Next Previous Contents

7. Filesystems

Filesystems are containers of files, that are stored, probably in a directory tree, together with attributes, like size, owner, creation date and the like. A filesystem has a *type*. It defines how things are arranged on the disk. For example, one has the types minix, ext2, reiserfs, iso9660, vfat, hfs.

7.1 FAT

The traditional DOS filesystem types are FAT12 and FAT16. Here FAT stands for File Allocation Table: the disk is divided into *clusters*, the unit used by the file allocation, and the FAT describes which clusters are used by which files.

Let us describe the FAT filesystem in some detail. The FAT12/16 type is important, not only because of the traditional use, but also because it is useful for data exchange between different operating systems, and because it is the filesystem type used by all kinds of devices, like digital cameras.

Layout

First the boot sector (at relative address 0), and possibly other stuff. Together these are the Reserved Sectors. Usually the boot sector is the only reserved sector.

Then the FATs (following the reserved sectors; the number of reserved sectors is given in the boot sector, bytes 14-15; the length of a sector is found in the boot sector, bytes 11-12).

Then the Root Directory (following the FATs; the number of FATs is given in the boot sector, byte 16; each FAT has a number of sectors given in the boot sector, bytes 22-23).

Finally the Data Area (following the root directory; the number of root directory entries is given in the boot sector, bytes 17-18, and each directory entry takes 32 bytes; space is rounded up to entire sectors).

Boot sector

The first sector (512 bytes) of a FAT filesystem is the *boot sector*. In Unix-like terminology this would be called the superblock. It contains some general information.

First an explicit example (of the boot sector of a DRDOS boot floppy).

```
0000000 eb 3f 90 49 42 4d 20 20 33 2e 33 00 02 01 01 00
0000020 02 e0 00 40 0b f0 09 00 12 00 02 00 00 00 00 00
0000040 00 00 00 00 00 00 00 00 00 00 70 00 ff ff 49 42
0000060 4d 42 49 4f 20 20 43 4f 4d 00 50 00 00 08 00 18
...
```

(See here for the complete sector. And also a MSDOS example)

The 2-byte numbers are stored little endian (low order byte first).

```
Bytes   Content
0-2     Jump to bootstrap (E.g. eb 3c 90; on i86: JMP 003E NOP.
        One finds either eb xx 90, or e9 xx xx.
        The position of the bootstrap varies.)
3-10    OEM name/version (E.g. "IBM  3.3", "MSDOS5.0", "MSWIN4.0".
        Various format utilities leave their own name, like "CH-FOR18".
        Sometimes just garbage. Microsoft recommends "MSWIN4.1".)
        /* BIOS Parameter Block starts here */
11-12   Number of bytes per sector (512)
        Must be one of 512, 1024, 2048, 4096.
13      Number of sectors per cluster (1)
        Must be one of 1, 2, 4, 8, 16, 32, 64, 128.
        A cluster should have at most 32768 bytes. In rare cases 65536 is OK.
14-15   Number of reserved sectors (1)
        FAT12 and FAT16 use 1. FAT32 uses 32.
16      Number of FAT copies (2)
17-18   Number of root directory entries (224)
        0 for FAT32. 512 is recommended for FAT16.
19-20   Total number of sectors in the filesystem (2880)
        (in case the partition is not FAT32 and smaller than 32 MB)
21      Media descriptor type (f0: 1.4 MB floppy, f8: hard disk; see below)
22-23   Number of sectors per FAT (9)
        0 for FAT32.
24-25   Number of sectors per track (12)
26-27   Number of heads (2, for a double-sided diskette)
28-29   Number of hidden sectors (0)
        Hidden sectors are sectors preceding the partition.
        /* BIOS Parameter Block ends here */
30-509  Bootstrap
510-511 Signature 55 aa
```

The signature is found at offset 510-511. This will be the end of the sector only in case the sector size is 512.

The ancient media descriptor type codes are:

```
For 8" floppies:
fc, fd, fe - Various interesting formats
```

For 5.25" floppies:

Value	DOS version	Capacity	sides	tracks	sectors/track
ff	1.1	320 KB	2	40	8
fe	1.0	160 KB	1	40	8
fd	2.0	360 KB	2	40	9
fc	2.0	180 KB	1	40	9
fb		640 KB	2	80	8
fa		320 KB	1	80	8
f9	3.0	1200 KB	2	80	15

For 3.5" floppies:

Value	DOS version	Capacity	sides	tracks	sectors/track
fb		640 KB	2	80	8
fa		320 KB	1	80	8
f9	3.2	720 KB	2	80	9
f0	3.3	1440 KB	2	80	18
f0		2880 KB	2	80	36

```
For RAMdisks:
fa

For hard disks:
Value  DOS version
f8     2.0
```

This code is also found in the first byte of the FAT.

IBM defined the media descriptor byte as 11111red, where r is removable, e is eight sectors/track, d is double sided.

FAT16

FAT16 uses the above BIOS Parameter Block, with some extensions:

```
11-27   (as before)
28-31   Number of hidden sectors (0)
32-35   Total number of sectors in the filesystem
        (in case the total was not given in bytes 19-20)
36      Logical Drive Number (for use with INT 13, e.g. 0 or 0x80)
37      Reserved (Earlier: Current Head, the track containing the Boot Record)
        Used by Windows NT: bit 0: need disk check; bit 1: need surface scan
38      Extended signature (0x29)
        Indicates that the three following fields are present.
39-42   Serial number of partition
43-53   Volume label or "NO NAME    "
54-61   Filesystem type (E.g. "FAT12   ", "FAT16   ", "FAT     ", or all zero.)
62-509  Bootstrap
510-511 Signature 55 aa
```

FAT32

FAT32 uses an extended BIOS Parameter Block:

```
11-27   (as before)
28-31   Number of hidden sectors (0)
32-35   Total number of sectors in the filesystem
36-39   Sectors per FAT
40-41   Mirror flags
        Bits 0-3: number of active FAT (if bit 7 is 1)
        Bits 4-6: reserved
        Bit 7: one: single active FAT; zero: all FATs are updated at runtime
        Bits 8-15: reserved
42-43   Filesystem version
44-47   First cluster of root directory (usually 2)
48-49   Filesystem information sector number in FAT32 reserved area (usually 1)
50-51   Backup boot sector location or 0xffff if none (usually 6)
52-63   Reserved
64      Logical Drive Number (for use with INT 13, e.g. 0 or 0x80)
65      Reserved (used by Windows NT)
66      Extended signature (0x29)
        Indicates that the three following fields are present.
67-70   Serial number of partition
```

```
71-81   Volume label
82-89   Filesystem type ("FAT32   ")
```

The old 2-byte fields "total number of sectors" and "number of sectors per FAT" are now zero; this information is now found in the new 4-byte fields.

An important improvement is the "First cluster of root directory" field. Earlier, the root directory was not part of the Data Area, and was in a known place with a known size, and hence was unable to grow. Now the root directory is just somewhere in the Data Area.

FAT

The disk is divided into clusters. The number of sectors per cluster is given in the boot sector byte 13.

The File Allocation Table has one entry per cluster. This entry uses 12, 16 or 28 bits for FAT12, FAT16 and FAT32.

The first two FAT entries

The first cluster of the data area is cluster #2. That leaves the first two entries of the FAT unused. In the first byte of the first entry a copy of the media descriptor is stored. The remaining bits of this entry are 1. In the second entry the end-of-file marker is stored. The high order two bits of the second entry are sometimes, in the case of FAT16 and FAT32, used for dirty volume management: high order bit 1: last shutdown was clean; next highest bit 1: during the previous mount no disk I/O errors were detected.

(Historically this description has things backwards: DOS 1.0 did not have a BIOS Parameter Block, and the distinction between single-sided and double-sided 5.25" 360K floppies was indicated by the first byte in the FAT. DOS 2.0 introduced the BPB with media descriptor byte.)

FAT12

Since 12 bits is not an integral number of bytes, we have to specify how these are arranged. Two FAT12 entries are stored into three bytes; if these bytes are uv,wx,yz then the entries are xuv and yzw.

Possible values for a FAT12 entry are: 000: free, 002-fef: cluster in use; the value given is the number of the next cluster in the file, ff0-ff6: reserved, ff7: bad cluster, ff8-fff: cluster in use, the last one in this file. Since the first cluster in the data area is numbered 2, the value 001 does not occur.

DOS 1.0 and 2.0 used FAT12. The maximum possible size of a FAT12 filesystem (volume) was 8 MB (4086 clusters of at most 4 sectors each).

FAT16

DOS 3.0 introduced FAT16. Everything very much like FAT12, only the FAT entries are now 16 bit. Possible values for FAT16 are: 0000: free, 0002-ffef: cluster in use; the value given is the number of the next cluster in the file, fff0-fff6: reserved, fff7: bad cluster, fff8-ffff: cluster in use, the last one in this file.

Now the maximum volume size was 32 MB, mostly since DOS 3.0 used 16-bit sector numbers. This was fixed in DOS 4.0 that uses 32-bit sector numbers. Now the maximum possible size of a FAT16 volume is 2 GB (65526 clusters of at most 64 sectors each).

