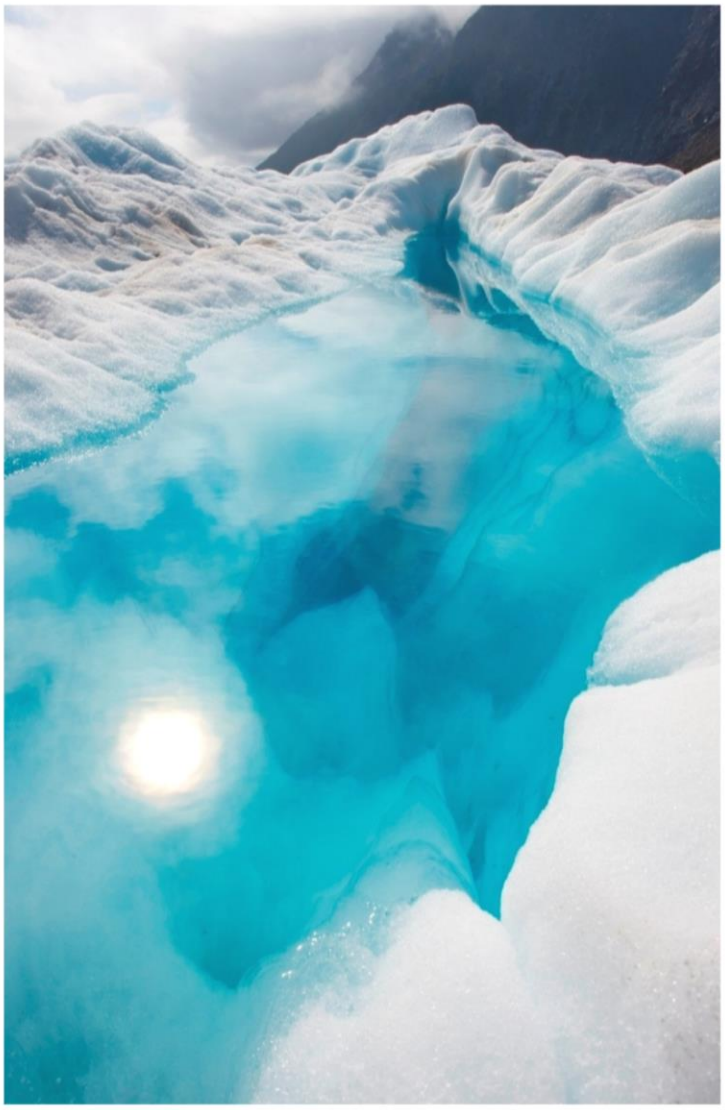


JOURNAL FILE SYSTEM

PHASE THREE

Mengyuan Zhang 1209583385 | CSE 536 Dr. Sandeep Gupta| April 15, 2016

Overview:

In this phase, we are considering the journal file system in the error-prone environment and the reliability mechanism to guarantee that the file system could recover from the undesirable errors and crashes. Compare to the former phase, this phase improves the file system by introducing logs keeping track of the actions dynamically for journal file system and providing mechanisms to recovery the actions or even file system from undesirable errors or crashes.

Assumption:

Old Assumptions in former phase:

1. The maximum number of files which could be stored in the journal file system is 128. (phase1)
2. The maximum size of each file is up to 128 Byte, and the size of file name is at most 8 Byte. (phase1)
3. In every action, at most 8 pending operation(writes) is allowed. (phase1)
4. There are only three threads being implemented used for demonstration, the user cannot modify the number of threads. (Phase2)
5. The threads can be concurrently implemented, however, the action in every threads would be locked to maintain the before and after atomicity. (Phase2)

New Assumptions in this phase:

1. The log could only store 1024 history operations (Including initializing a new action, committing a write, and exiting from an action).

Implementation procedures:

Let's review the architecture of the journal file system. As shown in the graph below, this system provides the following API: NEW_ACTION, WRITE_NEW_VALUE, COMMIT, ABORT, ACTION_CRASH, ACTION_RECOVERY, SYSTEM_RECOVERY, CLEAR_LOG, EXIT. We will describe how these API implement below:

2. NEW_ACTION:

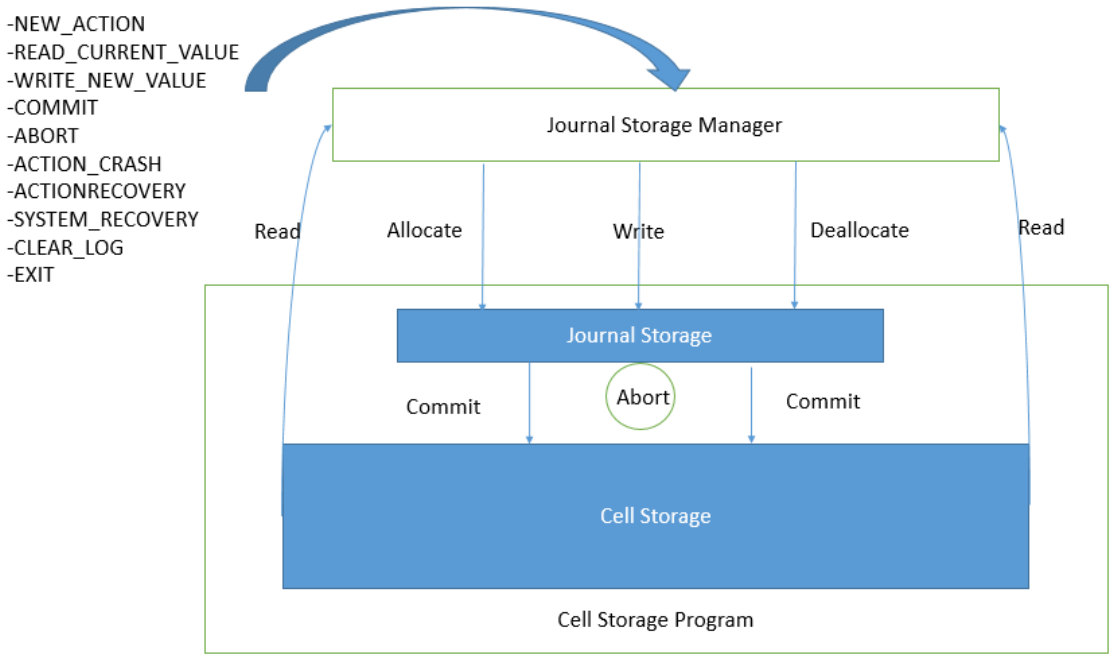
In the journal file system, actions are serialized in every thread. So, the initialization of a new action could only be implemented after the end of the former action. As the new action initialized, the journal storage management will allocate a new storage space for further implementation.

