

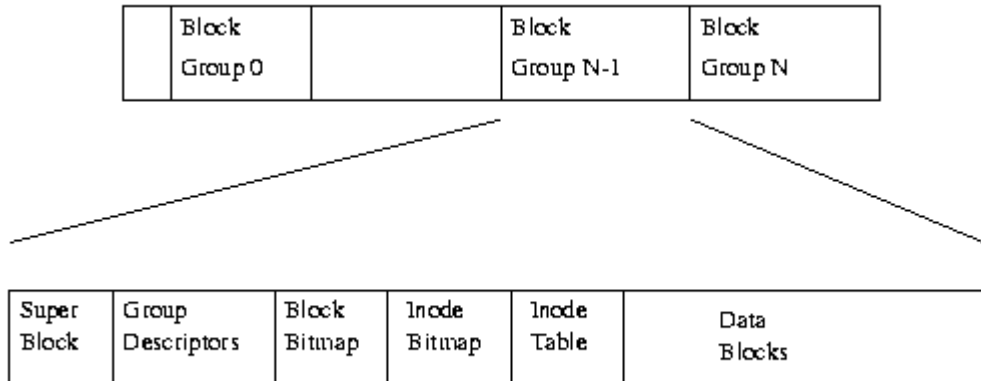
# ANALYZING A FILESYSTEM

Morgan David

The ext2 filesystem is the one that linux systems use most prevalently. It is described in some detail in sections 9.0 and 9.1 of "The Linux Kernel." Please read those two sections. In particular, it's important to understand the various data structures used and then how to traverse them to do the central job of finding a file.

## Filesystem Structures

The basic overall layout of a disk looks like this:



The unlabeled initial portion of the disk is the boot block. We will confine ourselves to diskettes. On a diskette there is just one block group. So a diskette has a boot block followed by the structure shown in the lower bar. Note that "every file in the EXT2 file system is described by a single inode and each inode has a single unique number identifying it. The inodes for the file system are all kept together in [an inode table (depicted above)]. EXT2 directories are simply special files (themselves described by inodes) which contain pointers to the inodes of their directory entries." In order to make sense of inodes and directory entries you need to know their internal structure too. That is described below. Given this overall layout however, let's look at the step-by-step file-fetch procedure.

## I say "Gettysburg," you say "Four score and seven..."

The whole purpose of a filesystem is to enable naming of data and making it findable by its name. I am the program, you are the filesystem. If I say the name of a file, you must come back with the content from your disk. But how do you do that? What is your "filing" system? The discussion in "The Linux Kernel" summarizes it:

```
"A Linux filename ... is a series of directory names separated by forward slashes (``/'') and ending in the file's name. One example filename would be /home/rusling/.cshrc where /home and /rusling are directory names and the file's name is .cshrc. ... To find the inode representing this file within an EXT2 file system the system must parse the filename a directory at a time until we get to the file itself.
```

```
"The first inode we need is the inode for the root of the file system and we find its number in the file system's superblock [this appears to be erroneous, the root inode is always inode number 2, but the process described here is how files are found]. To read an EXT2 inode we must look
```

# ANALYZING A FILESYSTEM

Morgan David

for it in the inode table .... If, for example, the root inode number is 42, then we need the 42nd inode from the inode table.... The root inode is for an EXT2 directory, in other words ... it's data blocks contain EXT2 directory entries.

"home is just one of the many directory entries and this directory entry gives us the number of the inode describing the /home directory. We have to read this directory (by first reading its inode and then reading the directory entries from the data blocks described by its inode) to find the rusling entry which gives us the number of the inode describing the /home/rusling directory. Finally we read the directory entries pointed at by the inode describing the /home/rusling directory to find the inode number of the .cshrc file and from this we get the data blocks containing the information in the file."

