**Digital Forensics Project**

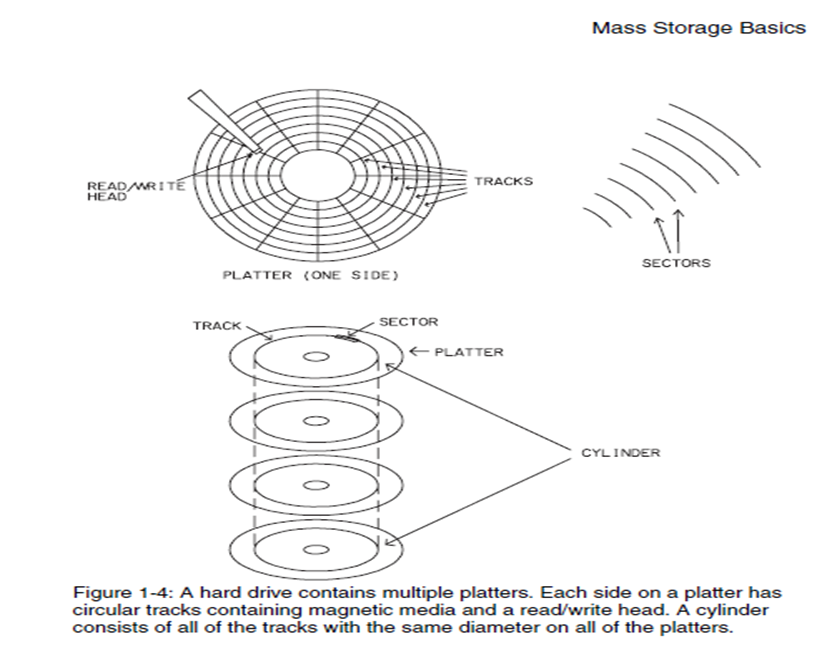
We look at two file systems – NTFS (windows) and Ext (linux). For our purpose, ext2, ext3, ext4 are same (for the time being).

The first main section is to define the objective of the project.

The second section describes the structure of the file systems

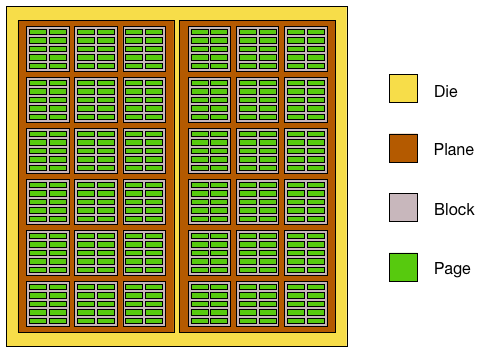
Nilesh-Start

# **Comparison between disc based and flash based storage**



As shown in above diagram, traditional disk based hard drive contains multiple platters. Each side of platter has circular track containing magnetic media and a read/write head. A cylinder consist of all tracks with same diameter on all platters. Sector is a part of a track.

# **Flash based storage Layout**



1. Each memory chip contains one or more dies.
2. Each die contains one or more planes. (Usually one or two).
3. Each plane contains number of blocks which are smallest unites that can be erased.
4. Each block contains number of pages which are smallest units which can be written.

Nilesh-End

# **A. Objective**

Files system can be corrupted accidentally or by human action. We want to recover the file system as much as possible. This means mainly two things:

1. The meta data should be recovered

2. The individual files AND the directories are recovered (the “lost” files and directories may be due to drive crashes or the action of deleting the file/directories).

3. Analyze what percentage of the disk has been recovered.

We will develop tools to achieve these objectives and use heuristics and other techniques to recover files.

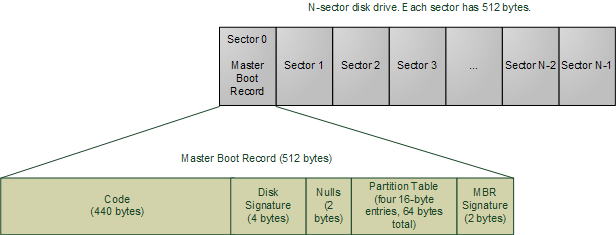
# **B. Boot sector (Vikram)**

### 1. Description

The first block of all hard drives, no matter what type of file system (NTFS or EXT etc), contains the boot sector. The first 1024 bytes of the block are reserved for Master Boot Record which is used to check the organization of partitions. It also contains the bootstrap code to load the installed operating system.

A standard MBR uses the following addresses for storing the information:

1. Bootstrap code: 440bytes of code starting at address 0 (00h)
2. Partition table: 4 partition entries, each 16 byte, the first starting at address 446 (1BEh)
3. Boot signature: ‘55AA’. This is stored in location 510-511 (1FEh – 1FFh).



The block addresses and sizes are stored in the partition table using 32 bit keys.

Each partition table entry has 16 bytes and has the following layout.

1. Status of the drive: 1 byte information either inactive (00h) or active (80h). If the 7th bit is set in this byte, then this drive is considered to be bootable and the boot information is loaded from this partition.
2. CHS address of the first sector: Stores the address of the first
   1. 1st byte stores the head address.
   2. 3rd byte stores the lower 7 bits of the cylinder address.
   3. 2nd byte stores the sector address as well as the 2 high bits of cylinder. (6th and 7th bit)
3. Partition Type: 1 byte to denote the type of partition
4. CHS address of the last sector: Stores in the same format as Starting CHS address
5. Logical block address of first sector: 4 bytes to denote this address.
6. Number of sectors in partition: 4 bytes to denote the total number of sectors



### 2. Corruption and Recovery

MBR can become corrupted when the disk is damaged either physically or by a virus or malware. Corruption of MBR can cause serious data losses. Verifying the boot signature which should have the constant value of 0x55AA and validating the entries in the partition table entry one can identify a corruption.

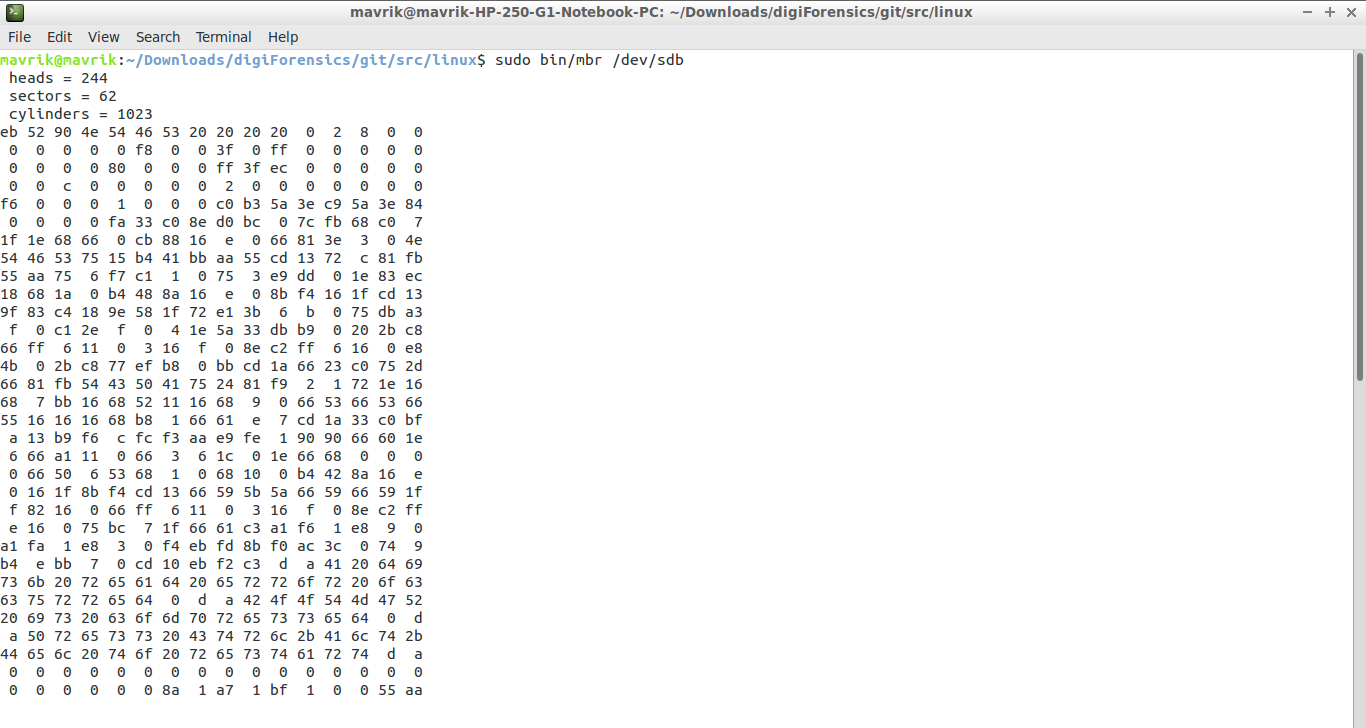
Corruption of ‘status of the drive’ field does not pose much problem since it does not concern location of the data. Even if this byte is corrupted, we just need to check whether the partition contain any boot information present in it, and set it to 0x00 or 0x80 accordingly.

In case the CHS values are corrupted, either starting or ending, then assuming that the total number of sectors shown in the partition entry are correct, these values can be recovered correctly.

In case of error in partition type byte, we need to confirm the type of system from the partition values itself.

**3. Running the program**

The compiled MBR program can retrieve the information of the diskname provided.





(Vikram - end)

# **C. EXT FILE SYSTEM**

Ext allocates storage space in units of “blocks”. A block is a group of sectors between 1KiB and 64KiB, and the number of sectors must be an integral power of 2. Blocks are in turn grouped into larger units called block groups.

Each block group mainly consists of the following blocks:

· Super block

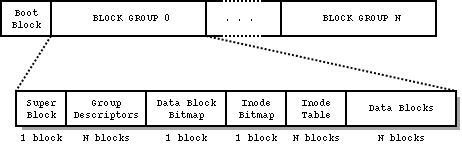
· Group Descriptors

· Data Block bitmap

· Inode bitmap

· Inode table

· Data blocks

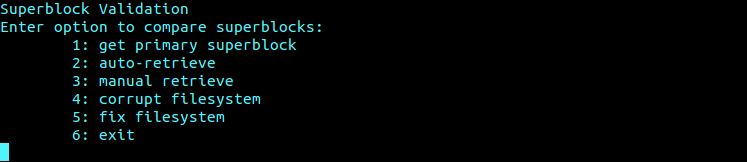


## 1. Super Block (by Jason Daza)

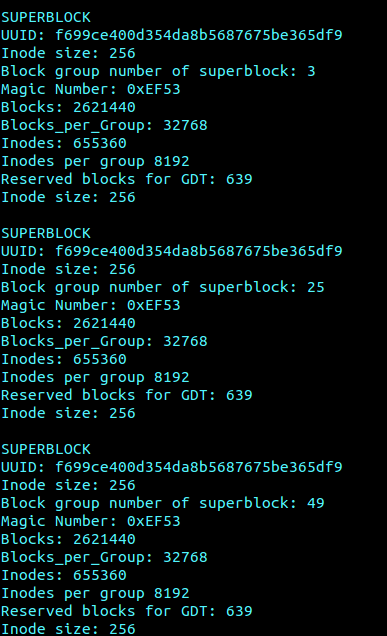
The superblock is a data structure which contains information regarding the configuration of the filesystem. This information, or metadata, can be categorized in three ways: 1) filesystem parameters that are fixed with respect to a particular installation (e.g. total number of blocks/inodes), 2) filesystem parameters that are unfixed (e.g. ID of user/group able to use reserved blocks), and 3) current state of the filesystem (e.g. number of free blocks/inodes). Superblocks are stored at the start of the block group and the primary copy of the superblock is stored at an offset of 1024 bytes from the start of the filesystem (block group 0). Superblocks are essential to correctly mounting the filesystem and typically it is the primary copy of the superblock (located in block group 0) that is read when a filesystem is mounted. The importance of superblocks demands that redundant copies be made available. Although copies can be stored in every block group, by default they are only stored in block groups 0 and 1, and block groups whose numbers are powers of 3, 5, and 7. There are three definitions of the superblock (one for each file system ext2, ext3, and ext4) which allow for variations in compatibility.

**Corruption of the Superblock**

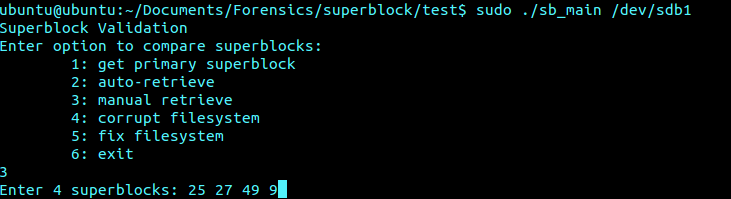
Corruption of superblocks is only a concern if it affects all copies; if any redundant copy remains uncorrupted, the filesystem can still be mounted correctly and all corrupt superblocks can be repaired. Suppose, though, that all copies were corrupted in some way. In that case some types of corruption are worse than others. For example, if the values representing the number of free blocks/inodes are corrupt, that may have no effect on accessing existing files. However, if the magic number is corrupt then the kernel will not recognize the superblock at all.



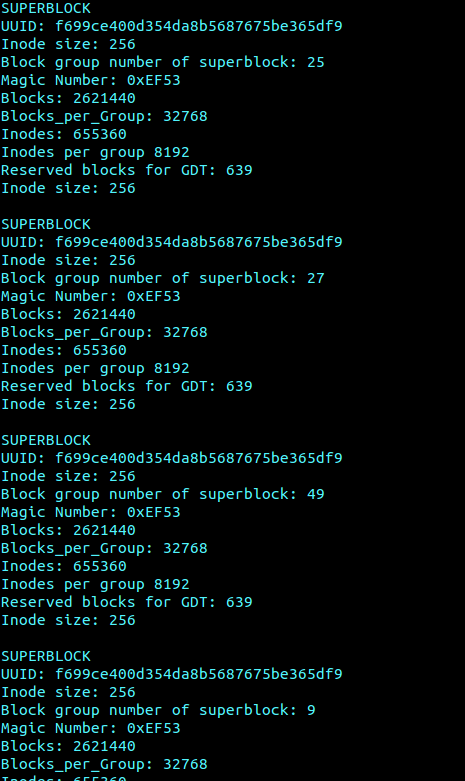
*Menu of superblock program*



*Example output of “auto-retrieve”.*



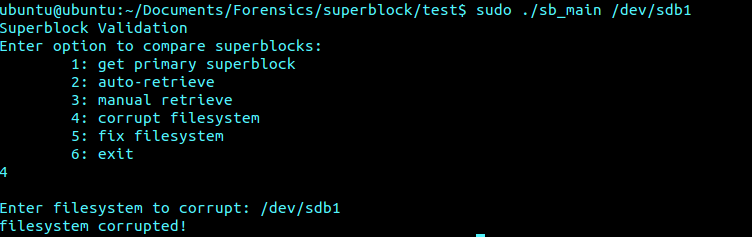
*“Manual retrieve allows user to select superblocks to view…”*



*Output of manual retrieve*

**Identifying and Verifying Corruption**

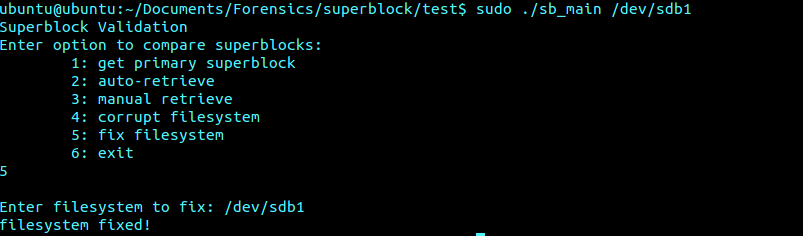
This program attempts to identify, verify, and correct corrupted superblocks through examination of an array of five random or user selected superblocks. Of the fields of the superblock struct which are not unique to each superblock (e.g. block group number of the superblock) there should be no variation between the superblock array elements. That is, information related to the state of the filesystem should unanimous. Before we can compare superblocks, however, we must first ensure the correct bytes are copied into our superblock structs. This is a simple task since the layout of the filesystem is dependent upon block size. For most systems there are only three available block sizes (1024 KB, 2048 KB, 4096 KB). This program will assume block size is unknown and allow the user to choose between the three (it should be noted that 4096 KB is the default block size and the program will inform the user as such). After a block size is entered the program will attempt to retrieve the superblocks. Once all five superblocks have been retrieved the program will use data from the superblock fields to begin looking for corrupted superblocks. Initially, the program will compare three fields; magic number, block size, and blocks per group. The decision to compare three fields was mostly arbitrary. While the author believes the number to be sufficient, if increasing the number of fields to compare is desired, doing so can be easily implemented. After our superblock array is created the program can then begin looking for signs of corruption. First, we know the magic number for an ext filesystem is 0xEF53. Therefore, if the magic number cannot be found at byte 56 of the superblock we know that the magic number is corrupted (we do not need to verify a corrupt magic number since it cannot vary). The second method of identification compares the values of s\_log\_block\_size and s\_blocks\_per\_group. [S]\_log\_block\_size is a four byte value located at byte 24 of the superblock and represents the number to shift 1024 left (i.e. 0 equals 1024, 1 equals 2048, 2 equals 4096). [S]\_blocks\_per\_group is the four byte value located at byte 32 of the superblock. Since s\_blocks\_per\_group is calculated as s\_log\_block\_size \* 8, any superblock whose values do not match this formula will be considered corrupted and flagged for verification. The program will utilize an array of ints to keep track of corrupted superblocks, where '0' represents a corrupted superblock and '1' represents a correct superblock. Below we can see the output of the program randomly selecting superblocks for comparison.Notice the array “sb\_array” display five “1’s”, this indicates that the retrieved superblocks are all correct.



*Corrupting the superbock*

### Repairing Superblocks

### To test the ability to repair the superblocks we give the user the option to manually corrupt a superblock. When the program runs again it will detect the corrupting superblock and automatically repair it. For testing purposes we only corrupted the magic number of a particular superblock. The bytes “EF53” were then written to the address of the corrupted magic number in order to repair it. Although the magic number is the only field of the superblock that’s constant, this example demonstrates the concept behind repairing a superblock. Since we are able to calculate the byte offset of any field of the superblock we can simply overwrite the corrupted bytes with bytes that have been determined to be correct.



*Repairing the superblock*

### Worst Case

### There is a possibility (though highly unlikely) that all superblocks are corrupted, and corrupted in different ways. However, it may still be possible to sufficiently rebuild a superblock such that data can be recovered. As mentioned earlier, the likelihood of data being corrupted such that no sense can be made from any superblock is extremely low. Thus, despite how corrupt a filesystem may be it is still very likely that recognizable superblock data can eventually be acquired (even through trial-and-error). The user knows that the primary superblock is located at offset 1024. Suppose by printing out the fields of a superblock a user can ascertain that at least some of the information is correct (an incorrect block size would doubtfully produce sensible results and almost certainly not produce such results consistently enough so as to appear legitimate). The user now knows the block size and with this information alone can locate the redundant superblocks. Once the redundant superblocks have been found the user can begin to reason about the correct field values (such as total/free inodes) which can aid in data recovery.

### Value of superblock to data recovery

### The superblock contains vital information regarding the metadata of the filesystem. As stated the previous paragraph, if we only knew the filesystem was EXT data recovery is still very possible. Values such as magic number and block size are either constant or trivial to determine, and through trial and error we could easily determine the structure of the block group (e.g. size of group descriptor table, block/inode bitmap, etc.). Thus, ensuring the data from the superblock is correct is critical for data recovery.

**Author :Anoop S S**

## 2. Group Descriptors

Each block group on the filesystem has one of the block group descriptors associated with it. It is defined as ext3\_group\_desc structure, which is of 32 bytes. Along with its own group descriptor, each block group also stores all other block group’s group descriptors in a sequential manner hence it is called as Block Group Descriptor Table. That means if we are looking at block group 3, then we can find its group descriptor at position 3 in descriptor table. Block group descriptor provides the location of the inode bitmap, inode table, block bitmap, number of free blocks and inodes, and some other useful information. Depending on the total number of block groups, this table can require multiple blocks of storage. Suppose there are 200 block groups in a drive. Then total size of block group descriptor table is 32 \* 200 = 6400 bytes. If the block size is 4K then block group descriptor table requires 2 blocks and for the block size of 2K it requires 4 blocks and so on.The block group descriptor table will be replicated in many block groups but not all the block groups. The block group descriptor table will always be present in the first block group i.e. block group number 0 and second block group i.e. block group number 1**(Note block group number are indexed from 0).** The replicated copies can be found in the block group number which is a multiple of 3, 5 or 7. Suppose if the drive with ext3 file system has 50 block groups then the block group descriptor table can be found in the block group numbers 0, 1, 3, 5, 7, 9, 25, 27 and 49. File System always uses the first block group descriptor table for its computation i.e. the block group descriptor at block group 0.

The definition of ext3\_group\_desc can be found in ext3\_fs.h:

struct ext3\_group\_desc

{

\_\_le32 bg\_block\_bitmap; /\* Blocks bitmap block \*/

\_\_le32 bg\_inode\_bitmap; /\* Inodes bitmap block \*/

\_\_le32 bg\_inode\_table; /\* Inodes table block \*/

\_\_le16 bg\_free\_blocks\_count; /\* Free blocks count \*/

\_\_le16 bg\_free\_inodes\_count; /\* Free inodes count \*/

\_\_le16 bg\_used\_dirs\_count; /\* Directories count \*/

\_\_le16 bg\_pad;

\_\_le32 bg\_reserved [3];

};

The bg\_free\_blocks\_count, bg\_free\_inodes\_count, and bg\_used\_dirs\_count fields are used by the block allocation algorithm when allocating new inodes and data blocks. These fields determine the most suitable block in which to allocate each data structure.

The descriptions of the ext2\_group\_desc structure attributes are as below:

1) **bg\_block\_bitmap** ->32 bit block id of the first block of the "block bitmap" for the group represented. The actual block bitmap is located within its own allocated blocks starting at the block ID specified by this value.

2) **bg\_inode\_bitmap** ->32 bit block id of the first block of the "inode bitmap" for the group represented.

3) **bg\_inode\_table** ->32 bit block id of the first block of the "inode table" for the group represented.

4) **bg\_free\_blocks\_count** ->16 bit value indicating the total number of free blocks for the represented group.

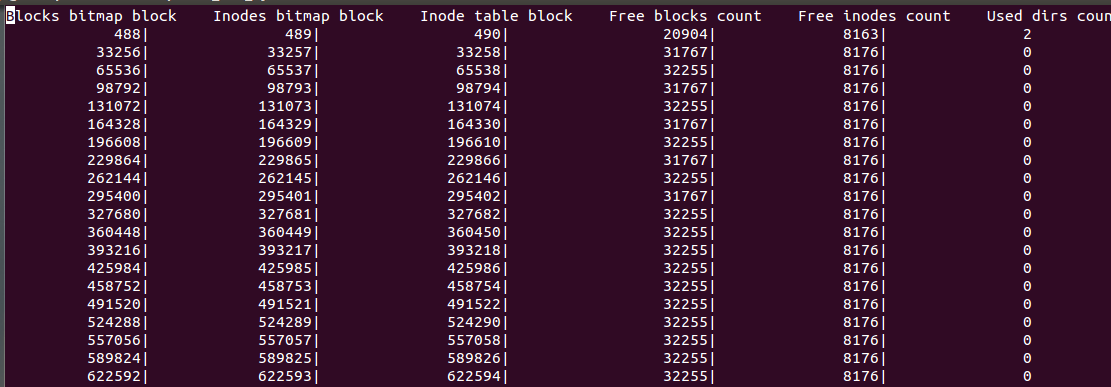
5) **bg\_free\_inodes\_count** ->16 bit value indicating the total number of free inodes for the represented group.

