JFS PROJECT PHASE 2

INTRODUCTION : This project aimed at implementing Journal File System(JFS) and consist of four phases of implementation. The phase II of the project aimed at implementing and testing the Journal File System for *Error-free environment with multi-threaded system*. This phases would be extending *All-or-Nothing* atomicity to *Before-or-After* atomicity to handle concurrent operations and make the system fault tolerant and consistent.

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 be using *mutex locks* for providing Before or After atomicity as the environment is considered 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.
- Written *is_valid* function to check whether pending write is valid for given data ID or not.
- Written *init_JFS* to initialize the lock at starting of the application.

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. 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. 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) 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.

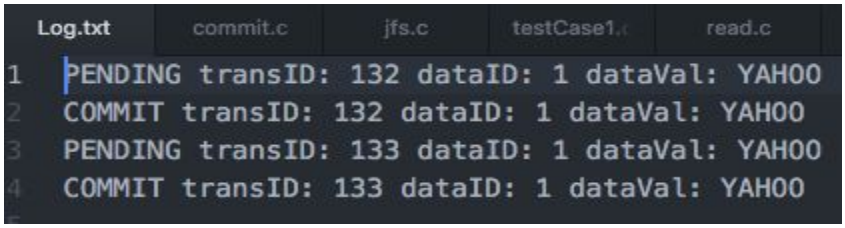
TEST CASES :
The following five test cases has been implemented for testing in Error-free environment and single threaded system.

1) 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
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 YAHOO and
Calling COMMIT for transID 132
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling READ_CURRENT_VALUE on dataId 1
Read the correct value
Calling COMMIT for transID 133
Calling READ_CURRENT_VALUE on dataId 1
Read the correct value
Saurabh:os_assignment saur_navigator$
```

Journal Storage Changes for TC 1:

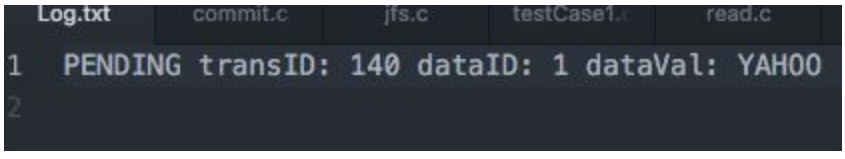


2) Checking Pending write for same data ID is not happening without commit or first pending write. The steps for the test cases are as follows :

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 "YAHOO" 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
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 141
Test Case 2 succesfully passed
Saurabh:os_assignment saur_navigator$
```

Journal Storage Changes for TC 2:

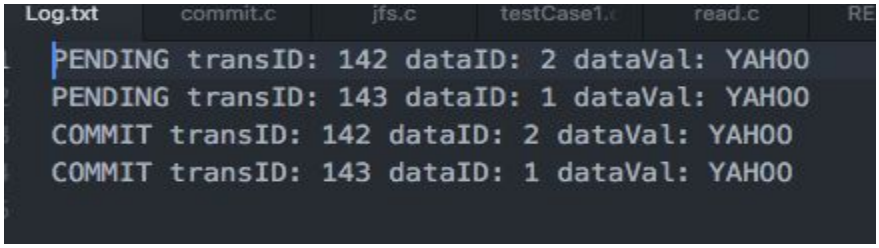


3) Checking COMMIT for the corresponding WRITE_NEW_VALUE in each thread working correctly or not for different data ID using Before-or-After atomicity. The steps are as follows :

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
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 YAHOO and
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 142
Calling COMMIT for transID 143
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$
```

Journal Storage Changes for TC 3:



4) 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 "YAHOO" 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
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 YAHOO and
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 147
Test Case 4 succesfully passed
Saurabh:os_assignment saur_navigator$
```

Journal Storage Changes for TC 4 :

commit.c | testCase4.c | Log.t

```
1  PENDING transID: 146 dataID: 2 dataVal: YAHOO
2  PENDING transID: 147 dataID: 1 dataVal: YAHOO
3  COMMIT transID: 147 dataID: 1 dataVal: YAHOO
4
```

5) 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 and data value "YAHOO" 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
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 NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 154
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling ABORT for transID 155
Calling READ_CURRENT_VALUE on dataId 1
Test Case 5 succesfully passed
Saurabh:os_assignment saur_navigator$
```

Journal Storage Changes for TC 5:

commit.c | testCase4.c | Log.txt

```
1  PENDING transID: 154 dataID: 1 dataVal: YAHOO
2  COMMIT transID: 154 dataID: 1 dataVal: YAHOO
3  PENDING transID: 155 dataID: 1 dataVal: YAHOO
4  ABORT transID: 155 dataID: 1 dataVal: YAHOO
5
```

SAMPLE JOURNAL STORAGE RECORD : It consist of four different data item which are as follows :

a) Status(char*)
b) Transaction ID(int)
c) Data ID(int)
d) Data Value (char*)

commit.c | testCase4.c | Log.txt | jfs.c

```
PENDING transID: 9 dataID: 10 dataVal: GOOGLE
COMMIT transID: 9 dataID: 10 dataVal: GOOGLE
PENDING transID: 10 dataID: 3 dataVal: YAHOO
COMMIT transID: 10 dataID: 3 dataVal: YAHOO
PENDING transID: 11 dataID: 3 dataVal: YAHOO
ABORT transID: 11 dataID: 3 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 II of Journal File System defines the behaviour for Error-free condition in Multi-threaded system. This phase implement all the basic functionality of JFS other than recovery system which would be implemented in further phases.In this phase I have used mutex locks during writing to journal and cell storage system and thus maitaining the cosistency between various threads by implementing Before-or-After atomicity . I have made makefile for building all the test cases and application.

CONCLUSION AND LESSONS LEARNED : The phase II of the project explains how JFS works for Multi-threaded system by extending All-or-Nothing atomicity to Before-or After Atomicity. In this phase we learnt how to create multiple threads using pthread and use mutex lock to avoid the corruption of data due to simultaneous change by different thread. This phase also let us know how concurrent operation can increase the performance of the system. As the journal storage is simultaneously accessed by different thread, therefore we can use different type of locks like flock, fcntl, mutex locks to keep the data consistent. As of now I have not implemented recovery function since this phase is taking account for Error-Free Environment.