Let's try it using the printed dump of a sample floppy. The floppy was produced carefully, to make the difficult job of tracing its contents easier. First it was formatted, thereafter overwritten with hex zeros from start to finish. The ext2 filesystem (i.e., the above described structures) was then written onto it. Then it was given some contents: 1) a file named "alpha1" containing the alphabet in upper case, 2) a subdirectory named "somedir," and 3) a file named "alpha2" containing the alphabet in upper case written backwards. (Remember in this filesystem a directory *is* a file.) The resulting 1.4MB floppy's contents were captured into a file of the same size, and the file dumped. That is what you see in the provided dump file. In reading it, note that the dump utility (the linux od command) prints an asterisk to represent long stretches of unchanging, solid data within the diskette. In this case, stretches of solid zeros appear because the diskette was deliberately "zeroed" before the filesystem was written onto it. This makes the dump relatively compact.

## Getting the lay of the land

First of all, noting that the entire first 1024 bytes are zeros we can take that block to be the boot block. That means the the information starting at byte 1024 (the 1025th) would correspond to the superblock. Does it? Let's test the hypothesis. Superblock contents are specifically documented.

size	start	end	Superblock
4	1	4	Total number of inodes
4	5	8	Filesystem size in blocks
4	9	12	Number of reserved blocks
4	13	16	Free blocks counter
4	17	20	Free inodes counter
4	21	24	Number of first useful block (always 1)
4	25	28	Block size
4	29	32	Fragment size
4	33	36	Number of blocks per group
4	37	40	Number of fragments per group
4	41	44	Number of inodes per group
4	45	48	Time of last mount operation
4	49	52	Time of last write operation

# ANALYZING A FILESYSTEM

Morgan David

2	53	54	Mount operations counter
2	55	56	Number of mount operations before check
2	57	58	Magic signature
2	59	60	Status flag
2	61	62	Behavior when detecting errors
2	63	64	Minor revision level
4	65	68	Time of last check
4	69	72	Time between checks
4	73	76	OS where filesystem was created
4	77	80	Revision level
2	81	82	Default user ID for reserved blocks
2	83	84	Default group ID for reserved blocks
4	85	88	Number of first nonreserved inode
2	89	90	Size of on-disk inode structure
2	91	92	Block group number of this superblock
4	93	96	Compatible features bitmap
4	97	100	Incompatible features bitmap
4	101	104	Read-only-compatible features bitmap
16	105	120	128-bit filesystem identifier
16	121	136	Volume name
64	137	200	Path of last mount point
4	201	204	Used for compression
1	205	205	Number of blocks to preallocate
1	206	206	Number of blocks to preallocate for directories
818	207	1024	Nulls to pad out 1024 bytes

Notice it says "always 1" for the value of the "Number of first useful block" field. And that this field occupies the 21st through 24th bytes of a superblock structure. So if we think our diskette's superblock starts at its byte 1024, we had better find "1" when we advance 21 bytes and examine bytes 1044 through 1047. Find those bytes. They contain "01 00 00 00." That's encouraging. So let's suppose this data at byte 1024 of the diskette *is* its superblock and evaluate its first several ostensible fields to confirm or deny our supposition. All the first several superblock fields are 4 bytes in size. So taking 4 bytes at a time from the diskette would align the following data with the following superblock fields:

Total number of inodes	b8 00 00 00
Filesystem size in blocks	a0 05 00 00
Number of reserved blocks	48 00 00 00
Free blocks counter	74 05 00 00
Free inodes counter	aa 00 00 00

You can convert these to decimal values if you understand that they are hexadecimal values, least significant to the left and most significant to the right. So the five hex numbers represented here are 000000b8, 000005a0, 00000048, 00000574, and 000000aa. If you convert them to decimal you get 184, 1440, 72, 1396, and 170. Sensible or not? Well there are 184 inodes. We aren't in a position to judge because we have no idea what number of inodes a diskette ought to have. And there are 1440

# ANALYZING A FILESYSTEM

Morgan David

blocks. By contrast this is meaningful to us, because we are familiar with diskettes. We do know their capacity is supposed to be 1.44 megabytes and that's what this appears to say. This quite strongly confirms the likelihood that here at the floppy's byte 1024 we do indeed have its superblock. Among the remaining 3 numbers it's notable that there should be 1396 free blocks and 170 free inodes. Since this is a largely empty diskette, 1396 free out of 1440, or 170 free out of 184, is definitely reasonable. It's safe to say we've located the superblock. It will help us locate other things. One thing important to locate would be the inodes for all the files, but the superblock content doesn't make it obvious where they are. So let's look further.