6) **bg\_used\_dirs\_count** ->16 bit value indicating the number of inodes allocated to directories for the represented group.

7) **bg\_pad** ->16 bit value used for padding the structure on a 32 bit boundary.

8) **bg\_reserved** -> 12 bytes of reserved space for future revisions.

Snapshot of Group descriptor table for the above first six fields:



The Block Group Descriptor (BGD) start offset can vary depending on the block size (1k, 2k, or 4k). In a partition, the first 1024 bytes are reserved, then followed with 1024 bytes of SUPER BLOCK. The block group descriptor table starts on the first block following the superblock. This would be the third block on a 1KB block file system, or the second block for 2KB and larger block file systems.

Depending on the block size, the BGD starts from:

Block Size =1K, the BGD starts at partition offset 2048 which is third block (1024 reserved + 1024 super block).

Block Size =2K, the BGD starts at partition offset 2048 which is second block(1024 reserved + 1024 super block).

Block Size =4K, the BGD starts at partition offset 4096, which is 1 block from start, that is the result you see 3072 bytes apart from the super block.

**Corruption of the Block Group Descriptor Table**

The block group descriptor table can get corrupted for many reasons. One of the reasons could be data overwritten at the location of block group descriptor table or the data at the block group descriptor table can be reset to NULL by the unintended memset NULL operation. Assuming all the replicas are consistent with each other initially and if the first replica gets corrupted then a verification algorithm is required to identify the valid block group descriptor table across the replicas. Ideally, all replicas should be identical. In the worst case, assuming some of the replicas other than the first replica might also get corrupted. Hence a comparison algorithm is needed to identify a valid block group descriptor table among all the replicas.For data recovery during corruption, block group descriptor contains crucial information like starting address of the block bitmap, starting address of the inode table etc. Once block group descriptor is recovered, it’s information can be used to find various kind of data blocks like text blocks, address blocks, directory blocks, null blocks. This categorization will help us during data recovery.

### Identifying and Verifying Corruption

The first six fields of the block group descriptor table i.e. bg\_block\_bitmap, bg\_inode\_bitmap, bg\_inode\_table, bg\_free\_blocks\_count, bg\_free\_inodes\_count and bg\_used\_dirs\_count must be same across all the replicas. The other two fields namely bg\_pad and bg\_reserved are not significant metadata for the block groups and hence can be ignored while comparing the consistency of all the replicas. Given any two block group numbers, the above mentioned six fields can be compared for the block group descriptor table of both the block group numbers to check if they are identical. A generic algorithm would be to compare all other replicas against each other and if at least 3 replicas are identical then it can be concluded that those three replicas are the valid ones. Then this valid block group table data is overwritten at all other replicas including the first replica.

This process will help us to recover the corrupted block group descriptor table and each replica is consistent with all other replicas which will help us recover if block group descriptor table gets corrupted again later.

**Finding block size from the super block**

The super block starts at the offset 1024 i.e. it starts from the 1025th byte. 24th byte of the superblock contains information about the block size i.e. at the 1048 ( 1024 + 24) byte offset. At this offset, read four bytes of data into an 32 bit integer variable. The value of this variable is logarithmic i.e. left shifting 1024 by this value gives the block size of the partition. For e.g. if the value read is 0 then block size is 1K, if the value read is 1 then the block size is 2K, if the value read is 2 then the block size is 4K and so on.

**Finding Group Descriptor Table Byte Offset of block groups**

Each block group contains as many as 8 \* block size (this is based on block bitmap) blocks. If the block group contains block group descriptor then the first block of that block group is super block followed by block group descriptor table i.e. block group descriptor is the second block from the beginning of the block group . Hence the block offset of any block group is given by:

Block\_offset = (block\_group\_number \* block\_size \* 8) + 1

The byte offset is calculated as below:

Byte\_offset = Block\_offset \* block\_size.

**Note:** If the block group doesn’t contain block group descriptor table i.e. block group number is not a multiple of 3, 5 or 7 then the first block of the given block group is data block.

**Finding total number of block groups in a partition**

The total number of block groups can be calculated as below:

Block\_groups\_count = total number of blocks in a partition/ number of blocks in a block group.

The total number of blocks and number of blocks in a group can be found from the super block. Super block can be located at byte offset 1024. From the beginning of super block, 5th byte and 32nd byte contains total number of blocks and number of blocks in a group respectively. Since both the fields are 32 bit integer, reading 4 bytes from 5th byte and 32nd byte is sufficient to find the total number of block groups in a partition.

There is an another way of finding the total number of block groups in a partition which is given by:

Block\_groups\_count = total number of inodes in a partition / number of inodes in a block group.

First 4 bytes from the beginning of the super block contains total number of inodes in a partition. Reading 4 bytes at the 40th byte from the beginning of the super block, gives the number of inodes in a block group. Using these values, we can calculate the total number of block groups in a given partition.

**Block Allocation Policy**

The ext3 block allocator has been replaced by a reservation-oriented version. The first time a block is needed for a file, the file system creates a "reservation window" which sets aside a range of blocks (**eight of them, initially**); the actual block allocations are then taken from the window. When the window is exhausted, a new, possibly expanded window is allocated, as near as possible to the old window, to replace it. Reservations only last until the process writing the file closes it; thereafter, the blocks become free once again.

Interestingly, nothing in the filesystem itself tracks block reservations; **they are all handled by a single, in-core linked list (per filesystem)**. A block reservation will not actually prevent blocks inside the window from being allocated to some other file. Since the filesystem allocates out of reservation windows whenever possible, however, and those windows do not overlap, the reservations are almost always honored. In some situations (such as when all remaining free blocks are reserved) the filesystem will forget about reservations and allocate blocks from anywhere.

Since disk latency is the key factor that affects the filesystem performance, modern filesystems always attempt to layout files on a filesystem contiguously. This is to reduce disk head movement as much as possible. However, if the file system allocates blocks on demand, then when two files located in the same directory are being written simultaneously, the block allocations for the two files may end up getting interleaved. To address this problem, some filesystems use the technique of pre allocation, by anticipating which files will likely need allocate blocks and allocating them in advance.

The core idea of the reservation based allocator is that for every inode that needs blocks, the allocator reserves a range of blocks for that inode, called a reservation window. Blocks for that inode are allocated from that range, instead of from the whole filesystem, and no other inode is allowed to allocate blocks in the reservation window. This reduces the amount of fragmentation when multiple files are written in the same directory simultaneously. The key difference between reservation and pre allocation is that the blocks are **only reserved in memory, rather than on disk**. Thus, in the case the system crashes while there are reserved blocks, there is no inconsistency in the block group bitmaps.

The first time an inode needs a new block, a block allocation structure, which describes the reservation window information and other block allocation related information, is allocated and linked to the inode. The block allocator searches for a region of blocks that fulfills three criteria.

1) First, the region must be near the ideal “goal” block, based on ext2/3's existing block placement algorithms.

2) Secondly, the region must not overlap with any other inode's reservation windows.

3) Finally, the region must have at least one free block. As an inode keeps growing, free blocks inside its reservation window will eventually be exhausted. At that point, a new window will be created for that inode, preferably right after the old with the guide of the “goal” block.

**The goal block is the next logical block after the current last block of the file**. If this goal block is free, it is allocated. If not, Ext2 attempts to find a free block within 32 blocks of the target. If it is unable to do so, it will attempt to find a block in the same block group. Finally, if this also fails, it will scan forward from the current block group, through each of the block groups until it finds an available block. When a block is found, Ext2 will attempt to pre-allocate up to 8 data blocks.

During block allocation, aim is to allocate a physical block = goal block. Hence, every allocation request has a **goal block**.

If the current block and previously allocated block have consecutive file block number then goal is logical block number of previous block + 1.

Else if at least one pre allocated block earlier, goal = that block.

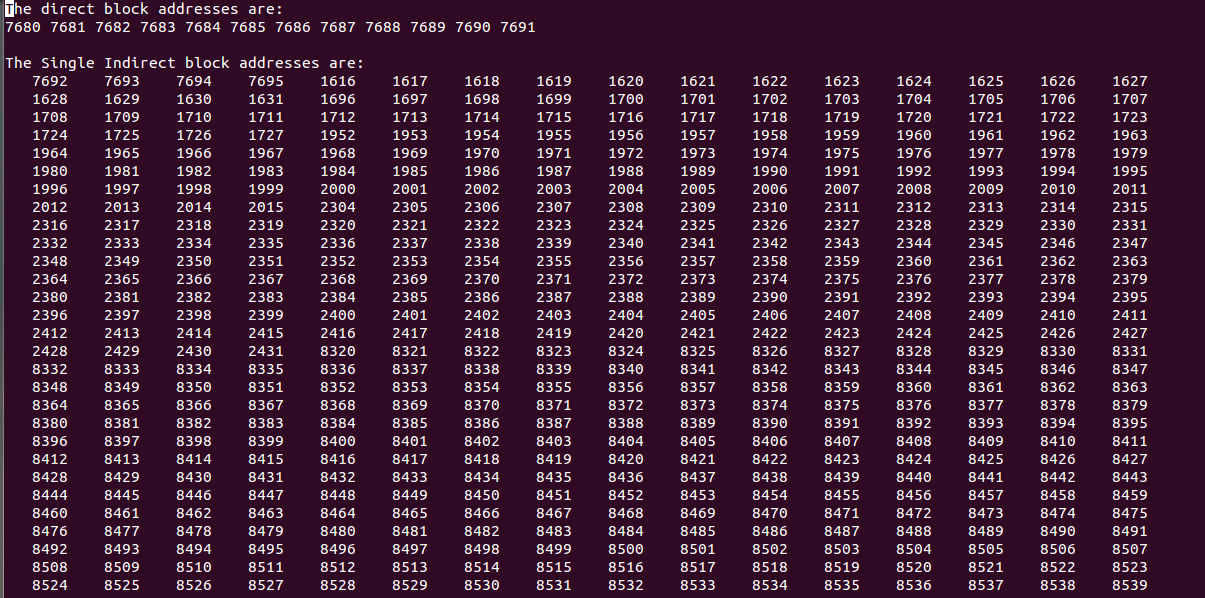
Else goal = first block in the block group with the file’s inode.

If the goal block is not free, try the next one. If not available, search all the block groups starting from the one containing the goal block which looks for a group of 8 adjacent free blocks. If not, look one.

All of the reservation windows are indexed via a per-filesystem red-black tree so the block allocator can quickly determine whether a particular block or region is already reserved by a particular inode. All operations on that tree are protected by a per-filesystem global spin lock.

Initially, the default reservation window size for an inode is set to **eight blocks**. If the reservation allocator detects the inode's block allocation pattern to be sequential, it dynamically increases the window size for that inode. An application that knows the file size ahead of the file creation can employ an ioctl command to set the window size to be equal to the anticipated file size in order to attempt to reserve the blocks immediately. The default block allocator for ext3 has been replaced by this reservation based block allocator. During the testing, it was observed that the number of blocks allocated follows a specific pattern. For e.g. the allocator first allocates 8 blocks for a file. If that file needs more blocks then another 8 blocks are allocated. If it needs some more blocks then allocator thinks the file size is growing gradually and hence tries to allocate the blocks which is equal to twice the previous allocation i.e. in our example 2 \* 8 = 16 blocks. Similarly for the next allocation, it will allocate 2 \* 16 = 32 blocks. The allocator also tries find contiguous blocks i.e. for the previous allocation it tries to find 32 contiguous blocks.

Snapshot of the block addresses allocated for some random file:



**Analysis of Block Allocation policy:**

**Experiment conducted :**

**Device Used:** USB Flash Drive (2GB)

**SuperBlock Contents:**

Filesystem volume name: <none>  
Last mounted on: <not available>  
Filesystem UUID: af1958cd-a4f8-4deb-a049-84462be2a746  
Filesystem magic number: 0xEF53  
Filesystem state: clean  
Errors behavior: Continue  
Inode count: 122400  
Block count: 489200  
Reserved block count: 24460  
Free blocks: 472587  
Free inodes: 122389  
First block: 0  
Block size: 4096  
Fragment size: 4096  
Reserved GDT blocks: 119  
Blocks per group: 32768  
Fragments per group: 32768  
Inodes per group: 8160  
Inode blocks per group: 510  
Filesystem created: Wed Mar 1 23:22:04 2017  
Last mount time: n/a  
Last write time: Wed Mar 1 23:22:04 2017  
Mount count: 0  
Maximum mount count: -1  
Last checked: Wed Mar 1 23:22:04 2017  
Check interval: 0 (<none>)  
Reserved blocks uid: 0 (user root)  
Reserved blocks gid: 0 (group root)  
First inode: 11  
Inode size: 256  
Journal inode: 8  
Default directory hash: half\_md4  
Directory Hash Seed: 08ecc44f-1869-4c0b-81fa-0c7b21f1fdb2  
Journal backup: inode blocks  
Directories: 2

**Initial Statistics of Block Groups:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123  
32129 free blocks, 8149 free inodes, 2 used directories  
Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610  
24055 free blocks, 8160 free inodes, 0 used directories  
Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754  
29936 free blocks, 8160 free inodes, 0 used directories

**Note:** Each Block group has 32000 blocks (close to 130 MB).

1. we created a new directory named dir1 in the USB.
2. Next, a 10 MB text file has been created under directory dir1. (**Note: This must take approximately 2500 blocks)**

**Directory Structure (along with inodes):**

.  
├── [ 8161] dir1  
│   └── [ 8162] user1.txt  
└── [ 11] lost+found [error opening dir]

**Block Group Statistics after file creation:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123  
29723 free blocks, 8149 free inodes, 2 used directories  
Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891  
32114 free blocks, 8158 free inodes, 1 used directory  
Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610  
24055 free blocks, 8160 free inodes, 0 used directories  
Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754  
29936 free blocks, 8160 free inodes, 0 used directories

**Note: We can see that Blocks were allocated from both block group 0 and block group 1.**

1. We then analyzed the inode of the txt file (8162) for the data blocks allocated to it.

**Inode Contents:**

Inode Analyser# inode 8162  
i\_mode = 33188  
i\_uid = 0  
i\_size = 9919779  
i\_atime = 1488435081  
i\_ctime = 1488435081  
i\_mtime = 1488435081  
i\_dtime = 0  
i\_gid = 0  
i\_links\_count = 1  
i\_blocks = 19408  
i\_block[0] = 33792  
i\_block[1] = 33793  
i\_block[2] = 33794  
i\_block[3] = 33795  
i\_block[4] = 33796  
i\_block[5] = 33797  
i\_block[6] = 33798  
i\_block[7] = 33799  
i\_block[8] = 33800  
i\_block[9] = 33801  
i\_block[10] = 33802  
i\_block[11] = 33803  
i\_block[12] = 33402  
i\_block[13] = 33403  
i\_block[14] = 0  
i\_flags = 0  
i\_generation = 1851121527  
i\_file\_acl = 0  
i\_dir\_acl = 0  
i\_faddr = 0  
i\_extra\_isize = 32  
i\_pad1 = 0

**Inferences:**

* Indirect pointers are not contiguous with the direct data pointers
* Indirect pointers are contiguous among themselves.

1. We can infer that all the direct block pointers, first and second indirect block pointers were from Block group 1.
2. We then analyzed the block addresses with in first indirect block pointer (33402). It contained the following addresses.

33804 33805 33806 33807  
656 ……………1407 1536  
…………………… 2059

**Note:** The first 4 block addresses were from Block group 1 and the remaining were from Block group 0. Evem though they are mostly contiguous, there were a lot of breaks in between. For example : After 1407 the next address was 1536.

Also the first 4 blocks are contiguous with the direct data block pointers.

1. We then analyzed the block addresses with in second indirect block pointer (33403). It contained the following two addresses, which are single indirect block pointers.

33408 33409

The first indirect block pointer 33408 contained contiguous block addresses from Group 0 as follows:

2060……...3071

4096……...4107

**Note: The first block address 2060 is contiguous with the last block address 2059 of the previous indirect pointer.**

The second indirect block pointer 33409 also contained contiguous block addresses from Group 0 as follows:

4108……….4469

We then created another 10 MB file in the same directory, the above properties did hold for this file also.

**Directory Structure After file creation:**

.  
├── [ 8161] dir1  
│   ├── [ 8162] user1.txt  
│   └── [ 8163] user2.txt  
└── [ 11] lost+found [error opening dir]

**Block group statistics:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123  
27317 free blocks, 8149 free inodes, 2 used directories  
Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891  
32094 free blocks, 8157 free inodes, 1 used directory  
Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610  
24055 free blocks, 8160 free inodes, 0 used directories  
Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754  
29936 free blocks, 8160 free inodes, 0 used directories

**Inode of the file (8163):**

Inode Analyser# inode 8163  
i\_mode = 33188  
i\_uid = 0  
i\_size = 9919779  
i\_atime = 1488439113  
i\_ctime = 1488439113  
i\_mtime = 1488439113  
i\_dtime = 0  
i\_gid = 0  
i\_links\_count = 1  
i\_blocks = 19408  
i\_block[0] = 33808  
i\_block[1] = 33809  
i\_block[2] = 33810  
i\_block[3] = 33811  
i\_block[4] = 33812  
i\_block[5] = 33813  
i\_block[6] = 33814  
i\_block[7] = 33815 i\_block[8] = 33816  
i\_block[9] = 33817  
i\_block[10] = 33818  
i\_block[11] = 33819  
i\_block[12] = 33406  
i\_block[13] = 33407  
i\_block[14] = 0  
i\_flags = 0  
i\_generation = 1851121528  
i\_file\_acl = 0  
i\_dir\_acl = 0  
i\_faddr = 0  
i\_extra\_isize = 32  
i\_pad1 = 0

**First Indirect Block (33406) Contents:**

33820 33821 33822 33823  
720 ………………..……….  
…………………….… 5131

**Second Indirect Block (33407) Contents:**

33408 33409

**Contents of 33408:**

5132………………...6155

**Contents of 33409:**

6156………………...6515

Thus we can easily infer that the above mentioned properties holds for all the files.

**Extended Analysis:**

We created around 18 files each of 10 MB and analysed the allocation of block groups.

**Tree structure:**

.  
├── [ 8161] dir1  
│   ├── [ 8171] user10.txt  
│   ├── [ 8172] user11.txt  
│   ├── [ 8173] user12.txt  
│   ├── [ 8174] user13.txt  
│   ├── [ 8175] user14.txt  
│   ├── [ 8176] user15.txt  
│   ├── [ 8177] user16.txt  
│   ├── [ 8178] user17.txt  
│   ├── [ 8179] user18.txt  
│   ├── [ 8180] user19.txt  
│   ├── [ 8162] user1.txt  
│   ├── [ 8181] user20.txt  
│   ├── [ 8163] user2.txt  
│   ├── [ 8164] user3.txt  
│   ├── [ 8165] user4.txt  
│   ├── [ 8166] user5.txt  
│   ├── [ 8167] user6.txt  
│   ├── [ 8168] user7.txt  
│   ├── [ 8169] user8.txt  
│   └── [ 8170] user9.txt  
└── [ 11] lost+found [error opening dir]  
  
2 directories, 20 files

**Block Group Statistics:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123  
7829 free blocks, 8149 free inodes, 2 used directories  
Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891  
9780 free blocks, 8124 free inodes, 1 used directory  
Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538  
10602 free blocks, 8160 free inodes, 0 used directories  
Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427  
15533 free blocks, 8160 free inodes, 0 used directories  
Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610  
24055 free blocks, 8160 free inodes, 0 used directories  
Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035  
32135 free blocks, 8160 free inodes, 0 used directories  
Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986  
32256 free blocks, 8160 free inodes, 0 used directories  
Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754  
29936 free blocks, 8160 free inodes, 0 used directories

**Thus we can see that blocks are always (we assume at this point of time) allocated from pairs of block groups.**

**ANALYSIS – 2:**

**Device Used** : 2GB USB Stick

Formatted the entire disk.

**File System Used** : ext3

**Experiment 1:**

Create a directory and check the location of the directory.

**Block Group Statistics :**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

32129 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

32135 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

32256 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

32135 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

32256 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

32135 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

24055 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

32135 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

32256 free blocks, 8160 free inodes, 0 used directories

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

32135 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

32256 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

32256 free blocks, 8160 free inodes, 0 used directories

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

32256 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32256 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

Inode of the dir:

89761 dir1 11 lost+found

Inode <89781>

Inode: 89761 Type: directory Mode: 0755 Flags: 0x0

Generation: 2597155628 Version: 0x00000000:00000001

User: 0 Group: 0 Size: 4096

File ACL: 0 Directory ACL: 0

Links: 2 Blockcount: 8

Fragment: Address: 0 Number: 0 Size: 0

ctime: 0x58d9aef8:7cf1b0dc -- Mon Mar 27 19:31:52 2017

atime: 0x58d9aef8:7cf1b0dc -- Mon Mar 27 19:31:52 2017

mtime: 0x58d9aef8:7cf1b0dc -- Mon Mar 27 19:31:52 2017

crtime: 0x58d9aef8:7cf1b0dc -- Mon Mar 27 19:31:52 2017

Size of extra inode fields: 32

BLOCKS:

(0):360960

TOTAL: 1

Directory has been created in block group 11.

We have repeated the above experiment 5 times. Every time the directory was created in one among the block groups 6,7,8,9,1.

**Block group stats:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

32129 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

32135 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

32256 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

32135 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

32256 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

32135 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

24055 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

32135 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

32256 free blocks, 8160 free inodes, 0 used directories

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

32135 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

32256 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

32255 free blocks, 8159 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

32256 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32256 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

**Experiment 2:** Create files within the directory to check the effect of directory on the location of files.

We then created a couple of files in dir1.

.

├── [ 89761] dir1

│   ├── [ 89762] user1.txt

│   └── [ 89763] user2.txt

└── [ 11] lost+found

The inodes of the files were in the same block group as its parent directory.

Regarding the data blocks,

Inode Analyser# inode 89762

i\_mode = 33188

i\_uid = 0

i\_size = 9919779

i\_atime = 1490661413

i\_ctime = 1490661413

i\_mtime = 1490661413

i\_dtime = 0

i\_gid = 0

i\_links\_count = 1

i\_blocks = 19408

i\_block[0] = 361472

i\_block[1] = 361473

i\_block[2] = 361474

i\_block[3] = 361475

i\_block[4] = 361476

i\_block[5] = 361477

i\_block[6] = 361478

i\_block[7] = 361479

i\_block[8] = 361480

i\_block[9] = 361481

i\_block[10] = 361482

i\_block[11] = 361483

i\_block[12] = 360961

i\_block[13] = 360962

i\_block[14] = 0

i\_flags = 0

i\_generation = 2597155629

i\_file\_acl = 0

i\_dir\_acl = 0

i\_faddr = 0

i\_extra\_isize = 32

i\_pad1 = 0

**user2.txt:**

Inode Analyser# inode 89763

i\_mode = 33188

i\_uid = 0

i\_size = 9919779

i\_atime = 1490661413

i\_ctime = 1490661413

i\_mtime = 1490661413

i\_dtime = 0

i\_gid = 0

i\_links\_count = 1

i\_blocks = 19408

i\_block[0] = 361984

i\_block[1] = 361985

i\_block[2] = 361986

i\_block[3] = 361987

i\_block[4] = 361988

i\_block[5] = 361989

i\_block[6] = 361990

i\_block[7] = 361991

i\_block[8] = 361992

i\_block[9] = 361993

i\_block[10] = 361994

i\_block[11] = 361995

i\_block[12] = 360965

i\_block[13] = 360966

i\_block[14] = 0

i\_flags = 0

i\_generation = 2597155630

i\_file\_acl = 0

i\_dir\_acl = 0

i\_faddr = 0

i\_extra\_isize = 32

i\_pad1 = 0

**Block group stats:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

27317 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

32135 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

32256 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

32135 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

32256 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

32135 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

24055 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

32135 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

32256 free blocks, 8160 free inodes, 0 used directories

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

32135 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

32256 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

32215 free blocks, 8157 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

32256 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32256 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

From the above data, we can easily observe that all the 0 – 13 block pointers were allocated from the same block group as their parent directory. But the indirect blocks were allocated from the first block group.

