The Ext2 Filesystem

CSCI780: Linux Kernel Internals
Spring 2003

Josh Glover

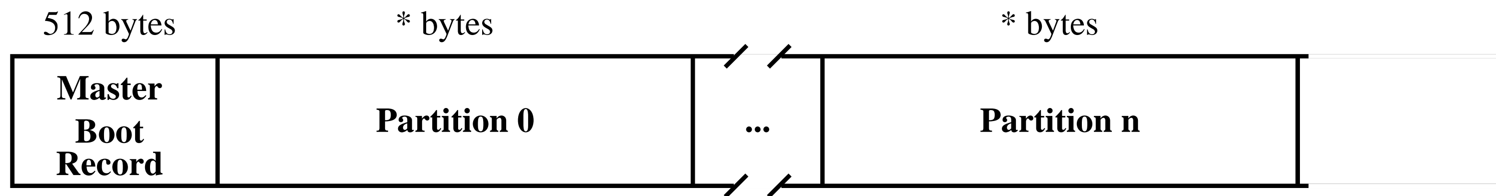
Overview

- General organisation of disks and filesystems
- Creating filesystems
- Mounting and unmounting filesystems
- Creating and deleting files (`open()` and `close()` are covered here)
- `read()`ing and `write()`ing and `lseek()`ing *...oh my!*

Neat Stuff that I Don't Have Time to Cover

- Transparent encryption
- Data block preallocation
- Filesystem Structure Caching (really a VFS topic)
- Read-ahead Caching

Organisation of a Disk



A partition must be a multiple of $cylinder_size$ bytes.

The Master Boot Record

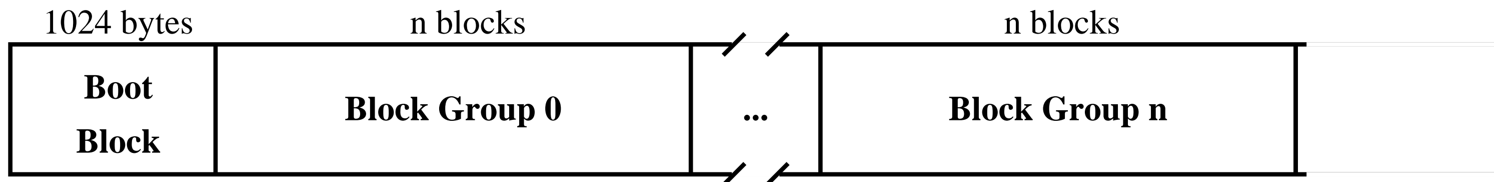
The Master Boot Record (MBR) lives at offset `0x0000` on each disk, and contains a program that starts the bootstrap process of the operating system. Each operating system tends to have its own MBR program (and Linux has two: `lilo` and GRUB), but they must fit in 512 bytes of disk (GRUB cheats, see link below), and they all do this:

1. Try to read the partition table, at offset `0x01be`
2. Look for a bootable partition
3. Read the boot block of said partition into memory offset `0x0000:7c00`
4. Jump to memory offset `0x0000:7c00`

If you are interested in the MBR, you can find more information:

http://www.ata-atapi.com/hiwmbr.htm
http://www.gnu.org/manual/grub/html_node/Bootstrap-tricks.html#Bootstrap%20tricks

Organisation of an Ext2 Filesystem



- The maximum size of a block group is $8 * block_size$, since the block bitmap must fit in one block (see block group figure, next slide). This means that the maximum number of block groups in any partition is: $partition_size/(8 * block_size)$
- All block groups are the same size, and addressed sequentially, so the kernel can find the offset of a block group in the partition quickly:
  $block_group_size * grp_index$
- The kernel tries to keep all data blocks belonging to a file in the same block group, which allows for speedy access (both sequential and random)

Organisation of a Block Group

1024 or 3072 bytes	n blocks	1 block	1 block	n blocks	n blocks
Super Block	**Group Descriptors**	**Block Bitmap**	**Inode Bitmap**	**Inode Table**	**Data Blocks**

- The kernel only uses the superblock and group descriptors from the first block group, even though superblock and group descriptors may live at the beginning of many or all block groups in the filesystem. The extra superblocks and group descriptors (which live in block groups 1 and powers of 3, 5, and 7) are simply backup copies, and can be used by the `e2fsck` program to restore a filesystem to a consistent state following a system crash or power outage. The backup copies are never touched by the kernel, but can be restored by `e2fsck -b superblock` if the primary superblock ever gets corrupted.
- You can use `tune2fs -s 0` to revert to the classic Ext2 behaviour of storing copies of the superblock at the beginning of each block group. You ***must*** run `e2fsck` immediately afterwards, or you will have an invalid filesystem.

Creating an Ext2 Filesystem

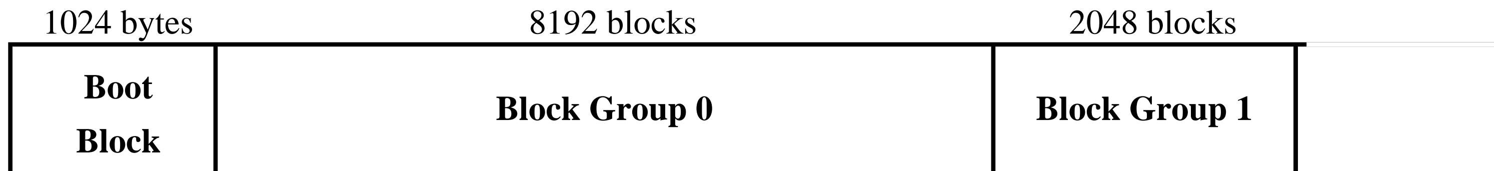
Let's create a filesystem with a block size of 1024 bytes on a 10MB "partition".

