Design Document

CSE 536

Name: Vishal Srivastava ID No.:1209824652

Project Description

We are designing a Journaling File system(JFS) to be used over a Unix system calls. It implements the all or nothing and before or after atomicity models. As normal file system generally writes the value on the same place, so in case of faults occurring during these writes, it might just corrupt the value of the entry. In order to handle such situations, we tend to make our commit operations atomic and for that purpose we are implementing the Journal File System which provides robust writes through atomicity.

The user is allowed to interact with the file system through available API. The API internally implements the logic to provide fault tolerant system model. The API accepts any read/write operation using semantics similar to read/write Unix semantics. This provides portability to switch to either Unix to JFS and JFS to Unix without making change in the application program. Also the API is able to handle multithreaded environments and concurrency conditions are handled internally.

Design Considerations

We are making certain assumptions for the model which may or may not be true in real life. As this fault tolerant system we are handling faults like power failure (which can corrupt memory during writes), bad writes(interrupted during writes) and memory overflow(overuse memory beyond allocated). The following are the assumptions we are making while designing the system:

- We are assuming that the underlying system which implements the Unix File system is robust and we will not encounter any fault from that file system.
- For phase I we assuming that the underlying system is fault tolerant and we will not encounter any problems or faults with the underlying file system. Also the system is designed for single threaded environment i.e. only when process/thread can write on the system.
- For Phase II, we are assuming that the underlying system is still fault tolerant and we will not encounter any problems or faults with the underlying file system. But now the

system can handle multithreaded environment i.e. it can handle multiple read/writes simultaneously.

- For Phase III, we are now considering that the system to be faulty. We will introduce faults artificially in the system to mimic the real life faults experienced in a file storage media. The system will be designed to recover from these unexpected faults. For this we are making a single threaded fault tolerance and we will later introduce multithreading in phase IV.
- For phase IV, we are still now considering the system to be faulty and the file system will try to recover from these faults but now we are also introducing multithreading environment which will increase the chance of faults due to concurrent operations.

Structure

Basic user interface is designed for command line interface making it suitable for any kind of linux systems.The current system is constructed in three parts:

1. API
2. Journal Storage Manager
3. Cell storage system

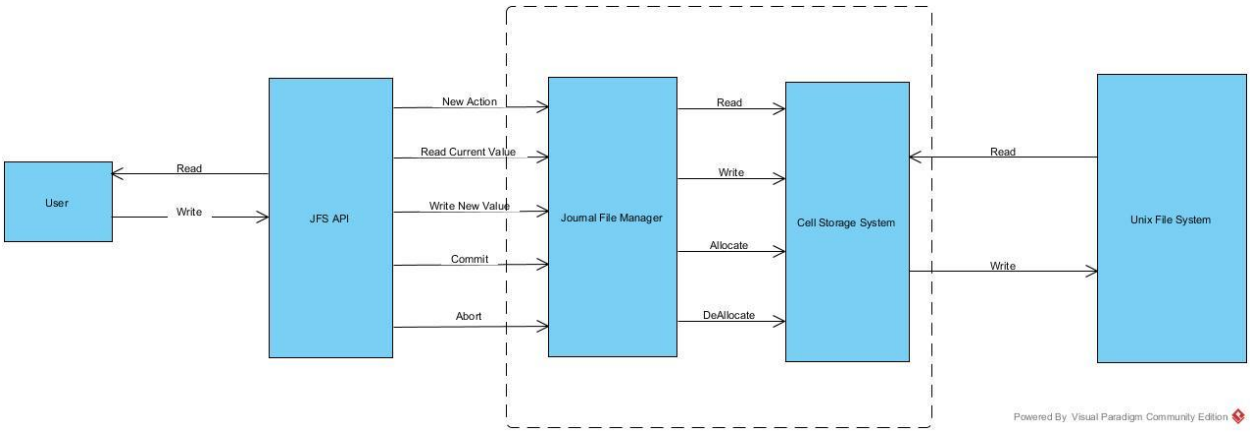

Figure 1: Journaling File System

API

We are designing an API that will provide the user with to interact with the Journaling File System using read and write commands having semantics similar to the Unix semantics. For now, we are implementing only read and write and later may include more functionalities. JFS API convert this read and write request to New action, Read Current Value, Write new value, Commit or Abort based on the context and deployment. Design is still amorphous and may change in later phases. The following functions/interfaces are thought about:

1. Read
   It uses the standard unix type read using file handles. They might not be compatible with the standard unix calls with respect to structures but will perform the fairly same way.
2. Write

   It also uses the standard unix type write using file handles. They might not be compatible with the standard unix calls with respect to structures but will perform the fairly same way.

Journal File System

Consists of two units:

1. Journal File Manager
   Journal File Manager manages all the incoming calls to the system and manages allocation, write, read and deallocation of cells from the cell storage system based on the requirement. It also manages a record called outcome record which contains logs for all the all-or-nothing actions and their states. Following are the functions used:
   1. New Action
      Inputs: Integer type process_id
      Output: Returns Integer type standard error codes or file_id
   2. Read Current Value
      Inputs: Integer type process_id, file_id, cell_no
      Output: Returns string type data from storage
   3. Write New Value
      Input: Integer type process_id, file_id, cell_no
      Output: Returns Integer type standard error codes
   4. Commit
      Input: Integer type process_id, file_id
      Output: Returns Integer type standard error codes
   5. Abort
      Input: Integer type process_id
      Output: Return standard status codes

2. Cell Storage System
   It is the container for storing all the new data value and the identifier of all-or-nothing actions. Data is stored as cell blocks which are implemented by files. Each block can hold upto 1024 bytes of data. It is managed by Journal File System and provides interface for Read, Write, Allocate and Deallocate. Later it implements the all-or-nothing-put to store

the data using Unix file system. It deals with exceptions such as It consists of catalogs, versions and outcome records. Following are the functions used:

1. Allocate
   Inputs: Integer type file_id
   Output: Returns Integer type standard error codes
2. Write
   Inputs: Integer type file_id, cell_no. String type data stored
   Output Returns Integer type standard error codes
3. Read
   Inputs: Integer type file_id,cell_no
   Output: Returns String type data stored
4. Deallocate
   Inputs: Integer type file_id,cell_no
   Output: Returns Integer type standard error codes

Unix File System

In Unix file system we are using the underling system call to commit an entry to the Unix File System which will be stored in a volatile memory. The architecture and system call is provided by the underlying operating system, so not much changes are expected in this unit.

Design Concepts

Here we will be discussing the design concepts that will be used in the project.

1. All-or-nothing atomicity
   This is done to insure that writes are fault tolerant. So, in cases of faults we don't expect any corruption of data. In this type of atomicity either the data is written to the system completely or the previous value is not changed at all. It could very well have multiple steps but all the steps are ensured to be completed before moving on the other actions.

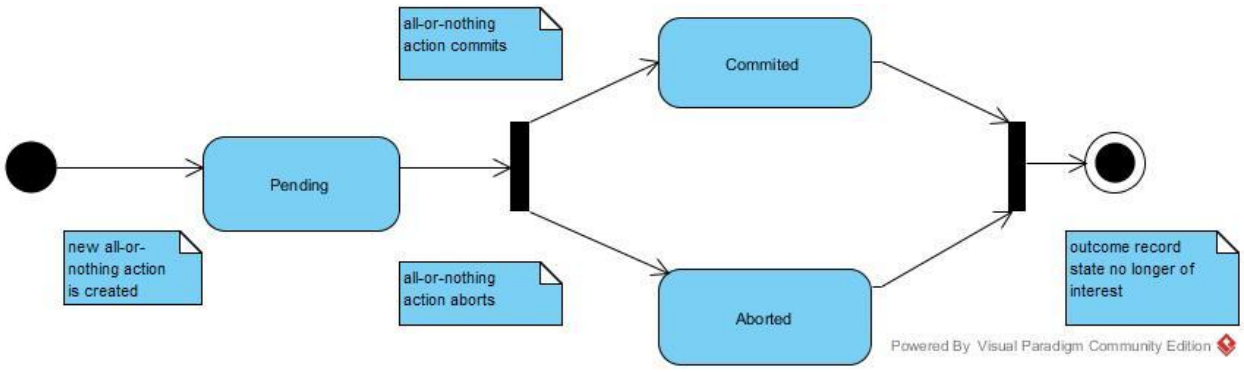

Figure 2:Implementation for All or Nothing (Reference: Principle of Computer System Design by Jerome H. Saltzer and M. Frans Kaashoek)

