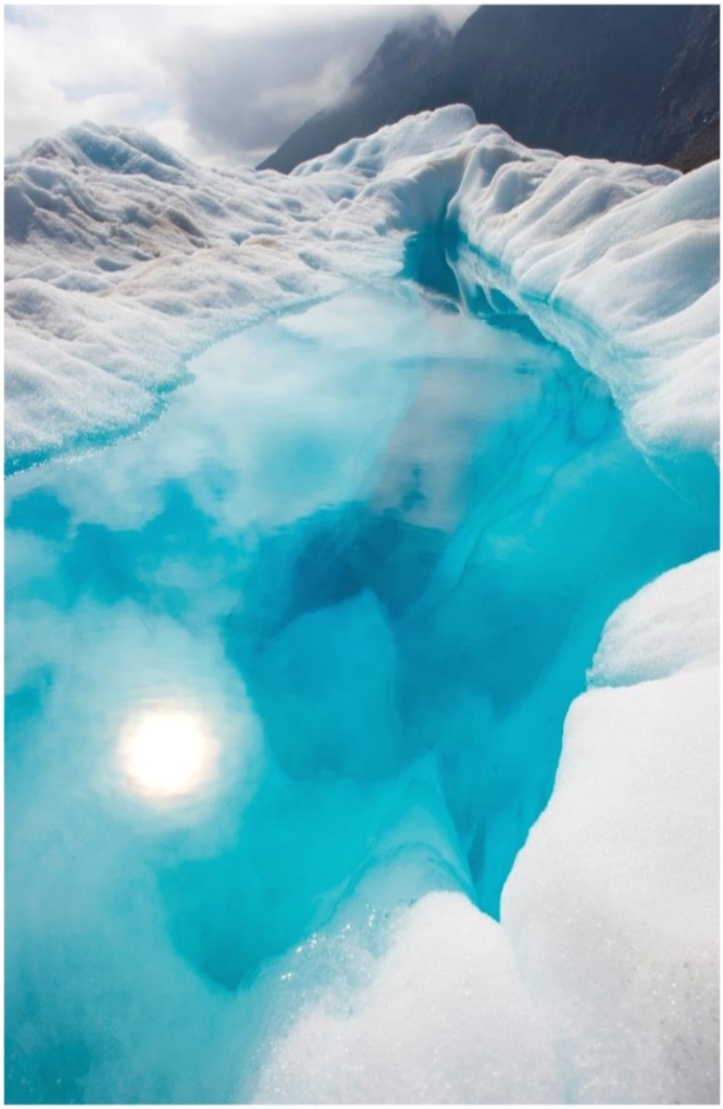


JOURNAL FILE SYSTEM

PHASE FOUR

Mengyuan Zhang 1209583385 | CSE 536 Dr. Sandeep Gupta| April 25, 2016

Overview:

In this phase, we are considering the multi_threads 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. Based on the former phases, this phase combines the concurrent mechanism in phase 2 and the error-prone program in phase 4 to complete the journal file system. And several modification is took as follows:

1. Different from former phase in which the commit and abort operation commits or aborts a specific file written before in the current action, in this phase commit and abort operation commits or aborts all file written before in the current action.
2. Before this phase, the multi-threads is implemented as strictly serialization order in which action is atomic, that means every action could only be run after the end of the previous action. In this phase, we introduce the mark point principle to increase the concurrency so that every action could interrupt each other, but each operation in the action still be atomic.
3. Also, We have modified the design document to give you a full view of the design details.

Assumption:

The assumption is based on the design of the journal file system, more details about these assumptions are available in the design document.

1. We suppose the journal storage is kept in the non-volatile storage but actually it is implemented in volatile storage.
2. The size of file data is no more than 128 byte, the size of file name is no more than 8 byte.
3. The maximum number of file pending in journal storage is 1024, the maximum number of file could be read in the cell storage is 128.
4. Every action could only write the same file one time.
5. The fault write could be detected and report to the upper interface.
6. The log size could not beyond 100, so frequently clear is necessary.
7. In each action, before mark point, writes are allowed but commits are not; after mark point, commits are allowed but writes are not.
8. The Fault prone environment has two kind of fault, one is the fault write which could be detected, and the other one is the fault cause crash so that the data in cell storage is erased.

In addition to design issue, we assume that the number of threads is fixed as 3 for demonstration.

Implementation procedures:

1. NEW_ACTION. (In my implementation, the program will ask if you want to enter a new action at the very beginning.) 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, new action id will be assigned. (The latest action id + 1)
2. READ_CURRENT_VALUE: The 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.
3. WRITE_NEW_VALUE: The operation writes the file into the journal storage by the file name and file data provided by the user.
4. COMMIT: This operation commits every pending files updated in the current action, so that the pending files will be removed in journal storage and be installed in the cell storage.
5. ABORT: This operation aborts every pending files updated in the current action, so that the pending files will be removed in journal storage.
6. AOUNCE_MARK_POINT. There is a global variable maintained in the program named mk. In this operation, if the current action id is ac_id, the current operation will be blocked till the action ac_id – 1 has announce the mark point. After the action announce mark point, mk would increase by 1.
7. FAULT_WRITE: This operation is simulating a fault injection. In this operation, the program could know the file is faultly wrote by detecting if the action id stored in Journal storage is 0xFF. And the fault write data cannot be committed successfully.
8. SYSTEM_CRASH: This operation simulate the program crash so that all the data in cell storage is erased.
9. SYSTEM_RECOVERY: This operation would reconstruct the cell storage after system crash, the system would appear the same as before system crash happen.
10. CLEAR_LOG: This operation is used to clear the data in the log file because the program keeps a limited size of buffer to carry the log information. So we have another assumption:
11. EXIT: This operation will close the current action.

Test cases and reasons for the choice

1. WRITE_READ_COMMIT_ABORT TEST:

Steps	Expected Result	Reason for Choices
1. Action 1: write file 1, Action 2: write file 2 Action 3: write file 3 2. Action 1: announce mark point Action 2: announce mark point Action 3: announce mark point 3. Commit action 1 Abort action 2 4. Read these three file no matter in which thread.	4. Only the file being wrote and committed in action 1 could be read.	This test could demonstrate that only the file committed could be accessed by the user. The file aborted or pending cannot be read.