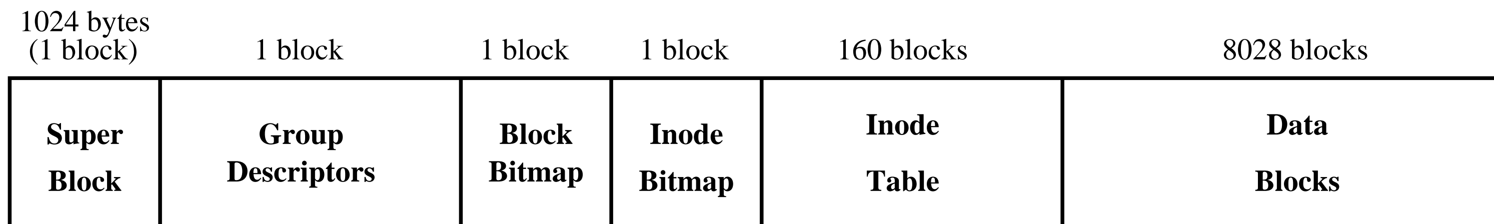
We use our formula for the maximum size of a block group:

$$8 * block_size \rightarrow 8 * 1024 = 8192$$

Therefore, we should two block groups, since

$$partition_size/(8 * block_size) \rightarrow 10240/8192 = 2$$





```
: jmglov@delyana; dd if=/dev/zero of=/tmp/test-01.fs bs=1k count=10240
10240+0 records in
10240+0 records out
: jmglov@delyana; mke2fs -b 1024 -F -L 'test-01' -v /tmp/test-01.fs
mke2fs 1.32 (09-Nov-2002)
Filesystem label=test-01
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
2560 inodes, 10240 blocks
512 blocks (5.00%) reserved for the super user
First data block=1
2 block groups
8192 blocks per group, 8192 fragments per group
1280 inodes per group
Superblock backups stored on blocks:
        8193
```

Now, what actually happened when we ran `mke2fs`? To find out, we need to take a look at the `e2fsprogs` source, in `misc/mke2fs.c` (I am using version 1.32–we use 1.27 in the department).

```
int main (int argc, char *argv[])
{
  errcode_t retval = 0;                                  errcode_t  is just a typedef long
  ext2_filsys fs;                                        Implementation of "Organisation of a Filesystem" figure
 . . . . . . . . . . .

  PRS(argc, argv);                                       Parse the args and set fs options in param
                                                         param  is an ext2_super_block  struct
  retval = ext2fs_initialize(device_name, 0, &param,     Create the superblock
     unix_io_manager, &fs);
 . . . . . . . . . . .                                   Bail on error, of course
```

The Ext2 Superblock

Before we see how the superblock is created, let's look at the actual data structure, in Linux kernel source file `include/linux/ext2_fs.h`:

```
struct ext2_super_block {
  __u32   s_inodes_count;      Total number of inodes
  __u32   s_blocks_count;      Size of filesystem, in blocks
  __u32   s_r_blocks_count;    Number of blocks reserved for the superuser--5% by default -m
  __u32   s_free_blocks_count; Number of free blocks
  __u32   s_free_inodes_count; Number of free inodes
  __u32   s_first_data_block;  Index of first useful block (always 0 or 1, because of the boot block)
  __u32   s_log_block_size;    Block size as a power of 2 (0 is 2^10, 1 is 2^11, 2 is 2^12)
  __s32   s_log_frag_size;     Fragments are not yet implemented in Ext2. Try Ext3.
  __u32   s_blocks_per_group;  Size of block group, in blocks
  __u32   s_frags_per_group;
  __u32   s_inodes_per_group;  Number of inodes in a group -i
  __u32   s_mtime;             Time of last mount operation, as a Unix epoch time
  __u32   s_wtime;             Time of last write operation, as a Unix epoch time
  __u16   s_mnt_count;         Number of times mounted since last fsck
  __s16   s_max_mnt_count;     When s_mnt_count reaches this value, fsck and reset
```

```
  __u16   s_magic;              Magic number to prove that this is a superblock--more on this later
  __u16   s_state;              0: mounted or not cleanly unmounted; 1: unmounted; 2: errors!
  __u16   s_errors;             1: continue; 2: remount read-only; 3: kernel panic
  __u16   s_minor_rev_level;    Minor revision; 0 in practice
  __u32   s_lastcheck;          Date of last fsck , as a Unix epoch time
  __u32   s_checkinterval;      Interval, in seconds, between fsck  (Defaults to six months)
  __u32   s_creator_os;         Linux is 0
  __u32   s_rev_level;          Major revision; 1, since we use Ext2 1.0 filesystems
  __u16   s_def_resuid;         Numeric userid for reserved blocks; 0 (root)
  __u16   s_def_resgid;         Numeric groupid for reserved blocks; 0 (root)
  __u32   s_first_ino;          Number of first unreserved inode; seems to be 11 in practice
  __u16   s_inode_size;         Size of on-disk inode structure; seems to be 128 in practice
  __u16   s_block_group_nr;     Block group number of this superblock
 . . . . . . . . . . .          Compatibility crap; deleted for your protection
  __u8    s_uuid[16];           128-bit Universally Unique Identifier
  char    s_volume_name[16];    Volume name (label); up to 16 characters
  char    s_last_mounted[64];   Pathname of last mount
  __u32   s_algorithm_usage_bitmap;  Used for compression
  __u8    s_prealloc_blocks;    Number of blocks to preallocate for file
  __u8    s_prealloc_dir_blocks;   Number of blocks to preallocate for directories
  __u16   s_padding1;           Align to word
  __u32   s_reserved[204];      Pad out to 1024 or 3072 bytes
};
```

e2fsprogs lib/ext2fs/initialize.c:ext2fs_initialize()

```
errcode_t ext2fs_initialize(const char *name, int flags,
    struct ext2_super_block *param,
    io_manager manager, ext2_filsys *ret_fs)
{
  ext2_filsys fs;                                   We don't want to return an invalid filesys...
  errcode_t retval;
  struct ext2_super_block *super;                   ...or an invalid superblock
 . . . . . . . . . . . .
  retval = ext2fs_get_mem(sizeof(struct struct_ext2_filsys),
                          (void **) &fs);           Just a wrapper around malloc()
 . . . . . . . . . . . .
  memset(fs, 0, sizeof(struct struct_ext2_filsys)); Zero out that memory
 . . . . . . . . . . . .
  retval = ext2fs_get_mem(SUPERBLOCK_SIZE, (void **) &super);  Grab enough memory for a superblock
 . . . . . . . . . . . .
  fs->super = super;                                Use super as an alias

  memset(super, 0, SUPERBLOCK_SIZE);                Zero it out
```

```
#define set_field(field, default) (super->field = param->field ? \ use the field from param
                                  param->field : (default))        or if unset, the default

  super->s_magic = EXT2_SUPER_MAGIC;                     On ix86: 0x53EF  (because ix86 is little endian)
  super->s_state = EXT2_VALID_FS;                        Set state to 0 (mounted or not cleanly unmounted)

  set_field(s_log_block_size, 0);                        Default blocksize is 1024 bytes
  set_field(s_log_frag_size, 0);                         So is default fragment size (this means nothing)
  set_field(s_first_data_block, super->s_log_block_size ? 0 : 1);   bs > 1024: 1st datblk 0
  set_field(s_max_mnt_count, EXT2_DFL_MAX_MNT_COUNT); Default is 20 mounts
  set_field(s_errors, EXT2_ERRORS_DEFAULT);              Default is 1 (continue)
 . . . . . . . . . . .
  set_field(s_rev_level, EXT2_GOOD_OLD_REV);             Default is 0, but most of us use Ext2fs 1.x
  if (super->s_rev_level >= EXT2_DYNAMIC_REV) {          If we are using Ext2fs 1.x or later...
    set_field(s_first_ino, EXT2_GOOD_OLD_FIRST_INO);    ...first inode will be 11
    set_field(s_inode_size, EXT2_GOOD_OLD_INODE_SIZE);   and inode size will be 128

  set_field(s_checkinterval, EXT2_DFL_CHECKINTERVAL);    Default is 15552000 sec, or about six months
  super->s_mkfs_time = super->s_lastcheck = time(NULL);

  super->s_creator_os = CREATOR_OS;                      Linux is 0

  fs->blocksize = EXT2_BLOCK_SIZE(super);               EXT2_MIN_BLOCK_SIZE << (s)->s_log_block_size
 . . . . . . . . . . .                                   Fragment crap, who cares?
```

```
  set_field(s_blocks_per_group, fs->blocksize * 8);
  if (super->s_blocks_per_group > EXT2_MAX_BLOCKS_PER_GROUP(super))
    super->s_blocks_per_group = EXT2_MAX_BLOCKS_PER_GROUP(super);
  super->s_frags_per_group = super->s_blocks_per_group * frags_per_block;

  super->s_blocks_count = param->s_blocks_count;
  super->s_r_blocks_count = param->s_r_blocks_count;
  if (super->s_r_blocks_count >= param->s_blocks_count) {
    retval = EXT2_ET_INVALID_ARGUMENT;
    goto cleanup;
  }

  /* If we're creating an external journal device, we don't need
   * to bother with the rest. */

retry:
  fs->group_desc_count = (super->s_blocks_count -
    super->s_first_data_block +
    EXT2_BLOCKS_PER_GROUP(super) - 1)
    / EXT2_BLOCKS_PER_GROUP(super);
  if (fs->group_desc_count == 0)
    return EXT2_ET_TOOSMALL;
```

