Operating Systems and Architecture (CO527) Spring 2016

Implementing File Systems

Fred Barnes and Radu Grigore

Virtual File System

In the previous lecture, we saw that UNIX-like operating systems pretend there is only one tree of files. We can identify nodes of this tree by paths, which are strings. Regular files are leafs of this tree. Once we get a handle on a regular file, we can open it, read it, modify it, and close it. More interestingly, to the tree of files, we can add whole subtrees, by mounting real file systems. Let us peek under the hood to see how Linux implements these operations.

The following three data structures correspond to several concepts we saw:

Concept	Data Structure	Defined in
path step	struct dentry	include/linux/dcache.h
tree node	struct inode	include/linux/fs.h
open file	struct file	include/linux/fs.h

Recall that for a path like /a/b/c we said that a, b and c are path steps. In our tree picture, path steps correspond to labelled edges. Each dentry (**d**irectory **entry**) contains its label, information about its neighbours in the tree, and a pointer to the tree node it points to.

```
struct dentry {
  struct qstr d_name;  // label on the edge; e.g. "a" in the  /a/b/c example
  struct dentry * d_parent; // edge from above
  struct list_head d_child; // sibling edges
  struct list_head d_subdirs; // edges from below
  struct inode * d_inode; // tree node, at the low end of the edge
  // ...
};
```

Each inode contains information about a file, such as its owner, permissions, and modification times.

```
struct inode { kuid_t i_uid; kgid_t i_gid; /* ... */ };
```

A user space file handle is essentially a reference to a kernel struct file. Each file tracks the current offset:

```
struct file { struct inode * f_inode;  loff_t f_pos; /* ... */ };
```

These data structures give a uniform way to represent the tree of files; that is, the virtual file system. But, recall that this tree is a patchwork of subtrees that correspond to real file systems. How are these different file systems mixed? Each of the data structures from above remembers how to perform operations specific to the real file system they are a part of:

```
struct dentry { const struct dentry_operations * d_op; /* ... */ };
struct inode { const struct inode_operations * i_op; /* ... */ };
struct file { const struct file_operations * f_op; /* ... */ };
```

For example, the file operations include

```
struct file_operations {
  int (*open)(struct inode *, struct file *);
```

```
  ssize_t (*read)(struct file *, char *, size_t, loff_t *);
  int (*iterate)(struct file *, struct dir_context *);
  // ...
};
```

One could view this scheme as a way to do inheritance in C.

If one creates a file, for example, the inode operation create gets called. This operation will be specific to the real file system within which we create a file. The operation creates an inode, and populates its i_op field. If the new inode is a directory, then its create operation was likely set to be the same as the create operation of the parent. But, in the virtual file system we sometimes cross boundaries from one real file system to another real file system. This happens at *mount points*. So, let us talk briefly about mounting.

Each file system, such as ext3 or VFAT or MINIX, is described by a struct file_system_type.

```
struct file_system_type {
  const char * name;
  struct dentry * (*mount)(struct file_system_type *, int, const char *, void *);
  // ...
};
```

You can see the names of the registered file system types by looking at the file /proc/filesystem. (Incidentally, /proc is the mount point of a file system of type procfs, where the kernel publishes all kinds of useful information.) The main characteristic of a file system type, however, is not its name, but its mount function. The mount function is a way to obtain the dentry that is the root of a real file system, also known as its mount point. The third argument could be, for example, the string '/dev/sda1'. Apart from returning the root dentry, the mount function also creates an instance of struct super_block.

```
struct super_block { const super_operations * s_op; /* ... */ };
struct super_operations {
  struct inode * (*alloc_inode)(struct super_block * sb);
  void (*delete_inode)(struct inode *);
  void (*dirty_inode)(struct inode *, int);
  int (*write_inode)(struct inode *, int);
  // ...
};
```

There is one instance of struct super_block for each mounted file system. Data is actually written to the backing store by the super operation write_inode.

MINIX File System

Now let us look in some detail at how one particular file system is implemented. MINIX is not widely used — we look at it because its implementation is simple, in comparison with the other file system types. Most of the relevant files are in fs/minix/. Several data structures related to the physical layout are in include/uapi/linux/minix_fs.h.

The struct file_system_type for MINIX is declared in fs/minix/inode.c.

```
static struct file_system_type minix_fs_type {
  .name = "minix",
  .mount = minix_mount,
  .fs_flags = FS_REQUIRES_DEV,
  // ...
};
```

From here we find out that MINIX file systems require a device. A device is typically identified by an absolute path in the virtual file system. It could be something like the block special file /dev/sda1, or it could be a regular file. The function minix_mount is defined in the same file. It delegates most of the work to the generic

mount_bdev, where bdev stands for block device. The function mount_bdev is given a callback, the function minix_fill_super, which does the MINIX-specific work. Let us take a look at it.