I also repeated this experiment for all the 5 times. This prediction looks consistent. Every time I created the files, the 0 – 13 blocks were allocated from the same block group of the directory.

**Experiment 3:**

Created some more directories in the first level i.e., the same level as dir1. (to check whether the directories at the same level are well separated)

Created 3 directories.

.

├── [ 89761] dir1 (360960 – BG 11)

│   ├── [ 89762] user1.txt

│   └── [ 89763] user2.txt

├── [ 106081] dir2 (426496 – BG 13)

├── [ 65281] dir3 (262656 – BG 8)

├── [ 40801] dir4 (164473 – BG 5)

└── [ 11] lost+found

**Block group stats:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

27317 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

32135 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

32256 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

32135 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

32256 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

32134 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

24055 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

32135 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

32255 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

32135 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

32256 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

32215 free blocks, 8157 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

32256 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32255 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

**Experiment 4:**

Let us consider dir1 in 11 th block group. Now lets first fill up all the blocks groups from 0 to 10 by creating files in dir1. Then free some blocks in groups 0 to 10 , to check whether the block allocator takes the recently freed blocks.

**Block group stats:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

7829 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

10481 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

8196 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

20771 free blocks, 8054 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

32256 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32255 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

From the below structure, we can confirm that the inodes for all the files are with in the same block group as their parent directories.

├── [ 89761] dir1

│   ├── [ 89762] user1.txt

│  ………………..

│   └── [ 89866] user100.txt

├── [ 106081] dir2

├── [ 65281] dir3

├── [ 40801] dir4

└── [ 11] lost+found

Also, the 0-13 blocks for all these files were allocated from the same block group as the directory.

Now the block allocator is currently allocating the indirect blocks from group 11 i.e., group of directory.

So, at this step both direct and indirect blocks must be allocated from the same block group. I have confirmed this for files 101 to 105.

Inode: 89866 Type: regular Mode: 0644 Flags: 0x0

Generation: 2597155736 Version: 0x00000000:00000001

User: 0 Group: 0 Size: 9919779

File ACL: 0 Directory ACL: 0

Links: 1 Blockcount: 19408

Fragment: Address: 0 Number: 0 Size: 0

ctime: 0x58d9b71e:642608dc -- Mon Mar 27 20:06:38 2017

atime: 0x58d9b71e:61499cdc -- Mon Mar 27 20:06:38 2017

mtime: 0x58d9b71e:642608dc -- Mon Mar 27 20:06:38 2017

crtime: 0x58d9b71e:61499cdc -- Mon Mar 27 20:06:38 2017

Size of extra inode fields: 32

BLOCKS:

(0-11):363152-363163, (IND):366967, (12-15):363164-363167, (16-31):363632-363647, (32-63):366080-366111, (64-127):366144-366207, (128-255):366464-366591, (256-511):371190-371445, (512-1023):371712-372223, (1024-1035):372736-372747, (DIND):366998, (IND):367093, (1036-2047):372748-373759, (2048-2059):374784-374795, (IND):366996, (2060-2421):374796-375157

TOTAL: 2426

Now lets free some blocks from group 0. (by deleting user 1… user 10.txt files)

**Block group stats:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

10481 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

8196 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

20971 free blocks, 8064 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

32256 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32255 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

Now lets create some files to check whether the freed blocks from group 0 are used or not.

**Block Group Stats:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

10481 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

8196 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

13693 free blocks, 8061 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

32256 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32255 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

**The blocks were allocated from group 11 and not from group 0;**

Lets us create some more files to check this; Because in the previous analysis, the freed blocks were reused only at the second time.

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

10481 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

8196 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

10967 free blocks, 8057 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

25278 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32255 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

**Note that the blocks are currently allocated from group 12.**

Then, freed some blocks in group 10, since it is at closest proximity to the directory.

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

13127 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

29610 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

11167 free blocks, 8067 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

25278 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32255 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

Some more files are created.

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

13127 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

29610 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

11107 free blocks, 8064 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

18060 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32255 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

Some more files are created

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

13127 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

29610 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

11007 free blocks, 8059 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

8196 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

30089 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

Some more files are created

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

13127 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

29610 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

10907 free blocks, 8054 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

8196 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

18059 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

free some blocks in group 12

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

13127 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

29610 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

11027 free blocks, 8060 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

22632 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

18059 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

Some more files are created.

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

31889 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

10481 free blocks, 8160 free inodes, 0 used directories

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

8196 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

10481 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

8196 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

10480 free blocks, 8159 free inodes, 1 used directory

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

7213 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

10481 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

10601 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

13127 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

29610 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

10947 free blocks, 8056 free inodes, 1 used directory

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

22632 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

10601 free blocks, 8159 free inodes, 1 used directory

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

27770 free blocks, 8160 free inodes, 0 used directories

Thus we can confirm from these analysis that the block allocator always looks for the blocks in the next block groups and it does not jump back.

**Experiment To Check How The (0-14) First Level Block Pointers Are Allocated When There Are No More Blocks In The Directory’s Block Group:**

Formatted the entire disk and created two directories.

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

32129 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

32134 free blocks, 8159 free inodes, 1 used directory

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

32256 free blocks, 8160 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

32135 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

32256 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

32135 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

24055 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

32135 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

32255 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

32135 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

32256 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

32256 free blocks, 8160 free inodes, 0 used directories

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

32256 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

32256 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

29936 free blocks, 8160 free inodes, 0 used directories

8161 filesDir 11 lost+found 65281 mainDir

→ Used all the free blocks by creating files. The block statistics are shown below.

→ Note that the directory mainDir is in 8 th block group. But it does not have any free blocks. Now let’s create some files in that directory and check where the first level blocks (0-14) are allocated from.

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

16842 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

380 free blocks, 7983 free inodes, 1 used directory

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

0 free blocks, 8152 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

0 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

0 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

0 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

0 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

12030 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

0 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

14676 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

2166 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

0 free blocks, 8160 free inodes, 0 used directories

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

0 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

0 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

0 free blocks, 8160 free inodes, 0 used directories

→ Created two files in the mainDir. The first level blocks were allocated from 10 th block group as shown below.

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

12404 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

380 free blocks, 7983 free inodes, 1 used directory

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

0 free blocks, 8152 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

0 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

0 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

0 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

0 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

11656 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

0 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

14676 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

2126 free blocks, 8158 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

0 free blocks, 8160 free inodes, 0 used directories

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

0 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

0 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

0 free blocks, 8160 free inodes, 0 used directories

**Files:**

81601 user1.txt 81602 user2.txt

Inode Analyser# inode 81601

i\_mode = 33188

i\_uid = 0

i\_size = 9919779

i\_atime = 1491087386

i\_ctime = 1491087386

i\_mtime = 1491087386

i\_dtime = 0

i\_gid = 0

i\_links\_count = 1

i\_blocks = 19408

i\_block[0] = 328704

i\_block[1] = 328705

i\_block[2] = 328706

i\_block[3] = 328707

i\_block[4] = 328708

i\_block[5] = 328709

i\_block[6] = 328710

i\_block[7] = 328711

i\_block[8] = 328712

i\_block[9] = 328713

i\_block[10] = 328714

i\_block[11] = 328715

i\_block[12] = 332148

i\_block[13] = 332149

i\_block[14] = 0

i\_flags = 0

i\_generation = 1758414871

i\_file\_acl = 0

i\_dir\_acl = 0

i\_faddr = 0

i\_extra\_isize = 32

i\_pad1 = 0

Inode Analyser# inode 81602

i\_mode = 33188

i\_uid = 0

i\_size = 9919779

i\_atime = 1491087386

i\_ctime = 1491087386

i\_mtime = 1491087386

i\_dtime = 0

i\_gid = 0

i\_links\_count = 1

i\_blocks = 19408

i\_block[0] = 329728

i\_block[1] = 329729

i\_block[2] = 329730

i\_block[3] = 329731

i\_block[4] = 329732

i\_block[5] = 329733

i\_block[6] = 329734

i\_block[7] = 329735

i\_block[8] = 329736

i\_block[9] = 329737

i\_block[10] = 329738

i\_block[11] = 329739

i\_block[12] = 332146

i\_block[13] = 332147

i\_block[14] = 0

i\_flags = 0

i\_generation = 1758414872

i\_file\_acl = 0

i\_dir\_acl = 0

i\_faddr = 0

i\_extra\_isize = 32

i\_pad1 = 0

→ Created 2 more files and checked. The same allocation happens.

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

12404 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

380 free blocks, 7983 free inodes, 1 used directory

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

0 free blocks, 8152 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

0 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

0 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

0 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

0 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

9624 free blocks, 8160 free inodes, 0 used directories

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

0 free blocks, 8159 free inodes, 1 used directory

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

11896 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

2086 free blocks, 8156 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

0 free blocks, 8160 free inodes, 0 used directories

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

0 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

0 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

0 free blocks, 8160 free inodes, 0 used directories

**Experiment :**

**→** Test to check whether the block allocator allocates the first level blocks from the block group which are in closer proximity with the block group of the directory.

**Initial State:**

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

14436 free blocks, 8149 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

220 free blocks, 7975 free inodes, 1 used directory

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

0 free blocks, 8152 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

0 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

0 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

0 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

0 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

0 free blocks, 8159 free inodes, 1 used directory

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

12030 free blocks, 8160 free inodes, 0 used directories

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

0 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

0 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

0 free blocks, 8160 free inodes, 0 used directories

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

0 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

0 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

0 free blocks, 8160 free inodes, 0 used directories

Note that the directory is in 7th block group. We also have free blocks in 8 th block group which are in closer proximity with it. Now lets check from where the blocks are allocated if we create some files in the directory.

12 user1.txt 13 user2.txt

Group 0: block bitmap at 121, inode bitmap at 122, inode table at 123

9958 free blocks, 8147 free inodes, 2 used directories

Group 1: block bitmap at 32889, inode bitmap at 32890, inode table at 32891

220 free blocks, 7975 free inodes, 1 used directory

Group 2: block bitmap at 65536, inode bitmap at 65537, inode table at 65538

0 free blocks, 8152 free inodes, 0 used directories

Group 3: block bitmap at 98425, inode bitmap at 98426, inode table at 98427

0 free blocks, 8160 free inodes, 0 used directories

Group 4: block bitmap at 131072, inode bitmap at 131073, inode table at 131074

0 free blocks, 8160 free inodes, 0 used directories

Group 5: block bitmap at 163961, inode bitmap at 163962, inode table at 163963

0 free blocks, 8160 free inodes, 0 used directories

Group 6: block bitmap at 196608, inode bitmap at 196609, inode table at 196610

0 free blocks, 8160 free inodes, 0 used directories

Group 7: block bitmap at 229497, inode bitmap at 229498, inode table at 229499

0 free blocks, 8159 free inodes, 1 used directory

Group 8: block bitmap at 262144, inode bitmap at 262145, inode table at 262146

11656 free blocks, 8160 free inodes, 0 used directories

Group 9: block bitmap at 295033, inode bitmap at 295034, inode table at 295035

0 free blocks, 8160 free inodes, 0 used directories

Group 10: block bitmap at 327680, inode bitmap at 327681, inode table at 327682

0 free blocks, 8160 free inodes, 0 used directories

Group 11: block bitmap at 360448, inode bitmap at 360449, inode table at 360450

0 free blocks, 8160 free inodes, 0 used directories

Group 12: block bitmap at 393216, inode bitmap at 393217, inode table at 393218

0 free blocks, 8160 free inodes, 0 used directories

Group 13: block bitmap at 425984, inode bitmap at 425985, inode table at 425986

0 free blocks, 8160 free inodes, 0 used directories

Group 14: block bitmap at 458752, inode bitmap at 458753, inode table at 458754

0 free blocks, 8160 free inodes, 0 used directories

Inode: 12 Type: regular Mode: 0644 Flags: 0x0

Generation: 420143985 Version: 0x00000000:00000001

User: 0 Group: 0 Size: 9919779

File ACL: 0 Directory ACL: 0

Links: 1 Blockcount: 19408

Fragment: Address: 0 Number: 0 Size: 0

ctime: 0x58e05847:0890eea0 -- Sat Apr 1 20:47:51 2017

atime: 0x58e05847:02d75b20 -- Sat Apr 1 20:47:51 2017

mtime: 0x58e05847:0890eea0 -- Sat Apr 1 20:47:51 2017

crtime: 0x58e05847:02d75b20 -- Sat Apr 1 20:47:51 2017

Size of extra inode fields: 32

BLOCKS:

(0-11):1536-1547, (IND):656, (12-15):1548-1551, (16-31):688-703, (32-63):736-767, (64-127):1072-1135, (128-255):896-1023, (256-511):1264-1519, (512-1023):3072-3583, (1024-1035):2048-2059, (DIND):657, (IND):658, (1036-2047):2060-3071, (2048-2059):10240-10251, (IND):659, (2060-2421):10252-10613

TOTAL: 2426

Inode: 13 Type: regular Mode: 0644 Flags: 0x0

Generation: 420143986 Version: 0x00000000:00000001

User: 0 Group: 0 Size: 9919779

File ACL: 0 Directory ACL: 0

Links: 1 Blockcount: 19408

Fragment: Address: 0 Number: 0 Size: 0

ctime: 0x58e05847:0f3ec55c -- Sat Apr 1 20:47:51 2017

atime: 0x58e05847:098531dc -- Sat Apr 1 20:47:51 2017

mtime: 0x58e05847:0f3ec55c -- Sat Apr 1 20:47:51 2017

crtime: 0x58e05847:098531dc -- Sat Apr 1 20:47:51 2017

Size of extra inode fields: 32

BLOCKS:

(0-11):1552-1563, (IND):660, (12-15):1564-1567, (16-31):880-895, (32-63):1216-1247, (64-127):3632-3695, (128-255):3712-3839, (256-511):3968-4223, (512-1023):6656-7167, (1024-1035):5120-5131, (DIND):661, (IND):662, (1036-2047):5132-6143, (2048-2059):274432-274443, (IND):663, (2060-2421):274444-274805

TOTAL: 2426

→ We can easily infer from the above block statistics that the blocks are allocated from the 1st block group and not from the closest block group.

**END**

**Reading data blocks from inode number**

The block group of the desired inode number can found using:

Block\_group\_number = (inode\_number - 1) / s\_inodes\_per\_group

**Note: Inode number start from the index 1 not index 0 i.e. first inode number is 1 not 0 unlike block number.**

Using block group number, inode table block number can be identified by looking at the block group descriptor table (bg\_inode\_table). Once the block of the inode table is identified, the local inode index for the local inode table can be identified using:

Local\_inode\_index = (inode\_number - 1) % s\_inodes\_per\_group

Sample Inode Computations where s\_inodes\_per\_group = 1712.

|  |  |  |
| --- | --- | --- |
| **Inode Number** | **Block Group Number** | **Local Inode Index** |
|  | | |
| 1 | 0 | 0 |
| 963 | 0 | 962 |
| 1712 | 0 | 1711 |
| 1713 | 1 | 0 |
| 3424 | 1 | 1711 |
| 3425 | 2 | 0 |

Since each inode entry is of fixed size, the default size of inode entry is 128 bytes. The exact offset of the desired inode number can be found by:

Inode\_table\_offset = Local\_inode\_index \* INODE\_SIZE

The inode table of the desired inode number can be retrieved by reading INODE\_SIZE bytes from Inode\_table\_offset.

**Locating Data Blocks from the Inode Table**

Once the desired inode’sinode entry is retrieved, the file size in blocks is calculated as below:

FileSizeInBlocks = file\_size / block\_size

The inode table has the file size of the corresponding inode and it can be retrieved by accessing the variable i\_size in the inodetable.If the file size in blocks is less than or equal to 12 then all the data blocks can be accessed by direct blocks addresses i.e. from i\_block[0] to i\_block[11]. The direct block addresses can hold up to 48 kilobytes of data if the block size is 4Kb (4Kb \* 12).

If the file size in blocks is greater than 12 and less than or equal to 1036 (1024 + 12) then the next data blocks can be accessed by single indirect block addresses i.e. from i\_block[12]. The block address i\_block[12] contains block addresses of 4 bytes which contains the data blocks. The single indirect block addresses can hold up to 4 megabytes of data if the block size is 4Kb (4Kb \* 1024)

If the file size in blocks is greater than 1036 and lesser than or equal to 1024 \* 1024 then the next data blocks can be accessed by double indirect block addresses i.e. from i\_block[13]. The block address i\_block[13] contains block addresses of 4 bytes which is a single indirect block addresses. The double indirect addresses can hold up to 4 gigabytes of data if the block size is 4Kb (4Kb \* 1024 \* 1024).

Finally, if the file size in blocks is greater than 1024 \* 1024 then the next data blocks can be accessed by triple indirect block addresses i.e. from i\_block[14]. The block address i\_block[14] contains block addresses of 4 bytes which is a double indirect block addresses. The triple indirect addresses can hold up to 4 terabytes of data if the block size is 4Kb (4Kb \* 1024 \* 1024 \* 1024).

**Note:** Whenever we encounter null address, we can stop looking for data blocks as it indicates the end of the file.

**Reserved Blocks**

According to tune2fs manual, reserved blocks are designed to keep your system from failing when you run out of space. It reserves space for privileged processes such as daemons like likesyslogd and other root level processes; also the reserved space can prevent the filesystem from fragmenting as it fills up. By default, this 5% regardless of the size of the partition.

On large partitions (500 GB drives and up are common these days), the default 5% reserved space can be quite a lot (25GB for 500 GB drive). For ext3 partitions, you can tune this parameter by using tune2fs with the parameter -m.

For e.g. to decrease this to 3% for the drive /dev/sdb1, you would run

Tune2fs -m 3 /dev/sdb1

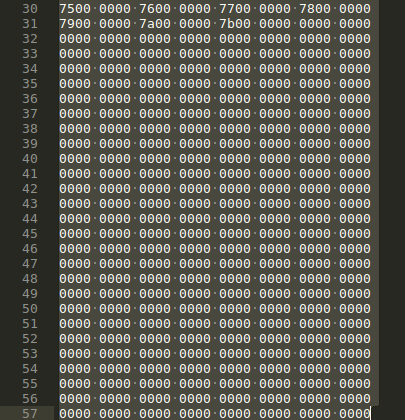
Also, tune2fs -l /dev/sdb1 command shows the total reserved block count.

If we set the reserved block count to zero, it won’t affect the performance much except if we run for long periods of time with lot of file creates and deletes while the system is almost full say 95% at which we will subject to fragmentation problems.If we are just using the filesystem for long term archive where files aren’t changing often (i.e. a huge video store), it obviously won’t matter.

### s\_r\_blocks\_count field in the superblock structure indicates the total number of blocks reserved for the usage of the super user. This is most useful if for some reason a user, maliciously or not, fill the file system to capacity; the super user will have this specified amount of free blocks at his disposal so he can edit and save configuration files.

**Block write observations**

In this section we will summarize some of the observations regarding the block write algorithm. The disk drive used for this behavior testing was of size 2GB with ext3 file system. The drive was filled to the maximum possible extent by repeatedly copying a test file of size 4 MB. These file were later deleted and a new text file was written of size 43KB, given the block size of 4KB, the file was written to the direct 12 data blocks. On using dd to observe the last block, it was noticed the rest of the block was zeroed out. And it contained no junk value. Following screenshot is the output from ‘dd’ command opened as a text file.



**Reserved GDT Blocks**

The resize\_inode feature creates a hidden inode( number 7, you can view it in debugfs stat <7> ) to reserve those blocks so that the GDT can be grown. By default it reserves enough space to grow the filesystem to 1024 times its original size.

Number of data blocks in a block group can be found as below:

Each block group with super block will have

Data Blocks = Blocks Per group - (Superblock + Group descriptors + Reserved GDT blocks + Data Block bitmap + Inode bitmap + Inodestable )

Each block group without super block will have

Data Blocks = Blocks Per group - (Data Block bitmap + Inode bitmap + Inodestable ) =

= 32768 - (1 + 1 + 512 ) = 32254

The total blocks required for inode table in a block group is given by:

Inode\_table\_size\_in\_blocks = total\_number\_of\_inodes\_in\_a\_block\_group \* inode\_table\_size / block\_size.

So if the partition has a block size of 4K and inode table size is 256 bytes then the inode table size for a block group is:

Inode\_table\_size = 8192 \* 256 / 4096 = 512 blocks

**Identifying Block of Addresses:**

Read the first 8 bytes of the data block into an integer variable. If the absolute difference between the integer variables is 1 then it can be concluded that the block is full of addresses as in most of the cases the block allocation algorithm allocates contiguous blocks. If the absolute difference is not 1 then read next 8 bytes of data into an integer variable. Again check the absolute difference between the integer variables, if it is not 1 then it can be concluded that the block is not an address block as it is highly unlikely that block addresses are non-contiguous. If the absolute difference between the integer variables in the second read is 1 then also it can be concluded that it is a block of addresses. Suppose if the block contains non integer values then the integer variable will contain some junk (very big) values and it is highly unlikely that it will be contiguous.

**Identify TEXT Blocks:**

Read the block size of data into a character array. Check if all the characters are in the ASCII range then it can be concluded that the data block is a TEXT block. If any of the character is not in ASCII range then without any further checking it can be concluded that the data block is not a TEXT block.

**Identifying NULL Blocks:**

Read the first four bytes of the data block into a character array. If those first bytes is null or if the string length of those four bytes is 0 then it can be concluded that it is a NULL block. Otherwise, it is not a NULL block.It can be noted that if the beginning of the data block is null then the whole data block should be null. Hence it is sufficient to check if the starting bytes of the data block is null to identify the null block.

**Identifying Directory Blocks:**

If the data block is a directory block then the first entry will be the entry for itself i.e. “.”. The size of first entry will always be 20 bytes. Read the first 20 bytes from the data block into ext3 directory structure and then check if the name field of the directory entry is “.” and the name\_len of the directory entry is 1, it can be concluded that the data block is directory entry.

**Identifying USED blocks:**

Read the block bitmap of the corresponding block group into a character array of size equal to the block size. For each byte, perform bitwise AND with 1, 2, 4, 8, 16, 32, 64 and 128. If the result of AND operation is greater than zero then the corresponding block number is USED otherwise it is UNUSED block. The total number of blocks in a group is block size \* 8 and the block numbers are contiguous i.e. if the block size is 4096 then total number of blocks will be 32768. So the block numbers of the zeroth block group is 0 to 32767 and the block numbers of the first block group is from 32768 to 65535 and so on.