*blocksize * 8 blocks/group, up to* 2ˆ16 *(GDT limit)*

Set up block count and reserved block count

Make sure reserved block count is not higher than block count

Ext3 bails here

Compute number of block groups

group_desc_count is misleading, it is actually the number of block groups

Make sure we have at least one block group

Code	Comment
`fs->desc_blocks = (fs->group_desc_count + EXT2_DESC_PER_BLOCK(super) - 1) / EXT2_DESC_PER_BLOCK(super);`	*Calculate how many blocks we need for group descriptors; each block group has one (32 byte) descriptor blocksize / size of group descriptor*
`i = fs->blocksize >= 4096 ? 1 : 4096 / fs->blocksize;`	*Default is one inode for every 4096 bytes*
`set_field(s_inodes_count, super->s_blocks_count / i);`	*same as* `-T news` *option to* `mke2fs`
. . . .	
`if (super->s_inodes_count < EXT2_FIRST_INODE(super)+1)`	*Make sure we have at least 11 inodes*
`super->s_inodes_count = EXT2_FIRST_INODE(super)+1;`	*(the minimum number needed for the filesystem)*
. . . .	
`ipg = (super->s_inodes_count + fs->group_desc_count - 1) / fs->group_desc_count;`	*Compute inodes-per-blockgroup*

Following this is a bevy of sanity checks and optimisations:

- Make sure we don't allocate more inodes than a one block inode bitmap can account for
- Make sure we don't exceed the limit of inodes per group: 65536 - (blocksize / inode size in bytes), because a group descriptor's free inode counter is only 16 bits
- Make sure we don't exceed the limit of inodes in a filesystem (number of inodes per group * number of block groups)
- Make sure the number of inodes per group completely fills the inode table blocks in the descriptor. If not, add some additional inodes per group.
- Finally, make sure the number of inodes per group is a multiple of 8 (to simplify the bitmap splicing code)

After all this, the inode count is adjusted to reflect the new inodes-per-group count.

```
overhead = (int) (3 + fs->desc_blocks + fs->inode_blocks_per_group);
. . . . . . . . . . .
rem = (int) ((super->s_blocks_count - super->s_first_data_block) %
  super->s_blocks_per_group);
if ((fs->group_desc_count == 1) && rem && (rem < overhead))
  return EXT2_ET_TOOSMALL;
if (rem && (rem < overhead+50)) {
  super->s_blocks_count -= rem;
  goto retry;
}
. . . . . . . . . . .
retval = ext2fs_allocate_block_bitmap(fs, buf, &fs->block_map);
. . . . . . . . . . .
retval = ext2fs_allocate_inode_bitmap(fs, buf, &fs->inode_map);
. . . . . . . . . . .
retval = ext2fs_get_mem((size_t) fs->desc_blocks * fs->blocksize,
  (void **) &fs->group_desc);
```

Annotations:

- *Number of book-keeping blocks per group (superblock backup, group descriptor backups, inode bitmap, block bitmap, and inode table)*
- *If the last group isn't big enough for the book-keeping blocks...*
- *If there is only one group...*
- *...bail with a filesystem too small error*
- *Otherwise...*
- *...just toss the last group and try again*
- *Allocate memory for the block and inode bitmaps*
- *Allocate memory for the group descriptor*

Now, we iterate through the block groups, filling in the group descriptor, block bitmap, and inode bitmap.

Here is the `ext2_group_desc` struct, from `include/linux/ext2_fs.h`:

```
struct ext2_group_desc
{
  __u32   bg_block_bitmap;            Block number of block bitmap
  __u32   bg_inode_bitmap;            Block number of inode bitmap
  __u32   bg_inode_table;             Block number of beginning of inode table
  __u16   bg_free_blocks_count;       Number of free blocks (in this group)
  __u16   bg_free_inodes_count;       Number of free inodes (in this group)
  __u16   bg_used_dirs_count;         Number of directories (in this group)
  __u16   bg_pad;                     Alignment to word
  __u32   bg_reserved[3];             Pad struct to 32 bytes
};
```

The last thing that `ext2fs_initialize()` must do before returning is to mark the superblock, block bitmap, and inode bitmap as dirty:

```
ext2fs_mark_super_dirty(fs);
ext2fs_mark_bb_dirty(fs);
ext2fs_mark_ib_dirty(fs);
```

Now, back in `misc/mke2fs.c`:

```
uuid_generate(fs->super->s_uuid);
. . . . . . . . . . .
/* Add "jitter" to the superblock's check interval so that we
 * don't check all the filesystems at the same time.  We use a
 * kludgy hack of using the UUID to derive a random jitter value. */
. . . . . . . . . . .
/* Override the creator OS, if applicable */
. . . . . . . . . . .
/* Set the volume label... */
. . . . . . . . . . .
/* Set the last mount directory */
```

