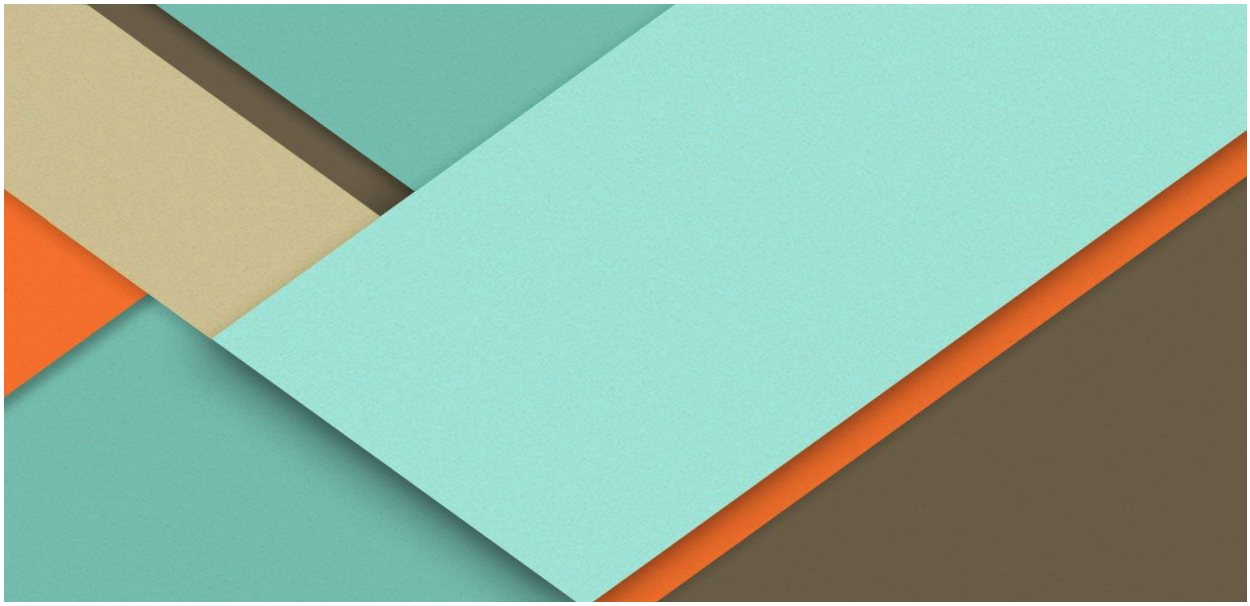


Cloud Computing

Fault Tolerant File System

Assignment 1

17.08.2017

Submitted by: [GROUP 15]

Aditi (2014CS10205)
Aditi Singla (2014CS50277)
Aditi Gupta (2017BSY7508)

File Systems

A File System is a structured data representation along with a set of metadata, which describes the stored data. It is used to keep track of files on a disk or a partition, i.e the whole organisation of files on the disk. File system operates on blocks, which are basically group of sectors that optimize storage addressing. Generally, block size ranges from 1 to 128 sectors. Different operating systems use different types of filesystems, based on their diverse requirements.

Current File systems have directory structures which keep the file attributes and pointers to data blocks. In case of a block read error, the information lost depends on information content of the block. In case the block contains very critical information of the directory structure, the whole file systems may become inaccessible. This has been discussed below for different filesystems :

Linux : Ext3, Ext4 File System

1. **Boot sector error :** We need to boot it using a CD. The files/inodes can be recovered.
2. **Inode read error :**
    a. Case 1 : Inode for file : The whole file is lost.
    b. Case 1 : Inode for file : All the files in that directory will be lost since we have lost the pointers.

    In both cases, loss is not proportional to the number of unreadable/corrupt blocks.
3. **File read error :** Only the data contained in the unreadable blocks is lost. Hence, the loss is proportional to the number of blocks corrupted.

MacOS : HFS+ File System

1. **File read error :**
    a. At the filesystem level, when the filesystem is checked and if a block is corrupt or unreadable, it is zeroed and reallocated to a bad block inode table.
    b. At the hardware level, the disk drive automatically remaps inaccessible sectors to unused sectors in a way that is transparent to the OS.

Windows : NTFS System

1. **File read error :**
    a. It dynamically remaps the cluster containing the bad sector and allocates a new cluster for the data, as well as marking the cluster as bad and no longer using it.

