JFS PROJECT PHASE 3

INTRODUCTION : This project aimed at implementing Journal File System(JFS) and consist of four phases of implementation. The phase III of the project aimed at implementing and testing the Journal File System for *Error-Prone environment with single threaded system*. This phases 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 single 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 not be using any locks as it consider the environment to be single threaded system so there would not be any initialization of mutex locks in *initJFS()*.
- 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. 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) 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

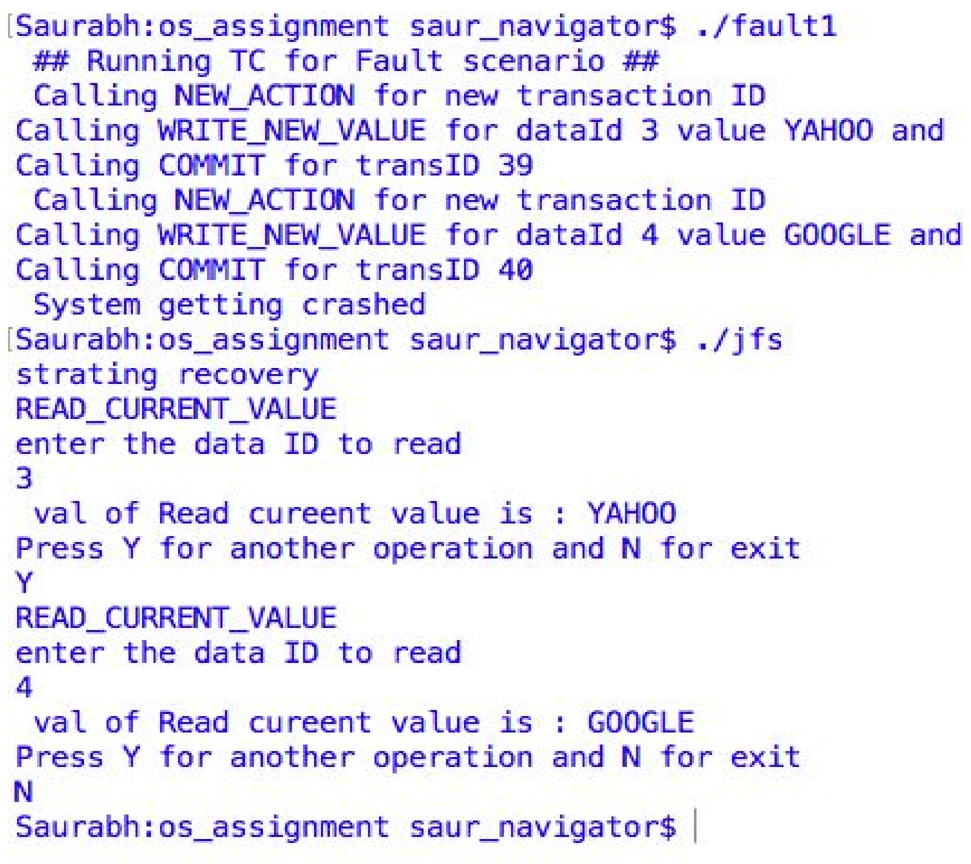
8) DEALLOCATE : Allocate function is used by free the in memory cell storage which is allocated by commit operation. It is written in *allocate.c* .
Input : Void
Output : Void

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 single threaded system.

1) In first fault scenario test case we are making pending write and then commit for two different data ID and then crashing the system and after that recovery is called and value is read from the in memory cell file to check the value. The septs of test case are as follows:

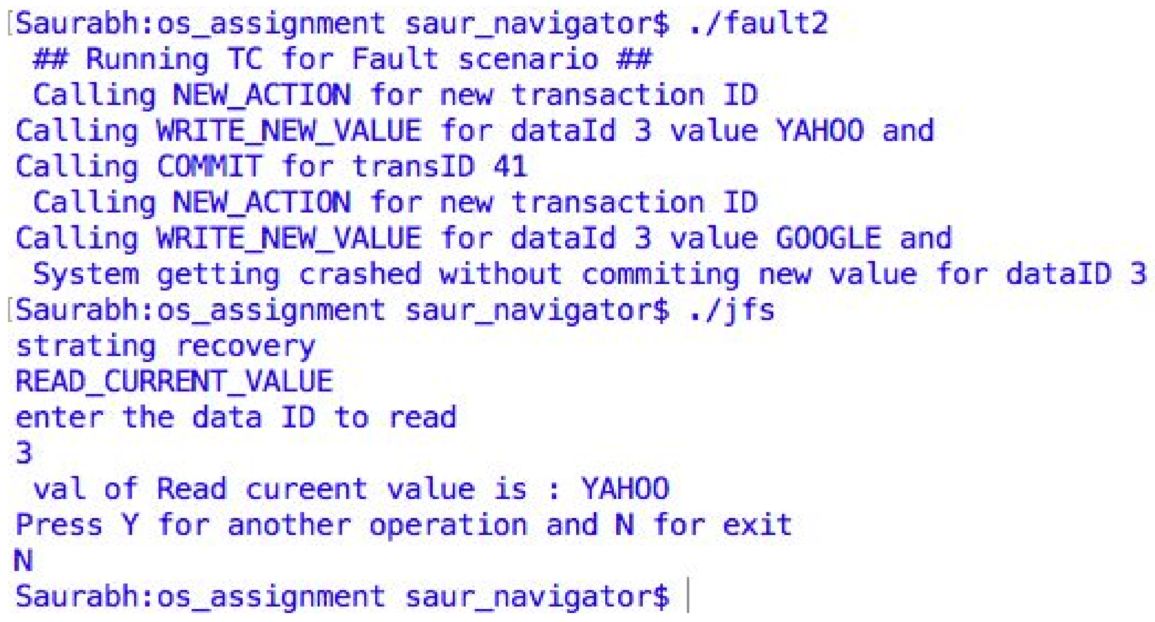
a) Call WRITE_NEW_VALUE for data ID 3 and data value "YAHOO".
b) Commit the data value for dataID 3.
c) Call WRITE_NEW_VALUE for data ID 4 and data value "GOOGLE".
d) Commit the data value for dataID 4
e) Crash the system
f) Calling RECOVERY for the system in initJFS()
g) READ_CURRENT_VALUE for dataID 3 and dataID 4.



```
[Saurabh:os_assignment saur_navigator$ ./fault1
  ## Running TC for Fault scenario ##
  Calling NEW_ACTION for new transaction ID
 Calling WRITE_NEW_VALUE for dataId 3 value YAHOO and
 Calling COMMIT for transID 39
  Calling NEW_ACTION for new transaction ID
 Calling WRITE_NEW_VALUE for dataId 4 value GOOGLE and
 Calling COMMIT for transID 40
  System getting crashed
[Saurabh:os_assignment saur_navigator$ ./jfs
 strating recovery
 READ_CURRENT_VALUE
 enter the data ID to read
 3
  val of Read cureent value is : YAHOO
 Press Y for another operation and N for exit
 Y
 READ_CURRENT_VALUE
 enter the data ID to read
 4
  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 making two pending write and one commit for same data ID and then crashing the system. 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 septs of test case are as follows:

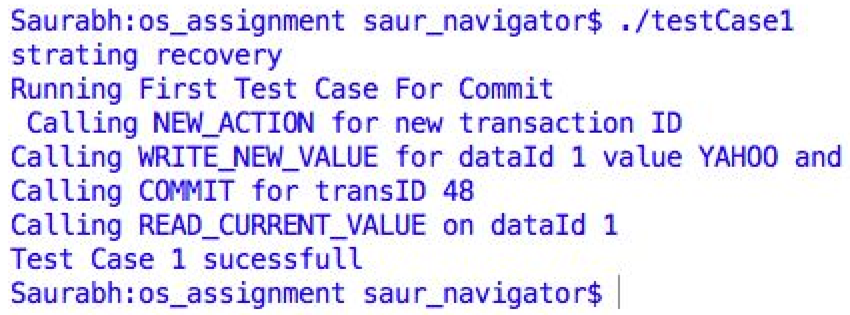
a) Call WRITE_NEW_VALUE for data ID 3 and data value "YAHOO".
b) Commit the value of dataID 3 .
c) Call again WRITE_NEW_VALUE for data ID 3 and data value "GOOGLE".
d) Crash the system.
e) Calling RECOVERY for the system in initJFS()
f) Call READ_CURRENT_VALUE for data ID 3 and it should print "YAHOO"