Test cases:

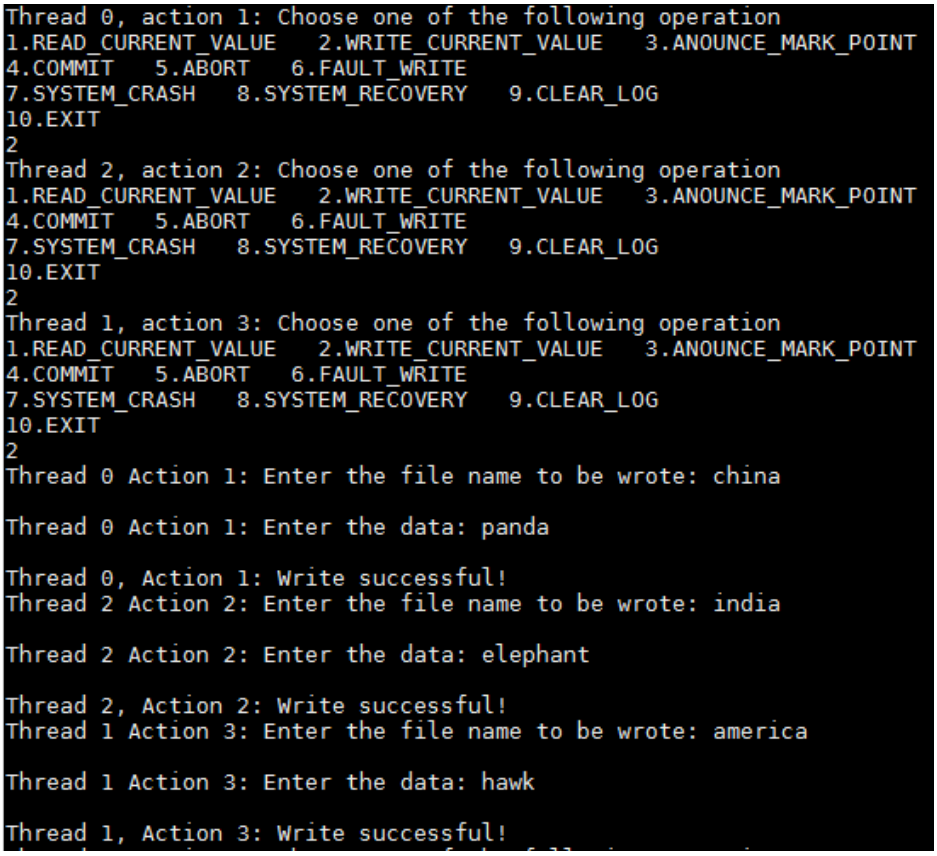
Action	File Name	File Data
1	**China**	**Panda**
2	**India**	**Elephant**
3	**America**	**Hawk**

Test run screenshot and explanation:

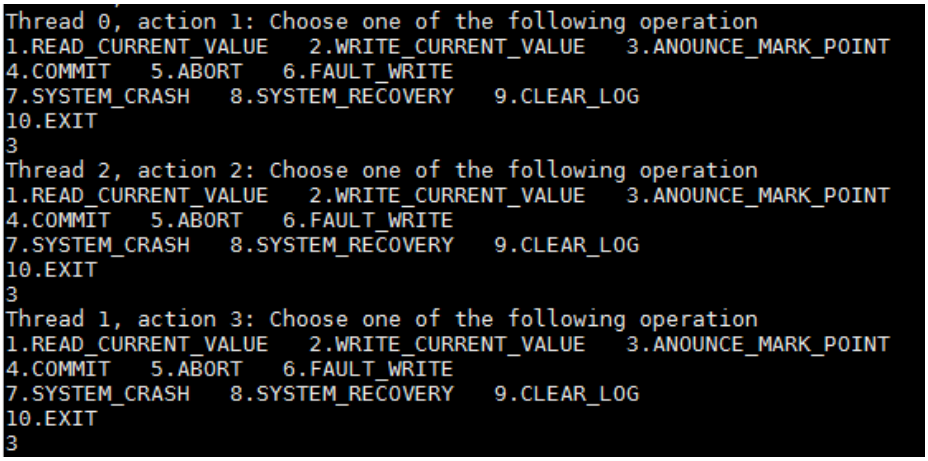
Notice: The multi-threads is designed that every threads operation is executed one by one. Because every operation will obtain the mutex_lock, if one operation is not complete, it will block other threads' operation. So if you don't want to run the current thread, just press "0". It will do nothing but finish the concurrent thread's operation.

Moreover, this test only verify the basic performance of abort and commit in multithread. The tester is welcome to verify by overwriting to the same file or exiting the current action, to initialize new action to test.

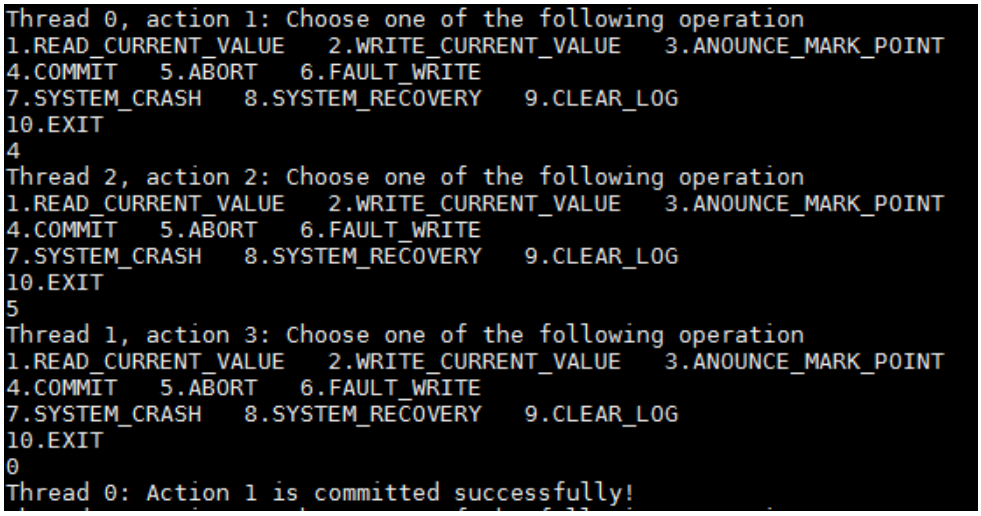
Screenshot of step one, write the files in each of the actions. (Threads)



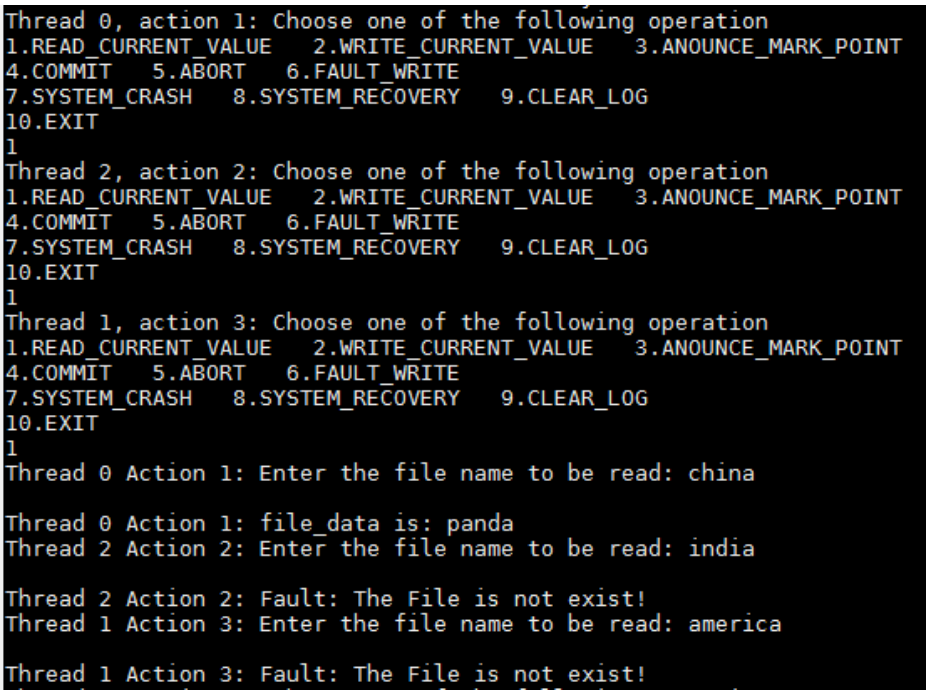
Screenshot of step two, announce the mark point in each of the actions. (Threads)



Screenshot of step three, commit action 1, abort action 2 and do nothing for action 3.