Q1. Minimisation of Information Loss

We observe that the current file systems may cause loss of lot of data, especially in cases where the inode corresponding to directory or even file gets corrupted/inaccessible. To address this issue, we propose a UNIX based filesystem, with the following modifications:

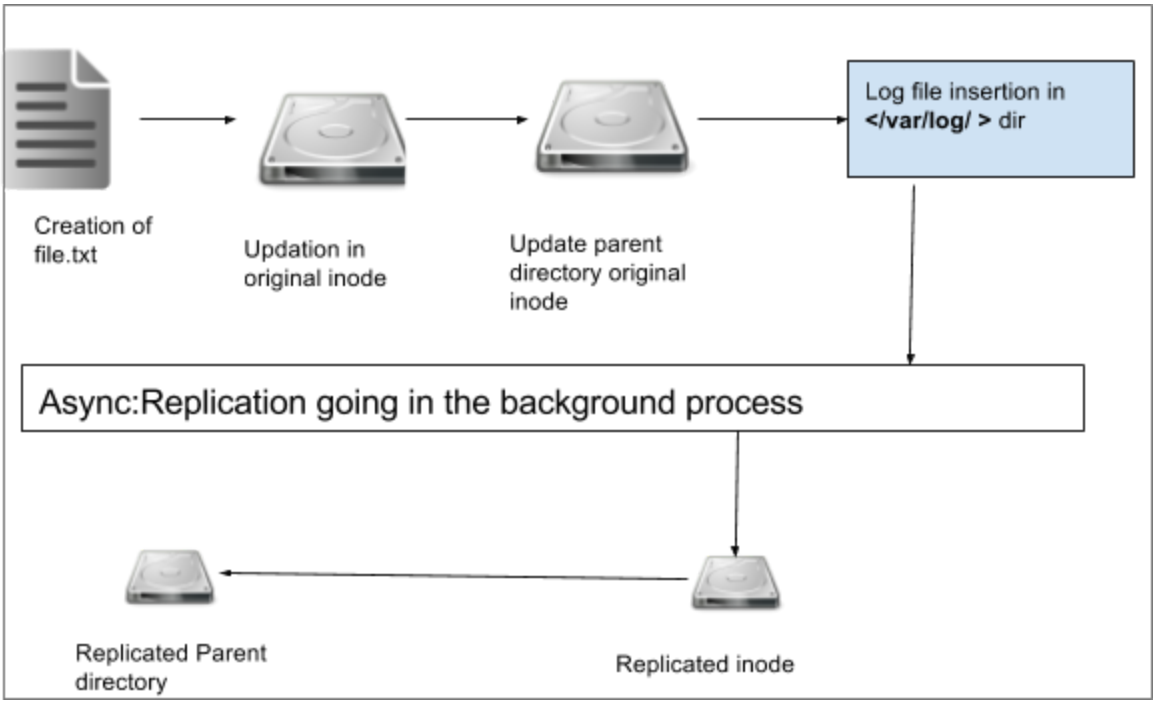
Preventing Metadata Loss

1. We maintain an Operation log file that contains any kind of updates done to the inodes. It would exist on two different locations on the disk, to avoid losing it.
2. The OS can also keep adding its own state as checkpoints to the log file, so that in case of recovery, one can pick up the last clean checkpoint and attain that state.
3. This log file is then used as reference to update the second copy of the inodes in the background.

Explanation using an example

Let us consider some operation such as creation of a file. The following diagram presents the steps that would occur:

Figure1



In the proposed file system, following steps will occur (corresponding to Figure1):

1. A new inode is created.
2. The inode of the parent directory is updated with the pointers and sizes for this newly added inode.
3. An entry is inserted into the log file in memory.
4. A background process keeps reading the log file and updates (and creates a new one) the inodes at the replicated location. This happens asynchronously with respect to the earlier steps.

Pros

We can observe that in the proposed file system, loss is always proportional to the number of blocks that get corrupted/unusable.
Let us consider different kinds of block read errors :

1. Boot sector / Super sector error :
   a. BIOS will first try to boot via the first copy of the boot sector. If it returns an error, BIOS will now point to a known location where its copy exists.
   b. Hence, assuming that both the copies of these sectors does not get unusable at the same time, there will be no loss of data on losing any one copy.

2. Inode sector error :
   a. In case any sector containing some inode data becomes corrupt/unreadable, the OS will need to :
      i. First, read the log file and check if there were any updates to that inode which have not been applied to the replicated inode.
      ii. If there are any such updates, it will first need to apply all the changes in the log file to ensure a consistent replica.
      iii. Now, the OS can refer to the location on the disk that contains the replica, and access the required information from there.
   b. Hence, with the assumption that both the locations containing the inode data cannot be lost, this design ensures no data loss in case of an inode sector error.

3. Data Sector Error :
   a. In the case when the corrupt block actually contained data of some file, we will not be able to recover it since that block had the only copy of the data. Since the file inode contains pointers to all data blocks, the rest of the blocks still remain accessible.
   b. Hence, loss will be proportional to the number of blocks lost in this case.