Generate the Universally Unique Identifier (sucks from `/dev/urandom` *if it exists, otherwise uses time)*

i.e. zero out `s_last_mounted`

Finally, the data blocks are zeroed out, and `mke2fs` is done, leaving us with a valid filesystem:

```
: jmglov@delyana; ~/bin/tune2fs -l /tmp/test-01.fs
tune2fs 1.32 (09-Nov-2002)
Filesystem volume name:   test-01
Last mounted on:          <not available>
Filesystem UUID:          a4657a50-53f9-4c6a-8517-5f042ac7d795
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      dir_index filetype sparse_super
Default mount options:    (none)
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              2560
Block count:              10240
Reserved block count:     512
Free blocks:              9898
Free inodes:              2549
First block:              1
Block size:               1024
```

```
Fragment size:            1024
Blocks per group:         8192
Fragments per group:      8192
Inodes per group:         1280
Inode blocks per group:   160
Filesystem created:       Sat Mar 29 09:04:30 2003
Last mount time:          n/a
Last write time:          Sat Mar 29 09:04:30 2003
Mount count:              0
Maximum mount count:      21
Last checked:             Sat Mar 29 09:04:30 2003
Check interval:           15552000 (6 months)
Next check after:         Thu Sep 25 10:04:30 2003
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               128
Default directory hash:   tea
Directory Hash Seed:      5ec1b5ea-7286-48e7-ba79-d39078a18850
```

I am using a hacked-up version of `tune2fs`, so I can verify that a backup superblock was made in block group 1, as I expected:

```
: jmglov@delyana; ~/bin/tune2fs -l -b 8193 /tmp/test-01.fs
tune2fs 1.32 (09-Nov-2002)
Filesystem volume name:   test-01
 . . . . . . . . . . .
```

If you want the patch, you can grab it here:

```
http://www.jmglov.net/src/e2fsprogs-1.32_tune2fs_alt-superblock.patch
```

Another fun (and sometimes, very useful) trick is to open up the filesystem in a hex editor (make ***sure*** it is not mounted first!):

```
: jmglov@delyana; hexedit /tmp/test-01.fs
```

Since the boot block is always going to take up the first 1024 bytes of the filesystem, we can find the primary superblock at offset 0x400:

```
                 s_inodes_count        s_blocks_count        s_r_blocks_count     s_free_blocks_count
                     (2560)                (10240)                 (512)                (9898)
00000400     00 0A 00 00   00 28 00 00   00 02 00 00   AA 26 00 00    .....(.......&..
              s_free_inodes_count   s_first_data_block    s_log_block_size     s_log_frag_size
                     (2549)                  (1)                   (0)                  (0)
00000410     F5 09 00 00   01 00 00 00   00 00 00 00   00 00 00 00    ................
              s_blocks_per_group    s_frags_per_group     s_inodes_per_group          s_mtime
                     (8192)                (8192)                 (160)       (0 == 1970-01-01 UTC)
00000420     00 20 00 00   00 20 00 00   00 05 00 00   00 00 00 00    . ... .........
                   s_wtime         s_mnt_count s_max_mnt_count s_magic  s_state    s_errors  s_minor_rev_level
           (Sat Mar 29 09:04:30 2003)  (0)          (21)                  (1)   (1 == continue)    (0)
00000430     EE A7 85 3E   00 00 15 00   53 EF 01 00   01 00 00 00    ...>....S.......
                 s_lastcheck            s_checkinterval        s_creator_os          s_rev_level
           (Sat Mar 29 09:04:30 2003) (15552000s == 6 months)  (0 == Linux)               (1)
00000440     EE A7 85 3E   00 4E ED 00   00 00 00 00   01 00 00 00    ...>.N..........
          s_def_resuid s_def_resgid        s_first_ino     s_inode_size s_block_group_nr
          (0 == root)  (0 == root)             (11)            (128)           (0)     s_feature_compat
00000450     00 00 00 00   0B 00 00 00   80 00 00 00   20 00 00 00    ............ ...
              s_feature_incompat    s_feature_ro_compat                     s_uuid ...

00000460     02 00 00 00   01 00 00 00   A4 65 7A 50   53 F9 4C 6A    .........ezPS.Lj
                             ... s_uuid                           s_volume_name ...

00000470     85 17 5F 04   2A C7 D7 95   74 65 73 74   2D 30 31 00    .._.*...test-01.
                        ... s_volume_name                            s_last_mounted ...

00000480     00 00 00 00   00 00 00 00   00 00 00 00   00 00 00 00    ................
                             ... s last mounted  continues to  0x4C8
```

With a blocksize of 1024 or 2048 bytes, the superblock is padded out to 1024 bytes, but with a blocksize of 4096 bytes, it is padded out to 3072 bytes. This means that for a 1024-byte blocksize, the boot block and the superblock together will take up two blocks, whereas for a 2048- or 4096-byte blocksize, they will only take up one block.

Mounting an Ext2 Filesystem

Of course, for a filesystem to be of much use, it must be mounted:

```
: jmglov@delyana; sudo mount -vv -t ext2 -o loop /tmp/test-01.fs /tmp/mnt/test-01
mount: going to use the loop device /dev/loop0
set_loop(/dev/loop0,/tmp/test-01.fs,0): success
mount: setup loop device successfully
/tmp/test-01.fs on /tmp/mnt/test-01 type ext2 (rw,loop=/dev/loop0)
```

Mounting a filesystem is done by the `mount` program, which is, on my system, part of the `util-linux-2.11y` package. Let's dive into `util-linux-2.11y/mount/mount.c` to see how mounting works:

```
int main (int argc, char *argv[]) {
 . . . . . . . . . . .                 Lots of code dealing with option parsing, mainly
  case 2:                              We care about this case
    /* mount [-nfrvw] [-t vfstype] [-o options] special node */
 . . . . . . . . . . .
    result = mount_one (spec, node, types, NULL, options, 0, 0);
    break;
```

mount_one() → try_mount_one() → guess_fstype_and_mount() → do_mount_syscall()