Screenshot of step four, only the file committed in action 1 could be read. The file aborted or pending cannot be read.



The test result completely meets the expectation!

2. Multithreads before and after atomicity test 1:

Steps	Expected Result	Reason for Choices
1. Action 1: write file 1 Action 2: write file 2 Action 3: write file 3 2. Announce mark point in action 2; Announce mark point in action 3 3. Do some empty operation.(enter "0" when listing the options) 4. Announce mark point in action 1 and commit action 1. 5. After one or two empty operation, it shows that action 1, 2 and 3 have committed. 6. Read the three files no matter the threads	3. We find the action 2 and 3 don't run, in fact they are blocked. 5. Soon after action 1 announcing mark point, screen will shows action 2 and action 1 are committed successfully. 6. Three files could be read correctly	This test demonstrate the mark point discipline that the concurrent action commission will be blocked until all previous action has announce the mark point so that the serialized order could be guaranteed.

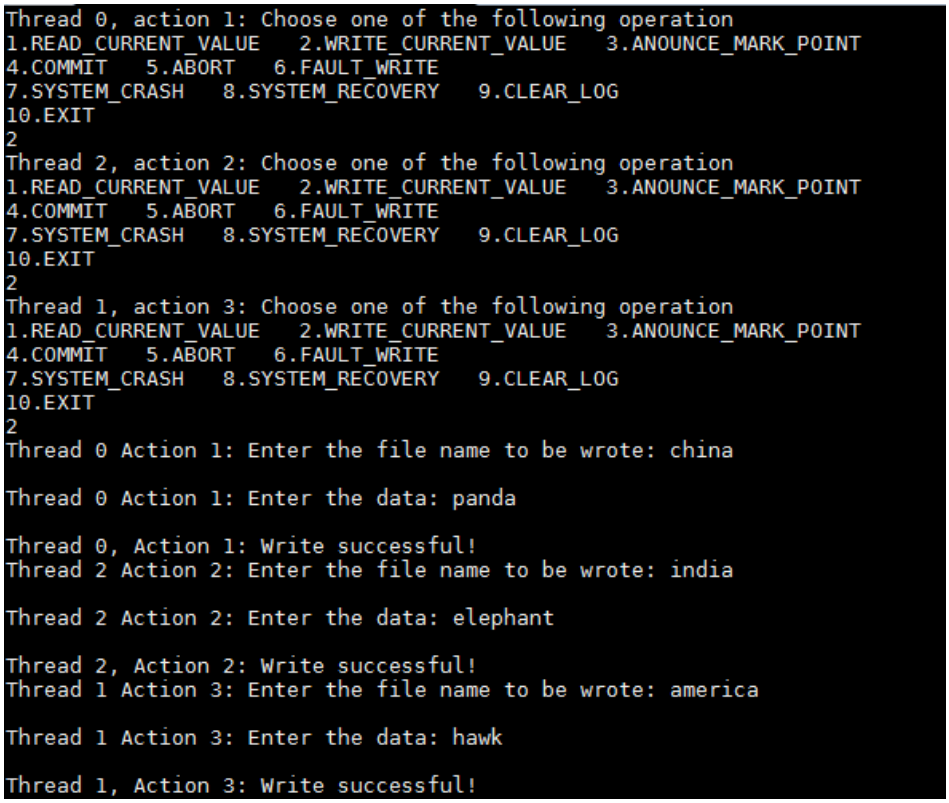
Test cases:

Action	File Name	File Data
1	**China**	**Panda**
2	**India**	**Elephant**
3	**America**	**Hawk**

Test run screenshot and explanation:

Notice: The multi-threads is designed that every threads operation is executed one by one. Because every operation will obtain the mutex_lock, if one operation is not complete, it will block other threads' operation. So if you don't want to run the current thread, just press "0". It will do nothing but finish the concurrent thread's operation.

Screenshot of step one, write three files in each actions.



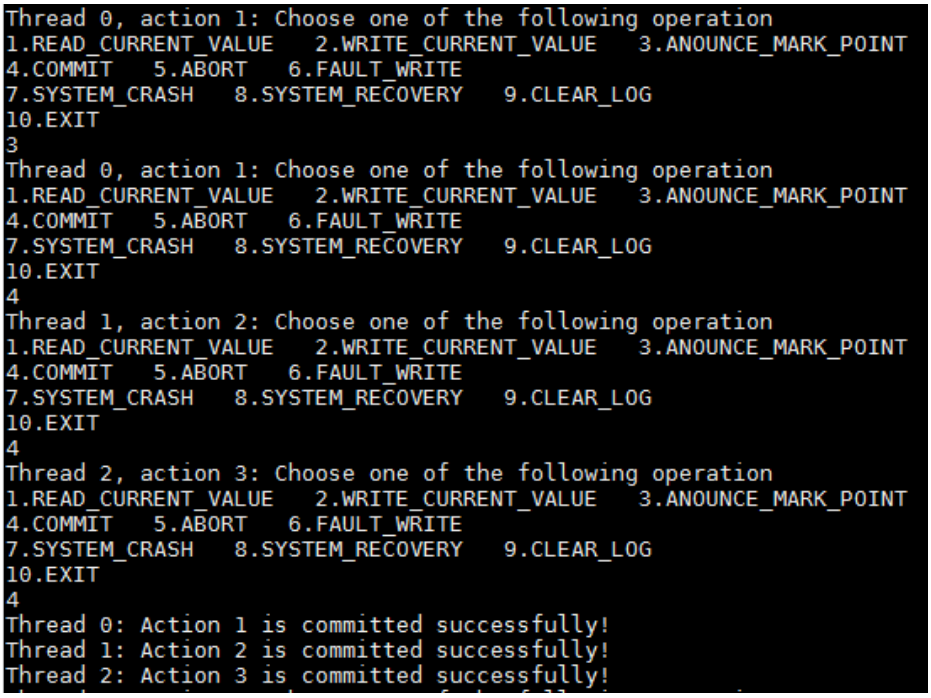
Screenshot of step two, announce mark point for action 2 and action 3 but do nothing for action 1.

```
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
0
Thread 2, action 2: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
3
Thread 1, action 3: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
3
```