3. READ_CURRENT_VALUE

The READ_CURRENT_VALUE operation searches the file directly from the cell storage. the user is supposed to provide the file name so that it will be used as the inode to locate the file in cell storage. If the program cannot get the used inode with the same name user provided, it will return an error to user that the file is not exist in cell storage.

4. WRITE_NEW_VALUE:

After entering an action, the WRITE_NEW_VALUE could be implemented. The user is supposed to provide the file name and file data so that the file could be stored in Journal storage. The replicated write for the same file will overwrite the original data. As the file being wrote into the journal storage, it is actually pending and cannot be accessed in the user's interface.



5. COMMIT:

In this operation, the user is supposed to provide the file name pending in the journal storage, so the program does the following two things: first, the file is located in journal storage and it will be copied into the cell storage. Second, the original file in journal storage will be removed. As file committed, it is able to be accessed in the user interface.

6. ABORT:

This operation is just to remove the file in journal system given the file name by user.

7. ACTION_CRASH:

This implementation simulates an occurrence of an action-inner error, so that the all the operations the action executed before have to roll back so that it appears the action never happened before.

8. ACTION_RECOVERY:
This implementation can run right after the ACTION_CRASH, so that all the updates of the action before crash could be recovered.

9. SYSTEM_RECOVERY:
This implementation is to simulates the recovery after crash of the whole program. Firstly, the user is supposed to press “ctrl + C” to kill the program simulating an unexpected error happens to crash the whole program, and then after restart the program, and implement SYSTEM_RECOVERY, it will appear that the program crash is never happened

10. CLEEAR_LOG
As the log could only store 1024 history operations (Including initializing a new action, committing a write, and exiting from an action), user will need clear the log sometimes.

11. EXIT:
This means the end of an action.

Test cases and reasons for the choice

1. Journal storage management API test(phase1):

Steps	Expected Result	Reason for Choices
1. Initialize action 0, write two files. **2. Read these two files** **3. Commit one of the former two file and read it** **4. Abort another file. Read it** **5. commit the file being aborted**	**2.The screen returns “file does not exist” Because the files do not commit to the cell storage** **3.the data of the file will be read and output in screen** **4. the file cannot be read** **5. commit operation will show no such file to commit**	**It demonstrates what happen if the file be wrote but not commit** **It demonstrates the commit function that the file being wrote to cell file storage** **It demonstrates the abort function that the file**

		being remove from journal file storage

Test cases:

Action	File Name	File Data
0	**China**	**Panda**
0	**India**	**Elephant**

2. Multithreads before and after atomicity test(phase 2):

Steps	Expected Result	Reason for Choices
1. Initialize three threads which show concurrently. **2. Execute the thread firstly gain the lock, write and commit a file, then quit the action of current thread** **3. Execute the thread secondly gain the lock, read the file committed in the former thread, write and commit another** **4. Execute the thread thirdly gain the lock, read the two files committed in the former thread**	**2. the program enter the entry of the thread firstly gain the lock** **3. the program enter the entry of the thread secondly gain the lock, and the data would be read and output** **4. the program enter the entry of the thread secondly gain the lock, and the data would be read and output**	**This test verifies the before and after atomicity of JFS, I want to show actions of threads will wait for locks which have been already obtained by other, the action won't be interrupt and works fine as single thread. More over every threads will have a common effect on the file system.**

Test cases:

Action	File Name	File Data
0	**China**	**Panda**
0	**India**	**Elephant**

3. Action crash and recovery test(phase 3):

Steps	Expected Result	Reason for Choices
1. Initialize action 0, write and commit two files, and then exit this action. 2. Exit action 0 and initialize action 1, write and commit two files. 3. Read the four files committed before in action 0 and action 1. 4. Implement ACTION_CRASH. 5. Read the four files as step 3. 6. Implement ACTION_RECOVERY 7. Read the four files as step 3 and step 5.	3.All the data of these four file could be read and print in the screen 5. Only two files commit in action 0 could be read, and the screen will return no existence of the files committed in action 1. 7.ALL of the four files could be read, same result as step 3.	This test is used to verify the function of action crash and recovery. I have compared two actions which face a crash and another does not. As the crash occurs in action 1, the program will erase all the operation action 1 did. As the action recovery implemented, the effect of action crash will disappear. So the test demonstrates that the journal file system is reliable in case of the inner-action error.