**Color Coding of all the blocks in a block group:**

Given a block group number, the starting block number of that block group can be calculated using block size i.e. if the block size is 4096 then the starting block number of the block group number 4 is 4096 \* 8 \* 4. Read each block from the starting block of the block group up to block size \* 8, check if it is a text block, null block, address block, directory blocks and update the color code bitmap accordingly. The color code bitmap will be of size block size \* 8. The color code bitmap will have a unique value for each of the text, null, address, directory blocks.

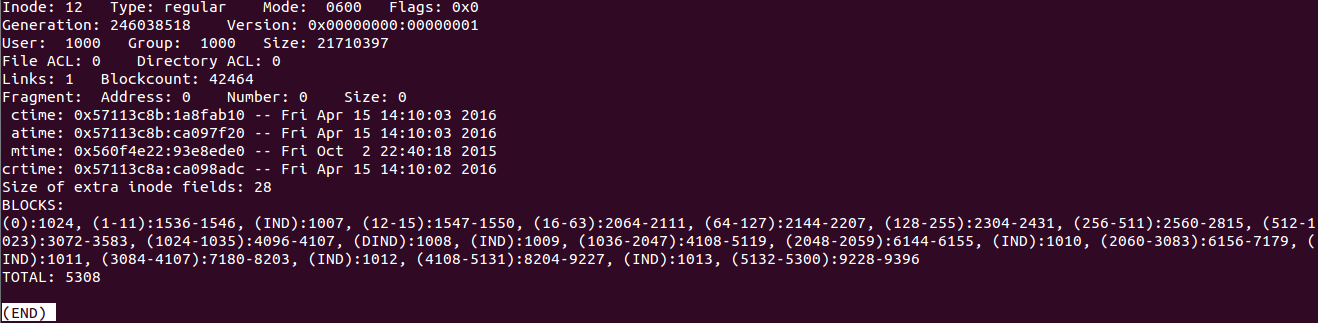
**Block allocation on file modification:**

1)If the existing file in the drive is opened in an editor and saved without any modification then the file system will create new inode for that file and performs new block allocation. It was observed that block allocation will be entirely different from the earlier block allocation and the block allocation starts from the beginning rather than the point from where the previous block allocation was stopped.

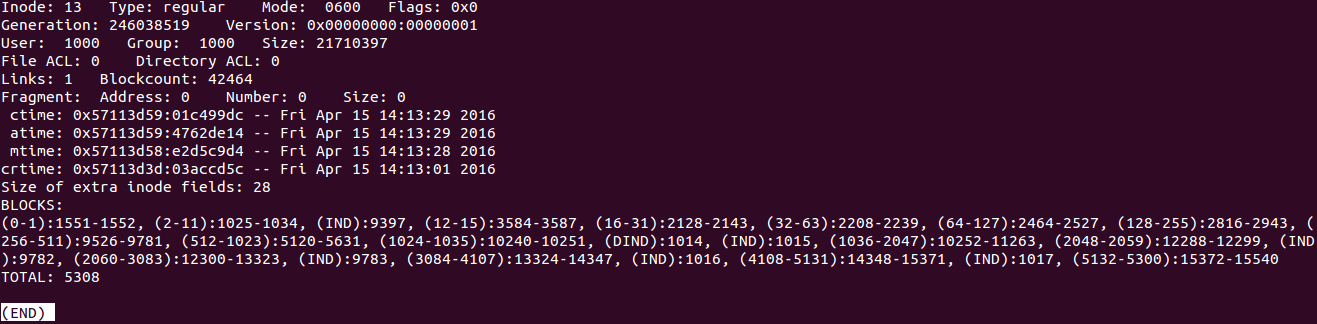
**Initial block allocation for a file:**

The following commands are used:

1. Debugfs /dev/sdb1(drive\_name)
2. Debugfs stat <file\_inode\_number>



**Block allocation after save without any content change:**



2)If the existing file in the drive is opened in an editor and some of the characters are modified without altering the size then also the file system will create new inode for that file and performs new block allocation.

3) If the existing file in the drive is opened in an editor and a new text is added at the end of the file then also the file system will create new inode for that file and performs new block allocation.

\*Note: Please note that the observations made above depends on the behavior of the application used to edit the file. In case of Linux’s native “LibreOffice Writer”, if you do not make any changes and save the file, the new inode is not created. However, if you were to delete a character and put the same character back, the applications does create a new inode on save. And if you use vi/vim any save would created a new inode regardless of the changes.

## 3. Data Block Bitmap

### a. Description

### b. Type of Corruption

### c. Recovery

## 4. Inode Bitmap

### a. Description

### b. Type of Corruption

### c. Recovery

## 5. Inode Table

### a. Description

*(NAVEEN - START)*

In the ext file system, Inode is the data structure used to represent a file system object (file, directory, etc..). Each object in the filesystem is represented by an inode. The inode structure contains pointers to the filesystem blocks which contain the data held in the object and all of the metadata about an object except its name. The metadata about an object includes the permissions, owner, group, flags, size, number of blocks used, access time, change time, modification time, deletion time, number of links, fragments, version and extended attributes (EAs) and/or Access Control Lists (ACLs).

**What is Extended Attributes (EA) ?**

Extended attributes are extensions to the normal attributes which are associated with all inodes in the system. They are often used to provide additional functionality to a filesystem—for example, additional security features such as Access Control Lists (ACLs) may be implemented using extended attributes.

**What is Access Control List (ACL) ?**

This allows file owners and system administrators to define who has access and what level of access they have beyond the level of fliesystem permissions. With extended ACLs, one can assign multiple users rather than just one as owners to a certain file.

The inode table is an array of inodes. Each block group contains an inode table and an inode bitmap. The inode bitmap is used to mark the inodes as free/used. The default size of an inode is 256 bytes (In Ext 4 default size is 256 bytes, in Ext 3 it is 128 bytes, This size is the space used to store the attributes of the inode ) and there would be 8176 inodes per Block Group. So, for a disk of size 16GB with 4KB block size (each block group would be 128MB), the number of block groups would be

16 \* 1024 / 128 = 128 Block Groups

and the total number of inodes in the filesystem would be

128 \* 8176 = 1046528 Inodes

These inodes would be numbered sequentially from 1 to 1046528.

**Storing objects using Inodes**

The root directory of the disk is always stored in the Inode 2(Inode 0 and Inode 1 are reserverd that is 0 is used as a NULL value, to indicate that there is no inode. Inode 1 is used to keep track of any bad blocks on the disk) and this root would be the parent/ancestor of all the objects that are present in the disk.

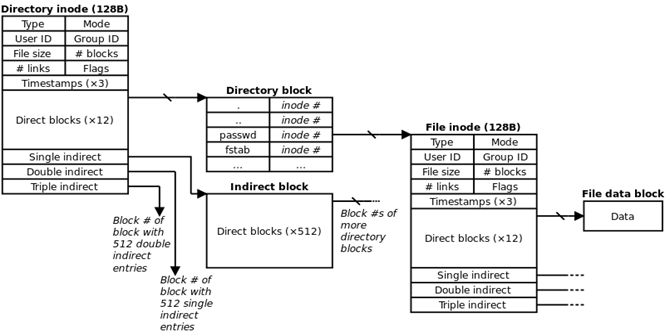


Figure 1: Structure of an inode (2KB block size)

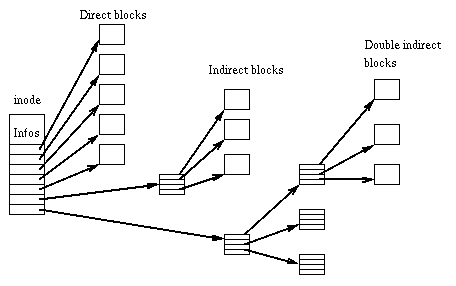
In ext2 filesystem an inode contains 15 block pointers that are used to identify the blocks which contain the actual data held by the file system object. The block pointers itself occupies 4 Bytes. These 15 blocks are classified as

1. Direct Block Pointers (12 blocks)

2. Single Indirect Block Pointers (1 block)

3. Double Indirect Block Pointers (1 block)

4. Triple Indirect Block Pointers (1 block)



*Direct Block Pointers:* The first 12 block pointers are called as Direct Block pointers. These pointers point to the address of the blocks containing the data of the file. 12 Block pointers can point to 12 data blocks. So, for 4KB block size, the Direct Block pointers can address only 48KB (12 \* 4KB) of data in total. This means if the file is only of 48KB or below in size, then inode itself can address all the blocks containing the data of the file.

*Single Indirect Block Pointers:* Whenever the size of the data goes above 48KB(by considering the block size as 4KB), the 13th pointer in the inode will point to the very next block after the data(adjacent block after 48KB of data),which in turn will point to the next block address where data is to be copied. The amount of data that can be addressed with this addressing scheme can be calculated as follows,

Size of a block = 4 KB

Size of a block pointer = 4 Bytes

No. of block pointers per block = 4096 / 4

= 1024

Now as we took our block size as 4KB, the indirect block pointer, can point to 1024 blocks containing data(by taking the size of a block pointer as 4 Bytes, one 4KB block can point to 1024 blocks because 4 Bytes \* 1024 = 4KB). This means an indirect block pointer can address, up to 4MB of data(4 bytes of block pointer in 4KB block, can point and address 1024 number of 4KB blocks which makes the data size of 4MB).

*Double Indirect Block Pointers:* Now if the size of the file is above 4MB + 48KB then the inode will start using Double Indirect Block Pointers, to address data blocks. Double Indirect Block pointer in an inode will point to the block that comes just after 4MB + 48KB data, which in turn will point to the blocks where the data is stored. Double Indirect block pointer also is inside a 4KB block as every blocks are 4KB, Now block pointers are 4 bytes in size, as mentioned previously, so Double indirect block pointer can address 1024 Indirect Block pointers(which means 1024 \* 4MB =4GB). So with the help of a double indirect Block Pointer the size of the data can go upto 4GB.

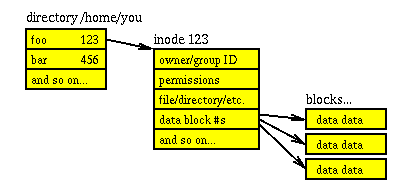
*Triple Indirect Block Pointers*:The fifteenth block pointer in the inode will point to the block just after the 4GB of data, which in turn will point to 1024 Double Indirect Block Pointers. This triple Indirect Block Pointers can address up to 4TB (4GB \* 1024).

**Block address and Block pointer values**

During initial creation of the partition, the file system blocks are numbered sequentially and these numbers are used as addresses to identify a block. The block pointers store these numbers to locate the contents of a file. For example, if a disk of size 16GB, formatted with 4KB block size, the total number of blocks would be 4194304 kilobits (16G\*4K\*8). These blocks would be numbered from 0 to 4194303. Now, if a file foo.txt (of size 1 KB) holds a data at block number 1410, the first direct block pointer will have the value 1410 and all the other pointers will have the value 0 (which indicates NULL).

**Filename and Inode Mapping**

As mentioned earlier, the inode structure does not store the name of the file. This information is maintained by the parent directory of the file. The data block of a directory inode contains information about the name, file type, inode number about all of its children (one record per child), its parent and the directory itself. The 'name' field for the parent record contains the value ”..” and the current directory contains the value “.”. So, given an inode, it is impossible to find the name of the file without knowing it's parent directory. Each record in the directory data block is represented using the structure ext3\_dir\_entry\_2. The 'file\_type' attribute can be used to determine if the file is a normal file or a directory.



Above image shows a simple unix file internals.

**Data block of a directory inode**

Ext2 implements directories as a special kind of file whose data blocks store file-names together with the corresponding inode numbers. In particular, such data blocks contain structures of type ext2\_dir\_entry\_2 . The structure has a variable length, since the last name field is a variable length array of up to EXT2\_NAME\_LEN characters (usually 255). Moreover, for reasons of efficiency, the length of a directory entry is always a multiple of 4 and, therefore, null characters ( \0 ) are added for padding at the end of the filename, if necessary. The name\_len field stores the actual file name length. The file\_type field stores a value that specifies the file type. The rec\_len field may be interpreted as a pointer to the next valid directory entry. It is the offset to be added to the starting address of the directory entry to get the starting address of the next valid directory entry. To delete a directory entry, it is sufficient to set its inode field to 0 and suitably increment the value of the rec\_len field of the previous valid entry.

struct ext3\_dir\_entry\_2 {

\_\_le32 inode; /\* Inode number \*/

\_\_le16 rec\_len; /\* Directory entry length \*/

\_\_u8 name\_len; /\* Name length \*/

\_\_u8 file\_type; /\* File Type \*/

char name[EXT3\_NAME\_LEN]; /\* File name \*/

};

For any directory, the first entry of the data block contains the inode number of the directory itself (. entry) and the second entry contains the inode number of the parent directory (.. entry). The structure of a directory block is shown below (for 4 KB block size).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| InodeNum | Record Length | Name Length | File Type | Name |
| <This Inode no.> | 12 | 1 | 2 | ./0/0/0 |
| <Parent Inode no.> | 4084 for empty directory  12 for non-empty directory | 2 | 2 | ../0/0 |
| <Child 1 Inode no.> | <rec\_len> | <name\_len> | <file\_type> | <name> |
| <Child 2 Inode no.> | <rec\_len> | <name\_len> | <file\_type> | <name> |
| <Child n Inode no.> | <4096 - bytes occupied by the previous records > | <name\_len> | <file\_type> | <name> |

**Locating an inode**

The inodes are numbered sequentially and distributed over the entire block group range. The number of inodes per block group can be found from the super block (refer super block documentation). Given an inode number, the block group where the inode belongs and the local index of the inode in that block group's inode table can be obtained using the following formulae,

block group = (inode - 1) / inodes per block group

Once the block is identified, the local inode index for the local inode table can be identified using:  
 local inode index = (inode - 1) % inodes per block group   
 The s\_inodes\_per\_group of the super block contains the value of inodes per block group.

Did you know the -i option in linux is used to locate inode number (in red). For example

vinay@oraclevm1% ls -i

2637825 bin 983041 etc 1572865 lib 2981889 media 2531329 root 106497 selinux 81921 usr

196609 boot 2 home 1761281 lib64 2129921 mnt 6416 run 2457601 srv 425985 var 312978 vinaytest

**Types of Corruption**

**Directory Inode**

Validating the Inode data block pointer:

A Directory inode is valid, if it's data block pointer points to the correct data block. This can be verified by the following steps:

1. Read the first record of the data block in ext\_dir\_entry\_2 structure.

2. Check if the ext\_dir\_entry\_2 record has an 'inode' value that matches with the 'actual inode number' and the directory 'name' is “.”.

3. If step 2 is TRUE then the inode is a valid directory inode otherwise it is not a valid inode.

IMPORTANT NOTE: A valid inode does not mean the data or the other inode fields is not corrupted. It simply makes sure the data block pointer points to the correct data block. All corrupted inodes are invalid inodes and some valid inodes can be corrupted inodes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Inode** | **Data Block Pointer** | **Data Block** | **Inference** |
| 1 | Non-Zero | Non-Zero | Valid | The inode is valid |
| 2 | Non-Zero | Non-Zero | Invalid | The inode is not a valid inode. There are two possibilities:  1. The data block pointer is corrupted or  2. The data block is corrupted |
| 3 | Non-Zero | Zero | - | This means the inode has a non zero value, but the block pointer is NULL. There are three possibilities:  1. The inode is actually unused and junk values corrupted the inode block.  2. The inode is actually used but the data block pointer is corrupted.  3. The inode is a file inode of zero size. |
| 4 | Zero | - | - | The inode is a free inode |

**Type 1:**

Now, that the inode is valid,

1. How to verify other fields of the inode are valid ?

2. How to verify the rest of the data block is not corrupted ?

**Type 2:**

How to determine whether the data block pointer or the data block is corrupted ?

Approach 1: Check how the data block pointer is allocated for an inode. With that, check the validity of the data block pointer.

Approach 2: A naive solution,

1. Iterate through all the data blocks

2. Read the first ext3\_dir\_entry\_2 record.

3. If it contains a valid record with name “.”,

a. Go to inode no. given by the ext3\_dir\_entry\_2 record.

b. Check if the block pointer points to the data block.

c. If not, make the block pointer point to this data block.

(Should the ext3\_dir\_entry\_2 be validated ?)

**Type 3:**

Case 1 will be fixed when Inode bitmap is reconstructed.

Case 2 is similar to Type 2 block pointer corruption.

Case 3, How to determine if the inode is a file or a directory ?

**Type 4:**

The inode is a free inode. No need to do anything.

The solutions discussed in this document deal with the corruption of data block pointer of a directory inode and not the actual data block.

**Directory tree recreation after corruption**

The files and directories of the file system are the nodes of a tree rooted at the partition root directory. When some of the directory inodes are corrupted, the file system may become a forest containing several trees. The recovery algorithm should be able to combine these forests into a single tree rooted at the partition root directory. The corrupted directory inode may not have information about itself, the parent and the children. An example for a good file system tree is shown below,



Every directory inode’s data block contains an entry for its parent and other entries for the children. The upward arrow in the image indicates the parent entry in an inode and the downward arrow indicates a child entry. A corruption scenario is described below,



In the above scenario, the data blocks of directory inodes Dir 1, Dir 2 and Dir 3 are corrupted. It can be observed that the links are unidirectional to these corrupted blocks. This means that the corrupted inodes lost its parent entry and the children entry. Now, the files and directories that are children of these corrupted directories are lost. We proposed two algorithms to recover the corrupted directories. These algorithms recovers only the directories (and its subtree) and not the files that are children of a corrupted directory. The first algorithm requires allocation of a new data block to the corrupted inode and the second does not. In the first algorithm the file system tree structure would be same before corruption and after recovery, but in the second algorithm it would be different.

**Assumptions:**

1. The inode of a parent will always be greater than the inode of its children

2. If the inode is valid, then it is not corrupted (i.e, The directory entries read from the data block are correct entries).

**Algorithm 1:**

This algorithm assumes that, if the inode is invalid, then it is corrupted and instead of recovering the actual data block, a new unused data block can be allocated.

In order to form the tree,

For every corrupted directory inode, the algorithm should

1. fix the link to it's parent and

2. fix the links to its children

*Fixing the Link from a corrupted inode to it's child*

MAX\_INODES = num of inodes in the partition

Initialize Bitmap[MAX\_INODES] = 0

for each inode in INODES\_LIST

{

read first record from inode->block\_pointer[0] in DirEntry

if (validateInode(inode, DirEntry) == FALSE)

{

/\* TODO: find which one is better, Recover Data block or allocate new data block ? \*/

Bitmap[inode] = 1 /\* Mark this inode as corrupted \*/

Allocate new data block;

continue;

}

read second record from inode->block\_pointer[0] in DirEntry

if (validateDirectoryEntry(DirEntry) == TRUE)

{

/\* The parent record is valid \*/

/\* TODO what if the parent inode is corrupted ? \*/

if (Bitmap[parent\_inode] == 1)

{

/\* The parent is corrupted \*/

write this child information (with an arbitrary name, eg: Recovery\_<inode no.>) to the newly created data block pointer of the parent.

/\* TODO: proper way to fill the name field of the child's directory entry at the parent ? The name of this directory is forever lost and can not be retrieved if the parent inode's data block is corrupted. \*/

}

}

}

The algorithm shown above would produce the following result,



It can be observed that, new data blocks are allocated to the directory inodes Dir 1, Dir 2 and Dir 3. The lost directories are now linked to these corrupted inodes by adding a child entry in the newly created data blocks.

*Fixing the Link from a corrupted inode to it's parent*

/\* Now Bitmap[] will contain 1 for corrupted inodes and 0 for non-corrupted inodes \*/

for each inode in INODES\_LIST

{

/\* If this is a corrupted inode, do nothing \*/

if (Bitmap[inode] == 1)

continue;

/\* Start reading from the third record (to skip . and ..)\*/

for each ChildDirEntry in inode→block\_pointer[0]

{

/\* Check if the child is corrupted using the Bitmap array updated by the previous algorithm \*/

if (Bitmap[ChildDirEntry.inode] == 1)

{

/\* This child is corrupted \*/

write the '..' entry to the data block of ChildDirEntry.inode

}

}

}

After execution the result would be as shown in the figure below,



It can be seen that, both the parent link and children link are fixed to the corrupted inodes.

This algorithm requires a bit array of size n, where n is the no. of inodes. For eg. If there are 10 million inodes in the system, the array can be a byte array of size 1250000 (10 million / 8) and can be created as char bitmap[125000]. This array takes 1.25MB in memory. The algorithm would fail if the assumption 1 is invalid. But that would happen only in rare cases.

The final file system object tree would look like,



Since, the algorithm only recovers lost directories (and its subtree), the files ‘File 2’ and ‘File 3’ are not a part of the tree after recovery. Notice that the parent- child relationship is not changed by the recovery algorithm.

*How to recover corrupted Root inode ?*

Since we are dealing only with data block pointer corruption and not the data block corruption, if the root inode’s data block pointer is corrupted, fixing the data block pointer would recover its children. The root inode always uses the first data block of the file system. So, if the root inode is invalid, then finding the first data block and setting the data block pointer of the root inode to this data block would fix the corruption. This is because, the root inode used the first data block to store the children entries, before corruption.

*Validating the directory entry*

This field rec\_len indicates the length of the record in bytes and is always a multiple of 4. The name\_len contains the length of the name. For example, If a record has a name length of 1 character, the actual record length would be 9 bytes, but rec\_len will have a value 12 bytes (multiple of 4) and name\_len will have a length 1 bytes. The 3 bytes after 9th byte will be set to '/0' and won't be available for the next record. The rec\_len for the last entry will be the length of last record + the free bytes. A sample data block (block size 4096 bytes) of a directory is shown below,

|--InodeNum: 760369 rec len: 24 name len: 9 file type: 2 file name: SplayTree

|--InodeNum: 760369 rec len: 12 name len: 1 file type: 2 file name: .

|--InodeNum: 2 rec len: 12 name len: 2 file type: 2 file name: ..

|--InodeNum: 760370 rec len: 48 name len: 14 file type: 1 file name: SplayTree.java

|--InodeNum: 760371 rec len: 4024 name len: 6 file type: 2 file name: newDir

An ext3\_dir\_entry\_2 record can be verified as follows,

if (rec\_len == (ceil((name\_len + 8) / 4) \* 4))

{

/\* All valid records except last record \*/

bytesReadSoFar += rec\_len;

/\* Action for VALID\_RECORD \*/

}

else if (rec\_len == (BlockSize - bytesReadSoFar))

{

/\* Last record \*/

/\* Action for VALID\_RECORD; \*/

}

else

{

/\* Still the record could be a valid record, if the next entry was already deleted. In that case the rec\_len would contain actual\_rec\_len + deleted\_entry\_rec\_len. When a file is deleted, the directory entry corresponding to that would contain a inode value 0 \*/

1. actual\_rec\_len = (8 Bytes + name\_len) as a multiple of 4

2. Goto CURRENT\_OFFSET + actual\_rec\_len

3. Read 4 bytes to get the inode of the next record in delInode

if (delInode == 0)

{

/\* Action for VALID\_RECORD \*/

}

else

{

/\* Action for INVALID\_RECORD \*/

}

But the above method does not validate the rec\_len field. As a result of this, if multiple next entries were deleted and the rec\_len field of the current entry is corrupted, this method will not detect it.