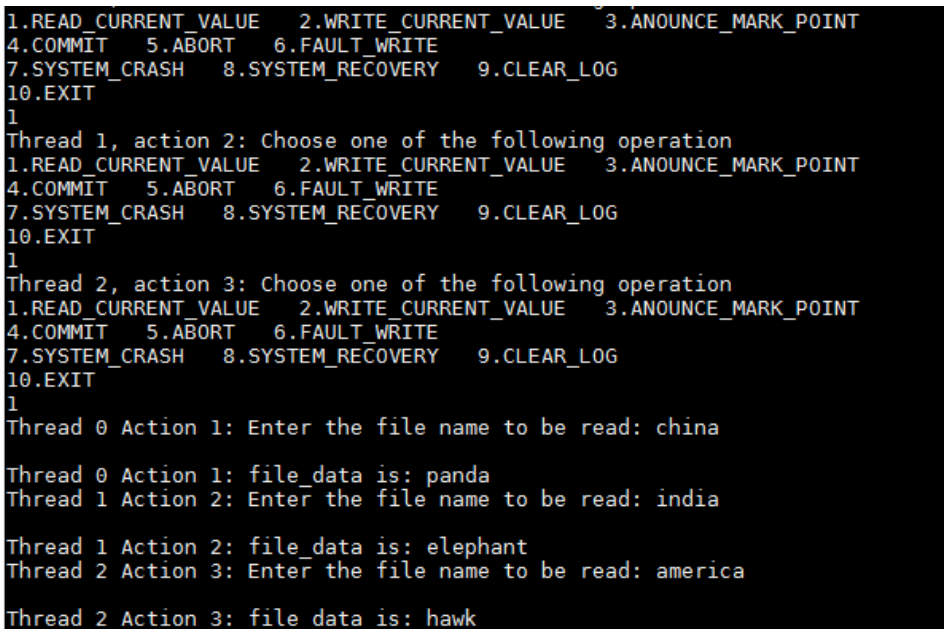
Screenshot of step three, here we can see that only action 1 runs again and again, that means action 2 and action 3 have been blocked.

```
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
0
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
0
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
0
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
0
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
0
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
```

Screenshot of step four and five, after announcing mark point for action 1, we can see action 2 and action 3 could run, and the three action could safely committed.



Screenshot of step six, the three files could be read successfully.



The test result completely meets the expectation!

3. Multithreads before and after atomicity test 2:

Steps	Expected Result	Reason for Choices
1. Action 1: write file 1 Action 2: write file 1 with different data Action 3: write file 2 2. Announce mark point in action 1, 2 and 3. 3. Commit action 2 and 3 4. Do some empty operations. 5. Commit action 1. 6. After the screen shows that action 1 and 2 has committed successfully, read file 1 in action 3	4. The commission in action 2 is blocked and action 3 commit successfully. 5. Soon after action 1 commit, screen will shows action 2 and 1 are committed successfully. 6. File 1 will be read as data in action 2. And file 2 will also be read correctly.	This test demonstrate the mark point discipline that the concurrent action commission will be blocked until all previous version of pending data have been committed so that the serialized order could be guaranteed.

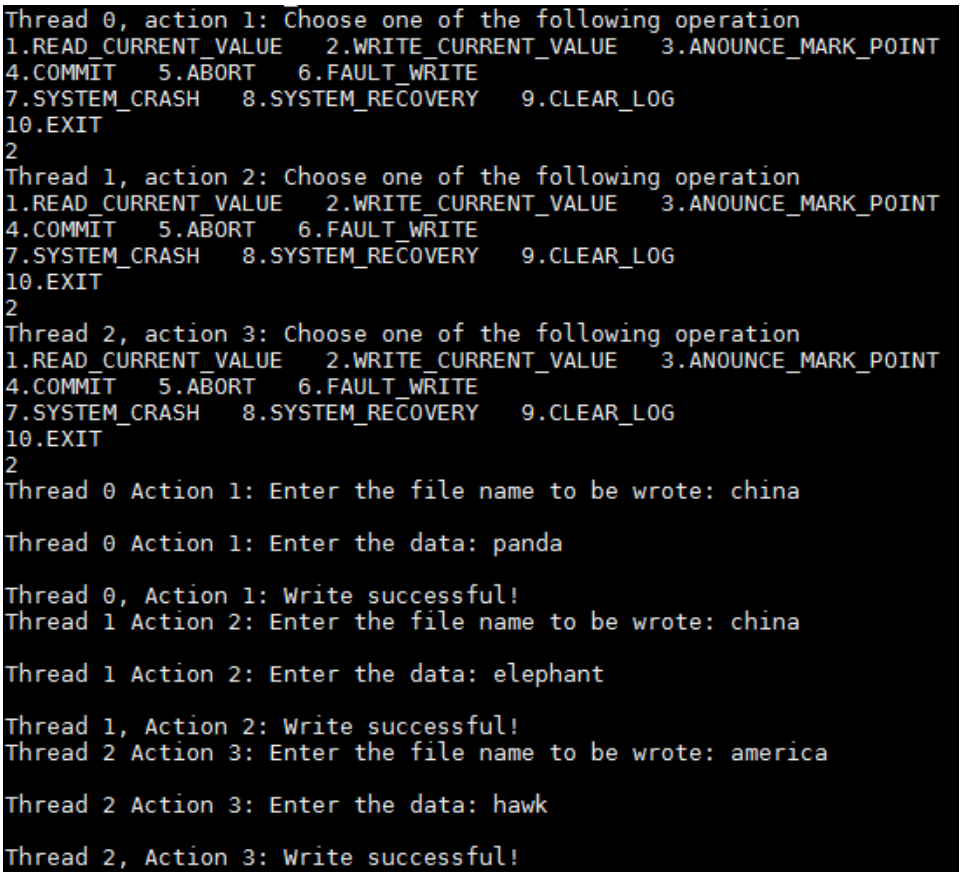
Test cases:

Action	File Name	File Data
1	**China**	**Panda**
2	**China**	**Elephant**
3	**America**	**Hawk**

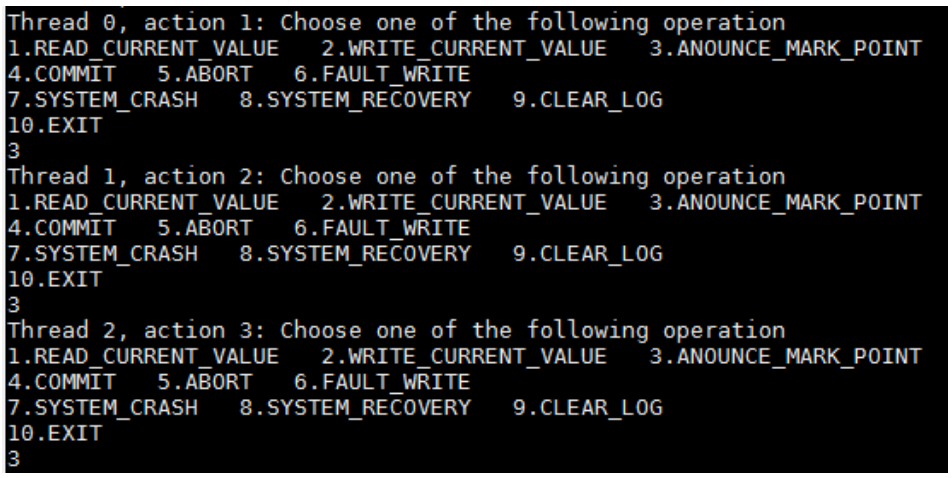
Test run screenshot and explanation:

Notice: The multi-threads is designed that every threads operation is executed one by one. Because every operation will obtain the mutex_lock, if one operation is not complete, it will block other threads' operation. So if you don't want to run the current thread, just press "0". It will do nothing but finish the concurrent thread's operation.

