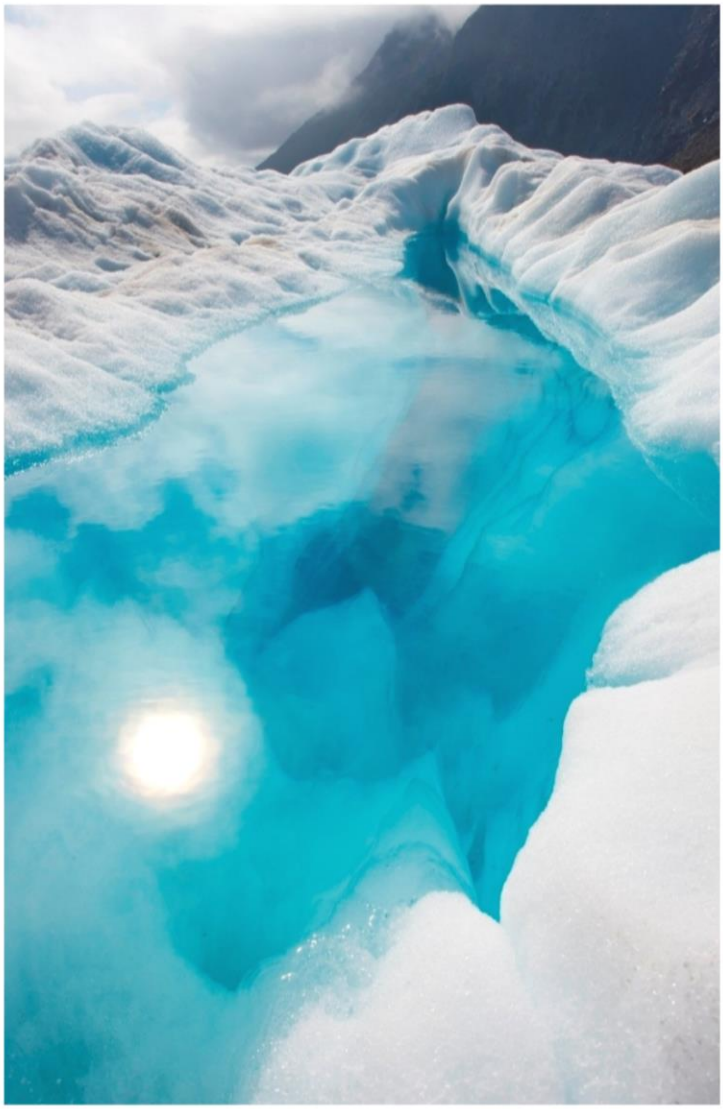

JOURNAL FILE SYSTEM

DESIGN DOCUMENT

Mengyuan Zhang 1209583385 | CSE 536 Dr. Sandeep Gupta| March 25, 2016(modified version)

1. Project Requirement

The objective of this project is to implement a file system using the C programming language and to test this file system for the following properties:

a) Error Free environment
b) Error Free environment with multi-threading(before or after atomicity)
c) Error prone system without multi-threading
d) Error-prone system with multi-threading

As shown in the figure below, The "Cell Storage System" can be abstracted as a file with multiple sectors, and it implement the "Read", "Write", "Allocate" and "Deallocate" procedures. The "Journal Storage Manager" should have all the logic to run all the scenarios mentioned above and have support for multi-threading.

