Design of Recoverable & Fault Tolerant File System

By Group No. 4

Members:

2017MCS2074 - Harish Chandra Thuwal
2017MCS2076 - Shadab Zafar
2017CSZ8058 - Kolluru Sai Keshav
2017MCS2102 - Chrystle Myrna Lobo

EXT

The extended file system (EXT) and its backward compatible successors namely, EXT2, EXT3 and EXT4, were designed for the linux kernel. Storage in the file system is divided into fixed number of contiguous blocks called block group (Figure 1).

BOOT BLOCK	BLOCK GROUP 0	BLOCK GROUP 1	BLOCK GROUP 2	...	BLOCK GROUP N

Figure 1: High level structure of a partition

A metadata block called *superblock* contains critical information about the file system such as block size, block per block group etc. Hence this block could be a single point of failure. To handle this issue, backup copies of the superblock are maintained at either some (Block 0, 1 and powers of 3, 5, 7) or all the block groups. During normal boot, the *primary superblock* (block 0 stored in block group 0) is used. In case of block read error, the *backup superblock* can be used.

For each block group a *block group descriptor* (BGD) is maintained, which contains pointer to data block bitmap, inode bitmap and start of inode table (Figure 2). The inode bitmap indicates which inodes are in use and which are not. Each inode table entry has information such as location of data block, permission etc, for a single file / directory. The inode table has inode entries of all files / directories in that block group. Data block bitmap indicates which data blocks are used and which are not.

The BGD for all the block groups are kept in a *block group descriptor table* (GDT). This information is critical and could be a single point of failure. Hence the GDT is duplicated in some or all block group. During normal operation, the GDT of Block group 0 is used. In case of block read error, the duplicated GDT copies can be used.

[SUPER BLOCK]	[GDT]	RESERVED GDT	DATA BLOCK BITMAP	INODE BITMAP	INODE TABLE	DATA BLOCKS

Figure 2: Physical structure of a block group

If there is a block read error in the inode table entry, then the information loss is not proportional to the number of corrupt blocks as losing an inode means that we lose any way of accessing all the data blocks of the file.

In case a block read error occurs at the data block, then only that block is unreadable. Here the loss is proportional to the data block size.

We observe that the EXT file system has no loss when superblock or GDT are corrupted. The file system is irrecoverable at the exceptional case when all the copies are corrupted simultaneously. There is no inherent self-heal mechanism.

A system where loss of information is minimised.

Design

We extend over EXT file systems, building upon the inode structure. Firstly, just like EXT we too keep redundant copies of the SuperBlock & Block Group Descriptors.

But since there are no copies of the inode structure itself, losing an inode will mean that we will not be able to access all blocks of the file/directory that this inode points to. Which shows that in EXT systems, the loss of information isn't proportional to the number of blocks lost.

To mitigate this, we propose the following approach:

1. Store the inode number of this file at the beginning of each data block of the file.

2. Store a flag at the last data block of the file to mark the end of the file.

3. Ensure that the data blocks to a file are allocated in a strictly increasing order of the block number and block group.

Now, in the event of a block read error:

1. If the block was a superblock - it would have multiple duplicate copies spread out on the disk. (A feature of EXT itself.)

2. If the block is a data block - the loss is already proportional.

3. If the block contained an inode - we'll traverse the data blocks of the block group to find all blocks that have have this inode number stored at the beginning.

   Since the data blocks to every file were allocated in increasing order, the entire file can be generated by simply appending these blocks in order.

Pros

1. First, the system is quite resilient against block read errors as the data loss is proportional to number of blocks lost.

2. This approach saves considerable space over the one which would duplicate the inode structures themselves.

Cons

1. First, this approach may result in fragmentation as we need to allocate blocks in increasing order of the block number. So defragmentation at specific intervals would be required.

2. Secondly, the recovery time at faults has been increased as we need to traverse the data blocks of the block group.

Completely Fault Tolerant Design

Design

Taking inspiration from HDFS, complete fault tolerance can be provided using redundant copies of the blocks on different sectors. The copies can be ensured to be on different platters for maximum reliability to address the issue of any region specific failures.

However, HDFS uses a fixed replication factor (by default r=3) for every block, which leads to a redundancy of 200%. The use of HDFS-RAID system reduces the redundancy to 133%, but at the cost of expensive hardware. In order to address this issue, we suggest to use a **Dynamic Replication Factor**, that decides the replication factor on a block-by-block basis.

The system automatically determines the replication factor for each block based on the expected health of the sector and disk, it is stored in. Blocks which reside in sectors that are predicted to be very healthy will be assigned a low replication factor while those that reside in unhealthy blocks will be given a high one.

Quantitatively, the health of a sector can be defined as the expected up-time of the sector before crashing. An estimate of it can be obtained using a **linear regression** model over a predefined set of parameters, whose values affect it directly.

We propose to use the following (non-exhaustive) set of parameters:

1. The physical state of the sector - such as number of reads and writes that occured, the up-time of the sector, etc

2. The age of the disk in which the sector is stored

3. The number of corrupted sectors in the neighbourhood (in that disk)

4. The manufacturing company of the disk

Based on historical data, a **linear regression** model can be learnt over the given parameters. Using the predicted health, system sets the replication factor to be between **min_r** (say, 2) and **max_r** (say, 3), which are two user-defined parameters.

Pros

- Reduces the redundancy factor from (max_r-1) * 100% to a number between (min_r-1)*100% and (max_r-1)*100%

- No extra hardware cost is required as in case of HDFS-Raid system, and can be easily implemented by updating the logic of the NameNode.

Cons

- The redundancy factor is not fixed and is dependent on the accuracy of the linear regression model.

- Small increase in latency as the replication factor must be computed for every block

- Related to the above, this method requires a linear regression implementation that works in real time

References

https://en.wikipedia.org/wiki/Ext2
http://www.nongnu.org/ext2-doc/ext2.html
http://www.science.unitn.it/~fiorella/guidelinux/tlk/node95.html#SECTION00111000000000000000
HDFS Slides from class