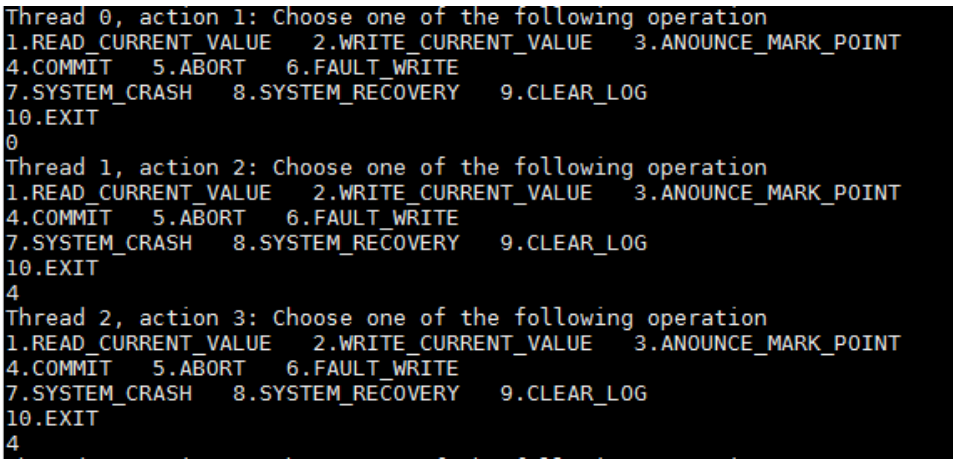
Screenshot of step one, we write the same file with different data in action 1 and 2, and write another file in action 3.



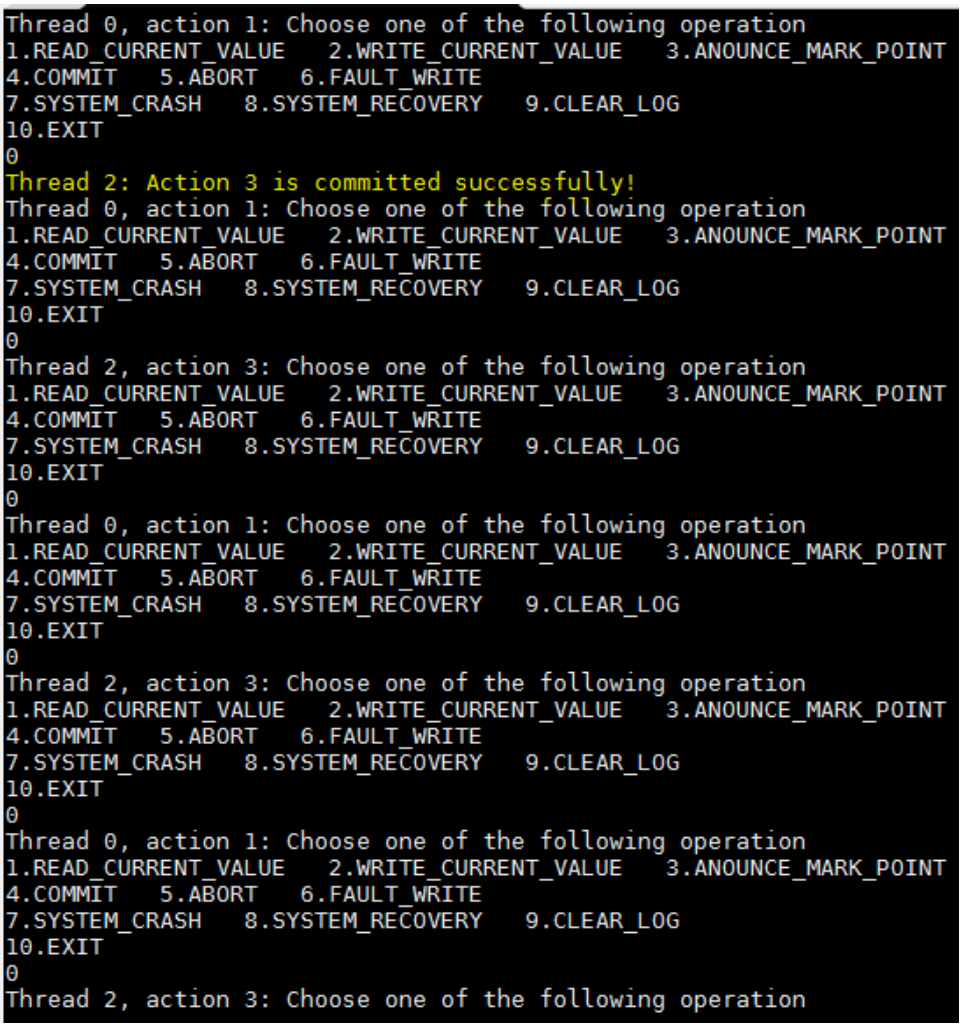
Screenshot of step two, we announce the mark point for every action



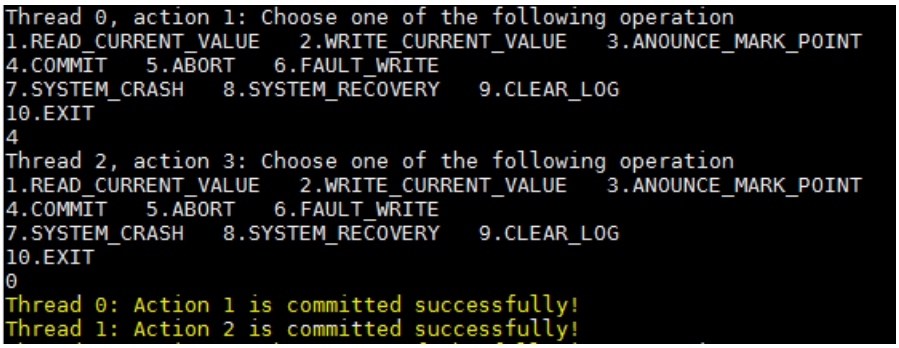
Screenshot of step three, we commit action 2 and action 3 but do nothing for action 1