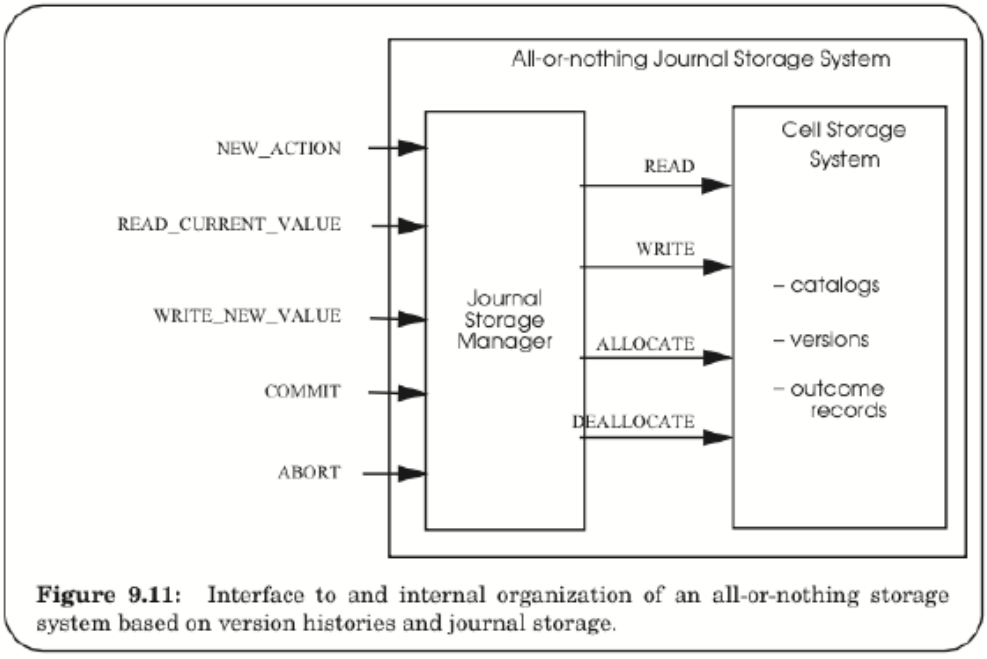

Figure 9.11: Interface to and internal organization of an all-or-nothing storage system based on version histories and journal storage.

2. Design Details

2.1. FILE SYSTEM ARCHITECTURE

In this journal file system, we are choosing one of the common logging configurations described in the textbook and shown as below. The log resides in non-volatile storage, and cell storage resides in non-volatile storage along with the log.

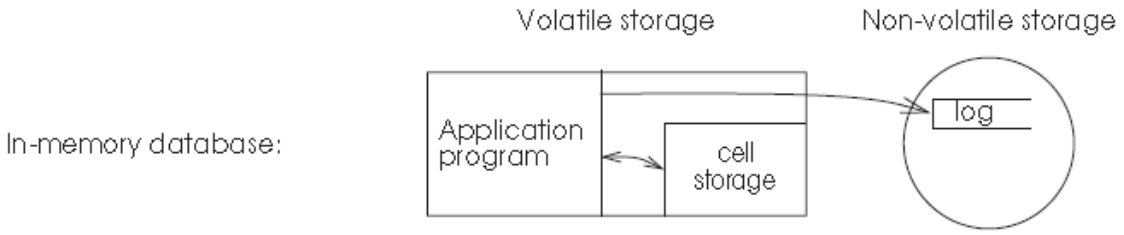

Figure 2. 1 logging configuration in this project

Specifically speaking, we are designing the journal file system as close as figure 9.11 in the textbook, though there are some differences. The Journal Storage Management is providing the user interface including "NEW_ACTION", "READ_CURRENT_VALUE", "WRITE_NEW_VALUE", "COMMIT" and "ABORT", and veil lower program interface such as "Write", "Read", "Commit", "Abort" and so on.