FAT32

FAT32 was introduced in Windows 95 OSR 2. Everything very much like FAT16, only the FAT entries are now 32 bits of which the top 4 bits are reserved. The bottom 28 bits have meanings similar to those for FAT12, FAT16. For FAT32: Cluster size used: 4096-32768 bytes.

Microsoft operating systems use the following rule to distinguish between FAT12, FAT16 and FAT32. First, compute the number of clusters in the data area (by taking the total number of sectors, subtracting the space for reserved sectors, FATs and root directory, and dividing, rounding down, by the number of sectors in a cluster). If the result is less that 4085 we have FAT12. Otherwise, if it is less than 65525 we have FAT16. Otherwise FAT32.

Microsoft operating systems use fff, ffff, xfffffff as end-of-clusterchain markers, but various common utilities may use different values.

Directory Entry

An example (6 entries on the same MSDOS floppy):

```
0009728 49 4f 20 20 20 20 20 20 53 59 53 27 00 00 00 00  IO      .SYS
0009744 00 00 00 00 00 00 08 5d 62 1b 1d 00 16 9f 00 00
0009760 4d 53 44 4f 53 20 20 20 53 59 53 27 00 00 00 00  MSDOS   .SYS
0009776 00 00 00 00 00 00 08 5d 62 1b 6d 00 38 95 00 00
0009792 43 4f 4d 4d 41 4e 44 20 43 4f 4d 20 00 00 00 00  COMMAND .COM
0009808 00 00 00 00 00 00 07 5d 62 1b b8 00 39 dd 00 00
0009824 44 42 4c 53 50 41 43 45 42 49 4e 27 00 00 00 00  DBLSPACE.BIN
0009840 00 00 00 00 00 00 08 5d 62 1b 27 01 f6 fc 00 00
0009856 4d 53 44 4f 53 20 20 20 20 20 20 28 00 00 00 00  MSDOS
0009872 00 00 00 00 00 00 1a 88 99 1c 00 00 00 00 00 00
0009888 46 44 49 53 4b 20 20 20 45 58 45 20 00 00 00 00  FDISK   .EXE
0009904 00 00 00 00 00 00 36 59 62 1b 02 00 17 73 00 00
```

```
Bytes   Content
0-10    File name (8 bytes) with extension (3 bytes)
11      Attribute - a bitvector. Bit 0: read only. Bit 1: hidden.
        Bit 2: system file. Bit 3: volume label. Bit 4: subdirectory.
        Bit 5: archive. Bits 6-7: unused.
12-21   Reserved (see below)
22-23   Time (5/6/5 bits, for hour/minutes/doubleseconds)
24-25   Date (7/4/5 bits, for year-since-1980/month/day)
26-27   Starting cluster (0 for an empty file)
28-31   Filesize in bytes
```

We see that the fifth entry in the example above is the volume label, while the other entries are actual files.

The "archive" bit is set when the file is created or modified. It is cleared by backup utilities. This allows one to do incremental backups.

As a special kludge to allow undeleting files that were deleted by mistake, the DEL command will replace the first byte of the name by 0xe5 to signify "deleted". As an extraspecial kludge, the first byte 0x05 in a directory entry means that the real name starts with 0xe5.

The first byte of a name must not be 0x20 (space). Short names or extensions are padded with spaces. Special ASCII characters 0x22 ("), 0x2a (*), 0x2b (+), 0x2c (,), 0x2e (.), 0x2f (/), 0x3a (:), 0x3b (;), 0x3c (<), 0x3d (=), 0x3e (>), 0x3f (?), 0x5b ([), 0x5c (\), 0x5d (]), 0x7c (|) are not allowed.

The first byte 0 in a directory entry means that the directory ends here. (Now the Microsoft standard says that all following directory entries should also have first byte 0, but the Psion's OS, EPOC, works with a single terminating 0.)

Subdirectories start with entries for . and .., but the root directory does not have those.

VFAT

In Windows 95 a variation was introduced: VFAT. VFAT (Virtual FAT) is FAT together with long filenames (LFN), that can be up to 255 bytes long. The implementation is an ugly hack. These long filenames are stored in special directory entries. A special entry looks like this:

Bytes	Content
0	Bits 0-4: sequence number; bit 6: final part of name
1-10	Unicode characters 1-5
11	Attribute: 0xf
12	Type: 0
13	Checksum of short name
14-25	Unicode characters 6-11
26-27	Starting cluster: 0
28-31	Unicode characters 12-13

These special entries should not confuse old programs, since they get the 0xf (read only / hidden / system / volume label) attribute combination that should make sure that all old programs will ignore them.

The long name is stored in the special entries, starting with the tail end. The Unicode characters are of course stored little endian. The sequence numbers are ascending, starting with 1.

Now an ordinary directory entry follows these special entries, and it has the usual info (file size, starting cluster, etc.), and a short version of the long name. Also the unused space in directory entries is used now: bytes 13-17: creation date and time (byte 13: centiseconds 0-199, bytes 14-15: time as above, bytes 16-17: date as above), bytes 18-19: date of last access. (And byte 12 is reserved for Windows NT - it indicates whether the filename is in upper or in lower case; byte 20 is reserved for OS/2.)

Old programs might change directories in ways that can separate the special entries from the ordinary one. To guard against that the special entries have in byte 13 a checksum of the short name:

```
int i; unsigned char sum = 0;

    for (i = 0; i < 11; i++) {
        sum = (sum >> 1) + ((sum & 1) << 7);  /* rotate */
        sum += name[i];                        /* add next name byte */
    }
```

An additional check is given by the sequence number field. It numbers the special entries belonging to a single LFN 1, 2, ... where the last entry has bit 6 set.

The short name is derived from the long name as follows: The extension is the extension of the long name, truncated to length at most three. The first six bytes of the short name equal the first six nonspace bytes of the long name, but bytes + , ; = [ ], that are not allowed under DOS, are replaced by underscore. Lower case is converted to upper case. The final two (or more, up to seven, if necessary) bytes become ~1, or, if that exists already, ~2, etc., up to ~999999.

VFAT is used in the same way on each of FAT12, FAT16, FAT32.

FSInfo sector

FAT32 stores extra information in the FSInfo sector, usually sector 1.

Bytes	Content
0-3	0x41615252 - the FSInfo signature
4-483	Reserved
484-487	0x61417272 - a second FSInfo signature
488-491	Free cluster count or 0xffffffff (may be incorrect)
492-495	Next free cluster or 0xffffffff (hint only)
496-507	Reserved
508-511	0xaa550000 - sector signature

Variations

One meets slight variations on the FAT filesystem in many places. Here a description of the version used for the Xbox.

7.2 Ext2

The ext2 filesystem was developed by Rémy Card and added to Linux in version 0.99pl7 (March 1993). It was a greatly improved version of his earlier ext filesystem (that again was a small improvement on the minix filesystem), and uses ideas from the Berkeley Fast Filesystem. It is really fast and robust. The main reason people want something else nowadays is that on the modern very large disks an invocation of `e2fsck` (to check filesystem integrity after a crash or power failure, or just after some predetermined number of boots) takes a long time, like one hour.

Layout

First, space is reserved for a boot block (1024 bytes). This is not part of the filesystem proper, and ext2 has no opinion about what should be there. This boot block is followed by a number of ext2 block groups.

Each block group starts with a copy of the superblock, then a copy of the group descriptors (for the entire filesystem - ach), then a block bitmap, then an inode bitmap, then an inode table, then data blocks.

An attempt is made to have the data blocks of a file in the same block group as its inode, and as much as possible consecutively, thus minimizing the amount of seeking the disk has to do.

Having a copy of superblock and group descriptors in each block group seemed reasonable when a filesystem had only a few block groups. Later, with thousands of block groups, it became clear that this redundancy was ridiculous and meaningless. (And for a sufficiently large filesystem the group descriptors alone would fill everything, leaving no room for data blocks.) So, later versions of ext2 use a sparse distribution of superblock and group descriptor copies.

The structure of ext2 can be read in the kernel source. The data structures are defined in `ext2_fs.h`, `ext2_fs_i.h`, `ext2_fs_sb.h` (in `include/linux`) and the code is in `fs/ext2`.