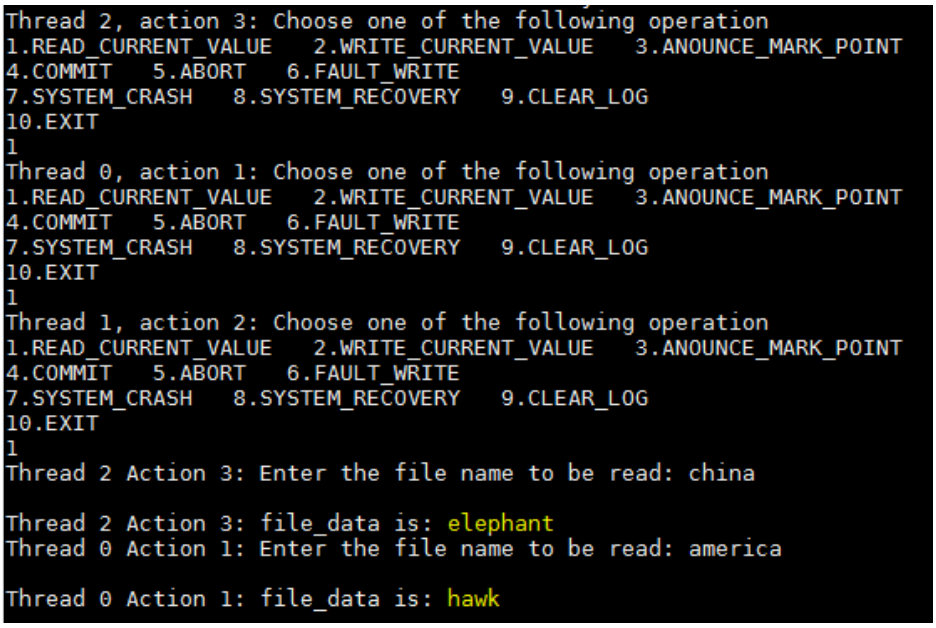
Screenshot for step four: we could see the action 3 can commit successfully but action 2 cannot come in the screen which means it has been blocked.



Screenshot of step five, after action 1 execute commit, soon the screen shows that action 1 and action 2 is committed successfully.



Screenshot of step six, when we read the two files, it shows that action 2 is committed successfully and overwrite the data committed by action 1, and the file action 3 committed could be read correctly.



The test result completely meets the expectation!

7. FAUTL_WRITE TEST:

Steps	Expected Result	Reason for Choices
1. Action 1: write file 1 Action 2: fault_write file 2 2. Action 1: announce mark point Action 2: announce mark point 3. Action 1: commit action. Action 2: commit action.	4. Only file 1 could be read.	This test demonstrates that the fault_write could be detected and could not installed into the cell storage. As a result, it could not be read and accessed by the user.

4. **Action 3: read file 1 and file 2**		

Test cases:

Action	File Name	File Data
1	**China**	**Panda**
2	**India**	**Elephant**

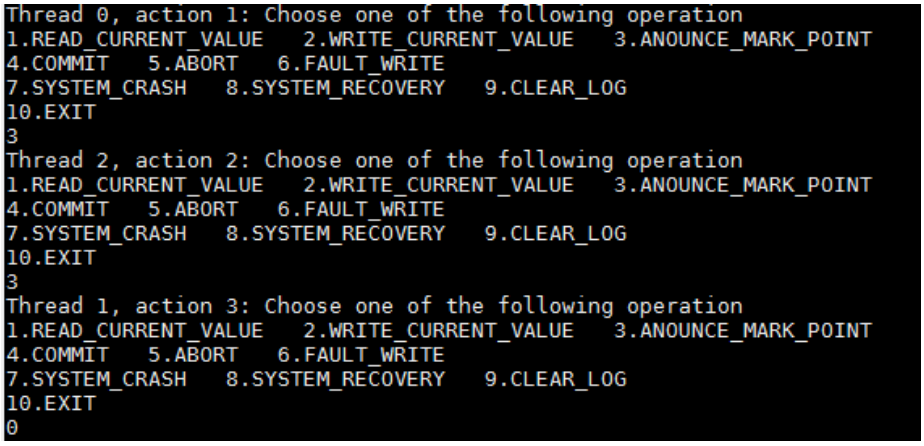
Test run screenshot and explanation:

Notice: The multi-threads is designed that every threads operation is executed one by one. Because every operation will obtain the mutex_lock, if one operation is not complete, it will block other threads' operation. So if you don't want to run the current thread, just press "0". It will do nothing but finish the concurrent thread's operation.

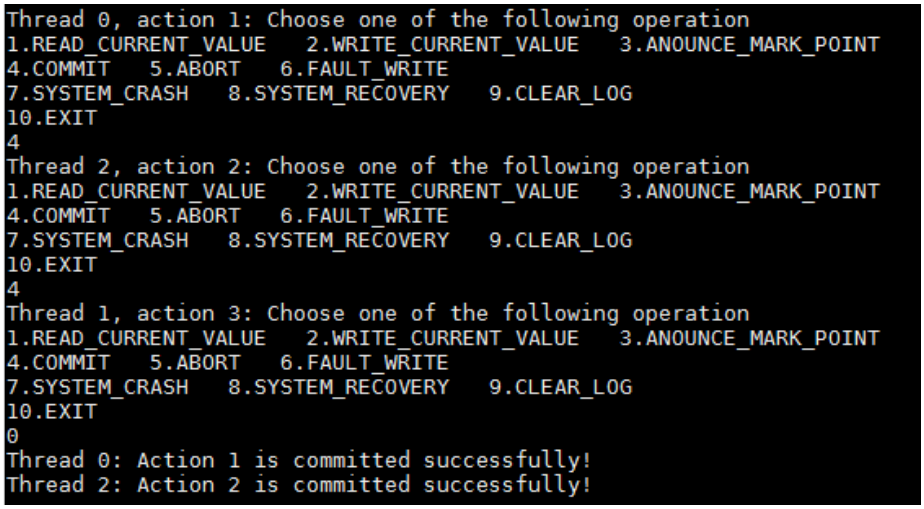
Screenshot of step one: write one file in action 1 and fault-write another file in action 2, do nothing for action 3.

```
Thread 0, action 1: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
2
Thread 2, action 2: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
6
Thread 1, action 3: Choose one of the following operation
1.READ_CURRENT_VALUE   2.WRITE_CURRENT_VALUE   3.ANOUNCE_MARK_POINT
4.COMMIT   5.ABORT   6.FAULT_WRITE
7.SYSTEM_CRASH   8.SYSTEM_RECOVERY   9.CLEAR_LOG
10.EXIT
0
Thread 0 Action 1: Enter the file name to be wrote: china

Thread 0 Action 1: Enter the data: panda

Thread 0, Action 1: Write successful!
Thread 2 Action 2: Enter the file name to be wrote: india

Thread 2 Action 2: Enter the data: elephant

Thread 2, Action 2: Write successful!
```