The first thing it does is to read block 1 (not 0, which is for booting):

```
static int minix_fill_super(struct super_block * s, void * data, int silent) {
  // ...
  if (!(bh = sb_bread(s, 1))) goto bad_sb;
  ms = (struct minix_super_block *) bh->b_data;
```

where minix_super_block is defined in minix_fs.h:

```
struct minix_super_node {
  __u16 s_ninodes;
  __u16 s_nzones;
  // ...
};
```

There are two things to note here. First, we were able to call the function sb_read to get a buffer for block 1: this was possible because of the work done by mount_bdev. Most of the interaction with the backing store is done through these sb_* functions, which read/write whole blocks, and cache them in memory, for speed. A block is often (but not always) the same size as a page: 4 KiB. Second, struct minix_super_node defines the physical layout of the superblock. Indeed, the next thing that happens is that the information is copied from struct minix_super_node into the in-memory representation of a MINIX super node.

```
  sbi = kzalloc(sizeof(struct minix_sb_info), GFP_KERNEL);
  s->s_fs_info = sbi;
  // ...
  sbi->s_ninodes = ms->s_ninodes;
  sbi->s_nzones = ms->s_nzones;
  sbi->s_imap_zones = ms->s_imap_zones;
  sbi->s_zmap_zones = ms->s_zmap_zones;
  // ...
```

The next thing being done is that s_imap_zones + s_zmap_zones blocks are loaded in memory.

```
  block = 2;
  for (i = 0; i < sbi->s_imap_zones; i++, block++) sbi->s_imap[i] = sb_bread(s, block);
  for (i = 0; i < sbi->s_zmap_zones; i++, block++) sbi->s_zmap[i] = sb_bread(s, block);
```

Finally, the root inode is read.

```
  root_inode = minix_iget(s, MINIX_ROOT_NODE); // MINIX_ROOT_NODE is 1
  s->s_root = d_make_root(root_inode);
```

The function d_make_root returns a dentry, which is the one eventually returned by minix_mount. The function minix_iget first invokes iget_locked, which searches a global cache for inodes, and then if that fails it reads from the device using V2_minix_iget, which in turn invokes minix_V2_raw_inode. This latter function does some interesting offset computation:

```
struct minix2_inode *
minix_V2_raw_inode(struct super_block * sb, ino_t ino, struct buffer_head ** bh) {
  struct minix_sb_info * sbi = minix_sb(sb);
  int minix2_inodes_per_block = sb->s_blocksize / sizeof(struct minix2_inode);
  // ...
  ino--;
  int block = 2 + sbi->s_imap_blocks + sbi->s_zmap->blocks
    + ino / minix2_inodes_per_block;
  *bh = sb_bread(sb, block);
```

```
10    // ...
11    struct minix2_inode * p = (void*) (*bh)->b_data;
12    return p + ino % minix2_inodes_per_block;
13  };
```

OK, maybe not that interesting, but at least not completely trivial. You should definitely make sure you understand what happens above.

It turns out that the two groups of blocks at the beginning are bitmaps: the first s_imap_blocks track which inodes are used, the next s_zmap_blocks track which data blocks are used.

TODO: maybe include impl of some char device

Tracking Free Space

After so much code, let us step back into idea-land. File systems, like memory allocators, need to track which space is available and which space is used. What is specific to file systems is that allocation is done in big chunks. Yes, even a one-byte file occupies a whole block, which takes a few kilobytes. Because allocation is done in big chunks, file systems can use simpler methods of tracking free space.

TODO: more explanations

Consider a file system with 2^n data blocks, each holding 2^k bits. We can track if one of these data blocks is used with 1 bit. Thus, we need 2^n bits, which fit in 2^{n-k} extra blocks. For example, if there are 8192 data blocks, each of 1024 *bytes*, then we can track which blocks are used by using *one* extra block. This scheme is used by MINIX.

Alternatively, one can simply list the blocks that are free, in a list. Consider again a file system with 2^n data blocks, each holding 2^k bits; and *no* extra blocks this time. To identify a block, we need a number from $\{0, 1, 2, \ldots, 2^n - 1\}$, which we can store in n bits. Thus, in one block we can fit $\lfloor 2^k/n \rfloor$ block addresses. If there are fewer free blocks, then we can store all their addresses in one block; say, in block 0. If there are $\sim 2^n$ free blocks, then we need $\sim 2^n/(2^k/n) = 2^{n-k}/n$ blocks to keep track of free blocks. We can chain these blocks in a list. Thus, we store only $\lfloor 2^k/n \rfloor - 1$ free block addresses in one block: the extra address is of the next block in the free list. How does the free list end? For example, by saying that the next block in it is block 0, which we have designated as the head of the list.

Exercises

1. ► Most of the information in inodes is readily available via the command stat. Read 'man stat'.
2. ► What is an inode? What is a dentry? What is a super_block?
3. Since a file can be memory mapped via mmap, it must be that the virtual file system interacts with the virtual memory. How does this interaction work?
4. Read 'man mount'.
5. ► Consider a file system with 10^6 blocks, each of 1024 bytes: k blocks are reserved for a bitmap that tracks which data blocks are in use; the other $10^6 - k$ blocks are data blocks. What is the minimum value of k?
6. ► Consider a file system with total size 16 GiB, and blocks of 1 KiB. It tracks free space using the free list method. How many block addresses fit in one block?
7. Which method for tracking free space in a file system would you use? Why? [Hint (not answer!): What happens to the free list when the file system fills up?]

References

[1] Linux kernel 4.2.0, Documentation/filesystems/vfs.txt, fs/*, and fs/minix/*