Exercise *Do* `cd /tmp; dd if=/dev/zero of=e2fs bs=1024 count=10000; mke2fs -F e2fs; od -Ax -tx4 e2fs`. *(This creates an empty ext2 filesystem in the file* `/tmp/e2fs`*, and prints its contents.) Compare the* `od` *output with the description below.*

The superblock

First, the superblock (of the original ext2, later more fields were added).

```
struct ext2_super_block {
        unsigned long s_inodes_count;     /* Inodes count */
        unsigned long s_blocks_count;     /* Blocks count */
        unsigned long s_r_blocks_count;   /* Reserved blocks count */
        unsigned long s_free_blocks_count;/* Free blocks count */
        unsigned long s_free_inodes_count;/* Free inodes count */
        unsigned long s_first_data_block; /* First Data Block */
        unsigned long s_log_block_size;   /* log(Block size) - 10 */
        long s_log_frag_size;             /* Fragment size */
        unsigned long s_blocks_per_group; /* # Blocks per group */
        unsigned long s_frags_per_group;  /* # Fragments per group */
        unsigned long s_inodes_per_group; /* # Inodes per group */
        unsigned long s_mtime;          /* Mount time */
        unsigned long s_wtime;          /* Write time */
        unsigned long s_pad;            /* Padding to get the magic signature*/
                                        /* at the same offset as in the */
                                        /* previous ext fs */
        unsigned short s_magic;         /* Magic signature */
        unsigned short s_valid;         /* Flag */
        unsigned long s_reserved[243];  /* Padding to the end of the block */
};
```

The superblock contains information that is global to the entire filesystem, such as the total number of blocks, and the time of the last mount.

Some parameters, such as the number of blocks reserved for the superuser, can be tuned using the `tune2fs` utility. Having reserved blocks makes the system more stable. It means that if a user program gets out of control and fills the entire disk, the next boot will not fail because of lack of space when utilities at boot time want to write to disk.

The magic signature is 0xef53. The signature is too small: with random data one in every 2^16 blocks will have this value at this offset, so that several hundred blocks will have the ext2 superblock signature without being a superblock. Modern filesystems use a 32-bit or 64-bit signature.

The group descriptors

Then, the group descriptors. Each block group has a group descriptor, and all group descriptors for all block groups are repeated in each group. All copies except the first (in block group 0) of superblock and group descriptors are never used or updated. They are just there to help recovering from a crash or power failure.

The size of a block group depends on the chosen block size for the ext2 filesystem. Allowed block sizes are 1024, 2048 and 4096. The blocks of a block group are represented by bits in a bitmap that fills one block, so with 1024-byte blocks a block group spans 8192 blocks (8 MiB), while with 4096-byte blocks a block group spans 32768 blocks (128 MiB). When disks were small, small blocks were used to minimize the loss in space due to rounding up filesizes to entire blocks. These days the default is to use larger blocks, for faster I/O.

Question *What percentage of the disk space is used (wasted) by copies of superblock and group descriptors on a 128 GiB filesystem with 1024-byte blocks? How large is the filesystem when all available space is taken by the group descriptors?*

```
struct ext2_group_desc
{
        unsigned long bg_block_bitmap;          /* Blocks bitmap block */
        unsigned long bg_inode_bitmap;          /* Inodes bitmap block */
        unsigned long bg_inode_table;           /* Inodes table block */
        unsigned short bg_free_blocks_count;    /* Free blocks count */
        unsigned short bg_free_inodes_count;    /* Free inodes count */
        unsigned short bg_used_dirs_count;      /* Directories count */
        unsigned short bg_pad;
        unsigned long bg_reserved[3];
};
```

Thus, a group descriptor takes 32 bytes, 18 of which are used. The field `bg_block_bitmap` gives the block number of the block allocation bitmap block. In that block the free blocks in the block group are indicated by 1 bits. Similarly, the field `bg_inode_bitmap` gives the block number of the inode allocation bitmap. The field `bg_inode_table` gives the starting block number of the inode table for this block group. These three fields are potentially useful at recovery time. Unfortunately, there is almost no redundancy in a bitmap, so if either a block with group descriptors or a block with a bitmap is damaged, `e2fsck` will happily go on and destroy the entire filesystem.

Project *Investigate how the redundancy present in an ext2 filesystem could be used. Is it possible to detect that a block bitmap or inode bitmap is damaged? Presently, the redundancy present in the repeated superblocks and group descriptors is used only when the user explicitly invokes* `e2fsck` *with parameter* `-b` *N, where N is the block number of the superblock copy. (Now superblock N and group descriptor blocks N+1.. are used.) How can* `e2fsck` *detect automatically that something is wrong, and select and switch to a copy that is better?*

The inode table

Each inode takes 128 bytes:

```
struct ext2_inode {
        unsigned short i_mode;          /* File mode */
        unsigned short i_uid;           /* Owner Uid */
        unsigned long i_size;           /* Size in bytes */
        unsigned long i_atime;          /* Access time */
        unsigned long i_ctime;          /* Creation time */
        unsigned long i_mtime;          /* Modification time */
        unsigned long i_dtime;          /* Deletion Time */
        unsigned short i_gid;           /* Group Id */
        unsigned short i_links_count;   /* Links count, max 32000 */
        unsigned long i_blocks;         /* Blocks count */
        unsigned long i_flags;          /* File flags */
        unsigned long i_reserved1;
        unsigned long i_block[15];      /* Pointers to blocks */
        unsigned long i_version;        /* File version (for NFS) */
        unsigned long i_file_acl;       /* File ACL */
        unsigned long i_dir_acl;        /* Directory ACL */
        unsigned long i_faddr;          /* Fragment address */
        unsigned char i_frag;           /* Fragment number */
        unsigned char i_fsize;          /* Fragment size */
        unsigned short i_pad1;
        unsigned long i_reserved2[2];
};
```

History

When 16-bit uid's no longer sufficed (a PC, and more than one user? yes, university machines handling the mail for more than 65000 people), the part

```
        unsigned long i_reserved2[2];
```

was replaced by

```
        __u16   l_i_uid_high;           /* High order part of uid */
        __u16   l_i_gid_high;           /* High order part of gid */
        __u32   l_i_reserved2;
```

Also, `i_version` was renamed into `i_generation`. This is for use with NFS. If a file is deleted, and later the inode is reused, then a client on another machine, that still had an open filehandle for the old file, must get an error return upon access. This is achieved by changing `i_generation`. The `i_generation` field of a file can be read and set using the `EXT2_IOC_GETVERSION` and `EXT2_IOC_SETVERSION` ioctls.

Exercise *Write the tiny program that can read and change the ext2 version field of an inode. Get the ioctl definitions from* `<linux/ext2_fs.h>`.

File sizes

Look at a filesystem with block size B (e.g., 1024 or 4096). The inode contains 12 pointers to direct blocks: the block numbers of the first 12 data blocks. Files of size not more than 12*B bytes (e.g., 12288 or 49152), do not need more. The 13th element of the array `i_block` is a pointer to an indirect block. That indirect block contains the block numbers of B/4 data blocks. That suffices for files of size not more than (B/4)*B + 12*B bytes (e.g., 274432 or 4243456). The 14th element of the array `i_block` is a pointer to a doubly indirect block. It contains the block numbers of B/4 indirect blocks. That suffices for size not more than (B/4)*(B/4)*B + (B/4)*B + 12*B bytes (e.g., 67383296 or 4299210752). The 15th and last element of the array `i_block` is a pointer to a triply indirect block. It contains the block numbers of B/4 doubly indirect blocks. That suffices up to (B/4)*(B/4)*(B/4)*B + (B/4)*(B/4)*B + (B/4)*B + 12*B bytes (e.g., 17247252480 or 4402345721856). Thus, this design allows for files not larger than about 4 TB. Other conditions may impose a lower limit on the maximum file size.

Sparse files are represented by having block numbers 0 represent holes.

Exercise *Explain the sizes given earlier for sparse files in some ext2 filesystem.*

Ext2 has *fast symbolic links*: if the file is a symlink (which is seen from its `i_mode` field), and the length of the pathname contained in the symlink is less than 60 bytes, then the actual file contents is stored in the `i_block[]` array, and the fact that this happened is visible from the fact that `i_blocks` is zero.

As an aside: how large are files in reality? That is of great interest to a filesystem designer. Things of course depend strongly on the type of use, but let me make the statistics on one of my machines.

```
Size of ordinary files:
("m bits" means "size less than 2^m but not less than 2^{m-1} unless m==0")
0 bits: 27635
1 bits: 207
2 bits: 712
3 bits: 2839
4 bits: 12343
5 bits: 66063
6 bits: 47328
7 bits: 45039
8 bits: 71593
9 bits: 104873
10 bits: 171541
11 bits: 356011
12 bits: 517599
13 bits: 283794
14 bits: 191133
15 bits: 132640
16 bits: 70352
17 bits: 38069
18 bits: 16614
19 bits: 8182
20 bits: 6045
21 bits: 3023
22 bits: 1433
23 bits: 1020
24 bits: 444
25 bits: 250
26 bits: 48
27 bits: 14
28 bits: 12
29 bits: 11
30 bits: 7
31 bits: 1
```

I see here 27635 empty files. The most common size is 12 bits: 2048-4095 bytes. Clearly, in this filesystem the majority of the files only need direct blocks. The fact that many files are small also means that a lot of space is wasted by going to a larger block size.

Other people will have a different distribution.

The designers of unionfs report: "the average file size is 52982 bytes on our groups file server's /home directory, which has over five million files belonging to 82 different users" (2005).

Google developed their own filesystem GFS, a large scale distributed fault-tolerant file system, designed for an environment where one has "a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case".

Reserved inodes

Inode 1 is reserved for a list of bad blocks on the device.

The root directory has inode number 2. This number must be fixed in advance, since the `mount()` system call must be able to find the root directory.

