JFS PROJECT PHASE 4

INTRODUCTION : This project aimed at implementing Journal File System(JFS) and consist of four phases of implementation. The phase IV of the project aimed at implementing and testing the Journal File System for *Error-Prone environment with multi-threaded system*. This phase has implemented a recovery system which is used to recover the data when any system failure or crash occurs.

ASSUMPTIONS: To implement the Journal File System for Error-free environment with multi-threaded system following assumption has been made :

- There could be system failure while writing data to journal or cell storage.
- The journal file system would be using *mutex locks* as this phase consider the environment to be multi-threaded system.
- The length of the data to be written while commit or write to journal storage will be restricted to given length.
- The *Record structure* has been maintained to hold the data ID, Transaction ID, data value, status for any given transaction.
- All the functions and structure has been declared in *proj.h* file.
- I am using in-memory logging configuration.
- The global variable *head* is used to store cell data in linked list data structure.
- Utility function populate() is written to create in memory cell storage.
- Utility function isValidForAbort() is implemented to check whether any transaction id is valid for aborting or not while recovery.

IMPLEMENTATION : The following function has to be implemented for first phase of project :

- NEW_ACTION
- WRITE_NEW_VALUE
- READ_CURRENT_VALUE
- ABORT
- COMMIT
- RECOVERY
- ALLOCATE
- DEALLOCATE

1) NEW_ACTION : This function is providing the unique transaction ID for every new write operation and is called before any WRITE_NEW_VALUE is invoked. I am storing the unique value in text file and incrementing its value by one after every call of NEW_ACTION. This transaction ID is used by COMMIT and ABORT

to perform the operation on Cell Storage. I have used mutex lock to keep transaction file data consistent when the file is simultaneously accessed by different thread. This function is implemented in *new_action.c* .
Input : Nothing
Output : transaction ID(Integer)

2) WRITE_NEW_VALUE : This function is used for writing the data to the log file (Journal Storage). The state of the record for this function is written as "pending" which has to be committed or aborted later on. I have used mutex lock for keeping journal storage data consistent when the file is simultaneously accessed by different thread. If for any given data ID there is previous pending write in journal storage which has to be committed or aborted, then the current call would not update the journal storage. This function is called after invoking NEW_ACTION and is implemented in file *write.c* .
Input = Record Structure pointer (Record *)
Output = Transaction ID (Integer)

3) READ_CURRENT_VALUE : This operation is used for reading the last committed value of given data ID from the cell storage. We have to remove race condition for simultaneous read access by different thread and therefore we have used locking system to lock in memory cell storage. This function is written in *read.c* .
Input : data ID(Integer), dataVal(Char *)
Output : Fills the value of dataVal(Char *)

4) COMMIT : This function is used for writing the pending write from log file to the Cell storage. The changes for given transaction ID is visible to the user once the commit function execute successfully. I have used mutex lock for keeping cell storage and journal storage data consistent when the storage is simultaneously updated by different thread. The function return 0 value if the data is written successfully to cell storage. This function is implemented in *commit.c* .
Input : Transaction ID(Integer)
Output : Result (Integer)

5) ABORT : This function is used to revert the transaction for any given transaction ID with pending write from journal storage. I have used mutex lock for keeping cell storage and journal storage data consistent when the storage is simultaneously updated by different thread. Once the pending write is aborted , it can not be committed in the Cell Storage again. It return 0 if abort operation takes

place otherwise it will return any other value then 0. This function is written in *abort.c* .
Input : Transaction ID(Integer)
Output : Result(Integer)

6) RECOVERY : In this function, we abort the pending write which has not been committed at the initialization of the Journal File System. Once recovery is done, we are populating the cell storage with the correct data from the journal storage using the function populate(). It is written in *recovery.c* .
Input : Void
Output : Void

7) ALLOCATE : Allocate function is used by Commit operation to allocate storage for given id in memory cell storage. It return 0 for successful allocation of a new record in the cell storage. It is written in *allocate.c* .
Input : Record Structure Pointer(Record*)
Output : Integer

8) DEALLOCATE : Allocate function is used by free the in memory cell storage which is allocated by commit operation. It is written in *allocate.c* .
Input : Void
Output : Void

USER INTERFACE : I have used command line input as the user interface to receive the input from user and display the output to the window.

TEST CASES :
The following seven test cases has been implemented for testing in Error-free environment and multi-threaded system.

