE536_Project'. The output consists of several status messages for file operations, including commit, read, allocation, and write, for different file IDs and cell numbers. The program ends with 'Thread finished', 'Memory freed', and 'Program complete'. The terminal prompt is 'vishal@vishal-virtual-machine:~/Documents/CSE536_Project\$'.

```
vishal@vishal-virtual-machine: ~/Documents/CSE536_Project
File Edit View Search Terminal Help
JFM-COMMIT-STATUS (File_ID:7):Commit Completed
CS-READ-STATUS (File_ID:7 Cell_no:13.txt):Read completed Successfully17
-----READCURR:ABCDEFGHIJKLMN
CS-ALLOC-STATUS (File_ID:8 Cell_no:19.txt):Allocation completed Successfully
JFM-NEW_ACTION-STATUS (File_ID:8, Cell No:19):Process Creation Completed
9 18 abcdefghijklmn
CS-WRITE-STATUS (File_ID:9 Cell_no:18.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:9):Commit Completed
JFM-WRITE_NEW_VALUE-STATUS (File_ID:8):Write Completed
CS-READ-STATUS (File_ID:9 Cell_no:18.txt):Read completed Successfully18
8 19 abcdefghijklmn
CS-WRITE-STATUS (File_ID:8 Cell_no:19.txt):Write completed Successfully
Something to commit was there
JFM-COMMIT-STATUS (File_ID:8):Commit Completed
CS-READ-STATUS (File_ID:8 Cell_no:19.txt):Read completed Successfully19
-----READCURR:abcdefghijklnm
-----READCURR:abcdefghijklnm
JFM-COMMIT-STATUS (File_ID:2):Commit Completed
CS-READ-STATUS (File_ID:2 Cell_no:9.txt):Read completed Successfully20
-----READCURR:abcdefghijklnm
Thread finished
Memory freed
Program complete
vishal@vishal-virtual-machine:~/Documents/CSE536_Project$
```