And finally, we are in the kernel, fs/super.c:sys_mount():

```
asmlinkage long sys_mount(char * dev_name, char * dir_name, char * type,
  unsigned long flags, void * data)
{
  int retval;
  unsigned long type_page, dev_page, data_page;          VFS type, device, mount options
  char *dir_page;                                        Mountpoint

  retval = copy_mount_options (type, &type_page);        Grab VFS type from userland
 . . . . . . . . . . .
  dir_page = getname(dir_name);                          Grab mountpoint from userland
 . . . . . . . . . . .
  retval = copy_mount_options (dev_name, &dev_page);     Grab device from userland
 . . . . . . . . . . .
  retval = copy_mount_options (data, &data_page);        Grab mount options from userland
 . . . . . . . . . . .
  lock_kernel();
  retval = do_mount((char*)dev_page, dir_page, (char*)type_page,
    flags, (void*)data_page);
```

fs/super.c:do_mount()

```
long do_mount(char * dev_name, char * dir_name, char *type_page,
  unsigned long flags, void *data_page)
{
  struct file_system_type * fstype;
  struct nameidata nd;
  struct vfsmount *mnt = NULL;
  struct super_block *sb;
  int retval = 0;

  /* Discard magic */
  if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
    flags &= ~MS_MGC_MSK;

  /* Basic sanity checks */

  if (!dir_name || !*dir_name || !memchr(dir_name, 0, PAGE_SIZE))
    return -EINVAL;
  if (dev_name && !memchr(dev_name, 0, PAGE_SIZE))
    return -EINVAL;
```

Some sort of deprecated option

Make sure the both mountpoint and device are not null and that they are null-terminated

Make sure that the mountpoint is not the empty string

```
/* OK, looks good, now let's see what do they want */
. . . . . . . . . . .
/* For the rest we need the type */

if (!type_page || !memchr(type_page, 0, PAGE_SIZE))
  return -EINVAL;

/* for the rest we _really_ need capabilities... */
if (!capable(CAP_SYS_ADMIN))
  return -EPERM;

/* ... filesystem driver... */
fstype = get_fs_type(type_page);
if (!fstype)
  return -ENODEV;
```

Handle `-o remount` *and* `-o bind` *options*

Make sure that VFS type is a valid, null-terminated string

Normally, only the superuser may mount filesystems

This will be Ext2fs in our case

To grab the filesystem “driver”, we jump to `get_fs_type()`, still in `fs/super.c`:

```
struct file_system_type *get_fs_type(const char *name)
{
  struct file_system_type *fs;

  read_lock(&file_systems_lock);
  fs = *(find_filesystem(name));
```

Following the White Rabbit a bit further, `find_filesystem()`:

```
static struct file_system_type **find_filesystem(const char *name)
{
  struct file_system_type **p;
  for (p=&file_systems; *p; p=&(*p)->next)
    if (strcmp((*p)->name,name) == 0)
      break;
  return p;
}
```

As long as the filesystem is registered with the kernel (`fs/super.c:register_filesystem()` does this), we should now have a valid `file_system_type`.

Back in `do_mount()`:

```
/* ... and mountpoint. Do the lookup first to force automounting. */
if (path_init(dir_name,                         The kernel automounter is beyond our scope
  LOOKUP_FOLLOW|LOOKUP_POSITIVE|LOOKUP_DIRECTORY, &nd))
  retval = path_walk(dir_name, &nd);
if (retval)
  goto fs_out;

/* get superblock, locks mount_sem on success */
if (fstype->fs_flags & FS_NOMOUNT)             Some filesystems may only be mounted by the kernel
  sb = ERR_PTR(-EINVAL);                        e.g. pipefs, shm
else if (fstype->fs_flags & FS_REQUIRES_DEV)   This is the normal case
  sb = get_sb_bdev(fstype, dev_name, flags, data_page);
else if (fstype->fs_flags & FS_SINGLE)         Some filesystems may have only one superblock
  sb = get_sb_single(fstype, flags, data_page);   e.g. procfs
else                                            But lots of filesystems don't need a device at all
  sb = get_sb_nodev(fstype, flags, data_page);    e.g. nfs, smbfs

retval = PTR_ERR(sb);
if (IS_ERR(sb))                                 If get_sb_foo() failed, bail
  goto dput_out;
```

Even though our example uses a filesystem that is not on a device, the `-o loop` option to `mount` associates our `test-01.fs` file with the `/dev/loop0` block device. This is necessary because Ext2 registers itself with the `FS_REQUIRES_DEV` flag.

So `get_sb_bdev()` gets called, which does some device sanity checking, then tries to find a `super_block` struct of the proper type (in this case, Ext2) by calling `get_super()` (see next slide). Failing this, it must call `read_super()` to read the `super_block` struct (see VFS slides; substitute `ext2_read_super()` for `minix_read_super()`).

Now, we should be ready to actually mount the filesystem.

```
struct super_block * get_super(kdev_t dev)     Not a /dev/foo  device;
{                                                a kind of unique ID for kernel drivers
 . . . . . . . . . . .                       The usual boring sanity checks ;)
restart:
  s = sb_entry(super_blocks.next);           A special invocation of list_entry() ; see below
  while (s != sb_entry(&super_blocks))       Walk through the list of VFS superblocks
    if (s->s_dev == dev) {                   If devices match, we have found the right list item...
      /* Yes, it sucks. As soon as we get refcounting... */
      lock_super(s);                         ...but there is a race condition
      unlock_super(s);
      if (s->s_dev == dev)                   If the devices still match, return the VFS superblock
        return s;
      goto restart;
    } else
      s = sb_entry(s->s_list.next);
  return NULL;
}
```

The `sb_entry` macro is found in `include/linux/fs.h`:

```
#define sb_entry(list)  list_entry((list), struct super_block, s_list)
```

Back in `do_mount()`:

```
  while(d_mountpoint(nd.dentry) && follow_down(&nd.mnt, &nd.dentry));
 . . . . . . . . . . . .                        If something is already mounted here, fail with -EBUSY
 . . . . . . . . . . . .                        If the mountpoint doesn't exist, fail with -ENOENT
  down(&nd.dentry->d_inode->i_zombie);          Semaphore P()  AKA wait()
  if (!IS_DEADDIR(nd.dentry->d_inode)) {        Make sure the mountpoint inode is alive
    retval = -ENOMEM;
    mnt = add_vfsmnt(&nd, sb->s_root, dev_name);   Attempt to mount the filesystem
  }
  up(&nd.dentry->d_inode->i_zombie);            Semaphore V()  AKA signal()
  if (!mnt)                                     If add_vfsmnt() failed, fail with -ENOMEM
    goto fail;                                  because we must be out of memory
  retval = 0;
unlock_out:
  up(&mount_sem);                               This semaphore was grabbed back in get_sb_bdev
dput_out:
  path_release(&nd);                            "Close" the mountpoint's dentry
fs_out:
  put_filesystem(fstype);                       Release the filesystem driver
  return retval;
fail:
  kill_super(sb);                               Give up the memory that we had allocated for the superblock
  goto unlock_out;
}
```

If `do_mount` succeeds, the successful return value percolates back down all the way back to the shell that originally invoked `mount` (yes, I am simplifying; if you really care about the error handling, UTSL).

