JFS PROJECT PHASE 1

INTRODUCTION : This project aimed at implementing Journal File System(JFS) and consist of four phases of implementation. The phase I of the project aimed at implementing and testing the Journal File System for *Error-free environment with single threaded system*. This phases would be implementing *All or Nothing* atomicity to make the system fault tolerant.

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

- There will be no crash while writing data to log file.
- The journal file system would not be using any locks as it consider the environment to be single 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.

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

- NEW_ACTION
- WRITE_NEW_VALUE
- READ_CURRENT_VALUE
- ABORT
- COMMIT
- 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. 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. 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. 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. 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. 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) ALLOCATE and DEALLOCATE : Allocate function is used by Commit operation and Deallocate function is used by Abort operation to manipulate cell storage. As we are using in-memory file to keep the data for commit and abort operation. Thus the implementation of these function are not required.

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 six test cases has been implemented for testing in Error-free environment and single threaded system.

1) Checking COMMIT for the pending write. The steps for the test cases are as follows :

a) Call NEW_BEGIN() to get a new transaction ID.
b) Call WRITE_NEW_VALUE for data ID 1 and data value "YAHOO".
c) Commit the transaction for given transaction ID.
d) Call READ_CURRENT_VALUE for given data ID and the value should match the data value "YAHOO".

```
Saurabh:os_assignment saur_navigator$
Saurabh:os_assignment saur_navigator$ gcc -o test1 testCase1.c
Saurabh:os_assignment saur_navigator$
Saurabh:os_assignment saur_navigator$ ./test1
Running First Test Case For Commit
 Calling NEW_ACTION for new transaction IDCalling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 1
Calling READ_CURRENT_VALUE on dataId 1
Test Case 1 sucessfull
Saurabh:os_assignment saur_navigator$
```

2) Checking the All or Nothing test case for JFS. The steps for the test cases are as follows :

a) Call NEW_BEGIN() to get a new transaction ID i.e 2.
b) Call WRITE_NEW_VALUE for data ID 1 and data value "YAHOO".
c) Commit the transaction for given transaction ID 2.
d) Call Abort for transaction ID 2
e) Call READ_CURRENT_VALUE for given data ID 1, it should read the value of previous COMMIT value "YAHOO".

```
Saurabh:os_assignment saur_navigator$
Saurabh:os_assignment saur_navigator$ gcc -o test2 testCase2.c
Saurabh:os_assignment saur_navigator$ ./test2
Running Second Test Case For Commit
 Calling NEW_ACTION for new transaction IDCalling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling COMMIT for transID 2
Calling ABORT for transID 2
Calling READ_CURRENT_VALUE on dataId 1
Test Case 2 sucessfull
Saurabh:os_assignment saur_navigator$
```

3) Checking the abort is not happening for a transaction ID which is already committed. The steps for the test cases are as follows :

a) Call NEW_BEGIN() to get a new transaction ID i.e 3.
b) Call WRITE_NEW_VALUE for data ID 1 and data value "YAHOO".
c) Commit the transaction for given transaction ID 3.
d) Call Abort for transaction ID 3 and it should fail as the commit for given transaction ID already occurred.

```
Saurabh:os_assignment saur_navigator$
Saurabh:os_assignment saur_navigator$ gcc -o test3 testCase3.c
Saurabh:os_assignment saur_navigator$ ./test3
Running third Test Case For Commit
 Calling NEW_ACTION for new transaction IDCalling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling COMMIT for transID 3
Calling ABORT for transID 3
 ABORT is not done as previously COMMIT
 TEST CASE three successfull
Saurabh:os_assignment saur_navigator$
```

4) Checking COMMIT should not happen for a transaction ID if it is already aborted. The steps for the test cases are as follows :

a) Call NEW_BEGIN() to get a new transaction ID i.e 7.
b) Call WRITE_NEW_VALUE for data ID 1 and data value "YAHOO".
c) Abort the transaction for given transaction ID 7.
d) Call Commit for transaction ID 7 and it should fail as the Abort for given transaction ID already occurred.

```
Saurabh:os_assignment saur_navigator$
Saurabh:os_assignment saur_navigator$ gcc -o test4 testCase4.c
Saurabh:os_assignment saur_navigator$ ./test4
Running Fourth Test Case For Commit
 Calling NEW_ACTION for new transaction IDCalling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling ABORT for transID 7
Calling COMMIT for transID 7
COMMMIT was not successful as Abort with same ID is called before
Test Case fourth sucessfull
Saurabh:os_assignment saur_navigator$
```