The next chunk of the diskette is the "Group Descriptor." Here's its layout:

Group Descriptor			
4	1	4	Block number of block bitmap
4	5	8	Block number of inode bitmap
4	9	12	Block number of first inode table block
2	13	14	Number of free blocks in the group
2	15	16	Number of free inodes in the group
2	17	18	Number of directories in the group
2	19	20	Alignment to word
4	21	24	Nulls to pad out 24 bytes

The important one here is the 3rd field, "Block number of the first inode table block." The inode table is the collection of all the inodes. It occupies multiple consecutive blocks on the disk and this indicates which block is the first of these. In order to read the Group Descriptor's 3rd field, we first have to locate the Group Descriptor itself. Let's take the super"block" at its word and assume it's exactly one block in size. We know where it starts, and that the Group Descriptor follows it, so what is the size of a block? That info is embedded in the superblock. It's the superblock's 7th field, which sits at the superblock's bytes 25-28. Read it. You'll be disappointed. Because it contains 00 00 00 00. That means our block size is zero, which is impossible, or our superblock is bogus. Or, we just don't know how to read this entry. It turns out that this "field expresses the block size as a power of 2, using 1024 bytes as the unit. Thus, 0 denotes 1024-byte blocks, 1 denotes 2048-byte blocks, and so on." ([Understanding the Linux Kernel](#), Bovet and Cesati). Good. So the superblock, which starts at byte 1024 is one block in size. One block is 1024 bytes. So the block after the superblock starts at byte 2048. The block after the superblock is the group descriptor. So the group descriptor starts at byte 2048. And now you can find its "Block number of first inode table block" field, and observe it contains 05 00 00 00. This represents the hex number 00000005, which is the number 5. The inode table starts in block 5.

It will be useful to be able to locate each block by its labeled byte position in the dump. It seems that the first block which we call block 0 starts with byte 0000000 in the dump (which gives these diskette offset numbers in decimal). And the next one, block 1, starts with byte 0001024. Then the one after, block 2, would start at 0002048 and in general

block n starts at byte (n x 1024)

It would be a good idea to label the dump with numbers in the left margin for each of the blocks it shows. Any offset that is a multiple of 1024 marks the starting point of a block, and you divide the offset by 1024 to get the block's number. Label it with that number.

# ANALYZING A FILESYSTEM

Morgan David

Now have a look at the place you labeled "5." That was next to offset 0005120 in the dump. That's the beginning of the inode table and of the first inode in it. How far does this inode table go? Well, if we know the number of inodes in it and the size of each we can figure it out. Do we? We saw already there appear to be 184 inodes (superblock's first field). And the superblock also contains a "Size of on-disk inode structure" field. Two bytes at offset 89 within the superblock. In our dump, where superblock's 1st byte is labeled 1024, its 89th (88 higher) is at position 1112 (88 higher) by the labels. There and in the ensuing byte we find 80 00. That means hex 0080 or decimal 128. Inode table therefore is 128 x 184 (or 23552) bytes in size. Starting as it does at diskette offset 5120, it goes up to (but not including) 5120 + 23552 or 0028672. That's the place you labeled 28. So the inode table occupies blocks 5 through 27 inclusive (most of which is empty, full of zeros according to the dump).

In addition to numbering the diskette's blocks, you should number the inode table's inodes. Or at least those several that are shown. Mark them down, every 128 bytes starting with 1 at 5120, 2 at 5248, and so on. In general the formula is:

$$\text{diskette offset} = 5120 + (n-1) \times 128$$

or equivalently

$$n = (\text{diskette offset} - 5120) / 128 + 1$$

where n is the inode's number and diskette offset is its position in the dump. The highest inode you'll mark (given the gaps in the dump where there are nothing but zeros) is number 14 at diskette offset 006784.