A few other inode numbers have a special meaning. Inode 11 is the first one for ordinary use.

Directories

Files have pathnames, paths from the root `/` of the tree to the file. The last element of the path is the file name. Directories are files that give the correspondence between file names and inodes. For the filesystem examined above:

```
Max level: 18
118355 directories
2176875 regular files
24407 other
```

A directory entry

```
struct ext2_dir_entry {
        unsigned long inode;            /* Inode number */
        unsigned short rec_len;         /* Directory entry length */
        unsigned short name_len;        /* Name length */
        char name[up to 255];           /* File name */
};
```

does not have a fixed length. It gives the inode number and the name. There are two lengths: the length of the name, and after how many bytes the next directory entry starts. These may differ - on the one hand because entries are aligned to start at an address that is a multiple of 4. On the other hand, the presence of a `rec_len` field allows efficient file deletion: it is not necessary to shift all names in a directory when a file is deleted; instead only the `rec_len` of the preceding entry is adapted, so that a directory search automatically skips over deleted entries.

Question *And what if the entry was the first in the directory? Read the answer in* `fs/ext2/namei.c`.

While names in a symlink are NUL-terminated, names in a directory entry are not.

These days the directory struct is slightly different:

```
struct ext2_dir_entry_2 {
        __u32   inode;                  /* Inode number; 0: unused */
        __u16   rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[up to 255];        /* File name */
};
```

Since `name_len` cannot be larger than 255, a single byte suffices, and the other byte, that used to be zero, is now a file type: unknown (0), regular (1), directory (2), character special device (3), block special device (4), fifo (5), socket (6), symlink (7). Some applications are sped up considerably by the presence of this file type, since they now can walk a tree without doing a `stat()` call on every file encountered.

Limits and efficiency

The macro `EXT2_LINK_MAX`, defined in `ext2_fs.h` to the value 32000, makes sure that a directory cannot contain more than 32000 subdirectories. That sometimes causes problems. (I ran into this when trying to extract the about 60000 different files from a collection of about 150000 similar files. Comparing pairwise would take too long, so I decided to move each file to a directory that had its md5sum as name. Now files in the same directory would be almost certainly identical, and a few diffs should remove the duplicates. The script failed after creating 31998 subdirectories.)

Something else is that ext2 does linear searches in its directories, and things get really slow with large directories. Timing on some machine here: create 10000 files in an empty directory: 98 sec; create 10000 files more: 142 sec; create 10000 files more: 191 sec; create 10000 files more: 242 sec. Clear quadratic behaviour. This means that it is a bad idea to store many files in one directory on an ext2 system. Try the same on a reiserfs system: 80 sec, 81 sec, 81 sec, 80 sec. No increase in running time.

Fragments

Fragments have not been implemented.

Attributes

Files on an ext2 filesystem can have various attributes. There are

A	Atime	Linux 2.0	`EXT2_NOATIME_FL`	Don't update the atime field upon access.
S	Sync	Linux 1.0	`EXT2_SYNC_FL`	Perform synchronous writes - do not buffer.
D	Dirsync	Linux 2.6	`EXT2_DIRSYNC_FL`	If this is a directory: sync.
a	Append Only	Linux 1.2	`EXT2_APPEND_FL`	Only allow opening of this file for appending.For directories: disallow file deletion.
i	Immutable	Linux 1.2	`EXT2_IMMUTABLE_FL`	Disallow all changes to this file (data and

				inode).
j	Journalling	none	`EXT2_JOURNAL_DATA_FL`	(ext3 only) Upon a write, first write to the journal.
d	No Dump	Linux 1.2	`EXT2_NODUMP_FL`	The dump(8) program should ignore this file.
c	Compress	none	`EXT2_COMPR_FL`	Transparently compress this file upon write, uncompress upon read.
s	Secure Deletion	Linux 1.0-1.2 only	`EXT2_SECRM`	When this file is deleted, zero its data blocks.
u	Undelete	none	`EXT2_UNRM_FL`	When this file is deleted, preserve its data blocks,so that later undeletion is possible.
T	Topdir	Linux 2.6	`EXT2_TOPDIR_FL`	Consider this directory a top directory for the Orlov allocator.

Attributes can be viewed with `lsattr` and changed using `chattr`. 'A' makes the system a bit faster since it saves some disk I/O. 'a' and 'i' are useful as a defense against hackers, even when they got root. (These bits are read and set using the `EXT2_IOC_GETFLAGS` and `EXT2_IOC_SETFLAGS` ioctls.)

Before Linux 2.1 there was a *securelevel* variable, so that the 'a' and 'i' bits could not be changed by root when it was larger than zero. Moreover, this variable could never be decreased, other than by rebooting. Or at least, that was the idea. However, with memory access one can do anything:

```
# cat /proc/sys/kernel/securelevel
1
# cat /proc/ksyms | grep securelevel
001a8f64 securelevel
# echo "ibase=16; 001A8F64" | bc
1740644
# dd if=/dev/zero of=/dev/kmem seek=1740644 bs=1 count=1
1+0 records in
1+0 records out
# cat /proc/sys/kernel/securelevel
0
#
```

Nowadays there is the capability system, see lcap(8), if you have that installed. Using

```
# lcap CAP_LINUX_IMMUTABLE
# lcap CAP_SYS_RAWIO
```

one disallows root to change the 'a' and 'i' bits, and disallows root to write to the raw disk or raw memory. Without this last restriction, root can do anything, since it can patch the running kernel code.

More

The above mostly describes an early version of ext2, and there are many details that were skipped. But this should suffice for our purposes.

7.3 Journaling filesystems

A crash caused by power failure or hardware failure or software bug may leave the filesystem in an inconsistent state. The traditional solution is the use of the utilities `icheck`, `dcheck`, `ncheck`, `clri`. However, with several hundred thousand files and a several GB filesystem checking the filesystem for consistency may take a long time, more than one is prepared to wait. A *journaling filesystem* has a *journal* (or *log*) that records all transactions before they are actually done on the filesystem. After a crash one finds in the journal what data was being modified at the moment of the crash, and bringing the filesystem in a consistent state is now very fast.

(If the crash happens during a write to the journal, we notice and need not do anything: the filesystem is OK and the journal can be erased. If the crash happens during a write to the filesystem, we replay the transactions listed in the journal.)

Of course there is a price: the amount of I/O is doubled. In cases where data integrity is less important but filesystem integrity is essential, one only journals metadata (inode and directory contents, not regular file contents).

There is a different price as well: the old check gave an absolute guarantee that after the, say, `e2fsck` the filesystem was clean. The check of a journaling filesystem only gives a conditional guarantee: *if* the power failure or hardware failure or software bug that caused the crash only affected the disk blocks currently being written to according to the journal, then all is well. Especially in the case of kernel bugs this assumption may be wrong.

And then a third price: code complexity.

Linux Journaling Filesystems

Currently, Linux supports four journaling filesystem types: ext3, jfs, reiserfs, xfs. Several other log-structured filesystems are under development.

(Ext3 is a journaling version of ext2, written by Stephen Tweedie. JFS is from IBM. Reiserfs is from Hans Reiser's Namesys. XFS is from SGI, and was ported from IRIX to Linux.) Each has its strengths and weaknesses, but reiserfs seems the most popular. It is the default filesystem type used by SuSE (who employ reiserfs developer Chris Mason). RedHat (employs Stephen Tweedie and) uses ext3.

Linux has a Journaling Block Device layer intended to handle the journaling for all journaling filesystem types. (See `fs/jbd`.) However, it is used by ext3 only.

7.4 NFS

NFS is a network filesystem. (It is the Network File System.) NFS v2 was released in 1985 by Sun. It allows you to mount filesystems of one computer on the file hierarchy of another computer. Other network filesystem types are smb (Samba, used for file-sharing with Windows machines), ncp (that provides access to Netware server volumes and print queues), Coda and the Andrew filesystem.

Setup

Let us first actually use it, and then look at the theory. On a modern distribution one just clicks "start NFS". Let us go the old-fashioned way and do the setup by hand.

Ingredients: two computers with ethernet connection. Make sure they see each other (`ping` works). Make sure both machines are running portmap (on each machine `rpcinfo -p` shows that portmap is running locally and `rpcinfo -p othermachine` shows that portmap is running remotely). If there are problems, check `/etc/hosts.allow`, `/etc/hosts.deny` and the firewall rules.

Is the intended server running NFS?

```
# rpcinfo -p knuth
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
#
```

Hmm. It isn't. Then let us first setup and start NFS on the server. We need to tell what is exported and to whom:

```
knuth# cat /etc/exports
/ 192.168.1.12(ro)
```

This says: Export `/`, that is, everything on the filesystem rooted at `/`. Export it to 192.168.1.12, read-only.