To fix this issue, in addition to checking the inode of the deleted next record, validate the actual next record (given by the rec\_len, skipping the deleted records). If it is a valid , then the current record is valid.

}

*(NAVEEN - END)*

*(Anusha - START)*

**Algorithm 2:**

The application currently uses the algorithm that is mentioned below, which is a modification of Algorithm 1. Instead of allocating a new unused data block for the corrupted inode’s directory entry, all the directories under this inode is linked to the root entry as recovered directories.

In this algorithm, the corrupted directory inode is not fixed, instead all the directories under this inode are recovered and added as the children of the root. Therefore,

1.Parent of the directories under the corrupted inode is changed to root.

2.New directory entries corresponding to the directories under the corrupted directories are added.

It is seen that the files directly under the corrupted directory inode are lost and cannot be recovered.

***Pseudocode***:

MAX\_INODES = num of inodes in the partition

Initialize Bitmap[MAX\_INODES] = 0

for each inode in INODES\_LIST

{

read first record from inode->block\_pointer[0] in DirEntry

if (validateInode(inode, DirEntry) == FALSE)

{

Bitmap[inode] = 1 /\* Mark this inode as corrupted \*/

continue;

}

read second record from inode->block\_pointer[0] in DirEntry

if ( Bitmap[second\_record.inode] == 1)

{

/\* The parent record is corrupted \*/

Update this child’s parent information as the root

Write this child information (with an arbitrary name, eg: Recovery\_<inode no.>) as a new entry to the directory entry of root inode data block.

}

}

The algorithm mentioned above will produce the following result,

afterRun (1).png

It is seen that the corrupted directories Dir1, Dir2, Dir3 data blocks are left as is and not modified. Instead, the correction is made to the children directories.

*Adding children entries:*

To add new child entries to the directory entry of root,

1.Iterate through all the directory entries in the data block up to the last entry.

2.Update the last entries *rec\_len*to the nearest multiple of 4 that can hold the record length for that entry.

3. Offset to *rec\_len* and write the new directory entry with the *rec\_len* as rest of the bytes in the data block.

From the illustration above, it is seen that the links to children entries to the corrupted directory inode is fixed (as in changed to root inode) and a new entry is added to the root for the child inode. Also, it is observed that regular files directly under the corrupted inode do not hold the information linking it to the parent (corrupted) inode and hence, these files are lost and cannot be recovered.

Similar to Algorithm 1, this algorithm uses a bit array of size n, where n is the number of inodes, to store the bitmap of corrupted inodes.

After the complete execution of the recovery algorithm, the following structure is observed in the file system.

Recovered.png

Unlike Algorithm 1, here, the tree structure is not maintained but it consumes less space and time for recovery. Also, the corrupted inodes are left as is.

***What if Root inode is corrupted?***

Similar to Algorithm 1, the root inode corruption deal with data block pointer corruption and not the data block itself. By fixing this pointer to the data block, the directory entries under the root can be read. This can be done easily because the root used the first data block for its directory entries.

**File storage:**

Files in an ext filesystem are of different types such as regular files (Stores data (text, binary, and executable)), directories etc. Regular files are the most common type of files stored. When they are first created they do not need any data blocks. The data blocks are allocated only when the file starts to have data in it. In ext file system, the directories are a special type of files whose data blocks have the file name and the inode information of the files under that directory. This data in the data blocks of the directory is contained in a structure called directory entry.

The directory entry usually contains the inode number, directory entry length, file name length, file type and the name of the files under it. This is where the information about the files can be obtained from. The root directory always has the inode number 2. Each file is associated with a unique inode which helps in locating the file.

**Inode creation and block group selection:**

When an inode is created, the block group in which the inode is stored is determined by whether the file the inode is pointing to is a directory or not.

If the new inode is a directory, the ext filesystem tries to find a block group such that the directories are evenly scattered through partially filled block groups.

· Particularly, ext filesystem allocates the new directory in the block group that has the maximum number of free blocks among all block groups that have a greater than average number of free inodes.

· The average number of free inodes can be calculated as

Avg. free inodes = (the total number of free inodes) / (the number of block groups)

If the newly created inode is not a directory, it allocates in the block group that as free inodes.

· In particular, it first selects the block group in which the parent directory is present and then moves further away for allocation.

· It usually performs a logarithmic search starting from the parent directory The algorithm jumps further ahead until it finds an available block group.

*o* ***Example****: if the number of the starting block group is i, the algorithm considers block groups i mod (n), i+1 mod (n), i+1+2 mod (n), i+1+2+4 mod (n), etc.*

· If the logarithmic search failed in finding a block group with a free inode, it performs an exhaustive linear search starting from the block group that has the parent directory

After finding the block group that has the free inode, it finds the first null bit in the inode bitmap of the selected block group, marks it and then decrements the total number of free inodes count by one.

**Corruption with respect to inodes:**

A corruption in the ext filesystem in terms of file corresponds to the corruption of the inode information such as the inode bitmap and the inode table.

**Inode bitmap corruption**:

The inode bitmap is usually used in the allocation of inodes where the bitmap is used to identify the free locations on the inode table. The bitmap just occupies one block and is a sequence of 0s and 1s where 0 means that the corresponding inodein the \_inode\_table\_ is free and 1 specified that it's used. In case of the corruption in the filesystem, the inode bitmap cannot be relied on to check back on the number of free inodes in the block group. It is required to access all the inode tables and find out the inode entries, reconstruct the inode bit map in order to recover the lost information.

Sometimes, a situation where both the inode bitmap and the inode table are corrupted can be encountered. In any casero, should the inode bitmap cannot be trusted to obtain any information which makes the recovery pcess tedious. In case of inode table corruption, there can be two types:

**Inode corruption - directories:**

If the inode of a directory is corrupted, the tree structure of the file system is lost, instead forest where the details of the files under the directory are lost (but not the actual data) is created. Therefore, there is no way to link the files under the directory to the directory itself.

Recovery: In this case, identification of all the sub trees in the forest and trace back to the root to create the original filesystem should be carried out. A new inode can be created to take the place of the lost directory inode in order to rebuild the tree.

From the block allocation algorithm, it is seen that the inode of a file is usually stored in the same block as that of the parent directory (corrupted inode in this case). With this information inode bitmap can be reconstructed to see where the file is located.

If the files’ inodes are not in the same block group as that of the lost directory’s inode, then it is required to scan all the data blocks as a directory entry to check if the data block belongs to the directory intended to be found. This would take a lot of processing time owing to the large number of data blocks in the file system.

**Inode corruptions - files:**

If the corruption in inode is considered, the other possibility is that the inode belongs to a regular file. In this case, only a directory entry that has the file information in it is present whereas the required inode to point to the exact data block where the file is stored is corrupted and cannot be used to located the data blocks of the file.

* In order to rectify this, the data blocks in the block group of the directory file can be searched to see if they belong to the lost file. But, there is no way to verify if the data blocks actually do belong to the file. If there is a certain algorithm that allocates data blocks to the inodes, it is possible to recover using reverse engineering technique.
* Form the inode number it is possible to locate the block group in which the file is stored in by using ,

*Block group = (inode - 1) / inodes per block group*

* This would also be a tedious process owing to the large number of data blocks. But by finding the block group of the file the search space can be reduced by total number of block groups time. It is only required to search and rearrange data blocks in one block group.

*(Anusha - END)*

## 6. Data Blocks

### a. Description

### b. Type of Corruption

### c. Recovery

A file contains an ordered sequence of markers, parameters, and entropyencoded segments that are spread over multiple blocks. For differentiating each file type data from other files data one can only rely on the markers.

First all data blocks of the storage device are scanned for known file markers. Each file type header marker is will have multiple bytes in length. Although there are many markers, few can be rather easily distinguished. Among these, the most important ones are the header marker which will be located at the beginning of a block and mark the beginning of a file. The end of file marker, which will be located in the last file block, and the restart markers, that will appear almost periodically to establish synchronization, are the other useful ones.

Especially in ext file sysytems all the data is stored in the 12 direct data blocks, 1 first indirect, 1 second indirect, 1 third indirect block. Please refer to the Inode section for more details about the inode.

Most of the time all the data blocks will be stored sequentially but the indirect blocks will be stored somewhere in the file system. The main goal of the file recovery process is to retrieve the block numbers of these 15 address blocks (12 direct + 3 indirect). Indirect blocks will store the 32 bit addresses of the data blocks, second indirect block will store n 32 bit first indirect block addresses and so on. So, for example, if you find the header in a block then the thirteenth data block will be stored sequentially. Now, the algorithm is to search in the remaining blocks whether that particular block number (address) is stored in any data block or not. If that is found then that particular data block is the indirect block.

Till now we found the first indirect block address and check whether the last four bytes have data or not. (“/x0/x0/x0/x0”). That means the file data is completed. If there is another address then search for thr footer marker if the footer is present then the file is end. Otherwise the process continues until it finds the third indirect block address. Now, we have the following information:

**typedef** **struct** file\_recovery {

**long** **long** directBlockOffset[12];

**long** **long** firstIndirectBlockOffset;

**long** **long** secondIndirectBlockOffset;

**long** **long** thirdIndirectBlockOffset;

} file\_recovery;

**Reconstruction:**

There are two ways of reconstructing the file with this information. The first one is to reconstruct the inode with this valuable information. Howver, this approach might not be reliable because different operating systems handle this information differently.

The other one is to write the file to a new disk using this data blocks information. First 12 blocks is written to the disk and then later if an indirect block is there then all the data blocks referred by the address location in the indirect block is retrieved and written to the file. In this way the file will be reconstructed and stored it in different disk.

## 7. List all deleted files

### Overview(Xinyi Li)

We need to list all deleted files, and their path and inode number, so that we can choose which file to be recovered. Moreover, knowing the inode number of the file we want to recover, we can search corresponding block group to find the data blocks. In that way, we can save time.

### Main idea

1. For deleted files (Xinyi Li)

Scan from root directory, read directory’s data block, if the entry’s real length is not equal to the attribute rec\_length, it shows that the following entries are deleted, so we should print the these entries until the rec\_length is end; moreover, if we meet directory entry, we should scan it as before recursively. Then we can list all deleted files and directories from root.

1. For deleted directories(Zhengkai Huang)

After a file deletion from disk, it marks as deletion from inode table. The directory entries in data blocks are still in the original location. The way to scan directory is to scan every data block for every data block group and look for a directory block. Hence, to get the deleted directories list, the group descriptor finds group descriptor table byte offset of block groups then scan for directory blocks.

### Problem and how to solve it

1. Repeatition(Xinyi Li):

Create many files in one directory, when the total length of these entries exteed the capacity of one block, then all these entries will be distributed into three blocks

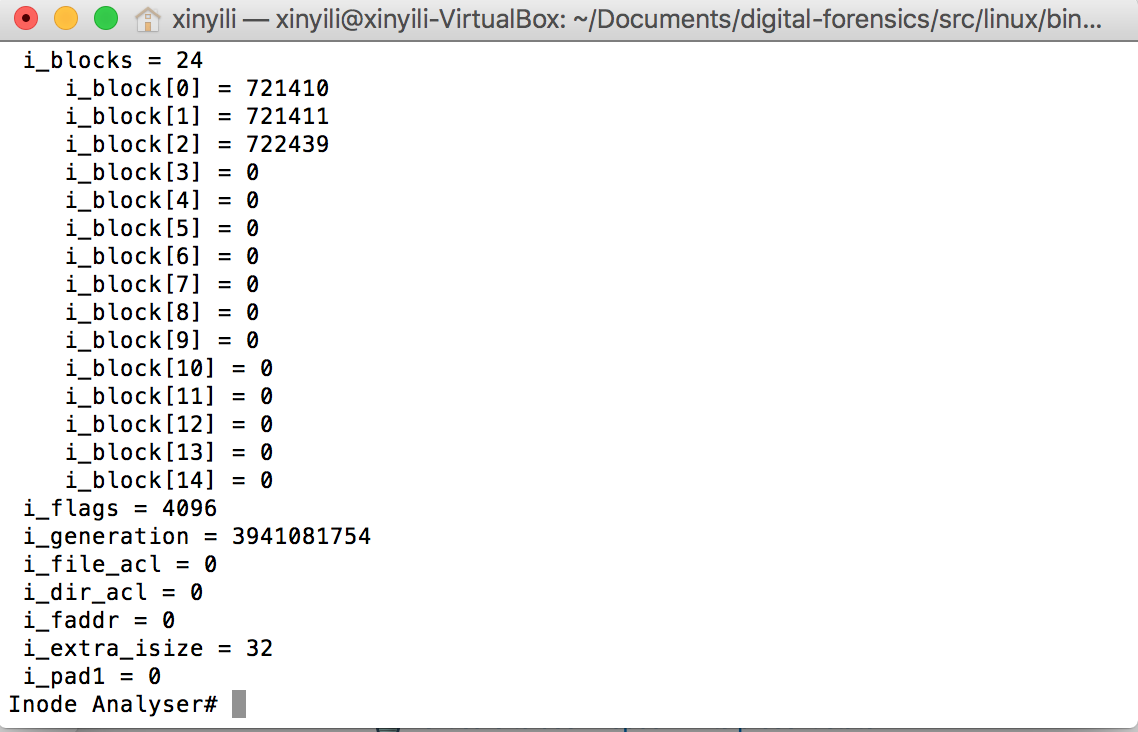


Figure 1 when not exteed one block size

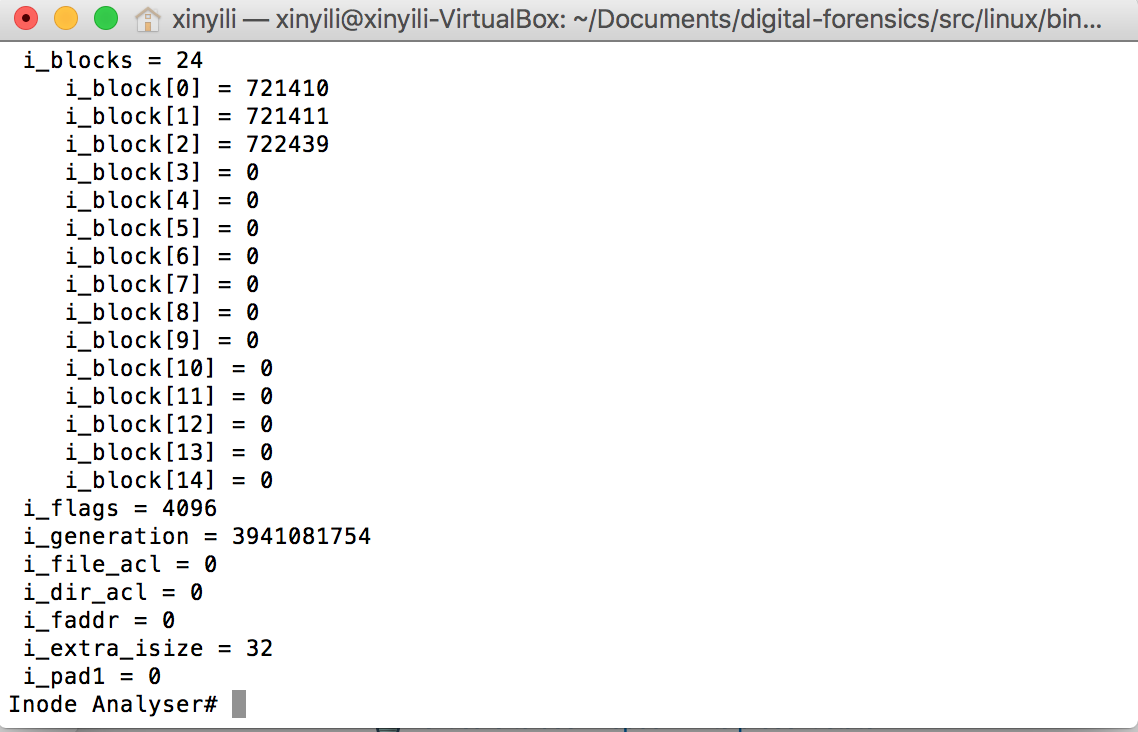


Figure 2 when just exteed one block size

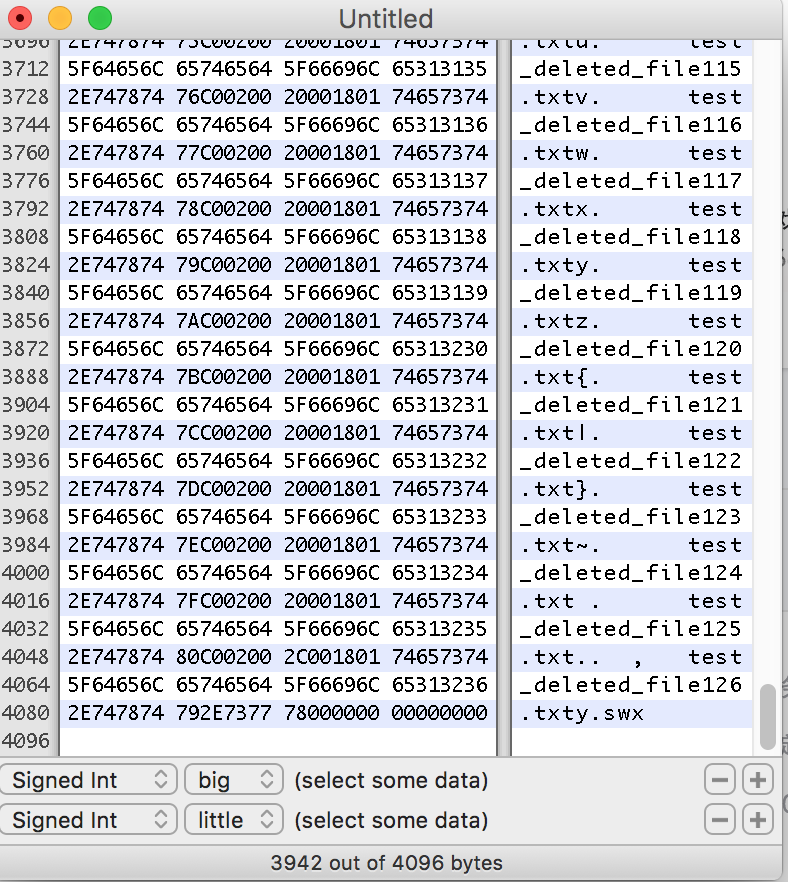


Figure 3 block contents exteed before

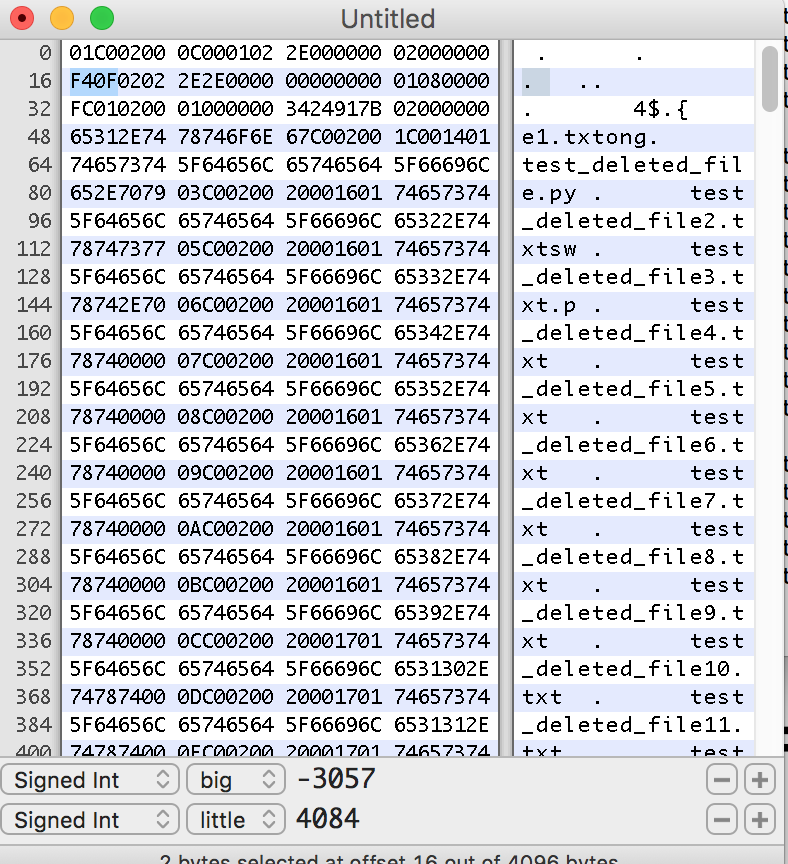


Figure 4 first block contents after exteeding

After exteeding, we can see in first block, the second entry’s(“..”) rec\_length is 4084, that means in the first block, there only exists 2 entries after exteeding. The remaining entries are distributed into the 2nd and 3rd block.

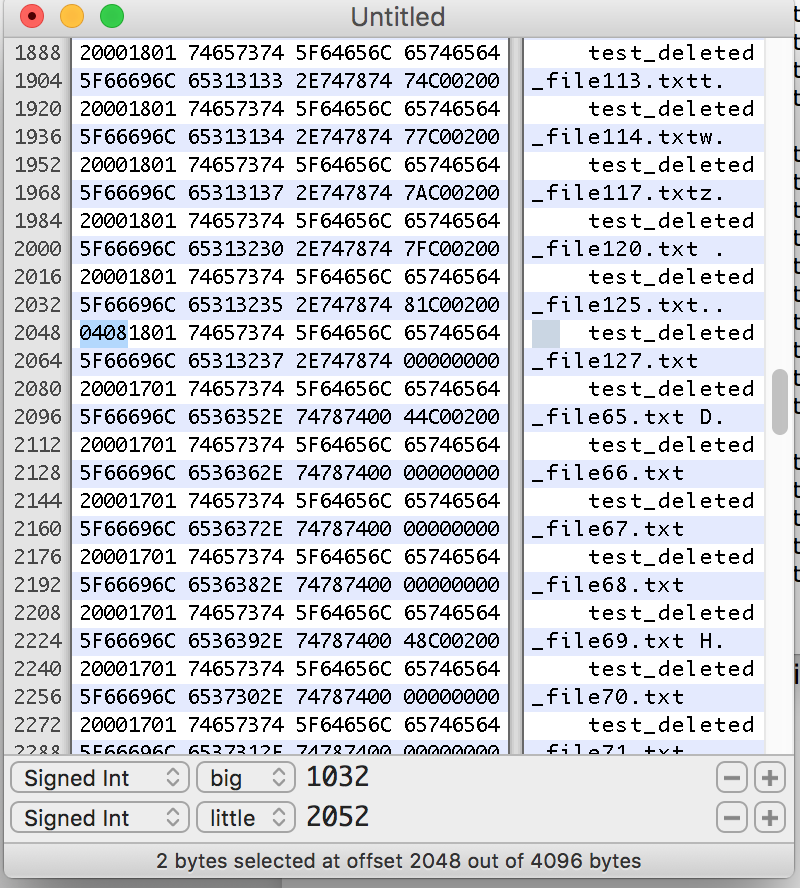


Figure 5 second block contents after exteeding

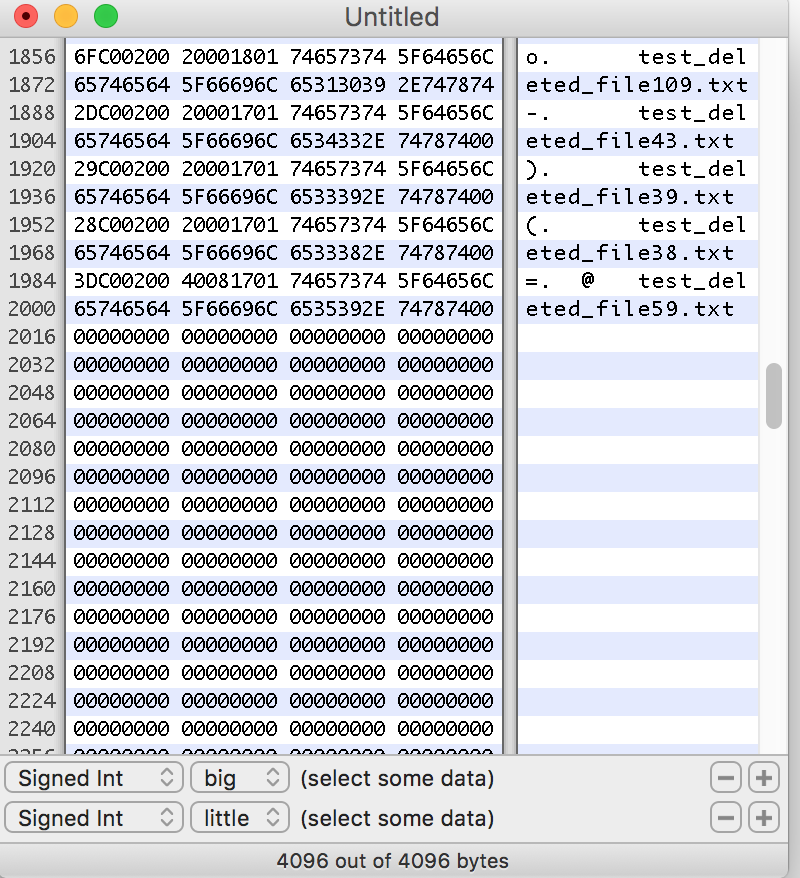


Figure 6 third block contents after exteeding

But as we can see, in the second block, in the about half of the block, the rec\_length of this entry is 1032, and after it, the following entry’s inode number is 0.