`http://info.astrian.net/jargon/terms/u/UTSL.html`

Unmounting an Ext2 Filesystem

When you are done with a filesystem, you should unmount it (if you don't, the `init` process certainly will on system shutdown):

```
: jmglov@delyana; sudo umount -vv /tmp/mnt/test-01/
Trying to umount /tmp/mnt/test-01
/tmp/test-01.fs umounted
del_loop(/dev/loop0): success
```

As you might suspect, `umount` is also part of the `util-linux-2.11y` package, and is implemented by `util-linux-2.11y/mount/umount.c`. We will not look at the source, but here is the bread-crumb trail of function calls that leads us into the kernel:

`umount_file()` → `umount_one()` → `umount()` syscall

`sys_umount()` lives in `fs/super.c`:

```
asmlinkage long sys_umount(char * name, int flags)
{
  struct nameidata nd;
  char *kname;
  int retval;

  kname = getname(name);          Grab the mountpoint from userland
 . . . . . . . . . . . .          Sanity checks on mountpoint string
  if (path_init(kname, LOOKUP_POSITIVE|LOOKUP_FOLLOW, &nd))
     retval = path_walk(kname, &nd);    See:
  putname(kname);                 http://www.jmglov.net/docs/kernel-hacking.html#pathwalk
 . . . . . . . . . . . .          A plethora of sanity checks
  down(&mount_sem);               Do the synchronisation samba...
  lock_kernel();
  retval = do_umount(nd.mnt, flags);    ...and call the workhorse function
  unlock_kernel();                Release all of your synch structures
  path_release(&nd);
  up(&mount_sem);
  goto out;
dput_and_out:
  path_release(&nd);
out:
  return retval;
}
```

fs/super.c:do_umount()

```
static int do_umount(struct vfsmount *mnt, int flags)
{
  struct super_block * sb = mnt->mnt_sb;
  struct nameidata parent_nd;
```

Just in case you thought kernel hacking was carefree...

```
  /* No sense to grab the lock for this test, but test itself looks
   * somewhat bogus. Suggestions for better replacement?
   * Ho-hum... In principle, we might treat that as umount + switch
   * to rootfs. GC would eventually take care of the old vfsmount.
   * The problem being: we have to implement rootfs and GC for that ;-)
   * Actually it makes sense, especially if rootfs would contain a
   * /reboot - static binary that would close all descriptors and
   * call reboot(9). Then init(8) could umount root and exec /reboot. */
  if (mnt == current->fs->rootmnt) {
  /* Special case for "unmounting" root ...
   * we just try to remount it readonly. */
    return do_remount("/", MS_RDONLY, NULL);
  }
```

```
spin_lock(&dcache_lock);        Grab a spinlock on the directory cache
                                If this filesystem is mounted more than once...
if (mnt->mnt_instances.next != mnt->mnt_instances.prev) {
  if (atomic_read(&mnt->mnt_count) > 2) {    Another typical misleading variable name:
    spin_unlock(&dcache_lock);               mnt_count  is the number of users with
    return -EBUSY;                           open files on this filesystem
  }
  if (sb->s_type->fs_flags & FS_SINGLE)    For filesystems with only one superblock in memory...
    put_filesystem(sb->s_type);            ...just decrement a reference count
  detach_mnt(mnt, &parent_nd);     Detach this mount instance from the list of mountpoints
  list_del(&mnt->mnt_list);     Delete this mount instance from the list that is used for /proc/mounts
  spin_unlock(&dcache_lock);  Release the spinlock
  mntput(mnt);                  The actual releasing of memory happens here
  if (parent_nd.mnt != mnt)
    path_release(&parent_nd);
  return 0;
}

spin_unlock(&dcache_lock);     Release the spinlock
```

. . . . *Quota-related stuff deleted for your sanity*

. . . . *Special case for* `-f` *flag to* `umount` *handled here*

Same deal as above; make sure the filesystem is not busy...

```
  spin_lock(&dcache_lock);
  if (atomic_read(&mnt->mnt_count) > 2) {
    spin_unlock(&dcache_lock);
    return -EBUSY;
  }
```

...and remove this instance from all the necessary places

```
  /* OK, that's the point of no return */
  detach_mnt(mnt, &parent_nd);
  list_del(&mnt->mnt_list);
  spin_unlock(&dcache_lock);
  mntput(mnt);
  if (parent_nd.mnt != mnt)
    path_release(&parent_nd);
  return 0;
}
```

Creating a File

You can use the `touch` program to create an empty file:

```
: jmglov@delyana; touch /tmp/foobar
: jmglov@delyana; ls -l /tmp/foobar
-rw-rw----    1 jmglov   jmglov          0 Mar 30 14:21 /tmp/foobar
```

`touch` is part of the `fileutils` package, and is implemented in `src/touch.c`. What it basically does is set some flags and invoke the `open()` syscall, then calls `close()`.

fs/open.c:sys_open()

```
asmlinkage long sys_open(const char * filename, int flags, int mode)
{
  char * tmp;                       Kernel memory for filename
  int fd, error;                    File descriptor
 . . . . . . . . . . .
  tmp = getname(filename);          Grab the filename from userland
  fd = PTR_ERR(tmp);
  if (!IS_ERR(tmp)) {               If the filename seems OK...
    fd = get_unused_fd();           ...grab the next free file descriptor
    if (fd >= 0) {
      struct file *f = filp_open(tmp, flags, mode);    Actually open the file
      error = PTR_ERR(f);
      if (IS_ERR(f))
        goto out_error;
      fd_install(fd, f);            Give the file descriptor to the user
    }
 . . . . . . . . . . .              Give back the memory used by tmp  and return the file descriptor
```

fs/open.c:filp_open()

```
struct file *filp_open(const char * filename, int flags, int mode)
{
  int namei_flags, error;
  struct nameidata nd;

  namei_flags = flags;
  if ((namei_flags+1) & O_ACCMODE)
    namei_flags++;
  if (namei_flags & O_TRUNC)
    namei_flags |= 2;

  error = open_namei(filename, namei_flags, mode, &nd);
  if (!error)
    return dentry_open(nd.dentry, nd.mnt, flags);

  return ERR_PTR(error);
}
```

Create a nameidata struct (which creates the file)

Opens the dentry associated with the nameidata structure and returns a file pointer to the file itself

fs/namei.c:open_namei()

```
int open_namei(const char * pathname, int flag, int mode, struct nameidata *nd)
{
  struct inode *inode;
  struct dentry *dentry;
  struct dentry *dir;
 . . . . . . . . . . .          Skipping the simplest case, when O_CREAT flag is not set
  /* Create - we need to know the parent. */    Standard path_init(), path_walk() dance,
  if (path_init(pathname, LOOKUP_PARENT, nd))   except LOOKUP_PARENT flag causes last, last_type
    error = path_walk(pathname, nd);            fields to be filled in (info on the parent)
 . . . . . . . . . . .          Sanity checks here that we don't care about
  if (!dentry->d_inode) {       If the inode does not yet exist...
    error = vfs_create(dir->d_inode, dentry, mode); ...actually create it
 . . . . . . . . . . .
```