1) In first fault scenario test case we are two threads with pending write and then commit for same data ID and then crashing the system for thread 2 without committing data value . Then recovery is called and value is read from the in memory cell file to check the value. The steps of test case are as follows:

a) Call WRITE_NEW_VALUE for data ID 1 for each thread for data value “GOOGLE” and “YAHOO” in thread 1 and thread 2 respectively.
b) Commit the data value for dataID 1 in thread 1 but not in thread 2.
c) Call RECOVERY for system.
d) Call READ_CURRENT_VALUE for data ID 1 from cell.

e) The value of data for data ID 1 should be “GOOGLE”

```
Saurabh:os_assignment saur_navigator$ ./fault1
strating recovery
Running First Test Case For Showing MultiThread Scenario
Thread 1 started for dataID 1
Thread 2 started for dataID 1
 Calling NEW_ACTION for new transaction ID
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value GOOGLE and
Calling COMMIT for transID 93
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
commited is successfull for thread 1
Calling COMMIT for transID 94
System is getting crashed for thread 2 and commit not happening
Saurabh:os_assignment saur_navigator$ ./jfs
strating recovery
READ_CURRENT_VALUE
enter the data ID to read
1
 val of Read cureent value is : GOOGLE
Press Y for another operation and N for exit
N
Saurabh:os_assignment saur_navigator$
```

2) In Second fault scenario test case we are creating two threads . One thread will do pending write for data ID 1. While in second thread we will do the pending write and commit for data ID 2. After that recovery is called and value is read from the in memory cell file to check the value is correct one or not. The steps of test case are as follows:

a) Call WRITE_NEW_VALUE for data ID 1 and data value “YAHOO” in thread 1.
b) Call again WRITE_NEW_VALUE for data ID 2 in thread 2 and data value “GOOGLE”.
c) Commit the value for dataID 2 in thread 2.
d) Calling RECOVERY for the system in initJFS()
e) Call READ_CURRENT_VALUE for data ID 2 and it should print “GOOGLE”

```
Saurabh:os_assignment saur_navigator$ ./fault2
strating recovery
Running First Test Case For Showing MultiThread Scenario
Thread 1 started for dataID 1
Thread 2 started for dataID 2
 Calling NEW_ACTION for new transaction ID
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 2 value GOOGLE and
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 101
System getting crashed for thread 1 without commiting value
Saurabh:os_assignment saur_navigator$ ./jfs
strating recovery
READ_CURRENT_VALUE
enter the data ID to read
2
 val of Read cureent value is : GOOGLE
Press Y for another operation and N for exit
N
Saurabh:os_assignment saur_navigator$
```

3) Checking COMMIT for the corresponding WRITE_NEW_VALUE in each thread working correctly or not using Before-or-After atomicity. The steps are as follows :

a) Create two thread for pending write and commit on same data ID.
b) Acquire mutex lock in NEW_BEGIN() and get a new transaction ID for each thread.
c) Acquire mutex lock for writing in Journal Storage and call WRITE_NEW_VALUE.
d) Commit the transaction for given transaction ID in each thread.
e) Call READ_CURRENT_VALUE for given data ID and the value should match the data value “YAHOO”.

```
Saurabh:os_assignment saur_navigator$ ./testCase1
strating recovery
Running First Test Case For Showing MultiThread Scenario
Thread 1 started for dataID 1
Thread 2 started for dataID 1
 Calling NEW_ACTION for new transaction ID
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value GOOGLE and
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 55
Other pending write for same dataID is still waiting to be committed
Not updating current pending WRITE
Calling COMMIT for transID 56
Calling READ_CURRENT_VALUE on dataId 1
Read the correct value
 There was no commit operation done as transID pending write was not there
Saurabh:os_assignment saur_navigator$
```

4) Checking Pending write for same data ID is not happening without commit or first pending write. The steps for the test cases are as follows :

a) Call NEW_BEGIN() to get a new transaction ID for two threads.
b) Call WRITE_NEW_VALUE for data ID 1 and data value "YAHOO" for first thread.
c) Call WRITE_NEW_VALUE for data ID 1 and data value "GOOGLE" for second thread and then Commit the transaction.
d) The WRITE_NEW_VALUE and COMMIT will not change the Journal Storage and Cell storage as previou pending write has to be committed first.

```
Saurabh:os_assignment saur_navigator$ ./testCase2
strating recovery
Running 2 TC : Pending write in first thread and commit on other thread
Thread 1 started for dataID 1
Thread 2 started for dataID 1
 Calling NEW_ACTION for new transaction ID
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Other pending write for same dataID is still waiting to be committed
Not updating current pending WRITE
Calling COMMIT for transID 58
Test Case 2 succesfully passed
Saurabh:os_assignment saur_navigator$
```