Because of this property, we will print files that still exist(because according to property deleted files is behind the entry that real length is not equal to the rec\_length, our program may print all files in block one).

**How to slove it:**

We may hold a existing files storage and “maybe” deleted files storage, and to get the real deleted files, we can use maybe deleted files subtract existing fils.

1. About inode number is equal to 0

There are two conditions that inode number is 0:

1. Delete the first entry that not in first block

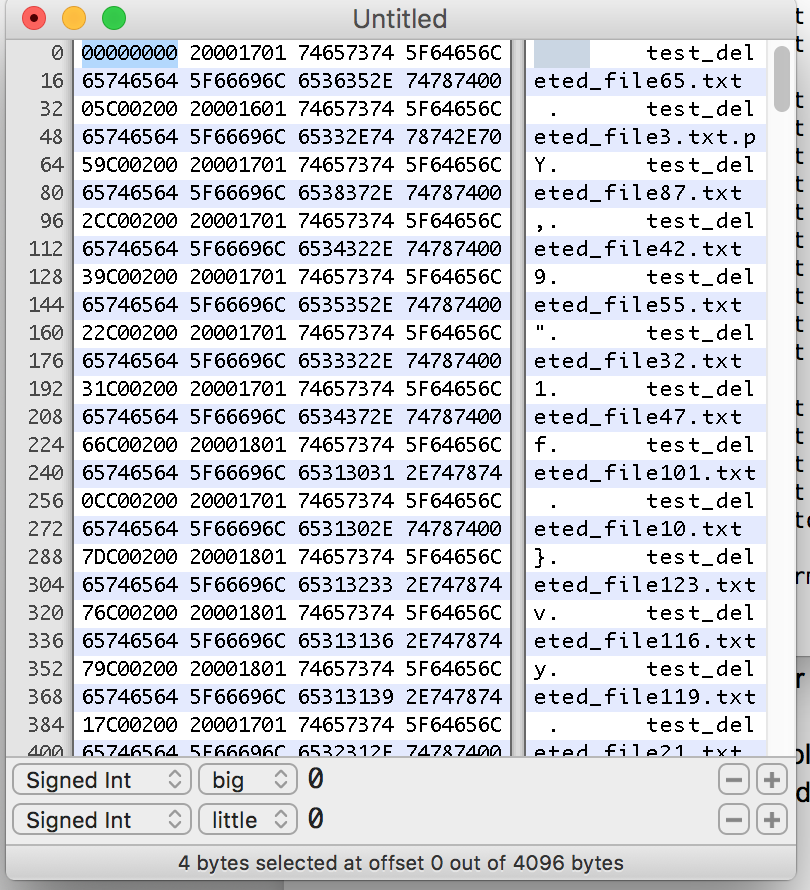


Figure 7 before deleting 65 Figure 8 after deleting 65

1. As shown before, in block 2, the remaining half entries inode number is 0.

**How to solve it:**

If inode number is 0, and it is not in deleted region, add it to maybe deleted storage, if inode number is 0, but it is in deleted region, do nothing.

1. Indirect block

If there are so many files in one directory, there would be indirct blocks to store entries.

**How to solve it:**

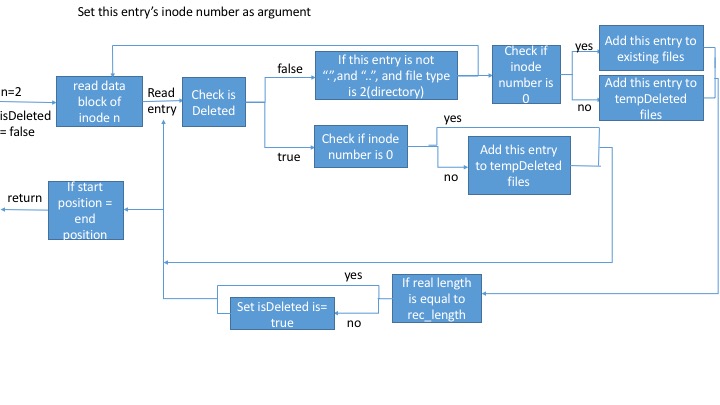
### Algorithm

1. Data structure List deleted files(Xinyi Li)

We need to create a data structure to store deleted files and existing files, moreover, this data structure need to easy to search so that we can use temp deleted files subtract the existing files to get deleted files. Therefore, we use hash table to store the data, and the structure of hash table is as following:

|  |  |
| --- | --- |
| Key(UINT4) | Value(hash table) |
| This dir inode number | |  |  | | --- | --- | | Key(UNIT4) | Value(struct) | | File inode number | |  |  | | --- | --- | | DirEntry(struct) | Path(char) | | |

1. Process



Now we get two hash tables: existingFiles and tempDeletedFiles, use tempDeletedFiles – existingFiles to get the deletedFiles;

***Pseudocode***:

RegionList(InodeNo, isDeleted, endPos, startPos){

If(endPos = startPos){return;}

entry = readEntry(InodeNo, startPos);

realLength = calculateRealLen(Entry);

if(isDeleted == false) {

if(entry.name != “.” && entry.name != “..” && entry.type == FILE\_TYPE){

RegionList(entry.InodeNo, false, endPos, startPos + entry.rec\_Length);

}

if(entry.InodeNo == 0){addToTempDeletedFiles(entry);}

else{addToExistingFiles(entry);}

if(entry.recLength != realLength){

RegionList(InodeNo, true, endPos, startPos + realLength);

}

}

else{

/\*isDeleted == true\*/

if(entry.inodeNo != 0){

addToTempDeletedFiles(entry);

}

if(startPos + realLength == endPos){

RegionList(entry.inodeNo,true,endPos,startPos+realLength)

}

else {

RegionList(entry.inodeNo,false,endPos,startPos+realLength)

}

}

}

1. List deleted files in deleted directories(Zhenkai Huang)

Group descriptor needs to get first level deleted directories’ name from inode list because group descriptor only scans for deleted directories and the first level directories still has files/directories remain. Group descriptor get the first level directory name and combine to the final deleted list. So make sure to run inode list command, i.e.: “./inode/dev/sdxx” and “list”. It will generate a deleted\_files.csv file under pre-defined DELETED\_FILES\_CSV\_PATH value in cmndef.h. After that, run group descriptor to get full deleted files/directories list. There are two way to run it. Both under filerecovery and gbd directory can execute.

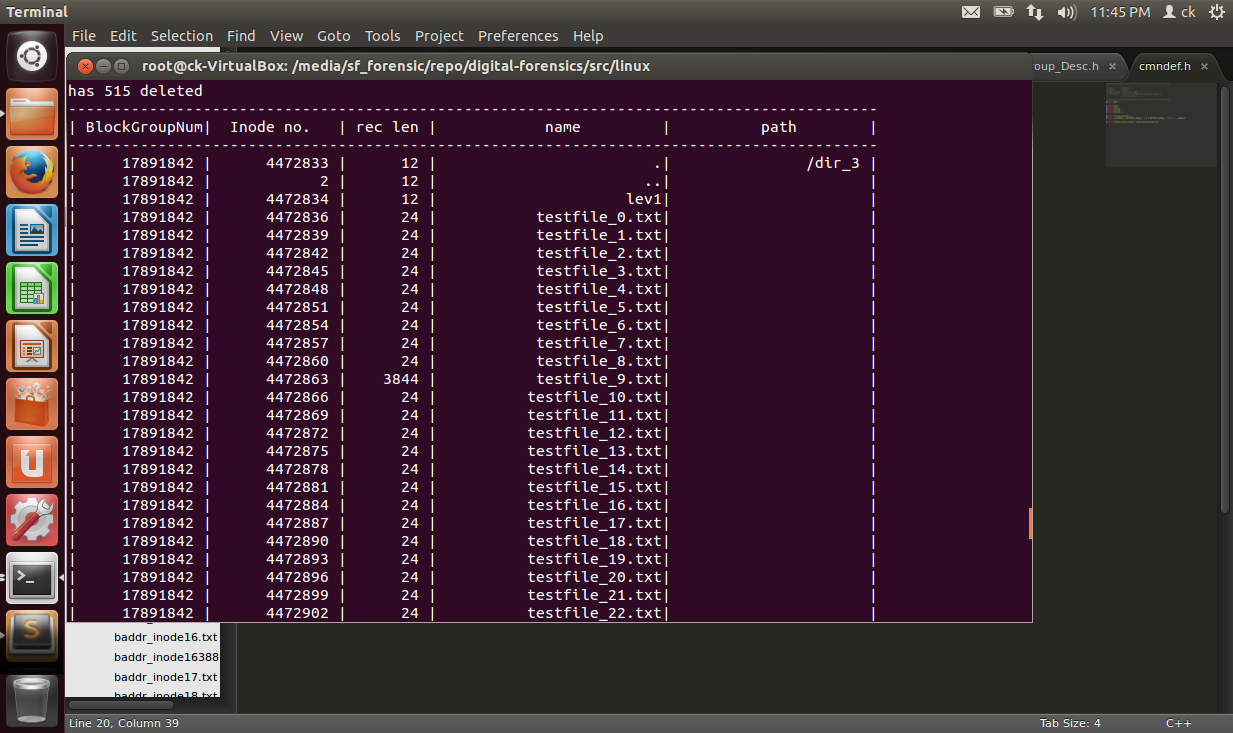
From filerecovery, make sure compile filerecovery by command “make filerecovery” and generated the executable filerecovery file under bin directory. Then by execute   
“./filerecovery /dev/sdxx 0“

to scan whole device disk as already-specified first argument of “bgdReadDeletedDir” function as a negative number. The result will show both on standard output(screen) and store as a file named “deletedDirectory.csv” in under same folder of executable file. The other way to execute group descriptor program is by execute group descriptor as two steps:

1. make gbd
2. ./gbd /dev/sdxx -1

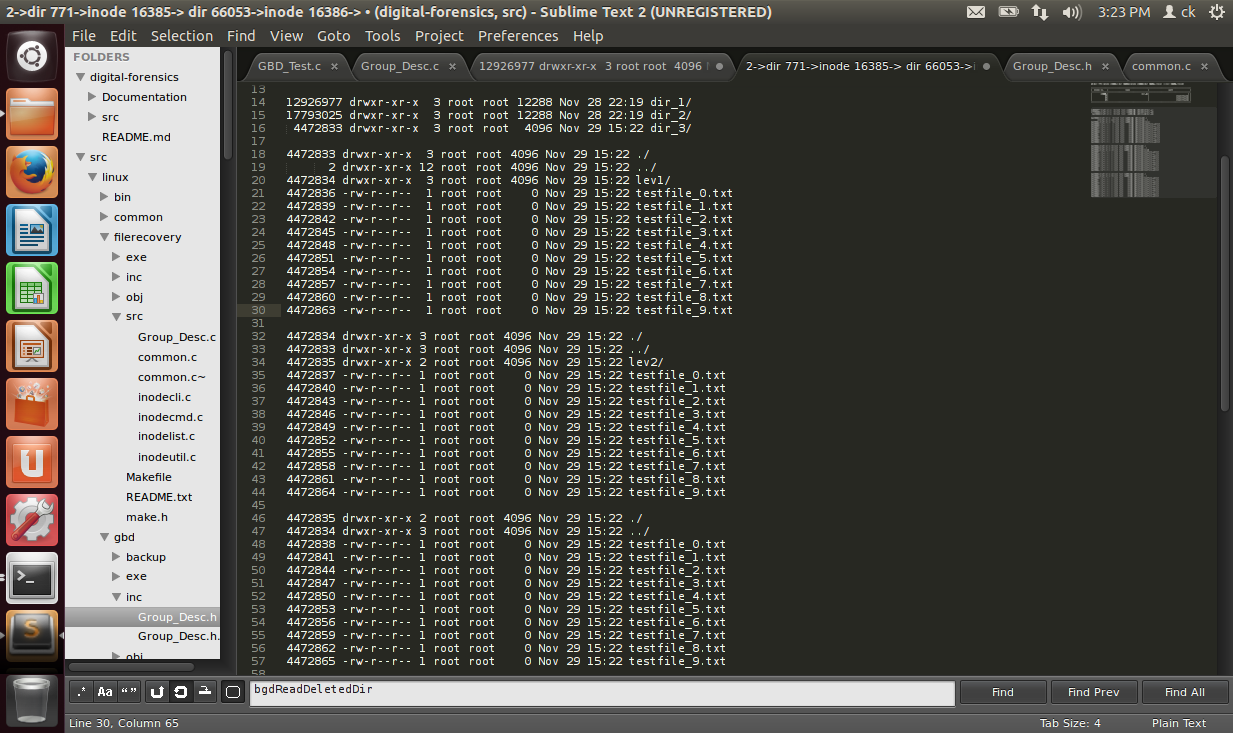
The second argument -1 means to scan whole device, which can be replaced as block group number. If the second argument is block group number, the group descriptor will only scan the specified block group. With group descriptor program, it will generate csv file by default. It can be disabled in GBD\_Test.c. There are five columns per row in csv file, they are, respectively:

Block group number, inode number, rec length, entry name, and path from root



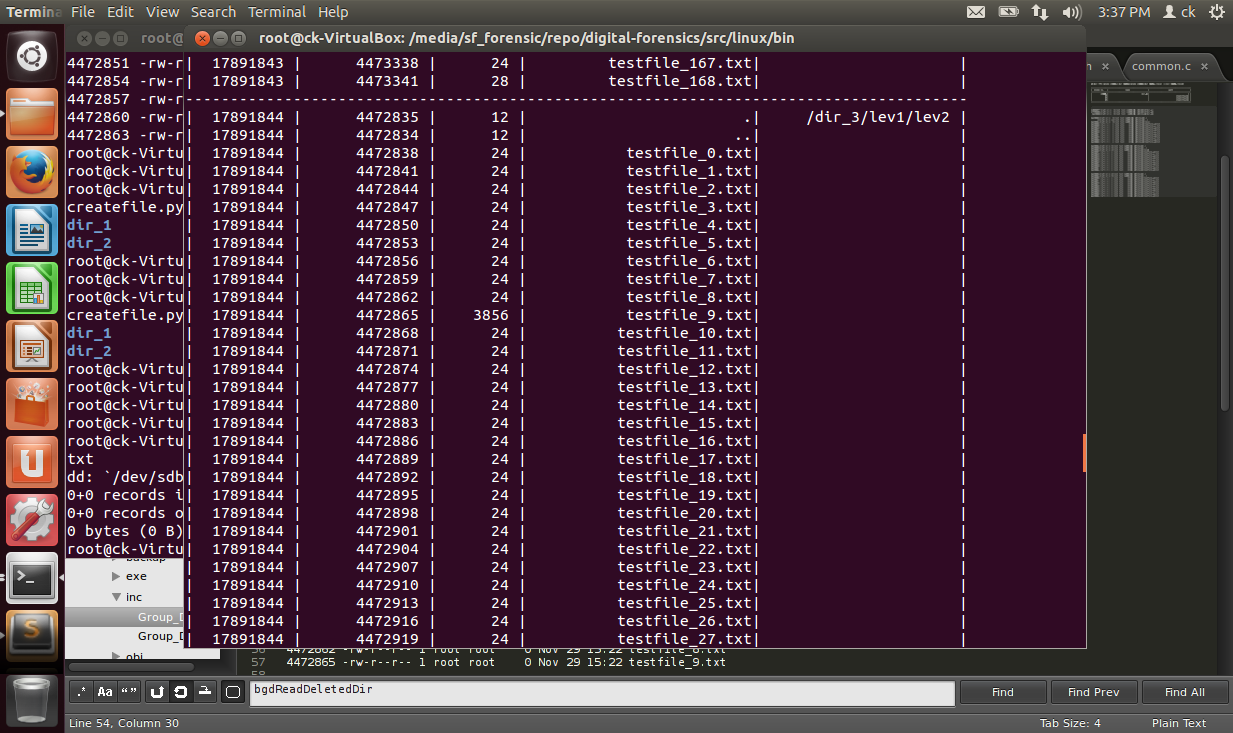
*Figure 1. List of all deleted files with inode numbers.*

For directories with file less than one block, one can execute “ll –i” to get inode number of listed files and directories. The transformation between inode number => block group number is divide inode number by number of inode per block group, i.e.:8192 in my case.



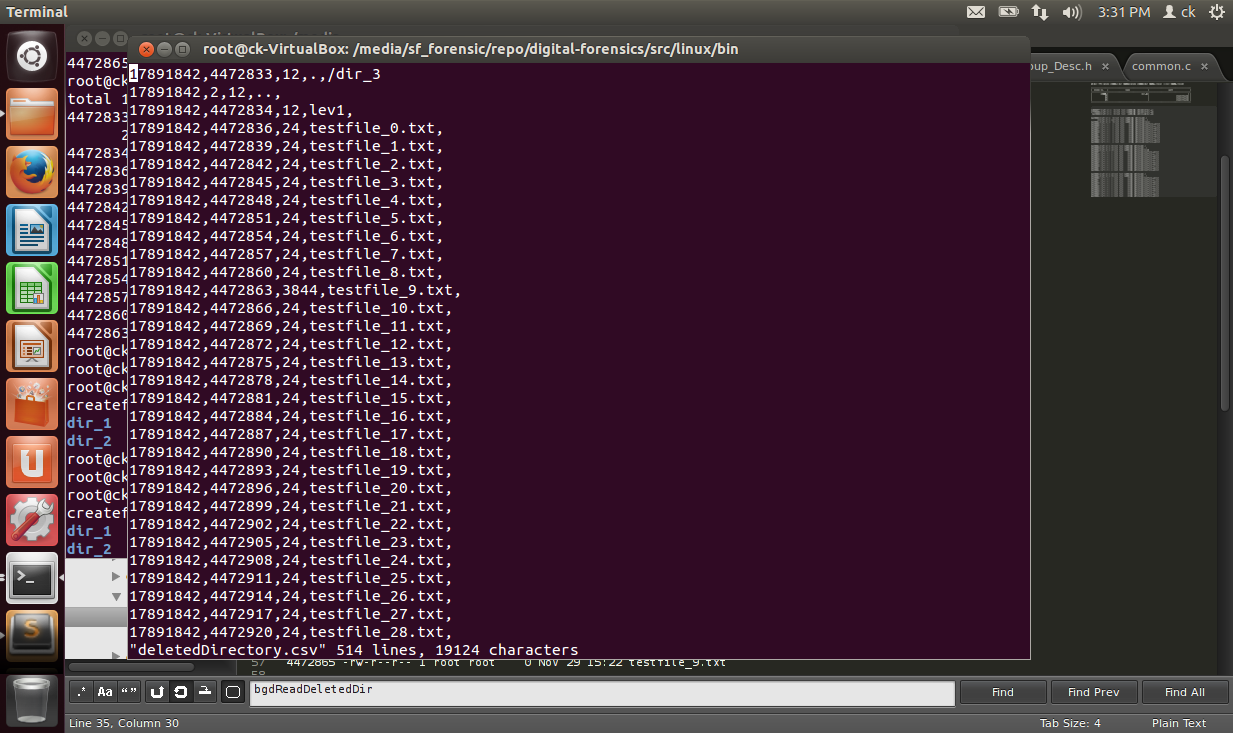
*Figure 2. List of all deleted files with inode numbers.*

Whenever the files or directories in one block exceeds block size, the original data block will be splitted into three data blocks. The first/original data block with file larger than one block, the group descriptor can list files and directories but may not contain all of them as figure 3



*Figure 3. Deleted files and directories from gbd.*  


*Figure 4. Block contents of deleted data block. The deleted files are there in data block and instead of deleted directories the marked region shows some zeros.*



*Figure 5. An example of part of deleted csv file.*

Transformation between block number and block group number is divide block number by block number per group, which is 32768 in my case. Both the inode/block number to block group number transformation can ignore the fraction. If you delete files or directories for a test, one very important thing is remember to unmount the device before use inode/gbd function. For somehow, it causes the data block be written back after unmount the device. Example in figure 5.

Currently reserved NUM\_RESERVED\_ENTRIES in Group\_Desc.h with 10000 entries of ext3\_deleted\_dir\_entry data type for deleted entries array. It is easy to overflow the heap memory with large device if not make a constraint on it.

One need to know that group descriptor uses absolute value to get the bit-offset data block address that the integer value variable may not sufficient to store it, so group descriptor uses long long variable to store the absolute bit offset value. It is important not to overflow.



*Figure 6. Compared with figure 4. The removed data block but not unmounts yet and has some problem in the marked line.*

Because this deleted directory program in group descriptor and file recovery only scans for deleted directories, make sure the deleted files are within the directory.

# **D. NTFS FILE SYSTEM**

# **E. FILE TYPE EXTENSIONS**

This section describes the various file types, eg jpeg, pdf etc. Each file type has a certain format. The sections below will describe the format of each file type. This will help in recovering the files.

Note: I have added the following extensions so far. Feel free to volunteer and add other well known extensions.

Please refer to the following link to get the file signatures (magic numbers for all the file extensions).

<https://en.wikipedia.org/wiki/List_of_file_signatures>

### 1. Format of jpeg extension

JPEG (Joint Photographic Experts Group) is a commonly used method of lossy compression for digital images, mostly for those images produced by digital photography. The degree of compression can be adjusted, allowing a tradeoff between storage size and image quality. JPEG compression is used in a number of image file formats. JPEG/Exif is the most common image format used by digital cameras and other image capture devices. JPEG/JFIF, it is the most common format for storing and transmitting photographic images on the Internet.