We are getting close to the Ext2-specific stuff now. `fs/namei.c:vfs_create()`:

```
int vfs_create(struct inode *dir, struct dentry *dentry, int mode)
{
 . . . . . . . . . . .            The usual bevy of sanity checks
  error = dir->i_op->create(dir, dentry, mode);  This will be ext2_create() , see below
 . . . . . . . . . . .
  if (!error)
    inode_dir_notify(dir, DN_CREATE);
```

Inode operations for Ext2 are defined in `fs/ext2/namei.c`:

```
struct inode_operations ext2_dir_inode_operations = {
  create:         ext2_create,
  lookup:         ext2_lookup,
  link:           ext2_link,
  unlink:         ext2_unlink,
  symlink:        ext2_symlink,
  mkdir:          ext2_mkdir,
  rmdir:          ext2_rmdir,
 . . . . . . . . . . .
};
```

fs/ext2/namei.c:ext2_create()

```
static int ext2_create (struct inode * dir, struct dentry * dentry, int mode)
{
  struct inode * inode = ext2_new_inode (dir, mode);   The inode is created here
 . . . . . . . . . . .
  inode->i_op = &ext2_file_inode_operations;   Make sure this inode has Ext2 operations
  inode->i_fop = &ext2_file_operations;
  inode->i_mapping->a_ops = &ext2_aops;
  inode->i_mode = mode;
  mark_inode_dirty(inode);       Force this inode to be synched to disk right away
  err = ext2_add_entry (dir, dentry->d_name.name, dentry->d_name.len,
  inode);
 . . . . . . . . . . .           Handle the error case
  d_instantiate(dentry, inode);      Fill in inode information for a dentry
  return 0;
}
```

fs/ext2/ialloc.c:ext2_new_inode()

```
/* There are two policies for allocating an inode.  If the new inode is
 * a directory, then [do some complicated shit.]
 *
 * For other inodes, search forward from the parent directory\'s block
 * group to find a free inode.
 */
struct inode * ext2_new_inode (const struct inode * dir, int mode)
{
  struct super_block * sb;
  struct buffer_head * bh;
  struct buffer_head * bh2;
  int i, j, avefreei;
  struct inode * inode;
  int bitmap_nr;
  struct ext2_group_desc * gdp;
  struct ext2_group_desc * tmp;
  struct ext2_super_block * es;
 . . . . . . . . . . .
  sb = dir->i_sb;
  inode = new_inode(sb);
```

Code	Comment
`sb = dir->i_sb;`	*Bind this inode to the superblock of the filesystem on which it is created*
`inode = new_inode(sb);`	*Calls* `get_empty_inode()` *, which slab allocates an inode*

```
  lock_super (sb);                   Grab a lock on the superblock (so no filesystem ops can breed race conditions)
  es = sb->u.ext2_sb.s_es;           Grab a copy of the pointer to the superblock in the buffer
repeat:
  gdp = NULL; i=0;                   Null out the pointer to the group descriptor

  if (S_ISDIR(mode)) {               This is not our case; we are just a file
 . . . . . . . . . . .
  else                               A normal file, that's us
  {
    i = dir->u.ext2_i.i_block_group;     Grab the index of the block group in which the parent directory lives
    tmp = ext2_get_group_desc (sb, i, &bh2);    Lookup stats on this block group
    if (tmp && le16_to_cpu(tmp->bg_free_inodes_count))  If there is a free block in this group...
      gdp = tmp;                                         ...use this group
    else                             Use a quadratic hash to find a group with a free block
 . . . . . . . . . . .               Failing that, fall back to a linear search
  bitmap_nr = load_inode_bitmap (sb, i);    Grab the inode bitmap
 . . . . . . . . . . .
  bh = sb->u.ext2_sb.s_inode_bitmap[bitmap_nr];     Look for the first zero in the bitmap
  if ((j = ext2_find_first_zero_bit ((unsigned long *) bh->b_data, ... ) {
    if (ext2_set_bit (j, bh->b_data)) {     Set the bit, since we now have an inode
 . . . . . . . . . . .
    mark_buffer_dirty(bh);           We will want to synch the buffer now
  } else {                           Report corruption of free inodes count, try to recover
```

Following this location of free space, the inode struct is filled out, locks are released, and we find ourselves back in `ext2_create()`, where we are immediately whisked off to `ext_add_entry()`, which adds our file entry to the parent directory. The new inode (encapsulated in a dentry, of course) percolates down:
`ext2_create()` → `vfs_create()` → `open_namei()` → `filp_open()`
where `fs/open.c/dentry_open()` opens the new file:

```
struct file *dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags)
{
  struct file * f;
  struct inode *inode;
  int error;
                                Default error is ''file table overflow''
  error = -ENFILE;              Slab allocate a file pointer
  f = get_empty_filp();         Sanity checking and flag setting
 . . . . . . . . . . .          Grab a pointer to the inode of the file we want to open
  inode = dentry->d_inode;      Sanity checks
 . . . . . . . . . . .
```

```
  f->f_dentry = dentry;          Give the dentry to the new file pointer
  f->f_vfsmnt = mnt;
  f->f_pos = 0;                  Set the file position pointer
  f->f_reada = 0;                Set the read-ahead flag to false
  f->f_op = fops_get(inode->i_fop);      Grab file operations from inode
  if (inode->i_sb)               If the inode is associated with a superblock (i.e. belongs to a filesystem)...
    file_move(f, &inode->i_sb->s_files);    ...put it on that filesystem's linked list of file objects
  if (f->f_op && f->f_op->open) {
    error = f->f_op->open(inode,f);      Strange, ext2_open_file() only disallows opening RW large files
 . . . . . . . . . . .                   on 32bit systems if the caller didn't specify O_LARGEFILE
  f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);

  return f;
 . . . . . . . . . . .
}
```

Returning to `sys_open()`, all we have left to do is bind the file to a file descriptor and return said file descriptor to the calling userland program.