4. Program crash and recovery(phase 3):

Steps	Expected Result	Reason for Choices
1. Initialize action 0, write and commit two files, and then exit this action. **2.Initialize action 1, write and commit two files.** **3. Read the four files committed before.** **4.Press "ctrl +C" to kill the program and restart it by enter a new action.** **5.Read the four files as step 3.** **6.Implement the SYSTEM_RECOVERY** **7.Read the four files as step 3 and step 5.**	**3.All the data of these four file could be read and print in the screen** **5. No files could be read, as the effect of all operation will be erased as the program ends.** **7.ALL of the four files could be read, same result as step 3.**	**This test is used to verify the program ability to recovery from a crash of the whole program. In the test, we are easily to see that the program could recover even we restart the program since it re-execute all the history operation in the back-end.**

Test cases:

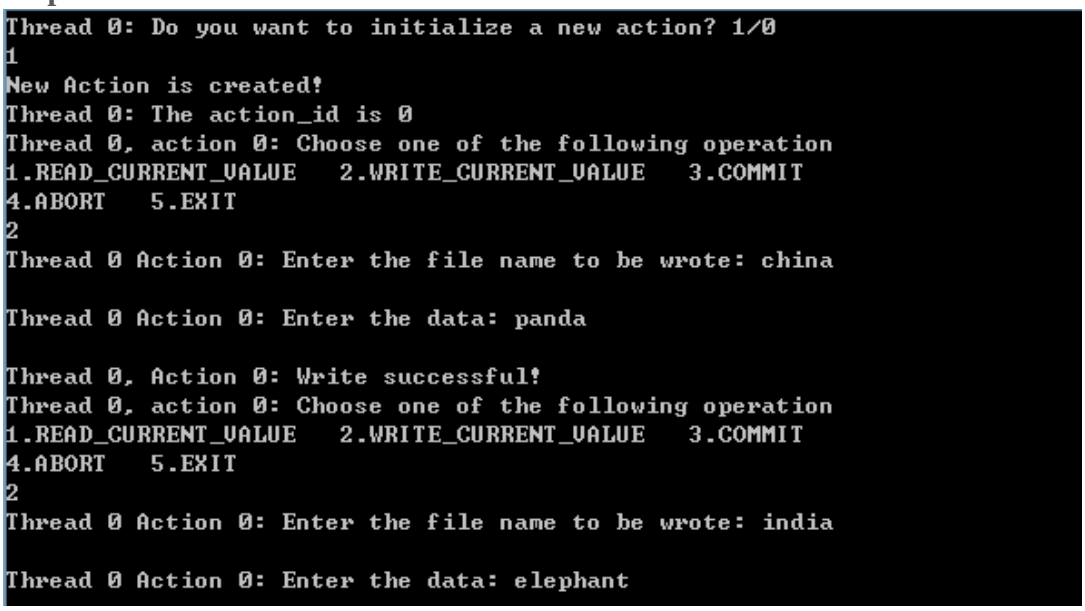
Action	File Name	File Data
0	**China**	**Panda**
0	**America**	**Hawk**
1	**Russia**	**Bear**

1	**India**	**Elephant**

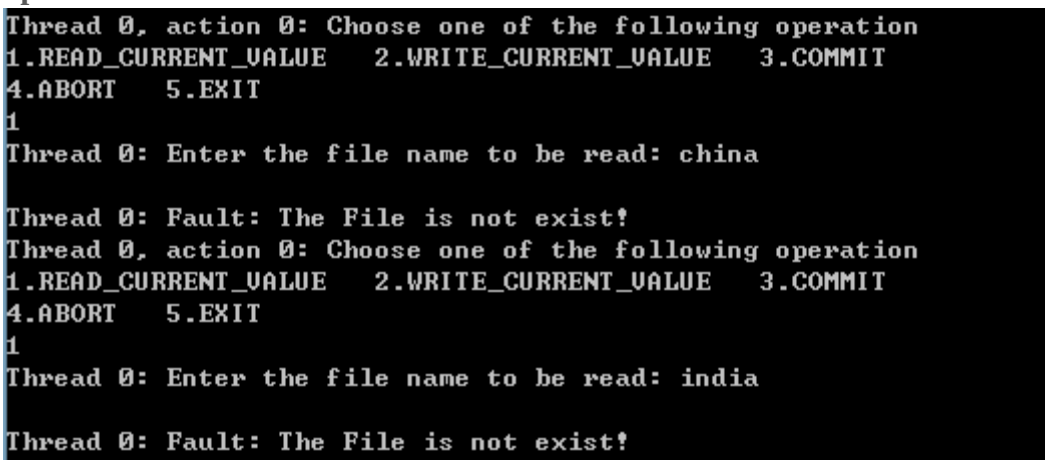
Test run screenshot and explanation:

1. Journal storage management API test:

i. Step one screenshot:



ii.Step two screenshot



iii. Step three screenshot

```
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
3
Thread 0: Enter the file name to be committed: china

Thread 0: Action 0 is committed successfully!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
1
Thread 0: Enter the file name to be read: china

Thread 0 Action 0: file_data is: panda
```

iv. Step four screenshot

```
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
4
Thread 0: Enter the file name to be aborted: india

Thread 0: Action 0 is aborted successfully!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
1
Thread 0: Enter the file name to be read: india

Thread 0: Fault: The File is not exist!
```

v. Step five screenshot

```
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
3
Thread 0: Enter the file name to be committed: india

Thread 0, Action 0: Error! The file could not found to be commited!
```