Now that you have brought order out of chaos by locating and labeling the blocks and inodes, you can use the dump in the same way as would a filesystem driver to locate and read a file.

## Finding a file

Let's find the file alpha1, or more literally let's find its contents (the alphabet A through Z) given its name. The above summary from "The Linux Kernel" gives us our starting point, "The first inode we need is the inode for the root of the file system and we find its number in the file system's superblock." That last part is not actually correct. It turns out, the root's inode is always inode 2. This information is not documented in several sources I used for this investigation, but a former student searched the web and found several references indicating block 2 (e.g., <http://www.linuxgazette.com/issue21/ext2.html>). Additionally, the linux "stat" utility, which prints out inode contents, gives the following:

```
[wright@sputnik wright]$ stat /
```

**File:** "/"

Size: 4096 Allocated Blocks: 8 Filetype: Directory

Mode: (0755/drwxr-xr-x) Uid: ( 0/ root) Gid: ( 0/ root)

Device: 3,1 **Inode:** 2 Links: 22

Access: Thu Dec 6 09:13:40 2001

Modify: Tue Oct 23 13:39:21 2001

Change: Tue Oct 23 13:39:21 2001

In other words, the inode for the "file" / is 2. / means root directory. (Note also that it, though a directory, is treated as a file. Directories *are* files in this filesystem. They do not differ from other files

# ANALYZING A FILESYSTEM

Morgan David

in structure; the difference lies in their content.) So the inode for the file which is the root directory is inode 2. Go to inode 2, where you marked it before. Can't read it, can you? Here's some help:

				Inode
2	1	2		File type and access rights
2	3	4		Owner identification
4	5	8		File length in bytes
4	9	12		Time of last file access
4	13	16		Time that inode last changed
4	17	20		Time that file contents last changed
4	21	24		Time of file deletion
2	25	26		Group identifier
2	27	28		Hard links counter
4	29	32		Number of data blocks of the file
4	33	36		File flags
4	37	40		Specific operating system information
4	41	44		Pointer to first data block
56	45	100		14 more pointers to data blocks
4	101	104		File version (for NFS)
4	105	108		File access control list
4	109	112		Directory access control list
4	113	116		Fragment address
8	117	124		Specific operating system information

Locate the "Pointer to first data block." It reads 1c 00 00 00, or decimal 28. This says that the content of the directory is in block 28. Go to block 28. In it you see "alpha1" and "somedir." Looks like the content of the root directory alright. This calls for our last structure table, that of an ext2 directory entry:

size	start	end		Directory Entry
4	1	4		Inode number
2	5	6		This directory entry's length
1	7	7		File name length
1	8	8		File type (1=regular file 2=directory)
	9?			File name

The key here is that when you've located the file's name in the directory, you can access its inode number by backing up 8 bytes. So for file alpha1, the inode number is 0c 00 00 00 or decimal 12. Go look at the inode you marked "12" earlier (starts at 0006528). Among other things its file length field (1b 00 00 00) indicates a 27-byte file. That would be the alphabet plus one. The extra byte is a linefeed or so-called "end-of-file" character, a hex 0a. Reassuring. But, getting down to the business of locating the file, the pointer to the first (and only) data block contains 29 00 00 00 or decimal 41. Slide down to the block you marked 41 earlier (at offset 0041984). There you see the alphabet.

# ANALYZING A FILESYSTEM

Morgan David

Let's recapitulate how to find a file. First we had to visit the superblock and group descriptor to map the layout of the structure superimposed on this diskette. That's not actually part of finding a file. The structure is a given, and a known, once you know it. So this was a one-time process for us. Now that we know the layout, finding a file using it is something we'll do time and time again. That process consists of:

- visiting the *inode* for the root directory (inode 2) to get the location of the *data* for the root directory
- visiting the data for the root directory to find the directory entry for alpha1
- visiting the entry for alpha1 to get the location of the inode for alpha1
- visiting the inode for alpha1 to get the location of the data for alpha1
- visiting the data of alpha1 (the alphabet)

You have now performed the job of the filesystem driver for the ext2 filesystem. The difference is, it can do it fast.