Several daemons (especially `tcpd`) use the files `/etc/hosts.allow` and `/etc/hosts.deny` (first checking the first to see whether access is allowed, then checking the second to see whether access is denied, finally allowing access). Just to be sure, add

```
knuth# cat /etc/hosts.allow
# Allow local machines
ALL:192.168.1.0/255.255.255.0

knuth# cat /etc/hosts.deny
# Deny everybody else
ALL:ALL

knuth#
```

Now start the NFS server:

```
knuth# /usr/sbin/rpc.mountd
knuth# /usr/sbin/rpc.nfsd
```

(Of course, once all is well one wants such commands in one of the startup scripts, but as said, we do things here by hand.) Check that all is well and mount the remote filesystem:

```
# rpcinfo -p knuth
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100003    2   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100005    1   udp    635  mountd
    100005    2   udp    635  mountd
    100005    1   tcp    635  mountd
    100005    2   tcp    635  mountd
```

```
# mount -t nfs knuth:/ /knuth -o ro
#
```

Done.

A portmapper may not be absolutely required - old versions of mount know that the port should be 635 - but in most setups both local and remote machine must have one.

There are other parts to NFS, e.g. the locking daemon.

What is exported is (part of) a filesystem on the server: the exported tree does not cross mount points. Thus, in the above example, if the server has the proc filesystem mounted at `/proc`, the client will see an empty directory there.

On some systems the file `/etc/exports` is not used directly. Instead, a file like `/var/lib/nfs/xtab` is used, and a utility `exportfs(8)` is used to initialize and maintain it. Now it is this utility that reads `/etc/exports`.

Theory

We shall mostly discuss NFS v2 (RFC 1094).

The most important property of NFS v2/v3 is that it is *stateless*. The server does not keep state for the client. Each request is self-contained. After a timeout, the request is repeated. As a consequence, the system is robust in the face of server or client crashes. If the client crashes the server never notices. If the server crashes the client keeps repeating the request until the server is up and running again and answers the request.

(This is the default, also called "hard" mounting. "Soft" mounting is when the client gives up after a while and returns an error to the user. "Spongy" mounting is hard for data transfers and soft for status requests: stat, lookup, fsstat, readlink, and readdir.)

Thus, it is expected that retransmissions of requests may occur. And since the server is stateless, it probably has not remembered what requests were served earlier. It follows that one would like to have *idempotent* requests, where it does not matter whether we do them once or twice. Fortunately, most requests (read, write, chmod) are naturally idempotent. An example of a non-idempotent request is delete. A repetition would get an error return.

Due to the fact that NFS is stateless, it cannot quite reproduce Unix semantics.

For example, the old Unix trick: create a temporary file, and delete it immediately, retaining an open file descriptor, doesn't work. The open descriptor lives on the client, and the server doesn't know anything about it. Consequently the file would be removed, data and everything.

As a partial workaround, the client, who knows that the file is open, does not transmit the delete command to the server, but sends a "silly rename" command instead, renaming the file to some unlikely name that hopefully won't conflict with anything else, like `.nfs7200205400000001`, and waits until the file has been closed before actually removing this file. This helps, but is less than perfect. It does not protect against other clients that remove the file. If the client crashes, garbage is left on the server.

Somewhat similarly, protection is handled a bit differently. Under Unix, the protection is checked when the file is opened, and once a file has been opened successfully, a subsequent `chmod` has no influence. The NFS server does not have the concept of open file and checks permissions for each read or write.

As a partial workaround, the NFS server always allows the owner of a file read and write permission. (He could have given himself permission anyway.) Approximate Unix semantics is preserved by the client who can return "access denied" at open time.

A problem on larger networks is that read and write atomicity is lost. On a local machine "simultaneous" reads and writes to a file are serialized, and appear as if done in a well-defined order. Over the network, a read or write request can span many packets, and clients can see a partly old partly new situation.

Thus, locking is important. But of course locking cannot work in a stateless setup. NFS comes with `lockd` (and `statd`). It supplies `lockf`-style advisory locking.

NFS v3 (RFC 1813) was released in 1993. It contains 64-bit extensions, and efficiency improvements. Only a small step from NFS v2.

NFS v4 (RFC 3010) became a proposed standard in 2000. It is stateful, allows strong security, has file locking, uses UTF8. Rather different from earlier NFS.

The protocol

XDR

One part of the protocol is the data representation. If the communication channel is a byte stream, and the transmitted data contains 32-bit integers, then an agreement is required in which order one transmits the four bytes making up an integer. And similar questions occur for other data types. Sun defined the Extended Data Representation (XDR) that specifies the representation of the transmitted data. For example, 32-bit integers are transmitted with high-order byte first. All objects and fields of structures are NUL-padded to make their length a multiple of 4. Strings are represented by an integer giving the length followed by the bytes of the string. More generally, variable-sized data is preceded by an integer giving the length. For more details, see RFC 1014.

RPC

Communication is done via Remote Procedure Calls (RPCs). Usually these are sent via (unreliable) UDP, and retransmitted after a timeout. One can also use NFS over TCP.

RPC uses XDR. As a consequence, all small integers take four bytes. Some detail of the protocol is given below.

NFS

NFS uses RPC. All procedures are synchronous: when the (successful) reply is received, the client knows that the operation has been done. (This makes NFS slow: writes have to be committed to stable storage immediately. NFS v3 introduces caching and the COMMIT request to make things more efficient.)

The NFS requests:

0	NFSPROC_NULL	Do nothing
1	NFSPROC_GETATTR	Get File Attributes
2	NFSPROC_SETATTR	Set File Attributes
3	NFSPROC_ROOT	Get Filesystem Root
4	NFSPROC_LOOKUP	Look Up File Name
5	NFSPROC_READLINK	Read From Symbolic Link
6	NFSPROC_READ	Read From File
7	NFSPROC_WRITECACHE	Write to Cache
8	NFSPROC_WRITE	Write to File
9	NFSPROC_CREATE	Create File
10	NFSPROC_REMOVE	Remove File
11	NFSPROC_RENAME	Rename File
12	NFSPROC_LINK	Create Link to File
13	NFSPROC_SYMLINK	Create Symbolic Link
14	NFSPROC_MKDIR	Create Directory
15	NFSPROC_RMDIR	Remove Directory
16	NFSPROC_READDIR	Read From Directory
17	NFSPROC_STATFS	Get Filesystem Attributes

Packets on the wire

On the wire, we find ethernet packets. Unpacking these, we find IP packets inside. The IP packets contain UDP packets. The UDP packets contain RPC packets. It is the RPC (remote procedure call) mechanism that implements NFS.

Ethernet

Just for fun, let us look at an actual packet on the wire: 170 bytes.

```
0000  00 10 a4 f1 3c d7 00 e0  4c 39 1b c2 08 00 45 00
0010  00 9c 00 20 40 00 40 11  b6 cc c0 a8 01 0c c0 a8
0020  01 08 03 20 08 01 00 88  d3 3d e2 c0 87 0b 00 00
0030  00 00 00 00 00 02 00 01  86 a3 00 00 00 02 00 00
0040  00 04 00 00 00 01 00 00  00 30 00 04 99 cd 00 00
0050  00 05 6d 65 74 74 65 00  00 00 00 00 03 e8 00 00
0060  00 64 00 00 00 05 00 00  00 64 00 00 00 0e 00 00
```

```
0070  00 10 00 00 00 11 00 00  00 21 00 00 00 00 00 00
0080  00 00 01 70 00 72 01 70  00 00 00 00 00 00 00 00
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00a0  00 00 00 00 00 03 61 65  62 00
```

An ethernet packet (frame) consists of

- An idle time of at least 9.6 microseconds,
- An 8-byte preamble (consisting of 62 alternating bits 1 and 0 followed by two bits 1) used for synchronization (not shown here),
- The 6-byte destination MAC [Medium Access Control] address (here 00:10:a4:f1:3c:d7),
- The 6-byte source MAC address (here 00:e0:4c:39:1b:c2),
- A 2-byte frame type (here `08 00`, indicating an IP datagram),
- The actual data,
- A 4-byte CRC (not shown here).

The specification requires the total length (including CRC, excluding preamble) to be at least 64 and at most 1518 bytes long. For the data that means that it must have length in 46..1500. Frames that are too short ("runts"), or too long ("jabbers"), or have a non-integral number of bytes, or have an incorrect checksum, are discarded. IEEE 802.3 uses a 2-byte length field instead of the type field.

IP

After peeling off this outer MAC layer containing the ethernet transport information, 156 data bytes are left, and we know that we have an IP datagram. IP is described in RFC 791.

An IP datagram consists of a header followed by data. The header consists of:

- A byte giving a 4-bit *version* (here 4: IPv4) and a 4-bit *header length* measured in 32-bit words (here 5). Thus, the header is

  ```
  45 00 00 9c  00 20 40 00  40 11 b6 cc  c0 a8 01 0c  c0 a8 01 08
  ```

- A byte giving a *service type* (here 0: nothing special),
- Two bytes giving the *total length* of the datagram (here `00 9c`, that is, 156),
- Two bytes *identification*, 3 bits *flags*, 13 bits *fragment offset* (measured in multiples of 8 bytes), all related to fragmentation, but this packet was not fragmented - indeed, the flag set is the DF [Don't Fragment] bit,
- A byte giving the TTL (time to live, here hex 40, that is, 64),
- A byte giving the protocol (here hex 11, that is, UDP),
- Two bytes giving a header checksum,
- The source IP address (here `c0 a8 01 0c`, that is, 192.168.1.12),
- The destination IP address (here `c0 a8 01 08`, that is, 192.168.1.8)
- Optional IP options, and padding to make the header length a multiple of 4.