JPEG files (compressed images) start with an image marker which always contains the marker code hex values **FF D8 FF**. These are the first 3 bytes of the file.

It does not have a length of the file embedded, thus we need to find JPEG trailer, which is **FF D9**. These are the last 2 bytes of the file.  
  
The following filename extensions also belong to JPEG: .jpg, .jpe, .jif, .jfif, .jfi.

**File Layout:**

A JPEG file is partitioned by markers. Each marker is immediately preceded by an all 1 byte (0xff). Although there are more markers, We will discuss the following markers:

|  |  |  |
| --- | --- | --- |
| **Marker Name** | **Marker Identifier** | **Description** |
| **SOI** | 0xD8 | Start of Image |
| **APP0** | oxE0 | JFIF application segment |
| **Appn** | oxE1-oxEF | Other APP segements |
| **DQT** | oxDB | Quantization Table |
| **SOF0** | oxC0 | Start of Frame |
| **DHT** | oxC4 | Huffman Table |
| **SOS** | oxDA | Start of scan |
| **EOI** | oxD9 | End ofImage |

**Screenshots:**

**MARKER NAME – HEX VALUE (DESCRIPTION) – LENGTH**

Legend of the image: (Byte offset – Data Bytes – ASCII Equivalent of data bytes)

SOI – FFD8 (First two bytes of the image/first block)

APP0 – FFE0 (APP0 marker for JFIF header) - 16

Screen%20Shot%202016-07-16%20at%2010

APP1 – FFE1 (APP1 marker for EXIF header) - 308

Screen%20Shot%202016-07-16%20at%2010

Another APP1 – FFE1 (Another APP1 marker with other information) – 2726

Screen%20Shot%202016-07-16%20at%2011

APP4 – FFE4 (APP4 marker with additional information) - 96

Screen%20Shot%202016-07-16%20at%2011

SOF0 – FFC0 (Start of frame header) - 17

Screen%20Shot%202016-07-16%20at%2011

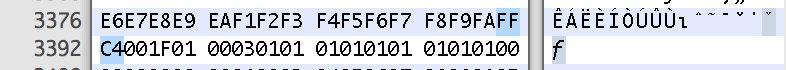
DHT – FFC4 (Huffman table - 00) – 31

Screen%20Shot%202016-07-16%20at%2011

DHT – FFC4 (Huffman table - 10) - 181

Screen%20Shot%202016-07-16%20at%2011

DHT – FFC4 (Huffman table - 01) - 31



DHT – FFC4 (Huffman table - 11) - 181

Screen%20Shot%202016-07-16%20at%2011

DQT – FFDB (Quantization Table) - 67

Screen%20Shot%202016-07-16%20at%2011

DQT – FFDB (Quantization Table - continue) - 67

Screen%20Shot%202016-07-16%20at%2011

DRI – FFDD (Define Restart Interval) – 4

SOS - FFDA (Start of scan) – 12 – 6+2\*3 (number of components in scan)

Screen%20Shot%202016-07-16%20at%2011

Image Data from here

EOI – FFD9 (End of Image)

Screen%20Shot%202016-07-16%20at%2011

**JPEG Header Format:**

Strictly speaking, JPEG files do not have formal headers but we can use the following relevant information from the file’s start of frame segment. There are many header formats available and out of them JFIF and EXIF are the mostly used headers.  
  
1) SOI is the start of image marker and always contains the marker code values FFh D8h.

2) APP0 is the Application marker and always contains the marker code values FFh E0h.

3) Length is the size of the JFIF (APP0) marker segment, including the size of the Length field itself and any thumbnail data contained in the APP0 segment. Because of this, the value of Length equals 16 + 3 \* XThumbnail \* YThumbnail.

4) Identifier contains the values 4Ah 46h 49h 46h 00h (JFIF) and is used to identify the code stream as conforming to the JFIF specification.

5) Version identifies the version of the JFIF specification, with the first byte containing the major revision number and the second byte containing the minor revision number. For version 1.02, the values of the Version field are 01h 02h; older files contain 01h 00h or 01h 01h.  
  
typedefstruct \_JFIFHeader

{

BYTE SOI[2]; /\* 00h Start of Image Marker \*/

BYTE APP0[2]; /\* 02h Application Use Marker \*/

BYTE Length[2]; /\* 04h Length of APP0 Field \*/

BYTE Identifier[5]; /\* 06h "JFIF" (zero terminated) Id String \*/

BYTE Version[2]; /\* 07h JFIF Format Revision \*/

BYTE Units; /\* 09h Units used for Resolution \*/

BYTE Xdensity[2]; /\* 0Ah Horizontal Resolution \*/

BYTE Ydensity[2]; /\* 0Ch Vertical Resolution \*/

BYTE XThumbnail; /\* 0Eh Horizontal Pixel Count \*/

BYTE YThumbnail; /\* 0Fh Vertical Pixel Count \*/

} JFIFHEAD;

**Recovering Technique:**

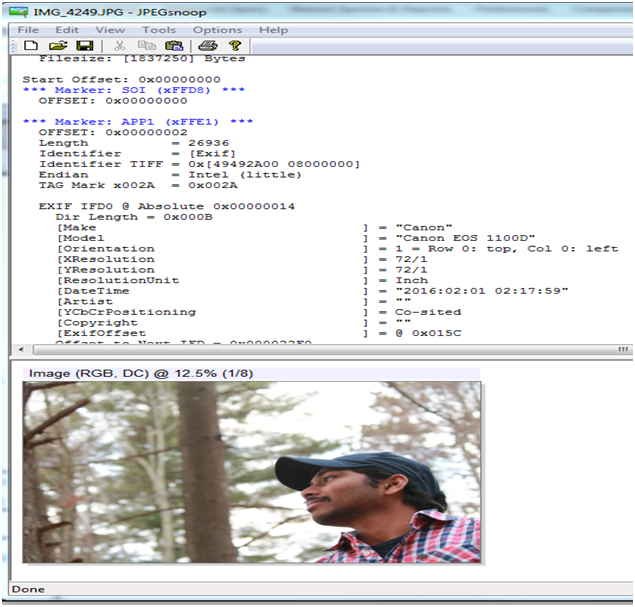
A JPEG file contains an ordered sequence of markers, parameters, and entropyencoded segments that are spread over multiple blocks. For differentiating JPEG file data from other files data one can only rely on the markers.

First all data blocks of the storage device are scanned for known file markers. Each marker is two bytes in length with the first byte always having 0xFF value and the second byte containing a code that specifies the marker type. Although there are many markers, few can be rather easily distinguished. Among these, the most important one is the start of image marker which will be located at the beginning of a block and mark the beginning of a file. The end of file marker, which will be located in the last file block, and the restart markers, that will appear almost periodically to establish synchronization, are the other useful ones.

In the second phase, data extracted from the first file block, containing the start of image marker, is combined together with the data extracted from the consecutive data blocks. Every time a new data block is merged with the previously merged file blocks a decoding operation is performed on the resulting data to test whether a fragmentation point is reached. In Garfinkel (2007), a fragmentation point is deemed to be reached when an error occurs during decoding and the lastly merged block is discarded from the partially recovered file data.

**Verification Tools:**

JPEGsnoop. This is an opensource library which is used to verify the images and gives the metadata after the complete image is constructed. (Collection of data).   
 Please note that this verification tool is available in windows currently but we can use the verification algorithm from the source.



**References:**

1) Garfinkel S. Carving contiguous and fragmented files with fast object validation. In: Proceedings of the 2007 digital forensics research workshop, DFRWS, Pittsburgh, PA, August 2007

### 2. Format of .doc and .ppt

**DOC file type**

|  |
| --- |
| **HEADER** |
| **TEXT** |
| **FORMAT TRAILER** |

1. Start of the file marker

**Header in Hex (Unique for all Microsoft files)**

**CFB header : D0 CF 11 E0 A1 B1 1A E1**



CFB header which shows that it is a microsoft document.

This is unique for all the microsoft documents.

**Word doc subtype: 57 6F 72 64 2E 44 6F 63 75 6D 65 6E 74 2E**



1. End of the file marker

Footer in hex

**57 6F 72 64 2E 44 6F 63 75 6D 65 6E 74 2E**



1. File size in the header

**Offset Bytes Field description**

**----------------------------------------------------------**

00H 4 Word signature 31H BEH 00H 00H

04H 8 Reserved (00H ABH 00H 00H00H

00H 00H00H)

0EH 4 Pointer to End-of-text

(1st character after text)

12H 2 Block pointer to the block containing

the paragraph format

14H 2 Block pointer to the block containing

the footnote table

16H 2 Block pointer to the block containing

the section formats

18H 2 Block pointer to the block containing

the nation table

1AH 2 Block pointer to the block containing

the table of page breaks

1CH 2 Block pointer to the block containing

file manager information

(author, date, and so on)

1EH 66 File name of print format, ASCII string

60H 2 Flag (reserved for Windows Write)

62H 8 Name of the printer driver,

ASCII string

6AH 2 Number of blocks used in the file

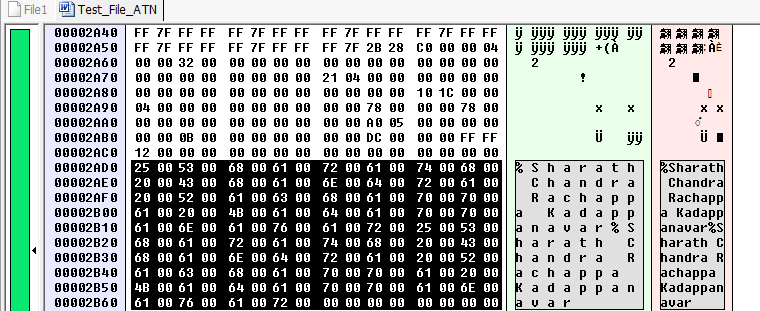
6CH 2 Bit field for corrected text areas

6EH 18 Reserved (in version 4.0 always 00H);

after version 5.0 used for unknown

code

**Author of the file is mentioned at offset 2AD0H**



***at offset 512 or 200 hex is a sub file signature that indicates if it is a word, powerpoint, excel document***

.**PPT file type**

Start of the file marker

* Header in Hex

**CFB header: D0 CF 11 E0 A1 B1 1A E1**

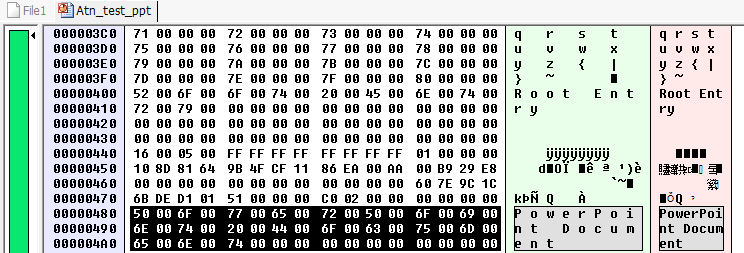


**PPT subtype:**

**PPT Header and Footer of the file**

* **PPT Header in hex (At offset 48h)**

**50 00 6F 00 77 00 65 00 72 00 50 00 6F 00 69 00 6E 00 74 00 20 00 44 00 6F 00 63 00 75 00 6D 00 65 00 6E 00 74**



* PPT footer

**50 00 6F 00 77 00 65 00 72 00 50 00 6F 00 69 00 6E 00 74 00 20 00 44 00 6F 00 63 00 75 00 6D 00 65 00 6E 00 74**

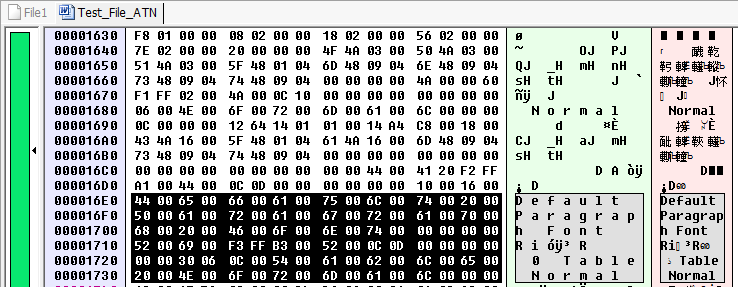
**Text Area which we need to recover:**

**The first byte containing text stored in ASCII format begins at offset 80H (when checked in FlexHex the text area is beginning at A0H).**

This text may extend over several blocks. In the last text block, the area from the last valid text character to the end of the block is undefined. **The end of the text is indicated in the header (offset 0EH)**. If Word stores a blank text window, the text block is omitted, and immediately the format information follows the header.



File formats , like paragraph fonts and , table fonts.



**Refernces:**

[**http://msxnet.org/**](http://msxnet.org/) **: Read about the word header format and the text area formats from here.**

**3. Format of png extension**

**Portable Network Graphics** is a raster graphics file format that supports lossless data compression. PNG was created as an improved, non-patented replacement for Graphics Interchange Format.

**PNG Signature**

The first eight bytes of a PNG datastream always contain the following (decimal) values:

**89 50 4E 47 0D 0A 1A 0A**

We see the corresponding value in Hex Editor, we also see the first 8 bytes marked as shown.



**Chunk layout**



**Length:** A four-byte unsigned integer giving the number of bytes in the chunk's data field. The length counts **only** the data field, **not** itself, the chunk type, or the CRC. Zero is a valid length.

In Hex Editor, we see that png signature is followed by a 4 byte length field.



**Chunk Type:** A sequence of four bytes defining the chunk type. Each byte of a chunk type is restricted to the decimal values 65 to 90 and 97 to 122

For example we see that after the length fields, we have chunk type **'IHDR'** as shown.



**Chunk Data:** The data bytes appropriate to the chunk type, if any. This field can be of zero length.

The below shows data of a small png file.



**CRC:** A four-byte CRC (Cyclic Redundancy Code) calculated on the preceding bytes in the chunk, including the chunk type field and chunk data fields, but **not** including the length field. The CRC can be used to check for corruption of the data.

We have the CRC indicated as below, which is used to check for corruption.



**Chunk Types**

There are two types

1) **Critical Chunk** - Critical chunks are those chunks that are absolutely required in order to successfully decode a PNG image from a PNG datastream.

### IHDR (Image Header) -- First chunk in PNG datastream

The four-byte chunk type field contains the HEX values

49 48 44 52

The **IHDR** chunk shall be the first chunk in the PNG datastream. It contains:

|  |  |
| --- | --- |
| Width | 4 bytes |
| Height | 4 bytes |
| Bit depth | 1 byte |
| Colour type | 1 byte |
| Compression method | 1 byte |
| Filter method | 1 byte |
| Interlace method | 1 byte |



**PLTE** Palette

The four-byte chunk type field contains the decimal values

80 76 84 69

The **PLTE** chunk contains from 1 to 256 palette entries, each a three-byte series of the form:

|  |  |
| --- | --- |
| Red | 1 byte |
| Green | 1 byte |
| Blue | 1 byte |

The number of entries is determined from the chunk length. A chunk length not divisible by 3 is an error.



**IDAT** Image data

The four-byte chunk type field contains the HEX values

49 44 41 54



The **IDAT** chunk contains the actual image data which is the output stream of the compression algorithm.

There may be multiple **IDAT** chunks; if so, they shall appear consecutively with no other intervening chunks.

**IEND** Image trailer

The four-byte chunk type field contains the HEX values

49 45 4E 44

The **IEND** chunk marks the end of the PNG datastream. The chunk's data field is empty.



2 ) **Ancillary chunks**

Since there are many ancillary chunks, only the important ones are mentioned here.

**gAMA Image gamma**

The four-byte chunk type field contains the decimal values

103 65 77 65

The **gAMA** chunk specifies the relationship between the image samples and the desired display output intensity



**bKCD Background Color**

The four-byte chunk type field contains the HEX values

62 4B 47 44

The **bKGD** chunk specifies a default background colour to present the image against 

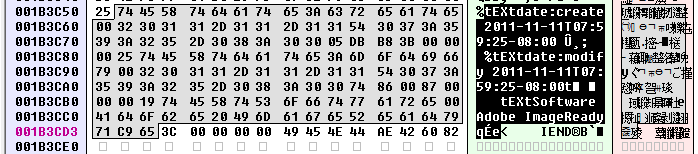
**tEXt Textual data**

The four-byte chunk type field contains the decimal values

116 69 88 116

Each **tEXt** chunk contains a keyword and a text string, in the format:

We see the information of the software which was used to create the file, the date it was modified and the date it was created.



**Recovery Process -**

To start the recovery process,

1) Build the project as per the instructions in 'Read Me File'

2) Take a storage device (disk / pen drive) and format it to 'Ext3' filesystem.

3) Place some png files in any directory and delete them using 'rm'.

4) Now navigate to $(PROJECT\_DIR)\linux\bin and run the below

5) sudo 'program\_name' 'disk\_location' '2' (option 2 is to search for PNG files only) Example usage (sudo ./filerecovery /dev/sdc1 2)

6) We see that we have successfully located deleted PNG data files as below

**Found PNG FILE file at offset 2439637301 and block address 341224**

**Found PNG FILE file at offset 2439641302 and block address 341225**

**Found PNG FILE file at offset 1432734631 and block address 411236**

**Found PNG FILE file at offset 1432738632 and block address 411237**

### 4. Format of odt extension

The **Open Document Format for Office Applications** (**ODF**), also known as **OpenDocument**, is an XML-based file format for spreadsheets, charts, presentations and word processing documents (ODT file). It was developed with the aim of providing an open, XML-based file format specification for office applications.

ODT File identification

ODT File can be identified by the file signature of size 4 bytes that is present in the first block at offset 0. Specifically the file signature is “**50 4B 03 04**” which is the ASCII equivalent of **PK••**.

While ODT file does have this file signature, the same however is used even by other formats such as zip file. This similarity of the two formats does extend to other fields as well.

Screen%20Shot%202016-07-14%20at%2010

File Structure

ODT files are stored using the the following XML structure

|-- META-INF

| `-- manifest.xml

|-- Thumbnails

| `-- thumbnail.png

|-- content.xml

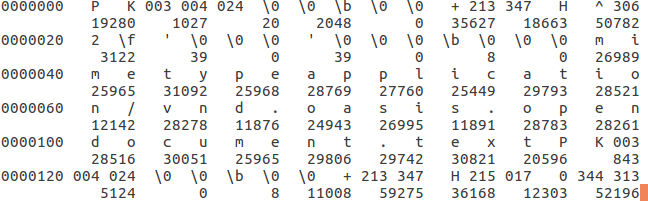
|-- meta.xml

|-- mimetype

|-- settings.xml

`-- styles.xml

The mimetype for the ODT file specifically can be found in the first block of the file at the offset 38 containing the value “mimetypeapplication/vnd.oasis.open.document.text”. This along with the file signature can be used to identify the ODT file.

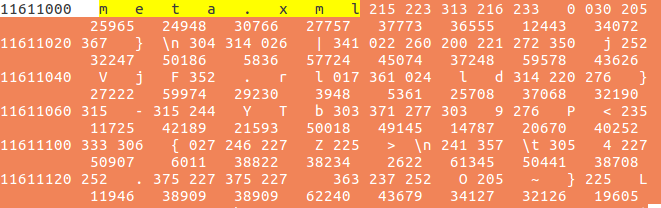


Other than the META-INF and meta.xml, all other XMLs contain only document relevant and rendering of the file data.

File metadata

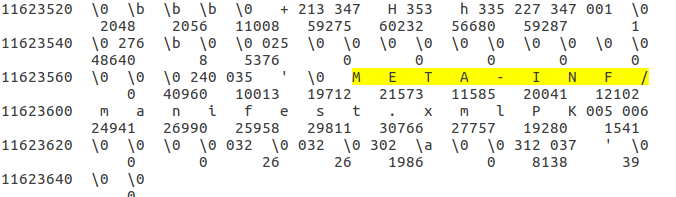
The meta.xml contains file related metadata that can be of use to recover the file. However, the meta.xml does not have a particular offset where it can be found, as it depends on the size of the content.xml that occurs before the same.

On encountering the meta.xml we need to access the elements under the tag “<meta:document-statistic>” to find meta data such as word-count, page count and character count. More on this can be found at [meta.xml information](http://books.evc-cit.info/odbook/ch02.html" \l "meta-statistics-table)



File sanity verification

META-INF is always present at the end of the file before the null character. Manifest.xml that is found here can be used to validated if all the XML tags found here have been recovered successfully.



Since **there is no linking of the data blocks**, the file systems default behavior of allocating the blocks contiguously is to be relied upon for recovery. We can however, make use of the XML file properties to check if the block recovered does indeed close the tags opened in the previously encountered blocks.

### 5. Format of mpg extension

### 6. Format of flv extension

Flash Video is a container file format used to deliver digital video content (e.g., TV shows, movies, etc.) over the Internet using Adobe Flash Player version 6 and newer. Flash Video content may also be embedded within SWF files. There are two different video file formats known as Flash Video: FLV and F4V. The audio and video data within FLV files are encoded in the same manner as they are within SWF files. Both formats are supported in Adobe Flash Player and developed by Adobe Systems.

**Header**

FLV files start with a standard header which is shown below:

**Field Data Type Default Details**

Signature byte[3] “FLV” Always “FLV”

Version uint8 1 Only 0x01 is valid

Flags uint8 bitmask 0x05 Bitmask: 0x04 is audio, 0x01 is video (so 0x05 is audio+video)

Header Size uint32\_be 9 Used to skip a newer expanded header

### 7. Format of gif extension

The Graphics Interchange Format (GIF) is a bitmap image format that was introduced by CompuServe in 1987 and has since come into widespread usage on the World Wide Web due to its wide support and portability.

The format supports up to 8 bits per pixel for each image, allowing a single image to reference its own palette of up to 256 different colors chosen from the 24-bit RGB color space. It also supports animations and allows a separate palette of up to 256 colors for each frame. These palette limitations make the GIF format less suitable for reproducing color photographs and other images with continuous color, but it is well-suited for simpler images such as graphics or logos with solid areas of color.

GIF files (compressed images) start with an image marker which always contains the marker code hex values **47 49 46 (GIF)**. These are the first 3 bytes of the file.

Immedietly after the GIF header format the version of GIF will also be mentioned. For example, **38 39 61** (89a).

**File Layout:**

A GIF file is partitioned by markers. Although there are more markers, We will discuss the following markers:

|  |  |  |
| --- | --- | --- |
| **Marker Name** | **Marker Identifier** | **Description** |
| **SOI** | 0x 47 49 46 | Start of Image |
| **VER** | 0x 38 39 61 | GIF file version |
| **LSD** | 0xF7 | Logical Screen Descriptor |
| **EOI** | Ox3B | End of Image |

**Screenshots:**

SOI marker. This is the first 3 bytes of the file / first block

Screen%20Shot%202016-07-30%20at%205

VER marker. This is immediately followed after SOI marker

Screen%20Shot%202016-07-30%20at%205

LSD marker (Logical Screen Descriptor)

Screen%20Shot%202016-07-30%20at%205

EOI marker (End of image)

Screen%20Shot%202016-07-30%20at%205

**Recovering Technique:**

A GIF file contains an ordered sequence of markers, parameters, and entropyencoded segments that are spread over multiple blocks. For differentiating GIF file data from other files data one can only rely on the markers.

First all data blocks of the storage device are scanned for known file markers. The most important one is the start of image marker which will be located at the beginning of a block and mark the beginning of a file. The end of file marker, which will be located in the last file block, and the restart markers, that will appear almost periodically to establish synchronization, are the other useful ones.