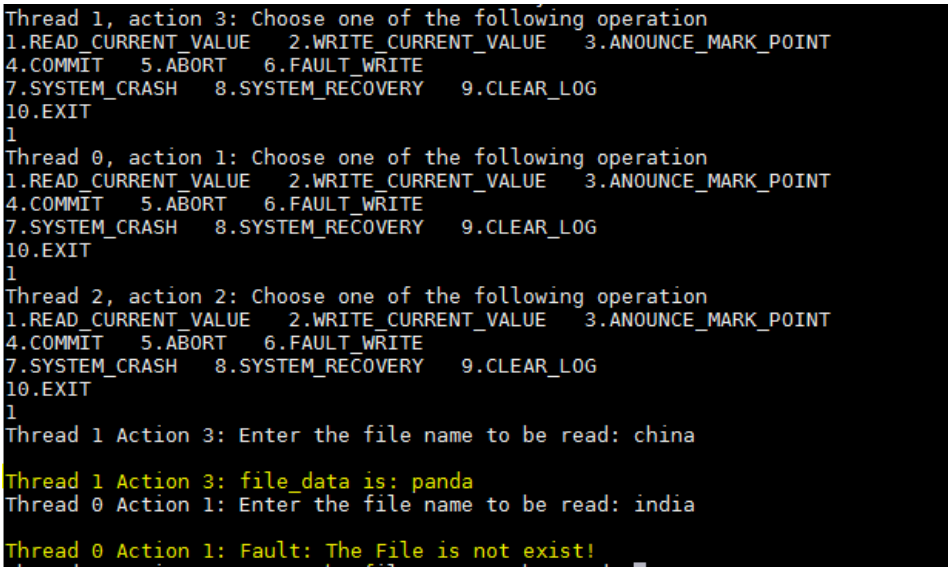
Screenshot of step two, announce mark point for action 1 and action 2, do nothing for action 3.



Screenshot of step three, commit action 1 and action 2, do nothing for action 3.



Screenshot of step four, read the two files, only the correctly-write file could be read.



The test result completely meets the expectation!

8. System crash and recovery:

Steps	Expected Result	Reason for Choices
1. In the action 1, 2, 3. Write, announce mark point and commit. 2. Exit action 1, 2, 3, initialize action 4, 5, 6. 3. Read the three files in any of the action 4, 5, 6. 4. System crash in any of the action 4, 5, 6. 5. Read the three files in any of the action 4, 5, 6.	3.All the data of the three file could be read and print in the screen 5. No files could be read.	This test is used to verify the program ability to recovery from a crash of the file system. In the test, we could see that the system_crash would erase all the data in the cell storage and the system_recovery could reconstruct the cell storage as system_crash never happen.

6. System recovery in any of the action 4, 5, 6. 7. Read the three files in any of the action 4, 5, 6.	7. ALL of the three files could be read, same result as step 3.	

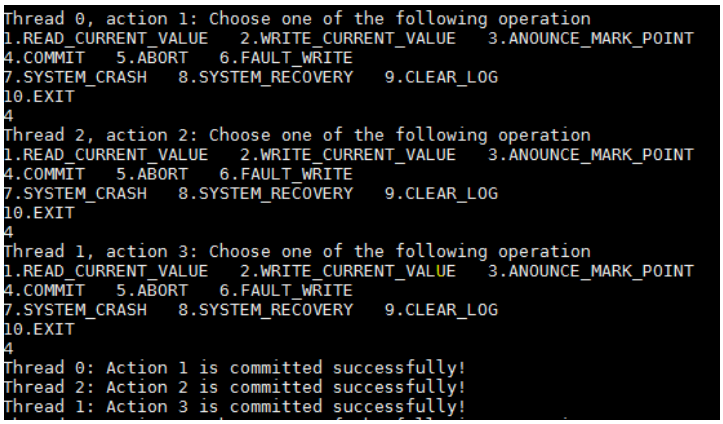
Test cases:

Action	File Name	File Data
1	**China**	**Panda**
2	**India**	**Elephant**
3	**America**	**Hawk**

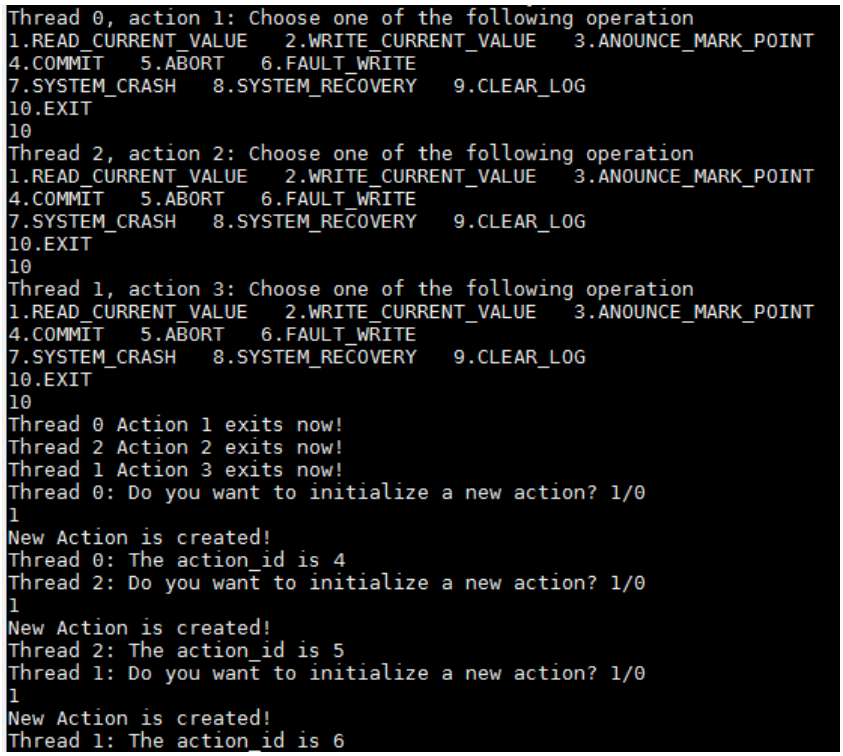
Test run screenshot and explanation:

Notice: The multi-threads is designed that every threads operation is executed one by one. Because every operation will obtain the mutex_lock, if one operation is not complete, it will block other threads' operation. So if you don't want to run the current thread, just press "0". It will do nothing but finish the concurrent thread's operation.

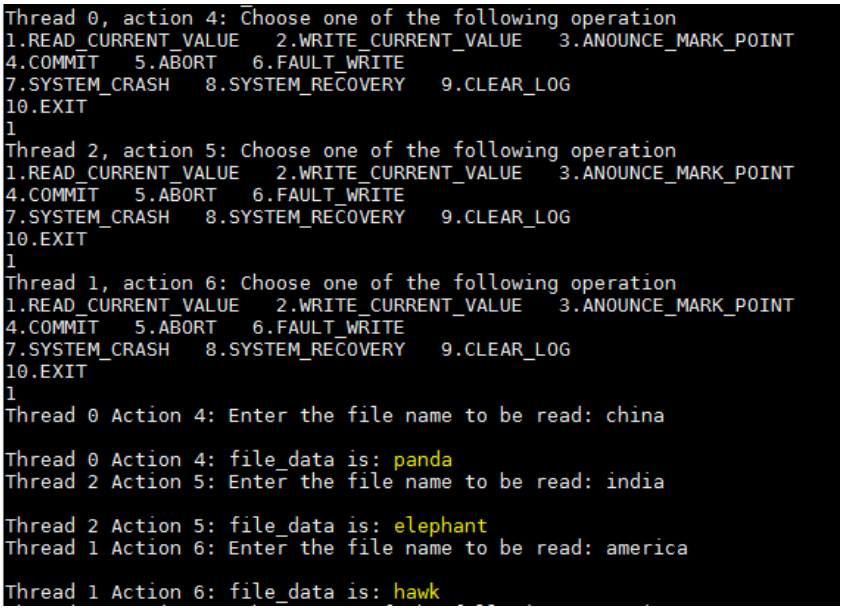
Screenshot of step one, there are some part of operation not included in the screenshot, but we do everything necessary to write, announce mark point and commit the three files in each of the three actions.