UDP

After peeling off the IP layer containing the internet transport information, 136 data bytes are left, and we know we have a UDP datagram:

```
03 20 08 01 00 88 d3 3d
e2 c0 87 0b 00 00 00 00 00 00 00 02 00 01 86 a3
00 00 00 02 00 00 00 04 00 00 00 01 00 00 00 30
00 04 99 cd 00 00 00 05 6d 65 74 74 65 00 00 00
00 00 03 e8 00 00 00 64 00 00 00 05 00 00 00 64
00 00 00 0e 00 00 00 10 00 00 00 11 00 00 00 21
00 00 00 00 00 00 00 00 01 70 00 72 01 70 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 03 61 65 62 00
```

UDP (User Datagram Protocol) is described in RFC 768. A UDP datagram consists of a 2-byte source port (here `03 20`, that is, 800), a 2-byte destination port (here `08 01`, that is, 2049), a 2-byte length (here `00 88`, that is 136), a 2-byte checksum, and data. So we have 128 bytes of data. On port 2049 the NFS server is running. It gets these 128 bytes, and sees a Remote Procedure Call.

RPC

RPC is described in RFCs 1050, 1057, 1831. An RPC request consists of

- A 4-byte `xid`, the *transmission ID*. The client generates a unique `xid` for each request, and the server reply contains this same `xid`. (Here `xid` is `e2 c0 87 0b`.) The client may, but need not, use the same `xid` upon retransmission after a timeout. The server may, but need not, discard requests with an `xid` it already has replied to.
- A 4-byte *direction* (0: call, 1: reply). (Here we have a call.)

In the case of a call, the fields following `xid` and direction are:

- A 4-byte *RPC version*. (Here we have RPC v2.)
- A 4-byte *program number* and a 4-byte *version* of the program or service called. (Here we have `00 01 86 a3`, that is, 100003, the NFS protocol, and `00 00 00 02`, that is NFS v2.)
- A 4-byte *procedure call number*. (Here we have `00 00 00 04`, procedure 4 of the NFS specification, that is, NFSPROC_LOOKUP.)
- Authentication info (see below).
- Verification info (see below).
- Procedure-specific parameters.

In the case of a reply, the field following `xid` and direction is:

- A 4-byte *status* (0: accepted, 1: denied)

In the case of a reply with status "accepted", the fields following are:

- Verification info (see below), that may allow the client to verify that this reply is really from the server.
- A 4-byte *result status* (0: call executed successfully, 1: program unavailable, 2: version unavailable, 3: procedure unavailable, 4: bad parameters).
- In case of a successfully executed call, the results are here. In case of failure because of unavailable version, two 4-byte fields giving the lowest and highest versions supported. In case of other failures, no further data.

In the case of a reply with status "denied", the fields following are:

- A 4-byte *rejection reason* (0: RPC version not 2, 1: authentication error).
- In case of RPC version mismatch, two 4-byte fields giving the lowest and highest versions supported. In case of an authentication error a 4-byte reason (1: bad credentials, 2: client must begin new session, 3: bad verifier, 4: verifier expired or replayed, 5: rejected for security reasons).

The three occurrences of "authentication/verification info" above each are structured as follows: first a 4-byte field giving a type (0: AUTH_NULL, 1: AUTH_UNIX, 2: AUTH_SHORT, 3: AUTH_DES, ...), then the length of the authentication data, then the authentication data.

In the case of the packet we are looking at the authentication info is `00 00 00 01` (AUTH_UNIX) `00 00 00 30` (48 bytes) `00 04 99 cd` (stamp) `00 00 00 05` (length of machine name: 5 bytes) `6d 65 74 74 65 00 00 00` ("mette" with three bytes padding) `00 00 03 e8` (user id 1000) `00 00 00 64` (group id 100) `00 00 00 05` (5 auxiliary group ids) `00 00 00 64 00 00 00 0e 00 00 00 10 00 00 00 11 00 00 00 21` (100, 14, 16, 17, 33).

The verification info is `00 00 00 00 00 00 00 00` (AUTH_NULL, length 0).

NFS

After peeling off the RPC parts of the packet we are left with the knowledge: this is for NFS, procedure NFSPROC_LOOKUP, and the parameters for this procedure are

```
01 70 00 72 01 70 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 03 61 65 62 00
```

Examining the NFS specification, we see that a lookup has as parameter a `struct diropargs`, with two fields, `fhandle dir` and `filename name`. A `fhandle` is an opaque structure of size FHSIZE, where FHSIZE=32, created by the server to represent a file (given a lookup from the client). A `filename` is a string of length at most MAXNAMLEN=255. Here we see length 3, and string "aeb".

Altogether, this packet asked for a lookup of the name "aeb" in a directory with specified fhandle.

The reply

```
00 e0 4c 39 1b c2 00 10  a4 f1 3c d7 08 00 45 00
00 9c 04 ec 00 00 40 11  f2 00 c0 a8 01 08 c0 a8
01 0c 08 01 03 20 00 88  16 ce e2 c0 87 0b 00 00
00 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 c6 98  01 72 02 70 03 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 02 00 00 41 ed 00 00
00 29 00 00 01 f4 00 00  01 f4 00 00 0c 00 00 00
10 00 00 00 00 00 00 00  00 06 00 00 00 01 72 01
98 c6 3d f0 ec fd 00 00  00 00 3d ee 07 54 00 00
00 00 3d ee 07 54 00 00  00 00
```

Exercise *Interpret this packet. Verify that this is a reply, and that the* `xid` *of the reply equals the* `xid` *of the request. By some coincidence request and reply have the same length.*

The call was successful, and returns

```
00 00 00 00 c6 98 01 72  02 70 03 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02  00 00 41 ed 00 00 00 29
00 00 01 f4 00 00 01 f4  00 00 0c 00 00 00 10 00
00 00 00 00 00 00 00 06  00 00 00 01 72 01 98 c6
3d f0 ec fd 00 00 00 00  3d ee 07 54 00 00 00 00
3d ee 07 54 00 00 00 00
```

The result of a lookup is a `status` (here NFS_OK=0), in case NFS_OK followed by the servers `fhandle` for the file and a `fattr` giving the file attributes. It is defined by

```
struct fattr {
      ftype        type;
      unsigned int mode;
      unsigned int nlink;
      unsigned int uid;
      unsigned int gid;
      unsigned int size;
      unsigned int blocksize;
      unsigned int rdev;
      unsigned int blocks;
      unsigned int fsid;
      unsigned int fileid;
      timeval      atime;
      timeval      mtime;
      timeval      ctime;
};
```

Here the file type is 2 (directory), the mode is hex 000041ed, that is octal 40755, a directory with permissions rwxr-xr-x, etc.

Security

NFS is very insecure and can be used only in non-hostile environments.

The AUTH_UNIX authentication uses user id and group id. This is inconvenient: both machines must use the same IDs, and a security problem: someone who has root on his client laptop can give himself any uid and gid desired, and can subsequently access other people's files on the server.

Doing root stuff on the server may be more difficult. Typically user id `root` on the client is mapped to user id `nobody` on the server.

7.5 The proc filesystem

The proc filesystem is an example of a *virtual* filesystem. It does not live on disk. Instead the contents are generated dynamically when its files are read.

Sometimes users are worried that space on their disk is wasted by some huge core file

```
% ls -l /proc/kcore
-r--------    1 root     root     939528192 Sep 30 12:30 /proc/kcore
```

but this only says something about the amount of memory of the machine.

Many of the files in the proc filesystem have unknown size, since they first get a size when they are generated. Thus, a `ls -l` will show size 0, while `cat` will show nonempty contents. Some programs are confused by this.

```
% ls -l /proc/version
-r--r--r--    1 root     root            0 Sep 30 12:31 /proc/version
% cat /proc/version
Linux version 2.5.39 (aeb@mette) (gcc version 2.96) #14 Mon Sep 30 03:13:05 CEST 2002
```

The proc filesystem was designed to hold information about processes, but in the course of time a lot of other cruft was added. Most files are read-only, but some variables can be used to tune kernel behaviour (using `echo something > /proc/somefile`).