In the second phase, data extracted from the first file block, containing the start of image marker, is combined together with the data extracted from the consecutive data blocks. Every time a new data block is merged with the previously merged file blocks a decoding operation is performed on the resulting data to test whether a fragmentation point is reached.

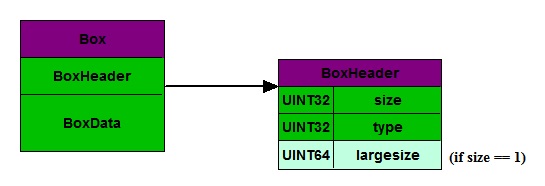
### 8. Format of tiff extension

### 9. Format of doc extension

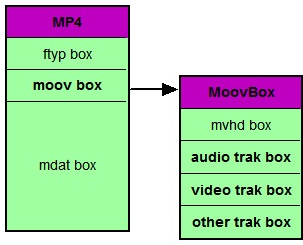
### 10. Format of mp4 extension

**MPEG-4 Part 14** or **MP4** is a digital multimedia container format most commonly used to store video and audio, but can also be used to store other data such as subtitles and still images. Like most modern container formats, it allows streaming over the Internet. The only official filename extension for MPEG-4 Part 14 files is **.mp4**, but many have other extensions, most commonly **.m4a** and **.m4p**.

Data inside mp4 files are contained in boxes (or atom in earlier version). Each box has the same structure as follows. The header of the box indicates its size and type. When the size set to 1, it means the it is a large box, the actuall size will be stored in the largesize section. If the size is 0, it means it is the last box of the file.



Some boxes are inside another box and create a nested structure.



In order to play a mp4 file, three most important boxes have to exist, which can be identified in a file as “ftyp”, “moov”, “mdat” respectively.

1. Recover the File Type Box (ftyp)

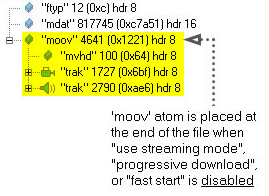
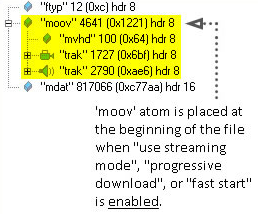
Every mp4 file starts with the ftyp box and can only have one ftyp box. It contains information about file type and compatibility needed for the media player. It has the following structure,



The first 4 byte **00 00 00 1c** is the size of the box, which is 28 bytes. The second 4 bytes **66 74 79 70** is the type of the box, which corresponds to “ftyp”. This box indicates the beginning of the file and is usually very small. It can be easily found by searching **66 74 79 70** on the disk, then check the previous 4 bytes to make sure the value is small.

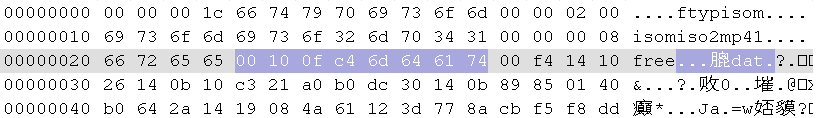
2. Recover the Movie Box (moov)

This box contains metadata of the media file, which is needed to parse the actually media data. Same as ftyp box, only one moov box can exsit in a file. The location of this box depends on whether the streaming mode is enabled or not, as explained in the pictures below.



Therefore, if streaming mode is enabled, the moov box follows immediately after the ftyp box. It is also very small and is therefore likely to be stored in contiguous blocks on the disk. So the recovery is relatively easy in this case.

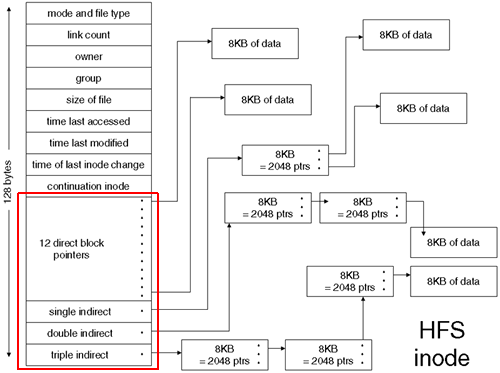
On the other hand, if streaming mode is not enabled, locating the moov box that belongs to the same file will be harder because there may be other mp4 files on the disk as well, which also have their own moov boxes. In this case, in order to find the moov box that belongs to the ftyp box we found earlier, more information are needed.



For example, in the picture above, the box that follows ftyp box is no longer moov box. **00 10 0f c4** indicates the size of this box is 1,052,612 bytesand **6d 64 61 74** means that it is a mdat box. It is called Media Data Box containing the actually media data. It can be as large as giga bytes.

3. Recover the Media Data Box (mdat)

As introduced before, this box stores the actually media data. The details of the data structure in this box are stored in the moov box. Since it is very large, the data are usually not on contiguous blocks on the disk. One way to locate all the data is to look for the inode of the file. The inode may no longer exsit in the file system, but its data may be still on the disk.



Since we have located the beginning of the file by finding the ftyp box, we can use the address of the ftyp box to locate its inode pointers. After that, we can assume that the adjacent blocks are pointers belonging to the same file and try to recover those blocks. The header of the mdat box must have data **6d 64 61 74**, which can be used to verify the correctness of the data.

4. Additional notes

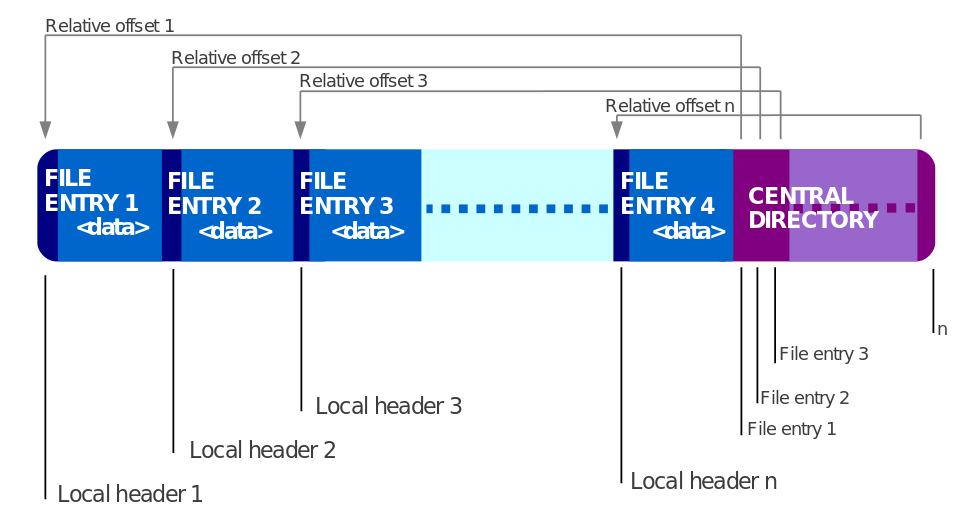
In reality, some data blocks on the disk may be corrupted and is beyond recovery. But as long as we can recovery ftyp box and moov box, it is still possible to partially recover the mp4 file. A mp4 file can still be played even if there are some corruption on its media data. In addition, the moov box contains information of different video and audio tracks of the file. If it is not possible to recovery the entire mp4 file, we can try to recover individual tracks. Therefore, the success of recoverying a mp4 file depends on recovering its metadata.

**11. Format of .zip files:**

.ZIP files are archives that store multiple files. .ZIP allows contained files to be compressed using many different methods, as well as simply storing a file without compressing itBecause the files in a .ZIP archive are compressed individually it is possible to extract them, or add new ones, without applying compression or decompression to the entire archive. This contrasts with the format of compressed tar files, for which such random-access processing is not easily possible.

A directory is placed at the end of a .ZIP file. This identifies what files are in the .ZIP and identifies where in the .ZIP that file is located. This allows .ZIP readers to load the list of files without reading the entire .ZIP archive

**Structure:**



A .ZIP file is correctly identified by the presence of an end of central directory record which is located at the end of the archive structure in order to allow the easy appending of new files. If the end of central directory record indicates a non-empty archive, the name of each file or directory within the archive should be specified in a central directory entry, along with other metadata about the entry, and an offset into the .ZIP file, pointing to the actual entry data.

Each entry stored in a ZIP archive is introduced by a local file header with information about the file such as the comment, file size and file name, followed by optional "extra" data fields, and then the possibly compressed, possibly encrypted file data. The "Extra" data fields are the key to the extensibility of the .ZIP format. "Extra" fields are exploited to support the ZIP64 format, WinZip-compatible AES encryption, file attributes, and higher-resolution NTFS or Unix file timestamps. Other extensions are possible via the "Extra" field. .ZIP tools are required by the specification to ignore Extra fields they do not recognize.

The .ZIP format uses specific 4-byte "signatures" to denote the various structures in the file. Each file entry is marked by a specific signature which is the **Local file header signature = 0x04034b50** (read as a little-endian number). The end of central directory record is indicated with its specific signature **End of central directory signature = 0x06054b50**, and each entry in the central directory starts with the 4-byte **central directory file header signature 0x02014b50**

There is no BOF or EOF marker in the .ZIP specification. Conventionally the first thing in a .ZIP file is a .ZIP entry, which can be identified easily by its local file header signature. However, this is not necessarily the case, as this not required by the .ZIP specification - most notably, a self-extracting archive will begin with an executable file header.

Tools that correctly read .ZIP archives must scan for the end of central directory record signature, and then, as appropriate, the other, indicated, central directory records. They must not scan for entries from the top of the ZIP file, because only the central directory specifies where a file chunk starts. Scanning could lead to false positives, as the format does not forbid other data to be between chunks, nor file data streams from containing such signatures. However, tools that attempt to recover data from damaged .ZIP archives will most likely scan the archive for local file header signatures; this is made more difficult by the fact that the compressed size of a file chunk may be stored after the file chunk, making sequential processing difficult.

Most of the signatures end with the short integer 0x4b50, which is stored in little-endian ordering. Viewed as an ASCII string this reads "PK", the initials of the inventor Phil Katz. Thus, when a .ZIP file is viewed in a text editor the first two bytes of the file are usually "PK". (DOS, OS/2 and Windows self-extracting ZIPs have an EXE before the ZIP so start with "MZ).

**End of Central Directory Signature:**

****

**Local Header Signature:**



**Central Directory Signature:**

****

**12. Format of .pdf files:**

A PDF file is a 7-bit ASCII file, except for certain elements that may have binary content. A PDF file starts with a header containing the magic number and the version of the format such as %PDF-1.7. The format is a subset of a COS ("Carousel" Object Structure) format. A COS tree file consists primarily of objects, of which there are eight types:

Boolean values, representing true or false

Numbers

Strings, enclosed within parentheses ((...)), may contain 8-bit characters.

Names, starting with a forward slash (/)

Arrays, ordered collections of objects enclosed within square brackets ([...])

Dictionaries, collections of objects indexed by Names enclosed within double pointy brackets (<<...>>)

Streams, usually containing large amounts of data, which can be compressed and binary

The null object

Furthermore, there may be comments, introduced with the percent sign (%). Comments may contain 8-bit characters.

Objects may be either direct (embedded in another object) or indirect. Indirect objects are numbered with an object number and a generation number and defined between the obj and endobj keywords. An index table, also called the cross-reference table and marked with the xref keyword, follows the main body and gives the byte offset of each indirect object from the start of the file. This design allows for efficient random access to the objects in the file, and also allows for small changes to be made without rewriting the entire file (incremental update). Beginning with PDF version 1.5, indirect objects may also be located in special streams known as object streams. This technique reduces the size of files that have large numbers of small indirect objects and is especially useful for Tagged PDF.

At the end of a PDF file is a trailer introduced with the trailer keyword. It contains

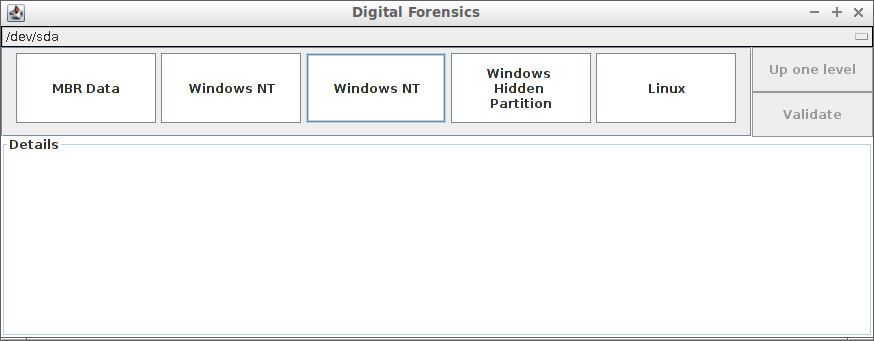
1. a dictionary
2. an offset to the start of the cross-reference table (the table starting with the xref keyword)
3. and the %%EOF end-of-file marker.

The dictionary contains

* a reference to the root object of the tree structure, also known as the catalog
* the count of indirect objects in the cross-reference table
* and other optional information.

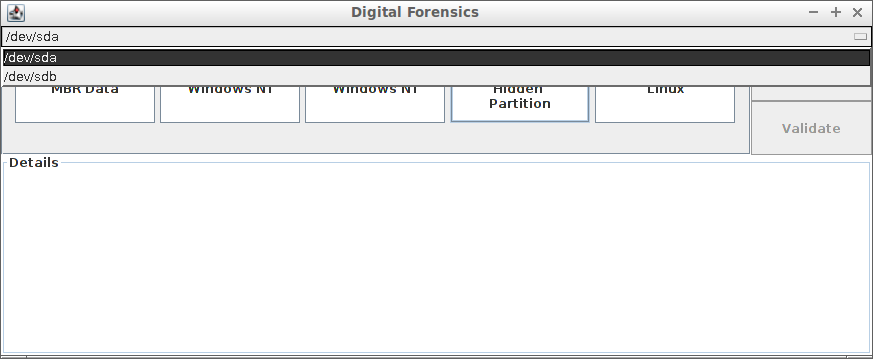
There are two layouts to the PDF files: non-linear (not "optimized") and linear ("optimized"). Non-linear PDF files consume less disk space than their linear counterparts, though they are slower to access because portions of the data required to assemble pages of the document are scattered throughout the PDF file. Linear PDF files (also called "optimized" or "web optimized" PDF files) are constructed in a manner that enables them to be read in a Web browser plugin without waiting for the entire file to download, since they are written to disk in a linear (as in page order) fashion.

# **F. GUI (Vikram)**

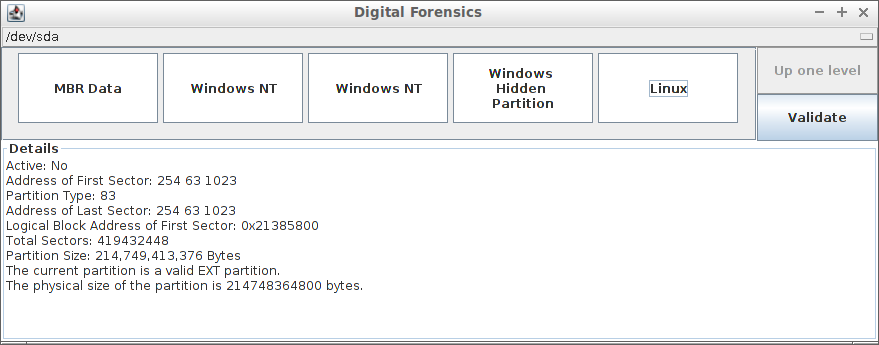
We have developed a GUI tool for displaying the information and to recover the data from a disk drive. There are two panels in GUI. The top panel corresponds to the selections available for use and the bottom part displays the details of the selected structure. For instance, if we have selected a partition inside the disk, the Details panel will be showing the information about the partition currently selected while the selection panel will show the different partitions of the disk. 

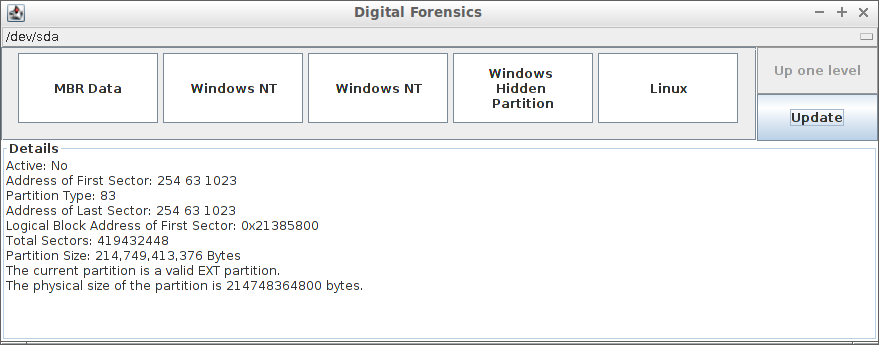
User can select or deselect the current option by single clicking the icon. Selection will display the information in Details panel and Deselecting the option will display the information of the parent structure in the Details panel.

The first screen shown on loading the software will be the layout of the default hard-drive or /dev/sda. This can be changed by the selection panel at top of the screen where the user can select the disk he prefers. This can be done at any point of time during the life of the software. This is shown in the figure below.

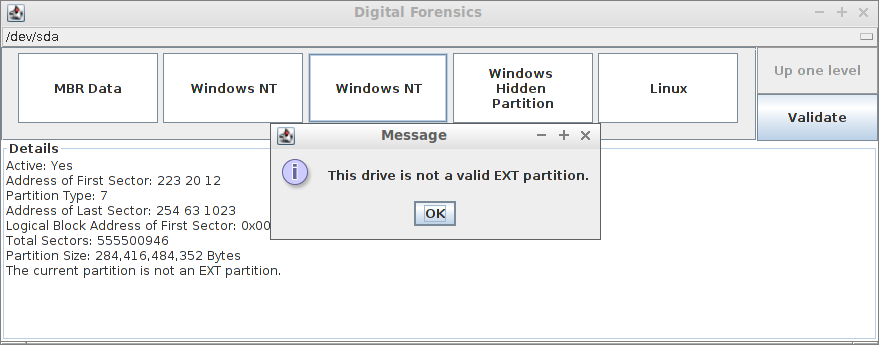


As soon as the user has selected one disk, the different partitions present on the disk are loaded. A single click on one of the partitions displays the information of this partition and also lets us know whether the partition is a valid EXT drive. In case it is a valid partition, we also display the physical size of the system. We can also click on the validate button for confirmation. After we validate the partition, the validate button changes to “Update” so that we can update the information in the partition itself. We can only update the information after it has been correctly validated from the GUI.

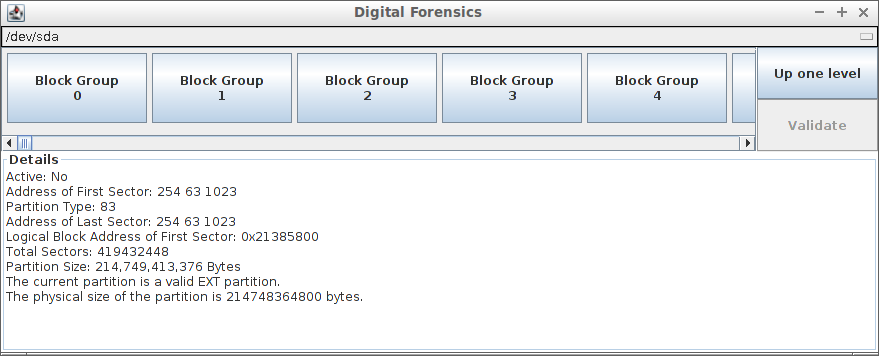




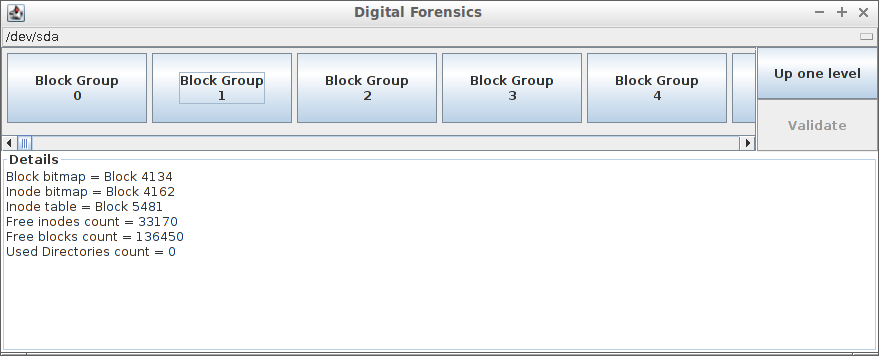
In case it is not a valid EXT partition, the user will be shown a popup about the same when he clicks on Validate button or when he tries to enter the partition by double-clicking.



These partitions, when double-clicked, will display the different block groups present inside the partition.



Each block-group provides information on where the different metadata blocks are located. It also covers the information on the count of used directories, free inodes and free blocks available in the block-group.



Each block-group contains many blocks present inside it. These can be displayed when we open the block-group. These blocks can be either metadata or normal data blocks. The description is listed as the name of the block.



(Vikram – end)

Nilesh-start

**Improvement of block bit map algorithm**

Block bit map algorithm scans the complete disk and finds out the classification of the blocks of disk into address block, data block, used block etc.

Existing algorithm to classify the blocks makes one pass through complete disk for each block type classification. It means it makes multiple passes through disk to create a block bit map.

Multiple passes through the disk caused lot of I/O operation and makes algorithm very slow.

This will induce much more time for large disks. While analyzing the code and file system meta-data, we came across a conclusion that this task can be done in single pass only. It means, each block of disk will be read only once from disk and analyzed.

As each block corresponds to only one type, this can be achieved in single pass. We made these changes. This change improved the run time of algorithm.

**Reducing the search space of data blocks**

Once we get block bit map, we can try to search 12 direct data blocks for particular file. But this will lead to scan large search space. To reduce this search space, we can exclude those data blocks which are not necessary to scan. All those data blocks which are connected via indirect address blocks are not needed. As link between them and indirect blocks are still intact. Hence we mark all these blocks as used. Number of such data blocks will be large and this will reduce search space drastically. It creates a foundation for next file recovery methods which will be dependent on this bitmap.

**Finding indirect Address blocks**

As we can find 12 address blocks from bit map we created in last step, we are remaining with task to find 3 indirect blocks for particular file.

Our, block bit map already has classification of blocks as address blocks. Using this bit map, we apply new algorithm and find out the level of address block.

Consider an address block which points to data block. It means its a First level indirect block.

An address block pointing to first level block will be second level address block.

An address block pointing to second level block will be third level address block.

All of this processing is done in block bit map which is available in memory, which leads to a fast algorithm as there is no disk read involved in this.

****

Above screen shot shows a sample run of code. Program prints all address blocks with their level. Below are the levels of blocks shown as codes.

11 → First Level

12 → Second Level

13 → Third Level

**Future Scope**

As we have block bit map ready with all data blocks and indirect blocks classification. It can be used to design future file recovery related algorithms. Search space has been reduced already which will lead to design efficient recovery algorithm.

**Open Points**

1. Logic to check null block need to be revisited. In Group\_Desc.c, bgdUpdateBlocks, we read 4 bytes and copy them to string array and then check if string length is zero we conclude block to be null. This scenario need to be tested on null blocks . One scenario might be when new disk is bought , it may have null blocks as nothing is written on disk.

2. In same function mentioned above, we check for “.” to conclude its directory block. We should check for “.” and “..”

3. Also, in same function, we check for Text block. There we check for particular byte to be in certain ASCII range. Value of that range need to be re-checked for printable characters.

Nilesh-end