```
[Saurabh:os_assignment saur_navigator$ ./fault2
  ## Running TC for Fault scenario ##
  Calling NEW_ACTION for new transaction ID
 Calling WRITE_NEW_VALUE for dataId 3 value YAHOO and
 Calling COMMIT for transID 41
  Calling NEW_ACTION for new transaction ID
 Calling WRITE_NEW_VALUE for dataId 3 value GOOGLE and
  System getting crashed without commiting new value for dataID 3
[Saurabh:os_assignment saur_navigator$ ./jfs
 strating recovery
 READ_CURRENT_VALUE
 enter the data ID to read
 3
  val of Read cureent value is : YAHOO
 Press Y for another operation and N for exit
 N
 Saurabh:os_assignment saur_navigator$ |
```

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

a) Calling RECOVERY for the system in initJFS()
b) Call NEW_BEGIN() to get a new transaction ID.
c) Call WRITE_NEW_VALUE for data ID 1 and data value “YAHOO”.
d) Commit the transaction for given transaction ID.
e) Call READ_CURRENT_VALUE for given data ID and the value should match the data value “YAHOO”.

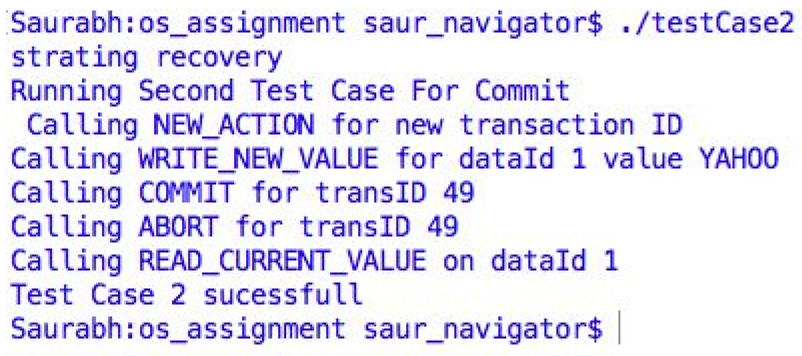


```
Saurabh:os_assignment saur_navigator$ ./testCase1
strating recovery
Running First Test Case For Commit
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO and
Calling COMMIT for transID 48
Calling READ_CURRENT_VALUE on dataId 1
Test Case 1 sucessfull
Saurabh:os_assignment saur_navigator$ |
```

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

a) Calling RECOVERY for the system in initJFS()
b) Call NEW_BEGIN() to get a new transaction ID i.e 2.
c) Call WRITE_NEW_VALUE for data ID 1 and data value “YAHOO”.
d) Commit the transaction for given transaction ID 2.

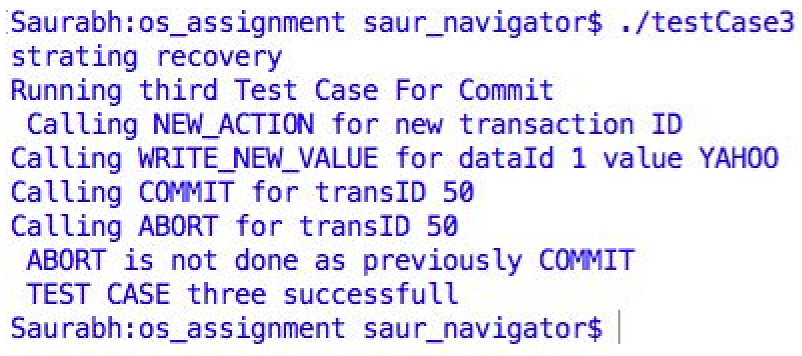
e) Call Abort for transaction ID 2
f) Call READ_CURRENT_VALUE for given data ID 1, it should read the value of previous COMMIT value “YAHOO”.