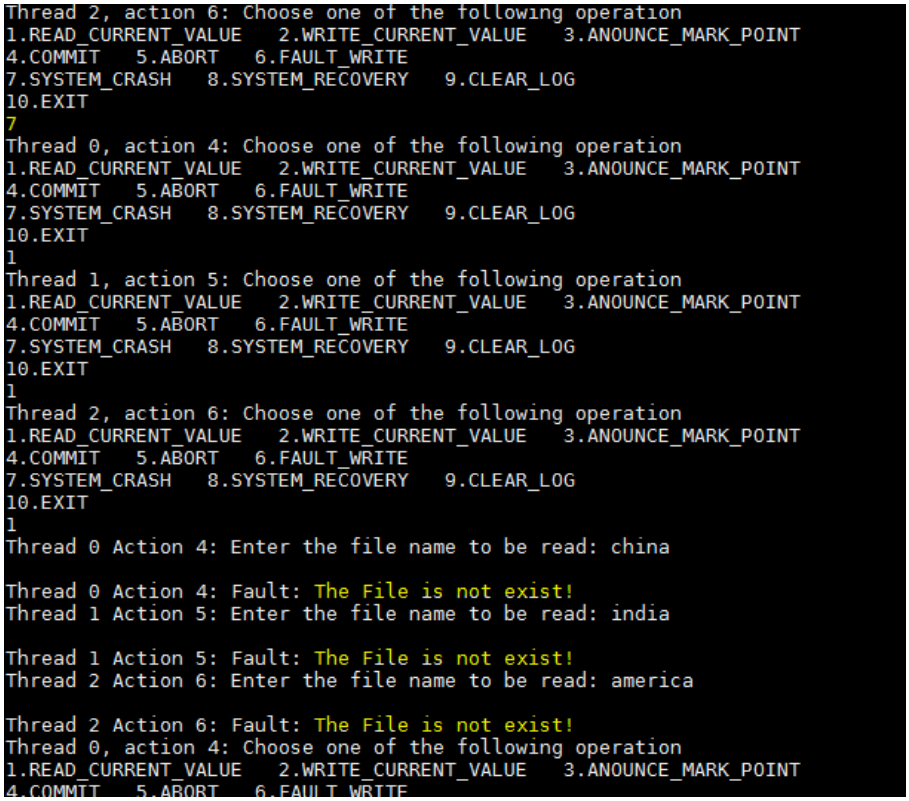
Screenshot of step two, we exit the concurrent actions and initialize three new actions.



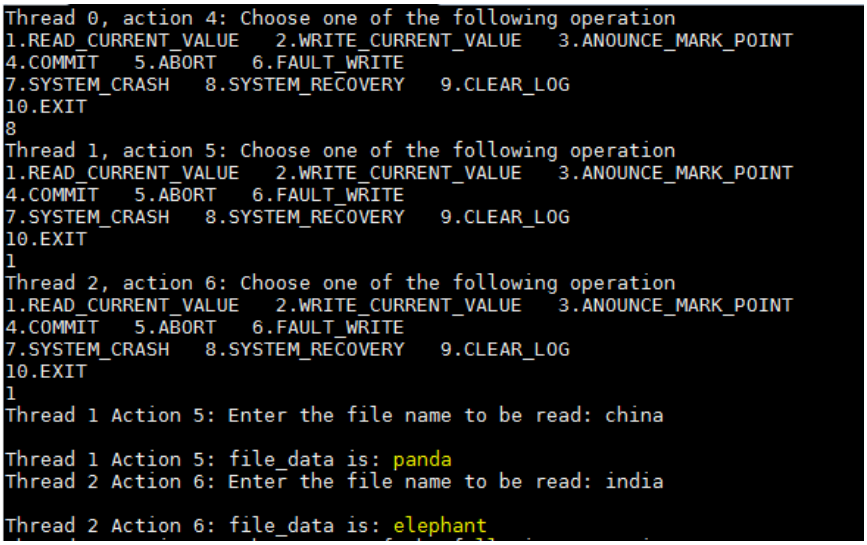
Screenshot of step three, the three files committed in the former three actions is committed correctly.

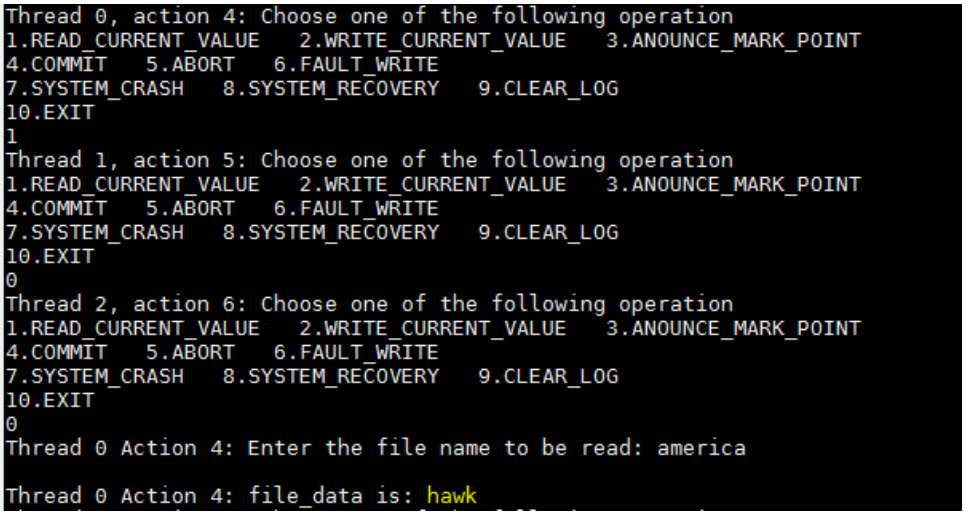


Screenshot of step four and five, as we execute the "SYSTEM_CRASH", everything committed is erased. So no file committed by former actions could be read.



Screenshot of step six, after system recovery. Every file committed before could be reconstructed after system crash, so every files could be read correctly.





The test result completely meets the expectation!

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 file system. In phase 2, I have implement the mark point principle described in the text book to guarantee the serialized ordering and increase the concurrency. In phase 3, I have utilized the log file for system recovery and did some implementation of fault injection. In this phase, I just modified the former phases and combine the techniques together to fulfill the project.

Lesson learned

In the four phases, I have got in-depth understanding in the following aspects:

a. Further understanding of the C language pointer.
b. The principle of journal file system journal storage and cell storage.
c. How to structure a file system.

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