An example

It is easy to add an entry to the proc tree. Let us make `/proc/arith/sum` that keeps track of the sum of all numbers echoed to this file.

```
# insmod sum-module.o
# ls -l /proc/arith
total 0
dr-xr-xr-x    2 root     root            0 Sep 30 12:40 .
dr-xr-xr-x   89 root     root            0 Sep 30 12:39 ..
-r--r--r--    1 root     root            0 Sep 30 12:40 sum
# cat /proc/arith/sum
0
# echo 7 > /proc/arith/sum
# echo 5 > /proc/arith/sum
# echo 13 > /proc/arith/sum
# cat /proc/arith/sum
25
# rmmod sum-module
# ls -l /proc/arith
ls: /proc/arith: No such file or directory
#
```

How was this achieved? Here is the code.

```
/*
 * sum-module.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

static unsigned long long sum;

static int show_sum(char *buffer, char **start, off_t offset, int length) {
        int size;

        size = sprintf(buffer, "%lld\n", sum);
        *start = buffer + offset;
        size -= offset;
        return (size > length) ? length : (size > 0) ? size : 0;
}

/* Expect decimal number of at most 9 digits followed by '\n' */
static int add_to_sum(struct file *file, const char *buffer,
                      unsigned long count, void *data) {
        unsigned long val = 0;
        char buf[10];
        char *endp;

        if (count > sizeof(buf))
                return -EINVAL;
        if (copy_from_user(buf, buffer, count))
                return -EFAULT;
        val = simple_strtoul(buf, &endp, 10);
        if (*endp != '\n')
                return -EINVAL;
        sum += val;     /* mod 2^64 */
        return count;
}

static int __init sum_init(void) {
        struct proc_dir_entry *proc_arith;
        struct proc_dir_entry *proc_arith_sum;

        proc_arith = proc_mkdir("arith", 0);
        if (!proc_arith) {
                printk (KERN_ERR "cannot create /proc/arith\n");
                return -ENOMEM;
        }
        proc_arith_sum = create_proc_info_entry("arith/sum", 0, 0, show_sum);
        if (!proc_arith_sum) {
                printk (KERN_ERR "cannot create /proc/arith/sum\n");
                remove_proc_entry("arith", 0);
                return -ENOMEM;
        }
        proc_arith_sum->write_proc = add_to_sum;
        return 0;
}

static void __exit sum_exit(void) {
        remove_proc_entry("arith/sum", 0);
        remove_proc_entry("arith", 0);
}

module_init(sum_init);
module_exit(sum_exit);
MODULE_LICENSE("GPL");
```

seqfiles

In the old days routines generating a proc file would allocate a page and write their output there. A read call would then copy the appropriate part to user space. Typical code was something like

```
        /* read count bytes from proc file starting at file->f_pos */
        page = get_free_page(GFP_KERNEL);
        length = get_proc_info(page);
        if (file->f_pos < length) {
                if (count + file->f_pos > length)
                        count = length - file->f_pos;
                copy_to_user(buf, page + file->f_pos, count);
                file->f_pos += count;
        }
        free_page(page);
```

and clearly this works only when the entire content of a proc file fits within a single page. There were other problems as well. For example, when output is generated using `sprintf()`, it is messy to protect against buffer overflow.

Since 2.5.1 (and 2.4.15) some infrastructure exists for producing generated proc files that are larger than a single page. For an example of use, see the Fibonacci module.

The code

```
        f = create_proc_entry("foo", mode, NULL);
        if (f)
                f->proc_fops = &proc_foo_operations;
```

will create a file `/proc/foo` with given mode such that opening it yields a file that has `proc_foo_operations` as struct file_operations. Typically one has something like

```
static struct file_operations proc_foo_operations = {
        .open           = foo_open,
        .read           = seq_read,
        .llseek         = seq_lseek,
        .release        = seq_release,
};
```

where `foo_open` is defined as

```
static int foo_open(struct inode *inode, struct file *file)
{
        return seq_open(file, &foo_op);
}
```

and `foo_op` is a `struct seq_operations`:

```
struct seq_operations {
        void * (*start) (struct seq_file *m, loff_t *pos);
        void (*stop) (struct seq_file *m, void *v);
        void * (*next) (struct seq_file *m, void *v, loff_t *pos);
        int (*show) (struct seq_file *m, void *v);
};

struct seq_operations foo_op = {
        .start    = foo_start,
        .stop     = foo_stop,
        .next     = foo_next,
        .show     = foo_show
};
```

To make the proc file work, the foo module has to define the four routines `start()`, `stop()`, `next()`, `show()`. Each time some amount of data is to be read from the proc file, first `start(); show()` is done, then a number of times `next(); show()`, as long as more items fit in the user-supplied buffer, and finally `stop()`.

The start routine can allocate memory, or get locks or down semaphores, and the stop routine can free or unlock or up them again. The values returned by `start()` and `next()` are cookies fed to `show()`, and this latter routine generates the actual output using the specially provided output routines `seq_putc()`, `seq_puts()`, `seq_printf()`.

The routines `seq_open()` etc. are defined in `seq_file.c`. Here `seq_open()` initializes a `struct seq_file` and attaches it to the `private_data` field of the file structure:

```
struct seq_file {
        char *buf;
        size_t size;
        size_t from;
        size_t count;
        loff_t index;
        struct semaphore sem;
        struct seq_operations *op;
        void *private;
};
```

(Use a buffer `buf` of size `size`. It still contains `count` unread bytes, starting from buf offset `from`. We return a sequence of items, and `index` is the current serial number. The `private` pointer can be used to point at private data, if desired.)

As an example, look at `/proc/mounts`, defined in `proc/base.c`. The open routine here is slightly more complicated, basically something like

```
static int mounts_open(struct inode *inode, struct file *file)
{
        int ret = seq_open(file, &mounts_op);
        if (!ret) {
                struct seq_file *m = file->private_data;
                m->private = proc_task(inode)->namespace;
        }
}
```

and the start, next, stop, show routines live in `namespace.c`.

7.6 A baby filesystem example

Let us write a baby filesystem, as an example of how the Virtual File System works. It allows one to mount a block device and then shows the partition table as a file hierarchy, with partitions as files, and links in a chain of logical partitions as directories.

(It helps to know how partition tables work. Very briefly: Start at sector 0, the Master Boot Record. Each partition sector contains 4 descriptors, that either describe a partition, or point at the next partition sector. The partitions described in the MBR are called *primary*. The others are called *logical*. The box containing all logical partitions is called the *extended* partition.)

```
     # fdisk -lu /dev/hda
        Device Boot    Start       End     Blocks   Id  System
```

```
/dev/hda1   *  16450560  33736499   8642970    5  Extended
/dev/hda4            63  16450559   8225248+  63  GNU HURD or SysV
/dev/hda5      16450623  20547134   2048256   83  Linux
/dev/hda6      20547198  33656174   6554488+  83  Linux
/dev/hda7      33656238  33736499     40131   82  Linux swap
# insmod fdiskfs.o; mount -t fdiskfs /dev/hda /mnt
# ls -R /mnt
/mnt:
.  ..  1  2  3  4

/mnt/1:
.  ..  1  2  3  4

/mnt/1/2:
.  ..  1  2  3  4

/mnt/1/2/2:
.  ..  1  2  3  4
# cat /mnt/4
sectors 63-16450559, type 63
# cat /mnt/1/2/2/?
sectors 63-80324, type 82
empty slot
empty slot
empty slot
# ls -al /mnt
total 7
dr-xr-xr-x    3 root     root            6 1970-01-01 01:00 .
drwxr-xr-x   44 root     root         4096 2002-12-18 20:18 ..
dr-xr-xr-x    3 root     root            6 1970-01-01 01:00 1
-r--r--r--    1 root     root           16 1970-01-01 01:00 2
-r--r--r--    1 root     root           16 1970-01-01 01:00 3
-r--r--r--    1 root     root           16 1970-01-01 01:00 4
# umount /mnt; rmmod fdiskfs
```

We see a disk with four partition sectors. There are four actual partitions, and three links in the chain of logical partitions, and a lot of empty slots.

Below the code that does this. I wrote it this evening under 2.5.52 - correctness not guaranteed. Don't try it on a disk that is in use - obscure errors may result. (Now adapted to 2.6 kernels, otherwise just as silly as before.)