2. Multithreads before and after atomicity test:

vi. Step one screenshot:

```
Thread 1: Do you want to initialize a new action? 1/0
1
New Action is created!
Thread 1: The action_id is 0
Thread 0: Do you want to initialize a new action? 1/0
1
New Action is created!
Thread 0: The action_id is 0
Thread 2: Do you want to initialize a new action? 1/0
1
New Action is created!
```

vii.Step two screenshot

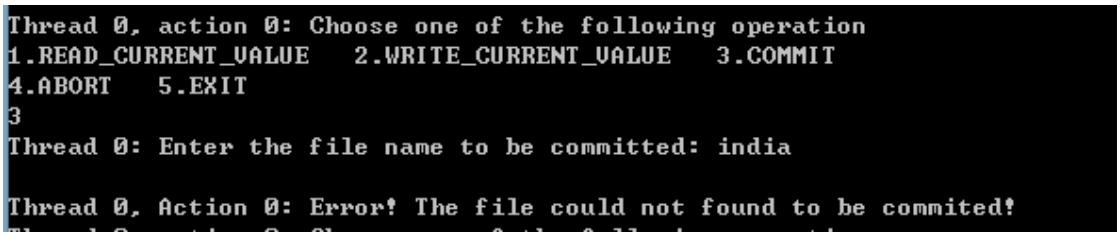
```
Thread 1, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
2
Thread 1 Action 0: Enter the file name to be wrote: china

Thread 1 Action 0: Enter the data: panda

Thread 1, Action 0: Write successful!
Thread 1, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
3
Thread 1: Enter the file name to be committed: china

Thread 1: Action 0 is committed successfully!
Thread 1, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
5
Thread 1 Action 0 exits now!
```

viii. Step three screenshot

```
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
1
Thread 0: Enter the file name to be read: china

Thread 0 Action 0: file_data is: panda
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
2
Thread 0 Action 0: Enter the file name to be wrote: india

Thread 0 Action 0: Enter the data: elephant

Thread 0, Action 0: Write successful!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
3
Thread 0: Enter the file name to be committed: india

Thread 0: Action 0 is committed successfully!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
5
Thread 0 Action 0 exits now!
```

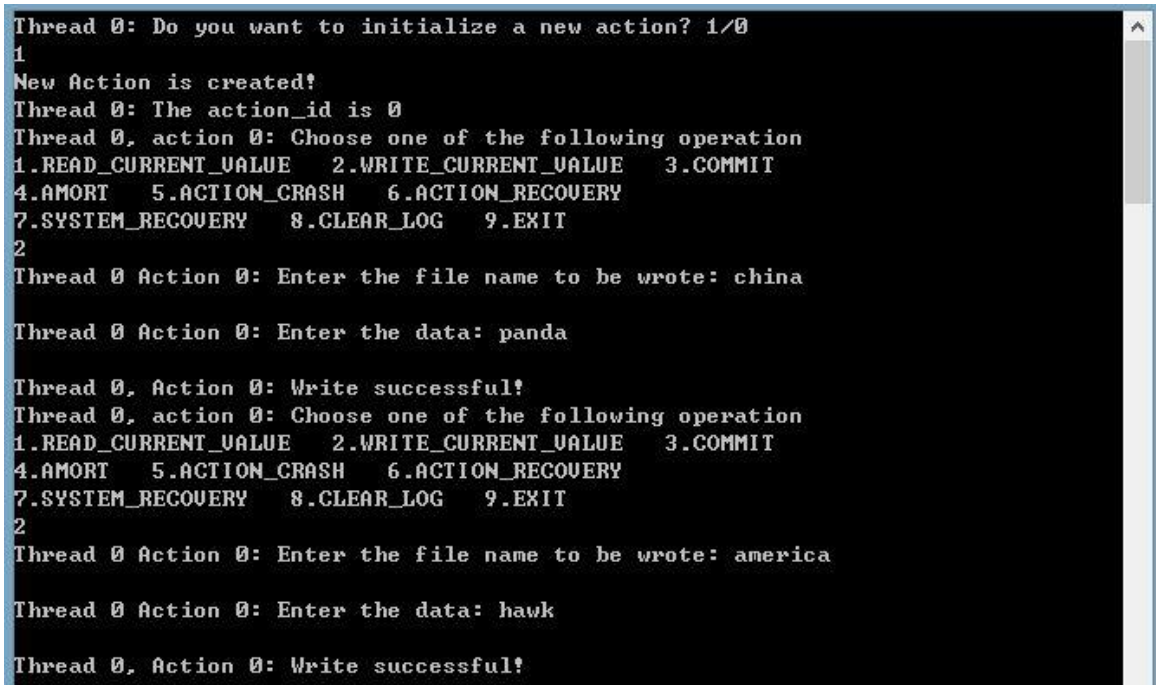
ix. Step four screenshot

```
Thread 2: Enter the file name to be read: china

Thread 2 Action 0: file_data is: panda
Thread 2, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
1
Thread 2: Enter the file name to be read: india

Thread 2 Action 0: file_data is: elephant
Thread 2, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.ABORT   5.EXIT
```



3. Action crash and recovery test:

x. Step one screenshot:



```
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
3
Thread 0: Enter the file name to be committed: china

Thread 0: Action 0 is committed successfully!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
3
Thread 0: Enter the file name to be committed: america

Thread 0: Action 0 is committed successfully!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
9
Thread 0 Action 0 exits now!
```

xi. Step two screenshot:

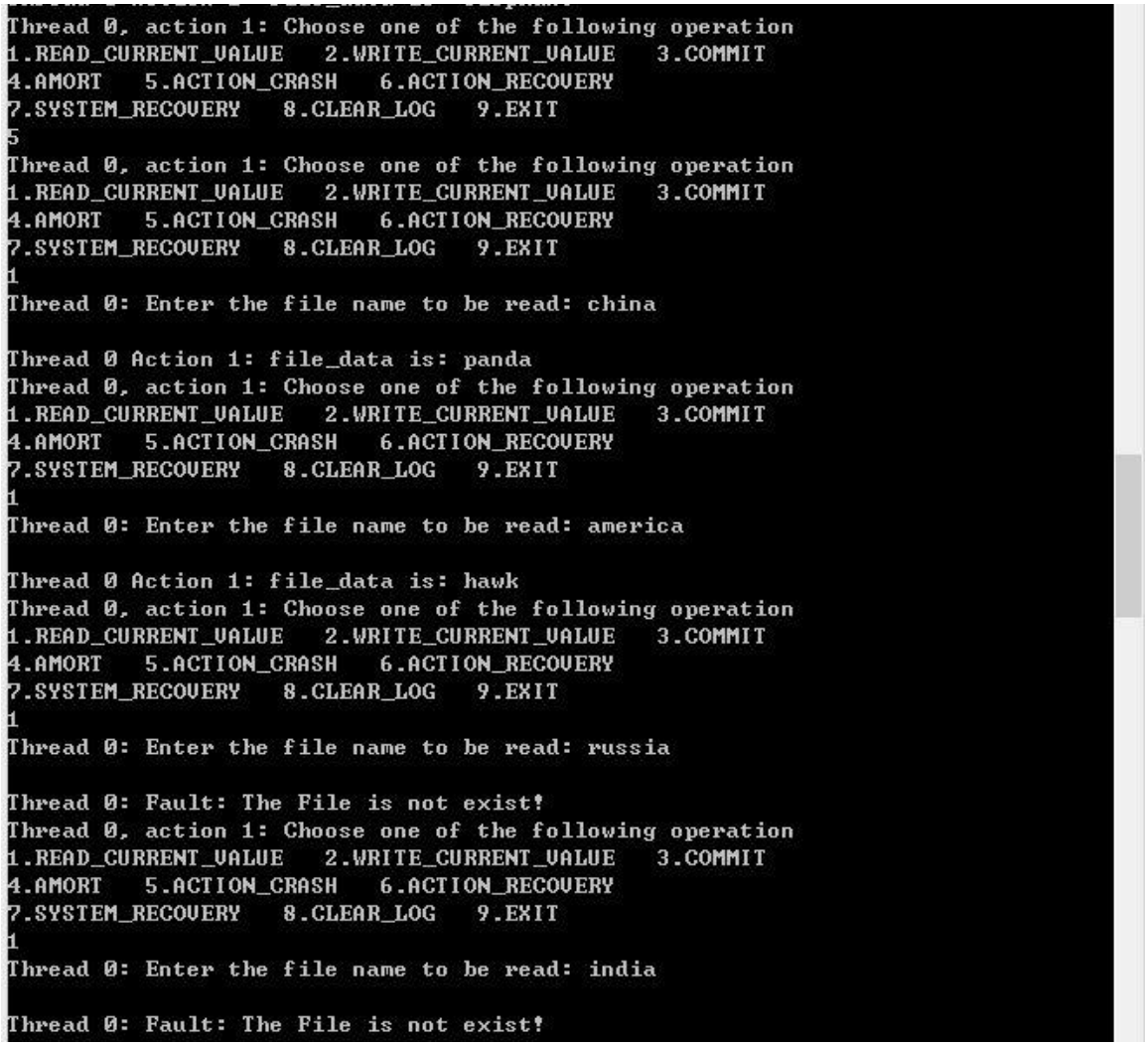
```
Thread 0: Do you want to initialize a new action? 1/0
1
New Action is created!
Thread 0: The action_id is 1
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
2
Thread 0 Action 1: Enter the file name to be wrote: russia

Thread 0 Action 1: Enter the data: bear

Thread 0, Action 1: Write successful!
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
2
Thread 0 Action 1: Enter the file name to be wrote: india

Thread 0 Action 1: Enter the data: elephant

Thread 0, Action 1: Write successful!
```

```
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
3
Thread 0: Enter the file name to be committed: russia

Thread 0: Action 1 is committed successfully!
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
3
Thread 0: Enter the file name to be committed: india

Thread 0: Action 1 is committed successfully!
```