5)Checking COMMIT operation is not happening for an unknown transaction ID. The steps for the test cases are as follows :

a) Call NEW_BEGIN() to get a new transaction ID i.e 8.
b) Call WRITE_NEW_VALUE for data ID 1 and data value "YAHOO".
c) Call Commit for unknown transaction ID 3 and it should fail as pending write for given transaction ID 3 does not exist.

```
Saurabh:os_assignment saur_navigator$
Saurabh:os_assignment saur_navigator$
Saurabh:os_assignment saur_navigator$ gcc -o test5 testCase5.c
Saurabh:os_assignment saur_navigator$ ./test5
Running Second Test Case For Commit
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling COMMIT for transID 3
 COMMIT is not SUCCESSFULL as no pending write with transID
 Test case 5 passed
Saurabh:os_assignment saur_navigator$
```

6)Checking for a given transaction ID only the first COMMIT occurs when more than one COMMIT is called consecutively.The steps for the test cases are as follows :

a) Call NEW_BEGIN() to get a new transaction ID i.e 6.
b) Call WRITE_NEW_VALUE for data ID 1 and data value "YAHOO".
c) Commit the transaction for given transaction ID 6.
d) Call Commit for transaction ID 6 twice and it should fail for second commit call as the commit was done previously for the same transaction ID.

```
Saurabh:os_assignment saur_navigator$
Saurabh:os_assignment saur_navigator$ gcc -o test6 testCase6.c
Saurabh:os_assignment saur_navigator$ ./test6
Running Second Sixth Case For Commit
 Calling NEW_ACTION for new transaction IDCalling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling COMMIT for transID 6
Calling COMMIT for transID 6
 Scond COMMIT was not done
Test Case sixth passed
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
PENDING transID: 2 dataID: 1 dataVal: YAHOO
COMMIT transID: 2 dataID: 1 dataVal: YAHOO
PENDING transID: 3 dataID: 1 dataVal: YAHOO
COMMIT transID: 3 dataID: 1 dataVal: YAHOO
PENDING transID: 4 dataID: 1 dataVal: YAHOO
ABORT transID: 4 dataID: 1 dataVal: YAHOO
PENDING transID: 32767 dataID: 1 dataVal: YAHOO
PENDING transID: 5 dataID: 1 dataVal: YAHOO
COMMIT transID: 5 dataID: 1 dataVal: YAHOO
PENDING transID: 6 dataID: 1 dataVal: YAHOO
COMMIT transID: 6 dataID: 1 dataVal: YAHOO
PENDING transID: 32767 dataID: 1 dataVal: YAHOO
PENDING transID: 7 dataID: 1 dataVal: YAHOO
ABORT transID: 7 dataID: 1 dataVal: YAHOO
PENDING transID: 8 dataID: 1 dataVal: YAHOO
COMMIT transID: 8 dataID: 1 dataVal: YAHOO
~
```

SAMPLE USE CASE :

```
Saurabh:os_assignment saur_navigator$ ./jfs
WRITE_NEW_VALUE
enter the data ID
10
enter the data value
GOOGLE
WRITE_NEW_VALUE is done for transID : 9
Press Y for another operation and N for exit
Y
COMMIT
enter the id to commit
9
Value is commited for transID : 9
Press Y for another operation and N for exit
Y
ABORT
enter the id to commit
9
Either there is already comit or abort for given ID
Press Y for another operation and N for exit
N
Saurabh:os_assignment saur_navigator$
```

SUMMARY : The phase I of Journal File System defines the behaviour for Error-free condition and single threaded system. This phase implement all the basic functionality of JFS other than recovery system which would be implemented in further phases. The different function are defined in different classes which makes the implementation less complex for further enhancement. I have made makefile for all test cases and building the application.

CONCLUSION AND LESSONS LEARNED : This phase defines the foundation and definition of Journal Storage and Cell Storage and Journal File System. This phase explain us how journal storage can be used for making the system more consistent and also explain the working scenario of All-or Nothing atomicity. It helps in getting the control flow during a single atomic operation which can be used further to implement recovery from system failure. As this phase is implemented for single threaded system therefore no locks have been used as there would not be any concurrent process creating race condition.