5)Checking COMMIT for the corresponding WRITE_NEW_VALUE in each thread working correctly or not for different data ID using Before-or-After atomicity. The steps are as follows :

a) Create two thread for pending write and commit on different data ID.
b) Call NEW_BEGIN() and get a new transaction ID for each thread.
c) Call WRITE_NEW_VALUE and update the journal with pending write.
d) Commit the transaction for given transaction ID in each thread.
e) The above operation is correctly updated in Journal Storage with its snapshot given below.

```
Saurabh:os_assignment saur_navigator$ ./testCase3
strating recovery
Running 3 TC: Showing MultiThread Scenario for Two different dataID
Thread 1 started for dataID 1
Thread 2 started for dataID 2
 Calling NEW_ACTION for new transaction ID
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 2 value GOOGLE and
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 81
Calling COMMIT for transID 82
Calling READ_CURRENT_VALUE on dataId 2
Read the correct value
Calling READ_CURRENT_VALUE on dataId 1
Read the correct value
Saurabh:os_assignment saur_navigator$
```

6) Checking Pending write for different data ID should be happening without commit or first pending write on different data ID. The steps for the test cases are as follows ::

a) Call NEW_BEGIN() to get a new transaction ID for two threads.
b) Call WRITE_NEW_VALUE for data ID 1 and data value "YAHOO" for first thread.
c) Call WRITE_NEW_VALUE for data ID 2 and data value "GOOGLE" for second thread and then Commit the transaction.
d) The WRITE_NEW_VALUE and COMMIT for second thread will change the Journal Storage and Cell storage as previous pending write for different data ID need not to be committe.

```
Saurabh:os_assignment saur_navigator$ ./testCase4
strating recovery
Running 4 TC : Pending write in one thread and commit in other thread for diff dataID
Thread 1 started for dataID 1
Thread 2 started for dataID 2
 Calling NEW_ACTION for new transaction ID
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 2 value GOOGLE and
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 62
Test Case 4 succesfully passed
Saurabh:os_assignment saur_navigator$
```

7) Checking COMMIT operation and ABORT operation in different thread for same dataID. The steps for the test cases are as follows :

a) Call NEW_BEGIN() to get a new transaction ID for each thread.
b) Call WRITE_NEW_VALUE for data ID 1 for each thread.
c) Call Commit for first thread and ABORT for next thread.
d) Call READ_CURRENT_VALUE to check the last commit value of given data ID.

```
Saurabh:os_assignment saur_navigator$ ./testCase5
strating recovery
Running 5 TC : MultiThread Scenario for abort and commit for same dataID
Thread 1 started for dataID 1
Thread 2 started for dataID 1
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value GOOGLE and
Calling COMMIT for transID 63
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling ABORT for transID 64
Calling READ_CURRENT_VALUE on dataId 1
Saurabh:os_assignment saur_navigator$
```

SAMPLE JOURNAL STORAGE RECORD :

```
PENDING transID: 32767 dataID: 1 dataVal: YAHOO
PENDING transID: 32767 dataID: 1 dataVal: YAHOO
COMMIT transID: 32767 dataID: 1 dataVal: YAHOO
PENDING transID: 1 dataID: 1 dataVal: YAHOO
COMMIT transID: 1 dataID: 1 dataVal: YAHOO
```

SUMMARY : The phase IV of Journal File System defines the behaviour for Error-prone scenario in multi-threaded system. This phase implement all the basic functionality of JFS including recovery system . In this phase during the initialisation of Journal File system, *recovery* is called to remove the corrupt data which have been occurred due to system crash or failure. Also during initialisation we would initialize the mutex lock for future locking and unlocking. I have made makefile for all test cases and building the application which can be called as make -f Makefile.

CONCLUSION AND LESSONS LEARNED : This phase implements the fault tolerant property of JFS system for multi-threaded system. As the JFS is initialised, we call the recovery system to abort the pending write present in journal file and populate in memory disk cell storage with correct data reading from the journal storage. This phase explain us how jFS can be used for system which are more vulnerable to system crash or failure, so that it can remove the corrupt data and load the correct data in in memory cell storage file. As this phase is implemented for multi-threaded system therefore mutex locks have been used in order to prevent race condition. This project has helped us in gaining the foundation for JFS for different environment which can be used in future.