2. Before-or-After atomicity
   In this type of atomicity, we have a guarantee that if there are two concurrent process operating on the same object then either of them will be processed either before or after the other process. This kind of scenario is handled/maintained using locks and leases.

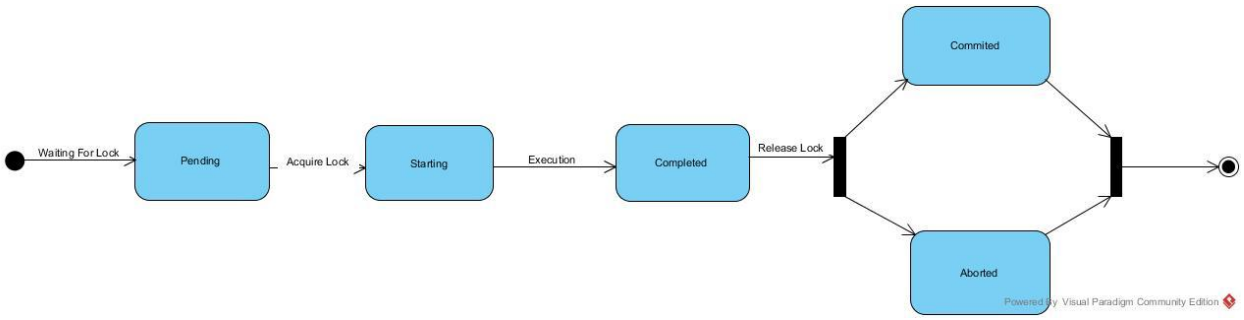

Figure 3: Implementation of Before-or-After using Locks (Reference: Principle of Computer System Design by Jerome H. Saltzer and M. Frans Kaashoek)

3. Version History
   Refers to the history created for each operation on values of the application data. The data is pushed in with an id assigned by journal file manager along with its state as pending. Then the value is either committed or aborted and in either case is discarded. This help in backtracking when we need to find the last committed value.
4. Logs
   Logs are advanced way of bookkeeping where we generally append with either of an operation and doesn't remove an entry from it. Even if we have to backtrack an action we generally add another entry in the log that nullifies the previous action but never delete it. Logs are used in times of faults and with its help we either process the request or rollback the operation.
5. Exceptions
   Certain exceptions have been taken care of and more will be added as design progresses. Exception like incorrect usage of cells by different file_id other than the allocated will throw an error to the user. This is inorder to maintain data security and avoid tampering of data other than the allocated file and process. Also we are using blocks of certain size. If the user tries to put more content then the cell size it will throw and error. Also If user tries to read unallocated spaces in the cell storage, it will throw an error. For now these exceptions have been taken care and more will be added later parts of design.

Design Decisions

For implementing the two types of atomicity we are considering the following:

1. All-or-nothing implementation
   Out of the two ways (Logs and version history) of implementing the ideology we are going with logs. Logs are one of the most robust ways of maintaining and recovering from fault tolerance. At this scale of implementation, logs seem to be most viable option. Also as we need to store that on non-volatile memory, they seem easier to implement instead of version history. As we are considering the design time also into consideration, we will go for logs for now.
2. Before-or after implementation
   Out of the many ways to implement it, we are considering it to implement through locks. Locks are easier to manage can provide serialization at the required critical sections. It might not be the most robust way of implementing the atomicity but if handled properly can be fast, efficient and simple way of implementing serialization.

As these are based on premature evaluation of the design they are subjected to changes. The later changes will be projected and highlighted in the later phases of the project document.