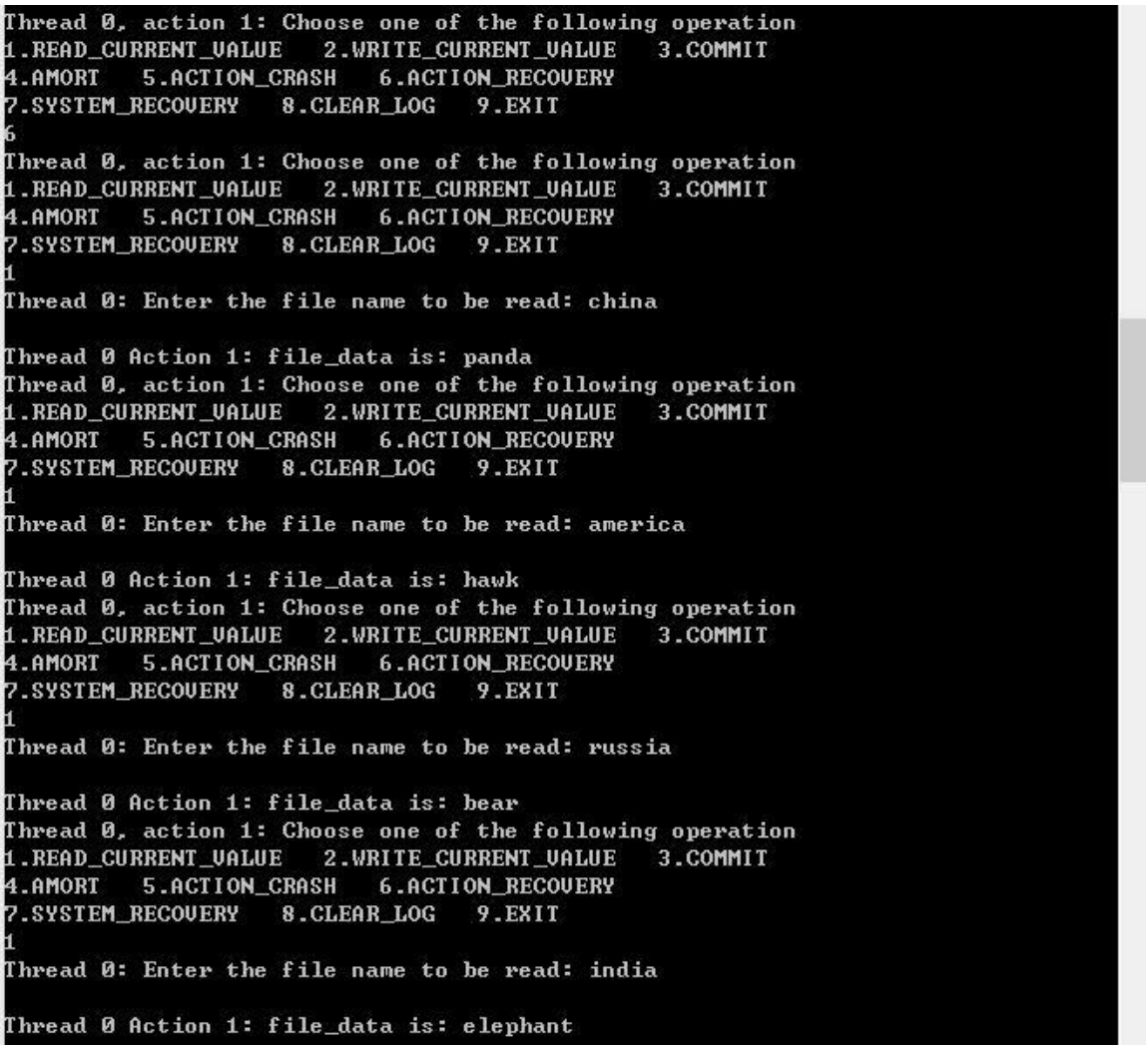
xii. Step three screenshot

```
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
1
Thread 0: Enter the file name to be read: china

Thread 0 Action 1: file_data is: panda
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
1
Thread 0: Enter the file name to be read: america

Thread 0 Action 1: file_data is: hawk
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
1
Thread 0: Enter the file name to be read: russia

Thread 0 Action 1: file_data is: bear
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
1
Thread 0: Enter the file name to be read: india

Thread 0 Action 1: file_data is: elephant
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
```

