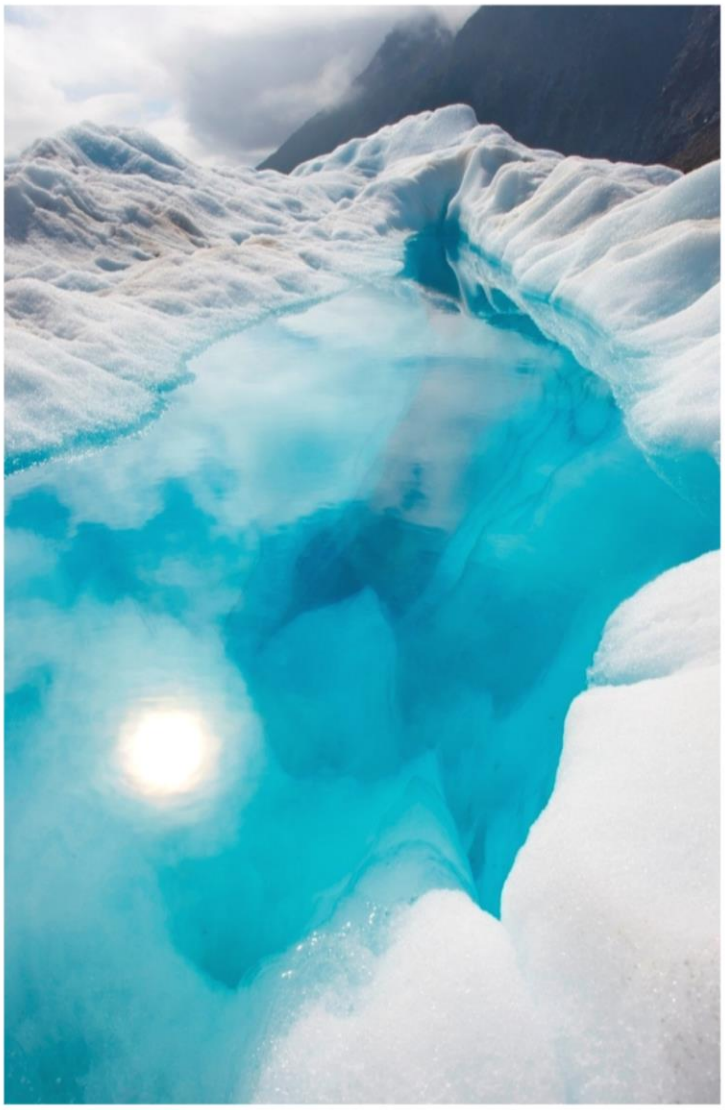


JOURNAL FILE SYSTEM

PHASE THREE

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

Overview:

In this phase, we are considering the journal file system in the error free environment. The phase work covers that the build-up of the primitive journal file system with two functional storages, and the application program interface for user to new_action, read_current_value, write_current_value, abort, and commit.

Assumption:

1. The maximum number of files which could be stored in the journal file system is 128.
2. The maximum size of each file is up to 128 Byte, and the size of file name is at most 8 Byte.
3. In every action, at most 8 pending operation(writes) is allowed.

Implementation procedures:

As shown in the graph below, this system provides the following API: NEW_ACTION, WRITE_NEW_VALUE, COMMIT, ABORT, EXIT. We will describe how these API implement below:

1. NEW_ACTION:

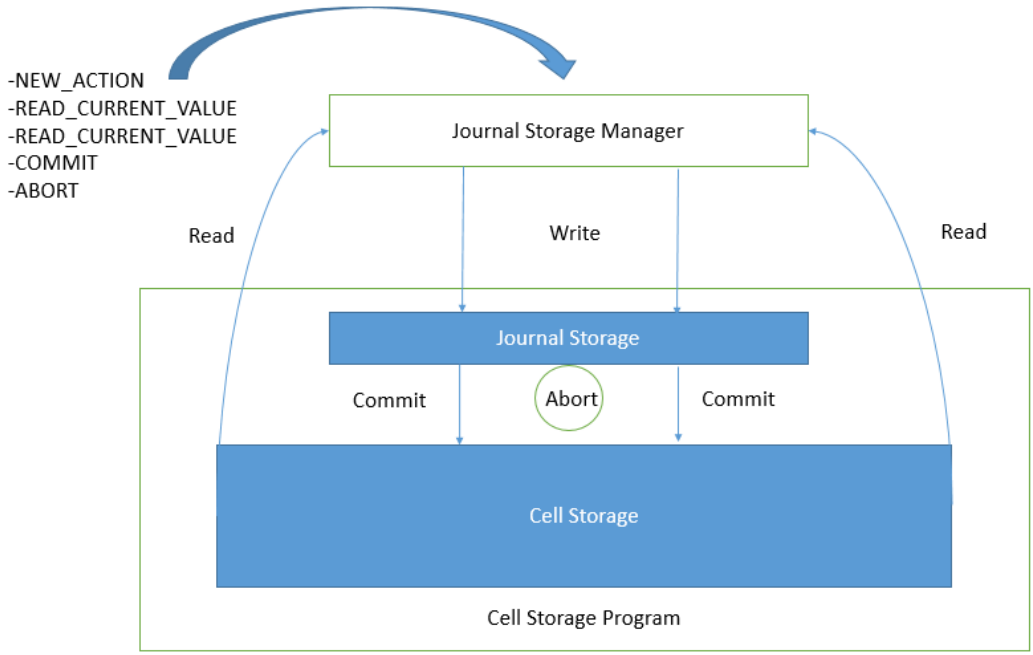
In the journal file system, actions are serialized in every thread. So, the initialization of a new action could only be implemented after the end of the former action. As the new action initialized, the journal storage management will allocate a new storage space for further implementation.