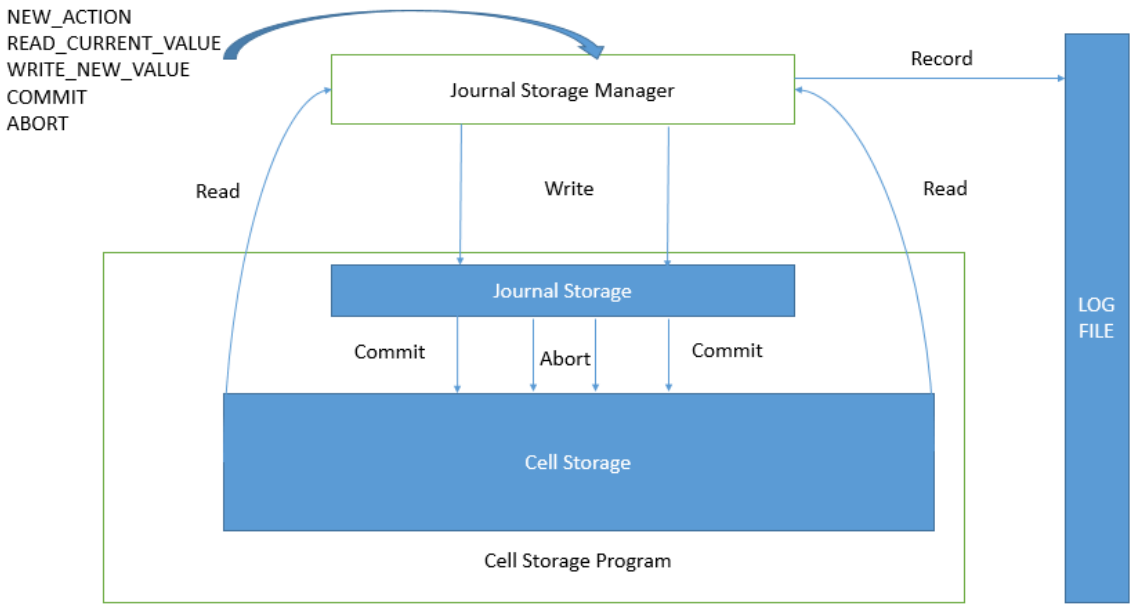

Figure 2. 2 Journal file system architecture

2.2. THE FUNCTION OF JFS COMPONENTS

2.2.1. JOURNAL STORAGE AND LOG FILE

In the textbook, the journal log is used to log the changes of data in the actions or transactions and it installs the change in cell storage. It maintains the version history and

could reconstruct the cell storage in case the cell storage is lost. Based on the observation that only the version history of pending data is needed to install the cell storage and ensure the All or Nothing and Before or After atomicity, and only the version history of committed data is needed for reconstruct the cell storage. So we could safely divide the journal storage into two parts, journal storage and log file, and it could largely simplify the implementation and save storage space.

So, the journal storage only maintains the new version of pending data with their action id as the form below. The characteristic of journal storage is:

a) The maximum number of files be maintained is 1024.
b) The size of file name is up to 8 byte.
c) The maximum action id is limited to 255. (The OxFF is defined as the fault write.)
d) The size of file data is up to 128 byte.
e) The same file in different version should be kept in different file structure.(file unit)

Bitmap	File 1 name	File 1 action ID	FILE 1 Data	…	File 1024 name	File 1024 action ID	File 1024 Data
128 Byte	8 Byte	1 Byte	128 Byte		8 Byte	1 Byte	128 Byte

The log file is used to maintain the commit history and redo & undo information. The log file could be viewed like the figure below:

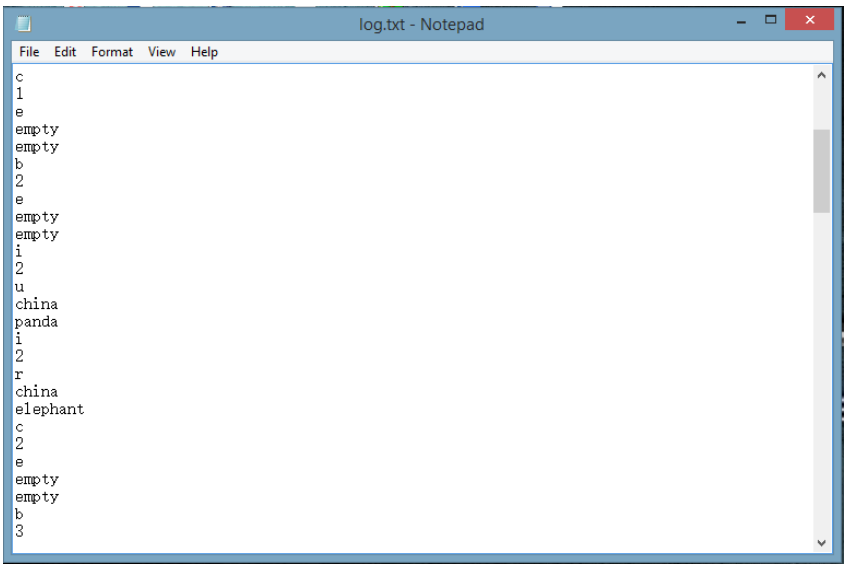

Figure 2. 3 the log file

There is one departure from the design of textbook. As described in the textbook, the journal log is supposed to be maintained in the non-volatile storage. However in this design, only log file is non-volatile. So we have to make this assumption:

> Assumption 1: We suppose the journal storage is kept in the non-volatile storage but actually it is implemented in volatile storage.

2.2.2. CELL STORAGE

The cell storage, in some sense, could be viewed as the cache of the storage log, so that it could intensively speed up the read. The cell storage only maintains the files that have already be committed and the storage structure is shown as below:

File 1 name	File 2 name	…	File 128 name	bitmap	File 1 data	…	File 128 data
8 Byte	8 Byte		8 Byte	16 Byte	128 Byte		128 Byte

The characteristics of cell storage is:

a) The maximum number of files be maintained is 128.
b) The size of file name is up to 8 byte.
c) The size of file data is up to 128 byte.

Because of the characteristics of both cell storage and journal storage, we have the further assumptions:

> Assumption 2: The size of file data is no more than 128 byte, the size of file name is no more than 8 byte.

> Assumption 3: the maximum number of file pending is 1024, the maximum number of file could be read in the cell storage is 128.

2.3. IMPLEMENTATION

From the beginning to the final phase, the journal file system totally provides 11 user interface based on the primitive interface described before. The implementation procedure will be described now:

a) 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)
b) 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.
c) WRITE_NEW_VALUE: The operation writes the file into the journal storage by the file name and file data provided by the user. We will make an assumption here:

> Assumption 4: Every action could only write the same file one time.

d) 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. At the same time, the operation record would be kept in the log file.
e) ABORT: This operation aborts every pending files updated in the current action, so that the pending files will be removed in journal storage.
f) 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.
g) FAULT_WRITE: This operation is simulating a fault injection, in this operation, we assume:

> Assumption 5: The fault write could be detected and report to the upper interface.

The program could know the file is fault wrote by detecting if the action id stored in Journal storage is 0xFF. And the fault write data cannot be committed successfully.

i) SYSTEM_CRASH: This operation simulate the program crash so that all the data in cell storage is erased.

j) SYSTEM_RECOVERY: This operation would reconstruct the cell storage after system crash, the system would appear the same as before system crash happen.
k) 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:

> Assumption 6: The log size could not beyond 100

l) EXIT: This operation will close the current action.

2.4 MECHANISM AND TECHNIQUES

2.4.1 MARK POINT PRINCIPLE

Mark point principle is used to guarantee the before and after atomicity as well as increasing the concurrency in the multi-thread program. Basically, the action id specifies the order in which should be implemented. And the principle has the following requirement, which could be our additional assumption:

> Assumption 7: In each action, before mark point, writes are allowed but commits are not; after mark point, commits are allowed but writes are not.

So the current action could write and commit only after all the former actions have announce the mark point, and when the current action commits the file, it need to wait the former version files pending have been committed.

2.4.2 SEMAPHORE & PTHREAD

To implement multi-threads program, p-thread and binary semaphore is used to create the threads except for the main thread. The binary semaphore functions the same as the mutex_lock so that it could allow only one operation work and operations on other threads would be blocked.

2.4.3 FAULT PRONE ENVIRONEMNT

In this system, we assume that:

> Assumption 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.

So the fault write cannot commit to cell storage and the cell storage could be reconstructed by log file.