```
Saurabh:os_assignment saur_navigator$ ./testCase2
strating recovery
Running Second Test Case For Commit
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling COMMIT for transID 49
Calling ABORT for transID 49
Calling READ_CURRENT_VALUE on dataId 1
Test Case 2 sucessfull
Saurabh:os_assignment saur_navigator$
```

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

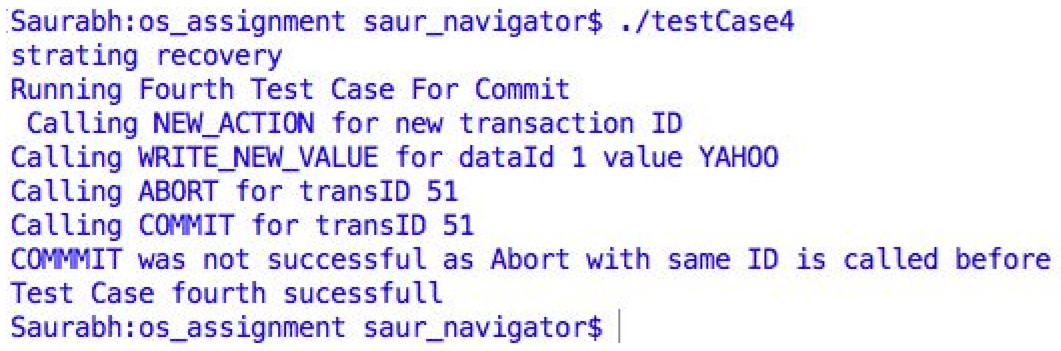
a) Calling RECOVERY for the system in initJFS()
b) Call NEW_BEGIN() to get a new transaction ID i.e 3.
c) Call WRITE_NEW_VALUE for data ID 1 and data value “YAHOO”.
d) Commit the transaction for given transaction ID 3.
e) Call Abort for transaction ID 3 and it should fail as the commit for given transaction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



```
Saurabh:os_assignment saur_navigator$ ./testCase3
strating recovery
Running third Test Case For Commit
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling COMMIT for transID 50
Calling ABORT for transID 50
 ABORT is not done as previously COMMIT
 TEST CASE three successfull
Saurabh:os_assignment saur_navigator$
```

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

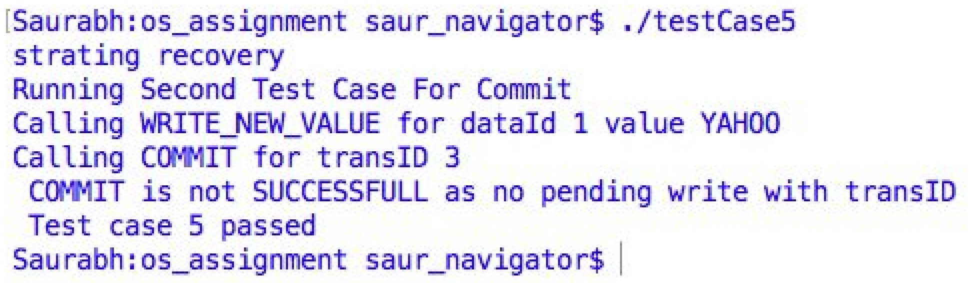
a) Calling RECOVERY for the system in initJFS()
b) Call NEW_BEGIN() to get a new transaction ID i.e 7.
c) Call WRITE_NEW_VALUE for data ID 1 and data value “YAHOO”.
d) Abort the transaction for given transaction ID 7.
e) Call Commit for transaction ID 7 and it should fail as the Abort for given transaction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