2. 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.

3. 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.



4. 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.

5. ABORT:

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

6. EXIT:

This means the end of an action.

Test cases and reasons for the choice

1. Journal storage management API test:

Steps	Expected Result	Reason for Choices
1. Initialize action 0, write two files. **2. Read these two files**	**2.The screen returns "file does not exist" Because the files do not commit to the cell storage**	**It demonstrates what happen if the file be wrote but not commit**

3. Commit one of the former two file and read it 4. Abort another file. Read it 5. commit the file being aborted	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 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

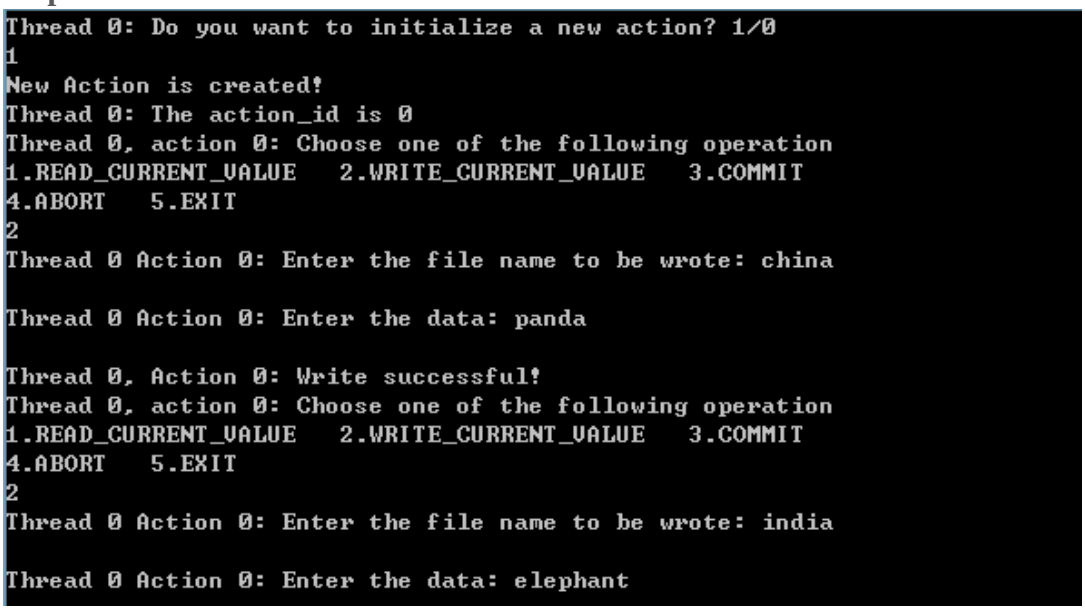
Test cases:

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

Test run screenshot and explanation:

1. **Journal storage management API test:**

i. Step one screenshot:



ii.Step two 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: 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.ABORT   5.EXIT
1
Thread 0: Enter the file name to be read: india

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

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
```

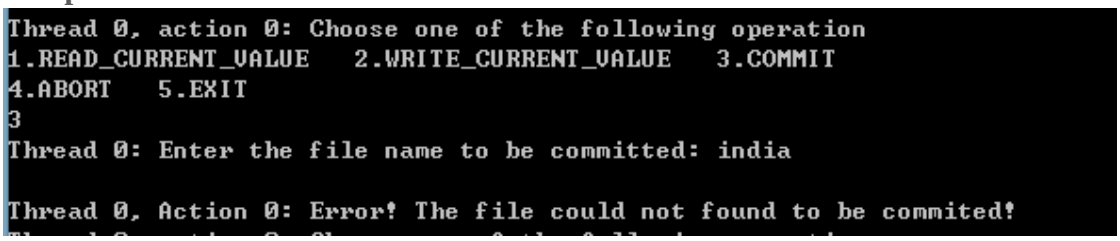
iv. Step four screenshot

```
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



Result and conclusion:

As the screenshot shown above, the test meets all the expectation discussed before to demonstrate the API functions.

Summary:

In this phase, 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.

Lesson learned

In this phase, I have got an 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.