xiii. Step four and five screenshot

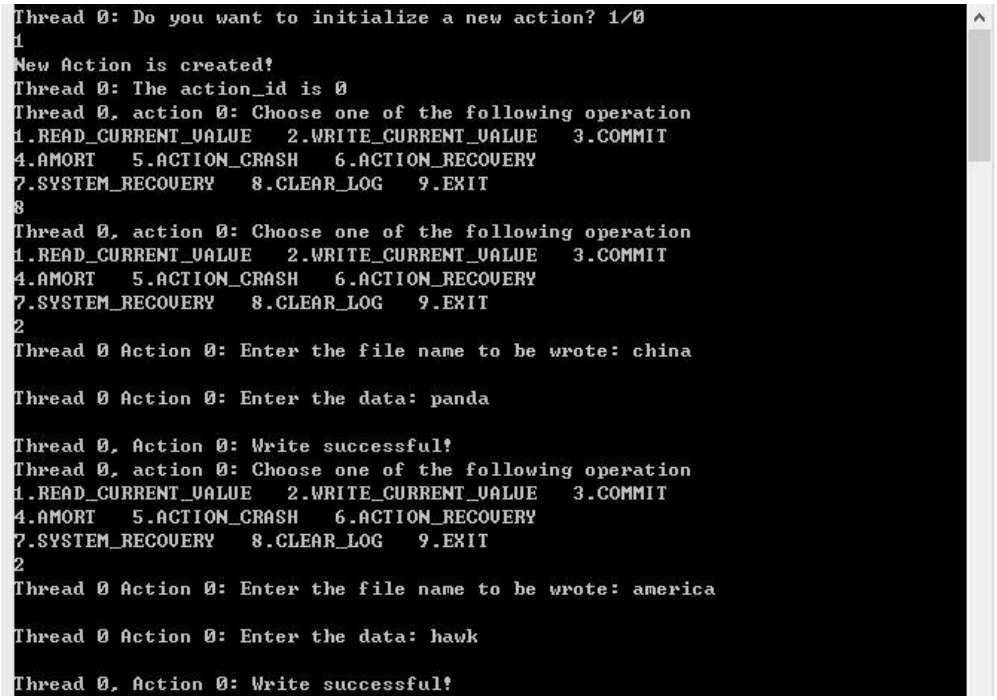


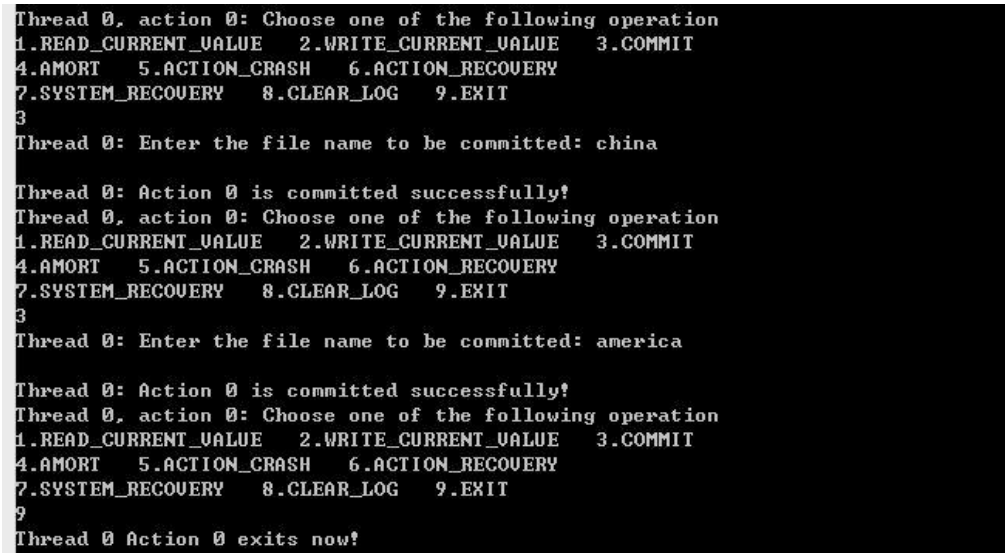
xiv. Step six and seven screenshot:



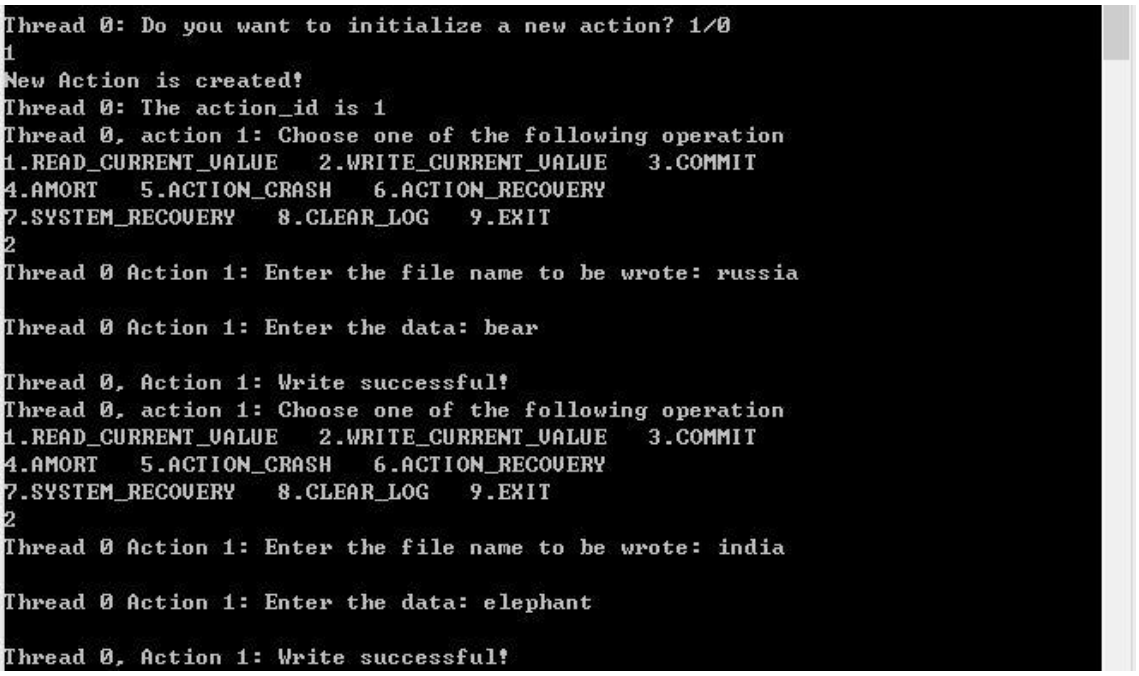
4. Program crash and recovery:

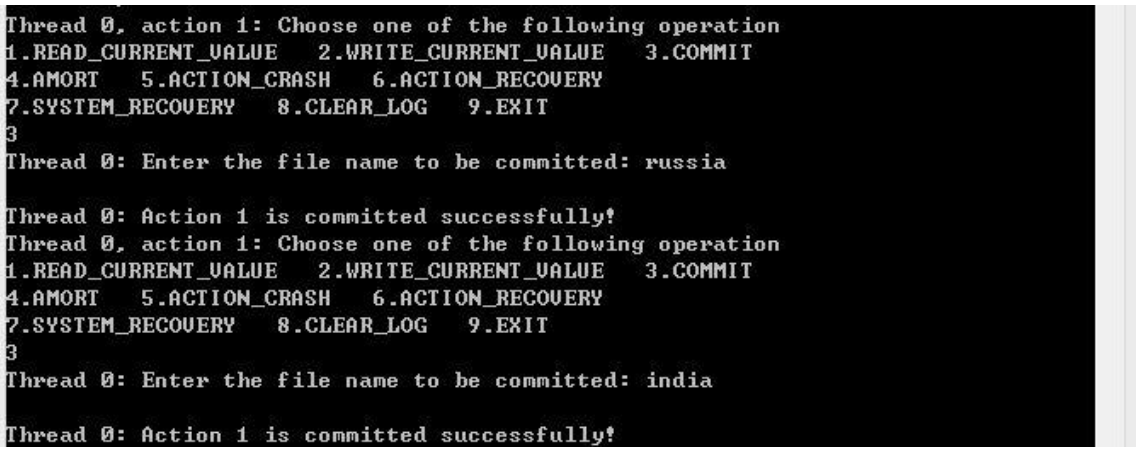
a) Step one screenshot



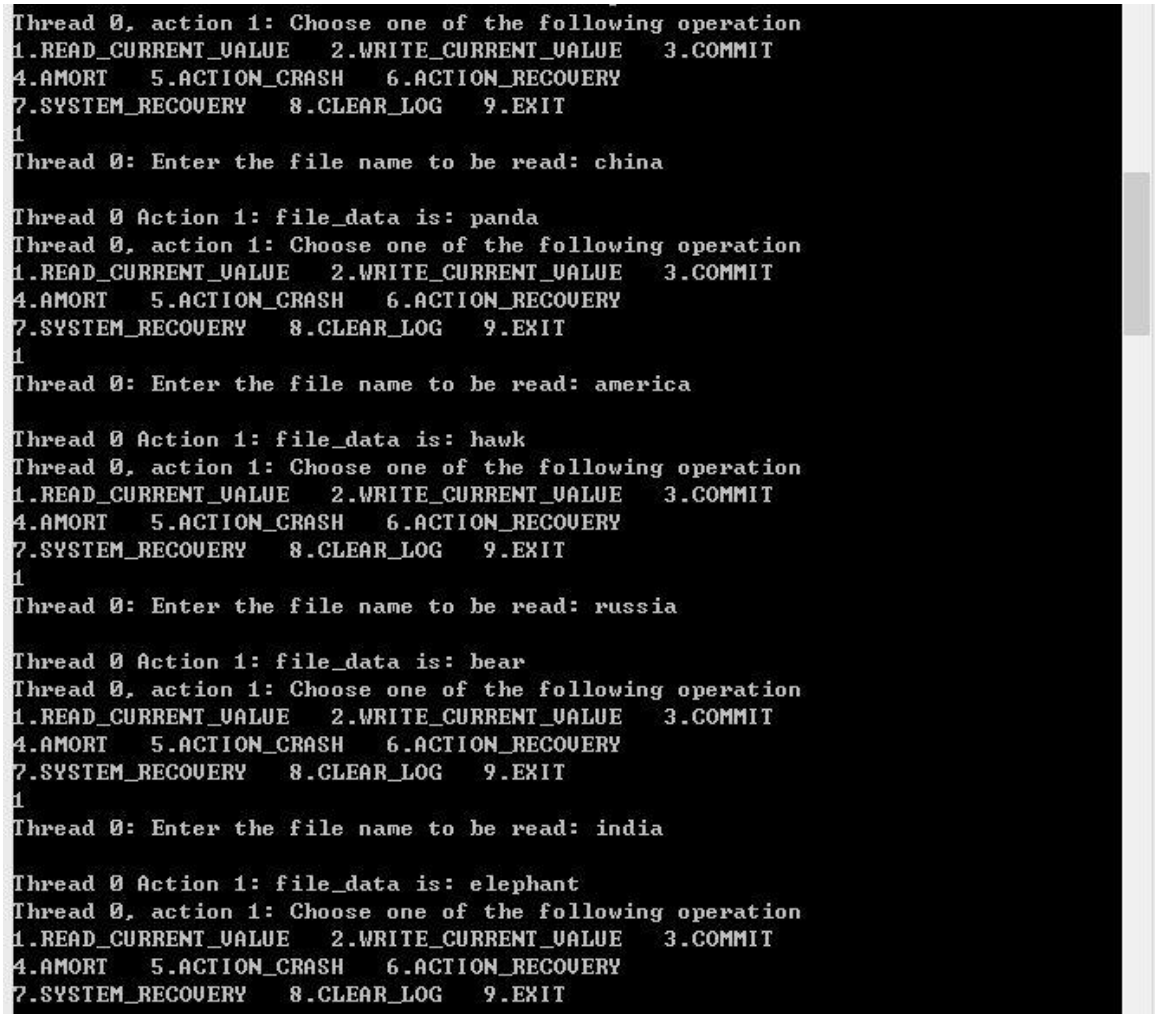


b) Step two screenshot:





c) **Step three screenshot:**



d) Step Four and five screenshot:

```
Thread 0: Do you want to initialize a new action? 1/0
1
New Action is created!
Thread 0: The action_id is 0
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
1
Thread 0: Enter the file name to be read: china

Thread 0: Fault: The File is not exist!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
1
Thread 0: Enter the file name to be read: america

Thread 0: Fault: The File is not exist!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
1
Thread 0: Enter the file name to be read: russia

Thread 0: Fault: The File is not exist!
Thread 0, action 0: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.COMMIT
4.AMORT   5.ACTION_CRASH   6.ACTION_RECOVERY
7.SYSTEM_RECOVERY   8.CLEAR_LOG   9.EXIT
1
Thread 0: Enter the file name to be read: india

Thread 0: Fault: The File is not exist!
```

e) **Step six and seven screenshot**

Result and conclusion:

As the screenshot shown above, the test meets all the expectation discussed before to demonstrate the reliability of the journal file system in Error-prone environment.

Summary:

In phase 1, I have built up the fundamental journal file system, with underlying mechanism to locate and mutate the file in the file structure, check and set the usage of the file by bitmap. All of the underlying file structure is implemented by utilizing the pointer By implementing the phase 1, I get a fundamental perspective of how to generate the operating system.

In this phase, in addition to the in-memory processing, the file system maintains a log keeping track of the operation history in non-volatile storage media. So that the file system could roll back and redo the operations before. The outcome completely meets the design objective and expectation.

Lesson learned

In this phase, I have got in-depth understanding in the following aspects:

a. The principle of journal file system with as far as journal storage and cell storage.
b. How to structure a file system.
c. Understanding of the C language pointer.

d. The log mechanism to keep track of history operation with redo and undo record.
e. The file I/0 processing, read and write the file stream with blended data type.