4. System Crash :
    a. In this scenario, our log file will be very useful in recovering the system to a consistent state.
    b. The latest clean checkpoint of the OS can be used to recover OS state.
    c. The operations logged in the file but not yet implemented can be read and applied.

Cons

1. This scheme uses double memory for some part since we are storing two copies of boot sector and the metadata comprising of all the inodes.
2. Consequently, any action that requires an update in any inode will now involve modifications at the original copy, as well as adding an entry to the log file regarding the change. The update at the replicated location will be executed by the background process reading the log file asynchronously. This is an overhead to ensure minimal loss.

Q2. Completely Fault Tolerant Systems

In the solution proposed above, we cannot recover data from a block if it gets corrupt / unreadable. To make the system completely fault tolerant, we need to use replication for data blocks also. We propose the following changes on top of the system designed in Q1 :

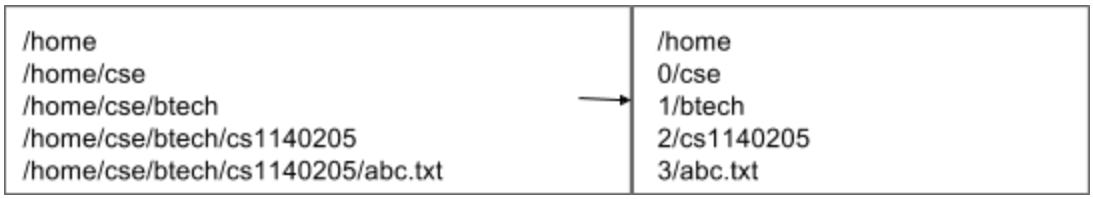
A) Namespace Management

We can reduce the space overhead of the two copies of metadata by using the following schemes :

1. Lookup tables :

    a. In place of the inode scheme, we can use the concept of lookup tables, to store pointers to blocks in the disk containing files' data.
    b. We can keep direct mappings from absolute file path names to their metadata, in the lookup table.
    c. Prefix Compression : To reduce the amount of space required for storing the lookup table, we can compress the pathnames to < 64 bytes of space . This is done by taking out the common prefixes of files, especially the ones in the same directory (*see Figure 2)*.

    Figure 2



2. Read - Write Lock :

    a. To handle concurrency issues over creating or updating files, we can keep a read - write lock over every entry in the lookup table.
    b. To update the entry corresponding to a particular file (or directory) in the table, we need to acquire read locks for all the parent (or ancestral) directories, and a read or write lock for the actual file (or directory) entry.
    c. The non-hierarchical structure of look up table also means that one does not need to update anything on the parent directory's metadata, and hence simplifies the task of creating/updating/deleting a file.

d. Hence, we don't need a write lock on the parent directory's entry, But we will need a read lock to avoid the directory getting deleted.
e. The locks can be ordered by level (in namespace tree) of the file(or directory) and then lexicographically on the same level, to avoid deadlock.

B) Replication Management

1. To avoid losing file data contained in any block on the disk, we need to maintain two copies of the data blocks.
2. The second copy of the blocks should reside at a separate disk (if available), or at a different location on the same disk, to avoid getting both copies lost together.
3. Whenever the user puts a read request for some file, the OS will first look for it in the original location, and if it is unusable, it can refer to the second copy, and also restore the data at the original location from the operation logs.

Pros

1. Proposed file system is able to recover from any unexpected state (unreadable or corrupt data blocks) if system enters into or failure thus making it as a complete fault tolerant .
2. Replication of data blocks ensures no loss of information in case some blocks becomes unreadable thus achieving high availability and reliability.
3. Prefix compression is reducing the space overhead of metadata stored in memory, which actually saves space as well as time (in updating the parent directory nodes, etc.).

Cons

1. This scheme is utilising more space as it is maintaining two replicas of data blocks on the same disk on different locations . But then it's a tradeoff between efficient space utilisation and data reliability.
2. Also, there is the overhead of writing and maintaining consistent replicas, which might consume extra time and the background process might slow down the next reads and writes.

Hence, after discussing the pros and cons, the above proposed file system seems like a good completely fault tolerant model.

Bibliography:

1. Google File System: http://sgotiweb.epn.edu.ec/~emafla/Cursos/CD/docs/gfs-sosp2003.pdf
2. Hadoop Distributed File System: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.pdf
3. File Systems: http://www.ufsexplorer.com/und_fs.php

   https://support.microsoft.com/en-in/help/100108/overview-of-fat--hpfs--and-ntfs-file-systems