Exercise *Change in the code below the regular files into block device nodes giving access to the partition described by the partition descriptor.*

```
/*
 * fdiskfs.c
 */

#include <linux/module.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/genhd.h>
#include <linux/smp_lock.h>
#include <linux/buffer_head.h>
```

Three partition types indicate an extended partition, and will be treated as directory. The remaining types become regular files.

```
static inline int
is_extended_type(int type) {
        return (type == DOS_EXTENDED_PARTITION ||
                type == WIN98_EXTENDED_PARTITION ||
                type == LINUX_EXTENDED_PARTITION);
}
```

The structure of a DOS-type partition table entry.

```
typedef struct { unsigned char h,s,c; } chs;

struct fdisk_partition {
        unsigned char bootable;             /* 0 or 0x80 */
        chs begin_chs;
        unsigned char sys_type;
        chs end_chs;
        unsigned int start_sect;
        unsigned int nr_sects;
};
```

Read a partition sector from disk.

```
static struct buffer_head *
fdisk_read_sector(struct super_block *s, int sector) {
        struct buffer_head *bh;
        unsigned char *data;

        bh = sb_bread(s, sector);
        if (!bh) {
                printk ("fdiskfs: unable to read sector %d on dev %s\n",
                        sector, s->s_id);
        } else {
                data = (unsigned char *) bh->b_data;
                if (data[510] != 0x55 || data[511] != 0xaa) {
                        printk ("No aa55 signature on sector %d of dev %s\n",
                                sector, s->s_id);
                        brelse(bh);
                        bh = 0;
                }
        }
        return bh;
}
```

Invent some silly scheme of partition numbering. The assumption here is that chains do not fork.

```
/* inos: root: 1, primary: 2-5, elsewhere E+(H<<2)+(L<<4)
   E entry (0-3), H chain head (0-3), L chain length (1-max) */
#define ROOT_INO        1       /* must not be 0 */
#define PRIMARY_SHIFT   2

#define IS_ROOTDIR(a)   ((a) == ROOT_INO)
#define IS_PRIMARY(a)   ((a) < 4 + PRIMARY_SHIFT)

#define ROOT_SUB_INO            (PRIMARY_SHIFT)
```

```
#define PRIMARY_SUB_INO(p)      ((((p) - PRIMARY_SHIFT)<<2) + (1<<4))
#define OTHER_SUB_INO(p)        (((p) & ~3) + (1<<4))

static inline int
sub_ino(int ino, int pos) {
        return pos + (IS_ROOTDIR(ino) ? ROOT_SUB_INO :
                      IS_PRIMARY(ino) ? PRIMARY_SUB_INO(ino) :
                      OTHER_SUB_INO(ino));
}

static struct fdisk_partition *
fdiskfs_find_inode(struct super_block *s, int ino, struct buffer_head **abh) {
        int head, pos, depth;
        unsigned char *data;
        struct fdisk_partition *p;
        int sector, extd, i;

        if (IS_ROOTDIR(ino))
                return NULL;

        if (IS_PRIMARY(ino)) {
                pos = head = ino - PRIMARY_SHIFT;
                depth = 0;
        } else {
                pos = (ino & 3);
                head = ((ino >> 2) & 3);
                depth = (ino >> 4);
        }

        *abh = fdisk_read_sector(s, 0);
        if (!*abh)
                return NULL;
        data = (*abh)->b_data;
        p = (struct fdisk_partition *)(data + 446 + 16*head);

        if (depth == 0)
                return p;
        extd = sector = p->start_sect;

        for (;;) {
                brelse(*abh);
                *abh = fdisk_read_sector(s, sector);
                if (!*abh)
                        return NULL;
                data = (*abh)->b_data;
                p = (struct fdisk_partition *)(data + 446);

                if (--depth == 0)
                        return p+pos;

                for (i = 0; i< 4; i++)
                        if (p[i].nr_sects != 0 &&
                            is_extended_type(p[i].sys_type))
                                break;
                if (i == 4) {
                        brelse(*abh);
                        *abh = 0;
                        return NULL;
                }
                sector = extd + p[i].start_sect;
        }
}
```

So far the helper functions. Now VFS code. We need a routine to read a directory, one to lookup a name in a directory, and a routine to read a file.

```
static int
fdiskfs_readdir(struct file *filp, void *dirent, filldir_t filldir) {
        struct inode *dir;
        unsigned long offset, maxoff;
        int i, len, ino, dino, fino;
        int stored = 0;
        char name[3];

        lock_kernel();

        dir = filp->f_dentry->d_inode;
        dino = dir->i_ino;
        ino = sub_ino(dino, 0);
        offset = filp->f_pos;
        maxoff = 6;

        for(;;) {
                if (offset >= maxoff) {
                        offset = maxoff;
                        filp->f_pos = offset;
                        goto out;
                }
                filp->f_pos = offset;

                if (offset == 0) {
                        strcpy(name, ".");
                        fino = dino;
                } else if (offset == 1) {
                        strcpy(name, "..");
                        fino = parent_ino(filp->f_dentry);
                } else {
                        i = offset-2;
                        name[0] = '1' + i;
                        name[1] = 0;
                        fino = ino + i;
                }
                len = strlen(name);

                if (filldir(dirent, name, len, offset, fino, 0) < 0)
                        goto out;

                stored++;
                offset++;
        }
out:
        unlock_kernel();
        return stored;
}

static struct dentry *
fdiskfs_lookup(struct inode *dir, struct dentry *dentry,
               struct nameidata *nameidata) {
        struct inode *inode = NULL;
        const char *name;
        int dino, ino, len, pos;

        lock_kernel();

        name = dentry->d_name.name;
        len = dentry->d_name.len;
        if (len != 1 || *name < '1' || *name > '4')
                goto out;
        pos = *name - '1';

        dino = dir->i_ino;
        ino = sub_ino(dino, pos);
        inode = iget(dir->i_sb, ino);
 out:
        d_add(dentry, inode);
        unlock_kernel();
        return NULL;
}

static ssize_t
fdiskfs_read(struct file *filp, char *buf, size_t count, loff_t *ppos) {
        struct inode *inode;
        struct buffer_head *bh;
        struct fdisk_partition *p;
        int ino, len, offset;
        char file_contents[200];

        inode = filp->f_dentry->d_inode;
        ino = inode->i_ino;

        p = fdiskfs_find_inode(inode->i_sb, ino, &bh);
        if (!p)
                goto out;

        if (p->nr_sects == 0)
                sprintf(file_contents, "empty slot\n");
        else
                sprintf(file_contents, "sectors %d-%d, type %02X%s\n",
                        p->start_sect, p->start_sect + p->nr_sects - 1,
                        p->sys_type, p->bootable ? " boot" : "");
        brelse(bh);

        len = strlen(file_contents);

        offset = *ppos;
        if (offset >= len)
                return 0;
        len -= offset;
        if (len > count)
                len = count;

        if (copy_to_user(buf, file_contents+offset, len))
                return -EFAULT;
        *ppos += len;
        return len;
 out:
        return -EIO;
}

static struct file_operations fdiskfs_dir_operations = {
        .read           = generic_read_dir,
        .readdir        = fdiskfs_readdir,
};

static struct inode_operations fdiskfs_dir_inode_operations = {
        .lookup         = fdiskfs_lookup,
};

static struct file_operations fdiskfs_file_operations = {
        .read           = fdiskfs_read,
};
```

For the superblock operations we need a method to read an inode.

```
static void
fdiskfs_read_inode(struct inode *i) {
        struct buffer_head *bh = NULL;
        struct fdisk_partition *p;
        int ino, isdir;

        ino = i->i_ino;
        if (ino == ROOT_INO) {
                isdir = 1;
        } else {
                p = fdiskfs_find_inode(i->i_sb, ino, &bh);
                if (!p) {
                        printk("fdiskfs: error reading ino %d\n", ino);
                        return;
                }
                isdir = is_extended_type(p->sys_type);
                brelse(bh);
        }

        i->i_mtime.tv_sec = i->i_atime.tv_sec = i->i_ctime.tv_sec = 0;
        i->i_mtime.tv_nsec = i->i_atime.tv_nsec = i->i_ctime.tv_nsec = 0;
        i->i_uid = i->i_gid = 0;

        if (isdir) {
                i->i_op = &fdiskfs_dir_inode_operations;
                i->i_fop = &fdiskfs_dir_operations;
                i->i_mode = S_IFDIR + 0555;
                i->i_nlink = 3;         /* ., .., subdirs */
                i->i_size = 6;
        } else {
                i->i_fop = &fdiskfs_file_operations;
                i->i_mode = S_IFREG + 0444;
                i->i_nlink = 1;
                i->i_size = 16;
        }
}

static struct super_operations fdiskfs_ops = {
        .read_inode     = fdiskfs_read_inode,
};
```

For the struct file_system_type we need a method that reads the superblock.

```
static int
fdiskfs_fill_super(struct super_block *s, void *data, int silent) {
        struct buffer_head *bh;

        sb_set_blocksize(s, 512);
        s->s_maxbytes = 1024;

        bh = fdisk_read_sector(s, 0);
        if (!bh)
                goto out;
        brelse(bh);

        s->s_flags |= MS_RDONLY;
        s->s_op = &fdiskfs_ops;

        s->s_root = d_alloc_root(iget(s, ROOT_INO));
        if (!s->s_root)
                goto out;
        return 0;

out:
        return -EINVAL;
}

static int
fdiskfs_get_sb(struct file_system_type *fs_type, int flags,
               const char *dev_name, void *data, struct vfsmount *mnt) {
        return get_sb_bdev(fs_type, flags, dev_name, data,
                           fdiskfs_fill_super, mnt);
}

static struct file_system_type fdiskfs_type = {
        .owner          = THIS_MODULE,
        .name           = "fdiskfs",
        .get_sb         = fdiskfs_get_sb,
        .kill_sb        = kill_block_super,
        .fs_flags       = FS_REQUIRES_DEV,
};
```

Finally, the code to register and unregister the filesystem.

```
static int __init
init_fdiskfs(void) {
        return register_filesystem(&fdiskfs_type);
}

static void __exit
exit_fdiskfs(void) {
        unregister_filesystem(&fdiskfs_type);
}

module_init(init_fdiskfs)
module_exit(exit_fdiskfs)
MODULE_LICENSE("GPL");
```

That was all.

Next Previous Contents