```
Saurabh:os_assignment saur_navigator$ ./testCase4
strating recovery
Running Fourth Test Case For Commit
 Calling NEW_ACTION for new transaction ID
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling ABORT for transID 51
Calling COMMIT for transID 51
COMMIT was not successful as Abort with same ID is called before
Test Case fourth sucessfull
Saurabh:os_assignment saur_navigator$
```

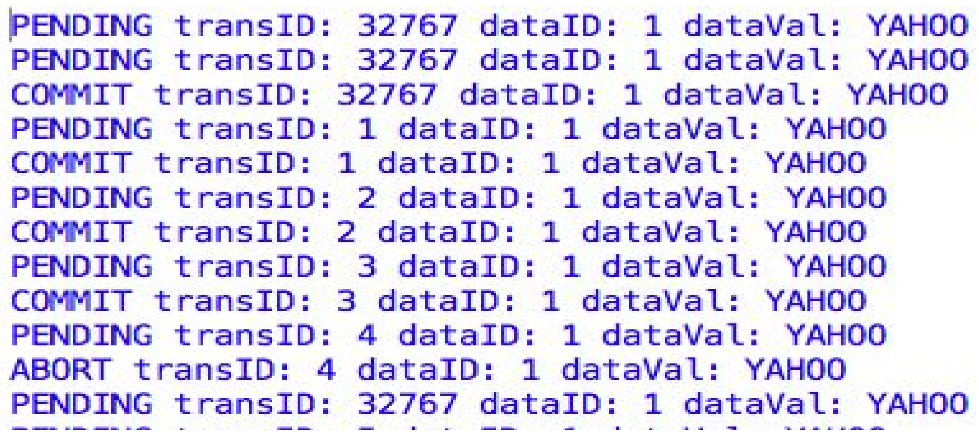
7)Checking COMMIT operation is not happening for an unknown transaction ID. The steps for the test cases are as follows :

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



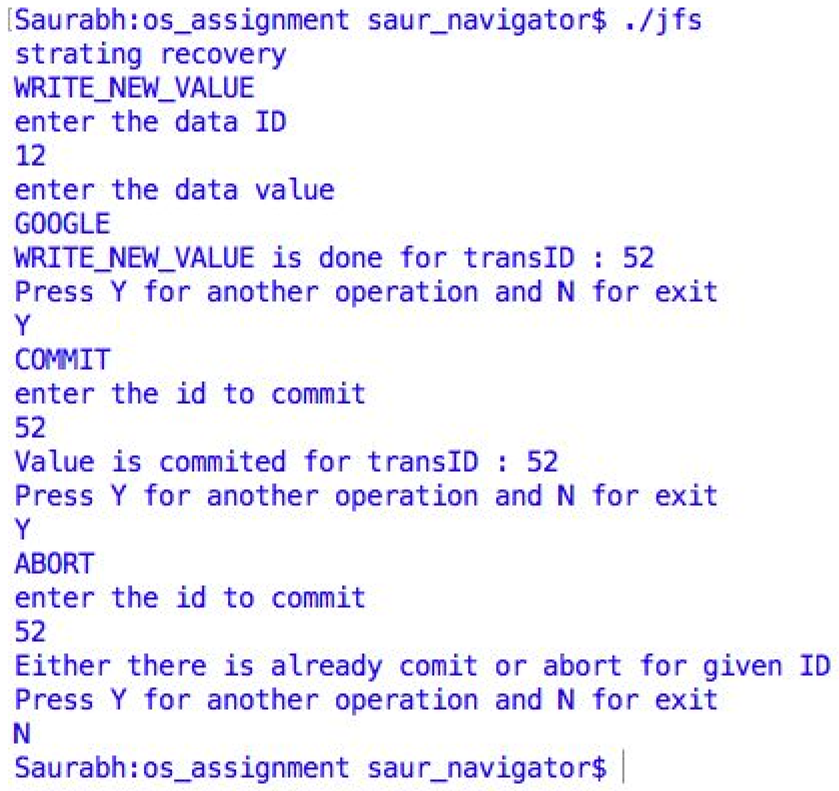
```
Saurabh:os_assignment saur_navigator$ ./testCase5
strating recovery
Running Second Test Case For Commit
Calling WRITE_NEW_VALUE for dataId 1 value YAHOO
Calling COMMIT for transID 3
 COMMIT is not SUCCESSFULL as no pending write with transID
 Test case 5 passed
Saurabh:os_assignment saur_navigator$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

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
```

SAMPLE USE CASE :



```
[Saurabh:os_assignment saur_navigator$ ./jfs
 strating recovery
 WRITE_NEW_VALUE
 enter the data ID
 12
 enter the data value
 GOOGLE
 WRITE_NEW_VALUE is done for transID : 52
 Press Y for another operation and N for exit
 Y
 COMMIT
 enter the id to commit
 52
 Value is commited for transID : 52
 Press Y for another operation and N for exit
 Y
 ABORT
 enter the id to commit
 52
 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 III of Journal File System defines the behaviour for Error-prone scenario in single 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. 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. 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 single threaded system therefore locks have not been initialized so we are not using locks as there would not be any concurrent process creating race condition.