Indian Institute Of Technology, Delhi

COL733: Cloud Computing Technology Fundamentals

Instructor: S. C. Gupta

Assignment 1: Design of Recoverable / Fault Tolerant File System

---

Report

August 08, 2019

Submitted To:
S. C. Gupta
Professor
Computer Science Department

Submitted By: (Group 4)
Shantanu Verma 2016CS10373
Pradyumna Meena 2016CS10375
Manav Rao 2016CS10523
Shubham 2016CS10371

Index

1. FAT file system

FAT file system is made by following 4 regions:

1.1. Reserved Sectors

The first reserved sector (Boot Sector) i.e. logical sector 0 is used to load the **OS**. However some versions of Windows have bootloaders bigger than a sector, hence end up taking more than one sector. This along with **BPB** (BIOS Parameter Block) make up what is known as **DBR** (DOS Boot Record). BPB contains information about the file system mainly its type and pointers to other sections. At the 6th logical sector we have the Backup Sector. However since bootloaders can take up more than a sector these distinctive boundaries vary with the size of bootloaders.

1.2. FAT Region

This region consists of two copies of **File Allocation Table** which are used for redundancy checking and specifies how the clusters are assigned. This can be considered as a map to traverse the data region. Contains mappings from cluster locations to cluster locations.

1.3. Root Directory Region

This region is a directory table which contains information about directories and files. It was used only in **FAT12** and **FAT16**. Its use was dropped in **FAT32** because this imposes a fixed maximum size constraint on the root directory which is pre-allocated at time of creation of the volume. **FAT32** stores the root directory in the data region so that it can be expanded if need be and therefore for **FAT32** there is no root directory region.

1.4. Data Region

This is the region where all of the directory data and existing files are stored. It uses up most of the partition.

2. Impact of Block Read Error

Bad block means that particular segment of memory is not reliable for storing or reading data. This can be of two types:

2.1. **Soft Bad block** - When **Cyclic Redundancy** (CRC) or **Error Correction Code** (ECC) do not match to the ones read by the disk. Occur when a write operation is stopped before completion possibly due to sudden shutdown.

2.2. **Hard bad block** - When there is a physical damage to the storage medium (damage to platter, depletion in constituents like in **NAND** flash, transistors in SSD)

Encountering damage to any block can make that memory region completely inaccessible by OS unless there is a copy of the pointer to its location. If any of the application files or booting related files are stored in a damaged sector this could result in application failure or OS level issues. Since the bad blocks cannot be used again, performance decreases as their number increases.

PROPOSED DESIGN - PART I

Limiting the Data Loss

The proposed file system focuses on the minimization of information loss in case of a read block error. To design a new system, we observe the UNIX file system and the corresponding issues it faces i.e Boot Sector Error, Data Sector Error, Inode Sector error. It uses inodes and whenever there is some issue with the access of the corresponding inode pointer, the data becomes inaccessible or gets corrupted. However this issue can be tackled by some slight modifications which are inspired from the distributed file systems over **HADOOP** and **GOOGLE** services.

- **Use of special log files for block's Metadata Storage:**

  We propose a design where we create a dynamic log file which contains the data of all the changes that a particular inode goes through during an operation. The OS maintains the log files and adds checkpoints randomly to check if there is some fault at any point of time. If the update in the log file doesn't match with the update reflected on the inode, the OS would recover data from the last clean checkpoint making a temporary copy. After the copy is made the OS uses the log file as reference and replicate the updates to the copy of the node.

  ### ❖ PROS:

  - ➢ The loss is directly proportional to the number of blocks that get corrupted as the action takes place once a corruption has been faced.
  - ➢ In case of a Boot sector error, the BIOS will observe the fault and create a copy of the recent most healthy checkpoint and try to boot from that first copy and update it with the new updates on the log file to see where it goes south.
  - ➢ In case of an Inode sector error, i.e some inode data becomes corrupt, the OS will first check the log file and figure out if there were any updates to the inode, if not then the data is either damaged, if not the OS creates a copy of the inode using itself as the reference. However, if there were updates in the inode data, it first have to apply all the changes in order to ensure a perfect replica.
  - ➢ In case of a Data Sector Error, since the data of a file is divided across various blocks, and since the fault has occurred that means the block has become corrupt and we're only saving the updates on the inode in the log file, hence the data is lost forever. Hence, the loss will be proportional to the number of blocks lost in this case.
  - ➢ In case of a System Crash however, the log file can be used as a recovery medium to restore the system to the last healthy checkpoint.

  ### ❖ CONS:

  - ➢ A big assumption here is that the log file is stored and maintained in such a way that it's data cannot be lost or get corrupted.
  - ➢ Extra memory space will be utilized for storing the copies in case of a faulty block read. In the worst case, half of the memory is utilized.
  - ➢ Cost operation is increased, i.e for one update the number of operations is doubled, i.e. the log file is updated along with the original copy.

PROPOSED DESIGN - PART 2

Completely Fault Tolerant

The above proposed design is fault tolerant to some extent but still, there are possibilities of data loss when inodes get corrupted or the file where the logs are stored is itself hampered. To tackle this issue, we provide a solution but it comes at the cost of space. We will require extra space to prevent complete data loss.

1. Create logs that would store the details of the operations done on inode.
2. The log files itself would be backed up, hence redundant use of space to eradicate the problem of data loss.
3. This log can be used to update the other copy of inode data concurrently in the background.

❖ PROS:

- ➢ In the proposed file system, the loss is always proportional to the number of blocks corrupted. For example, when the system tries to Boot, BIOS will read the original copy of boot sector, if there is any fault, it will try to read from the second copy.
- ➢ The next example we try to see is when there is an error in a block which contains inode data, we can simply read the data from the second copy of the inode data and simultaneously try to maintain the consistency between both the copies with the help of log files. This process is run by OS in background.
- ➢ The next case we look is whenever there is an error in data block, we cannot recover the data as we only have a unique copy. But we have the log file and the inode data intact hence, we can access all other blocks without any issue.

❖ CONS:

- ➢ The con in this model, as we mentioned above is that we have data redundancy. Since we have two copies of the same data, memory usage for the file system is doubled. Although this issue is not very significant as the data storage devices have become very cheap nowadays.

References

1. https://www.ufsexplorer.com/articles/file-systems-basics.php
2. https://support.microsoft.com/en-in/help/100108/overview-of-fat-hpfs-and-ntfs-file-systems