Said caller is, in our example, `touch`. Now that it has opened the file, all `touch` needs to do is close it. `fs/open.c:sys_close()` is invoked:

```
asmlinkage long sys_close(unsigned int fd)
{
  struct file * filp;
  struct files_struct *files = current->files;
 . . . . . . . . . . .          Sanity checking and locking
  files->fd[fd] = NULL;         No more errors can occur, so terminate the file descriptor
 . . . . . . . . . . .
  return filp_close(filp, files);   Close the file
 . . . . . . . . . . .          Error handling stuff, ho hum
}
```

fs/open.c:filp_close()

```
int filp_close(struct file *filp, fl_owner_t id)
{
  int retval;

  if (!file_count(filp))                    This file object is unused, so closing it is a NOOP
    return 0;

  retval = 0;                      Default to success (unusual for kernel functions)
  if (filp->f_op && filp->f_op->flush) {    Not defined for Ext2;
 . . . . . . . . . . .                      seems to be used for journalling and network filesystems
  fput(filp);                      The file object gets released here
  return retval;
}
```

The function that actually releases the file object, thus closing the file, is `fs/file_table.c:fput()`:

```
void fput(struct file * file)
{
 . . . . . . . . . . .           Variable names well-chosen for once...
  if (atomic_dec_and_test(&file->f_count)) {   If we cannot decrement f_count , just return
    locks_remove_flock(file);
    if (file->f_op && file->f_op->release)   For Ext2, this maps to ext2_release_file() , which just
      file->f_op->release(inode, file);      discards preallocated blocks for files opened in
    fops_put(file->f_op);       write mode
    file->f_dentry = NULL;
    file->f_vfsmnt = NULL;
    if (file->f_mode & FMODE_WRITE)
      put_write_access(inode);
    dput(dentry);               Release the dentry itself
 . . . . . . . . . . .
    file_list_lock();           Lock the file list
    list_del(&file->f_list);                Remove the file from the list...
    list_add(&file->f_list, &free_list);    ...and add it to the free list
    files_stat.nr_free_files++;   For accounting and limits
    file_list_unlock();         Unlock the file list
  }
}
```

Deleting a File

The rm utility, also part of the fileutils package, is the most common way to delete a file.

```
: jmglov@delyana; rm /tmp/foobar
```

What rm (implemented in src/rm.c and src/remove.c in the fileutils source tree) actually does is to invoke the unlink syscall, which in turn deletes a "name" referring to a file. In the common case, in which the file has only one link, the file is also deleted. Let's look at fs/namei.c:sys_unlink():

```
asmlinkage long sys_unlink(const char * pathname)
{
  int error = 0;
  char * name;                  Kernel memory for filename
  struct dentry *dentry;
  struct nameidata nd;
```

```
  name = getname(pathname);       Grab filename from userland
 . . . . . . . . . . .            Sanity checks on filename
  if (path_init(name, LOOKUP_PARENT, &nd))   Standard path_init(), path_walk() dance;
  error = path_walk(name, &nd);              fills out a nameidata struct
 . . . . . . . . . . .            Sanity check on nameidata struct
  error = -EISDIR;                    We cannot unlink a directory in Linux...
  if (nd.last_type != LAST_NORM)      ...so if this is a directory, we need to bail
    goto exit1;
  down(&nd.dentry->d_inode->i_sem);   Lock the inode
  dentry = lookup_hash(&nd.last, nd.dentry);   Grab the dentry
  error = PTR_ERR(dentry);
  if (!IS_ERR(dentry)) {
 . . . . . . . . . . .            "slashes" case deleted; seems to be a previously undetectable directory
    error = vfs_unlink(nd.dentry->d_inode, dentry);
  exit2:
    dput(dentry);                 Release the dentry
  }
  up(&nd.dentry->d_inode->i_sem);   Unlock the inode
 . . . . . . . . . . .            Exit on out, releasing kernel memory for file structures as we go
}
```

fs/namei.c:vfs_unlink()

```
int vfs_unlink(struct inode *dir, struct dentry *dentry)
{
 . . . . . . . . . . . .          The usual sanity checking and locking
    if (dir->i_op && dir->i_op->unlink) {
      DQUOT_INIT(dir);            Quota crap; who cares?
      if (d_mountpoint(dentry))      Returns success (0) if dentry's d_vfsmnt  list is empty
        error = -EBUSY;
      else {
        lock_kernel();            Lock the whole bloody kernel
        error = dir->i_op->unlink(dir, dentry);    For Ext2: ext2_unlink()
        unlock_kernel();          Release the kernel-wide lock
        if (!error)
          d_delete(dentry);       Delete the dentry object
      }
    }
 . . . . . . . . . . . .
}
```

`ext2_unlink()` basically does four things:

1. Calls `ext2_find_entry()`
2. Calls `ext2_delete_entry()`
3. Updates the directory's `i_ctime` and `i_mtime` fields, then marks the parent directory's inode as dirty
4. Decrements the link count on the file's inode (in our case, to zero), marks the inode as dirty, and sets the `i_ctime` field equal to that of the parent directory.

`fs/ext2/namei.c:ext2_find_entry()`:

```
static struct buffer_head * ext2_find_entry (struct inode * dir,
                                              const char * const name, int namelen,
                                              struct ext2_dir_entry_2 ** res_dir)
{
  struct super_block * sb;
  struct buffer_head * bh_use[NAMEI_RA_SIZE];    We can read ahead 2 chunks of 4 blocks each
  struct buffer_head * bh_read[NAMEI_RA_SIZE];   while searching a directory
  unsigned long offset;
  int block, toread, i, err;

  *res_dir = NULL;              We are returning a new ext2_dir_entry_2 here, so NULL it out
  sb = dir->i_sb;               Grab the superblock from the parent directory's inode
 . . . . . . . . . . .
  memset (bh_use, 0, sizeof (bh_use)); Zero out the buffer
  toread = 0;
  for (block = 0; block < NAMEI_RA_SIZE; ++block) { Read ahead 8 blocks...
    struct buffer_head * bh;
    if ((block << EXT2_BLOCK_SIZE_BITS (sb)) >= dir->i_size)  If this block is outside the directory...
      break;                                                  ...break out of the loop
    bh = ext2_getblk (dir, block, 0, &err);  Read the next block of the directory in...
    bh_use[block] = bh;                      ...and put it into the used buffer array
    if (bh && !buffer_uptodate(bh))     If the buffer is not up-to-date...
      bh_read[toread++] = bh;           ...stick it in the to-be-read buffer array
  }
```

```
for (block = 0, offset = 0; offset < dir->i_size; block++) {  Iterate through the dir
  struct buffer_head * bh;
  struct ext2_dir_entry_2 * de;
  char * dlimit;

  if ((block % NAMEI_RA_BLOCKS) == 0 && toread) { If block  is a multiple of four and we have blocks to read...
    ll_rw_block (READ, toread, bh_read);          ...read the blocks in bh_read  and update the buffers
    toread = 0;
  }
  bh = bh_use[block % NAMEI_RA_SIZE];
  if (!bh) {                              If the buffer head is invalid, we have a hole;
    offset += sb->s_blocksize;    that's OK, just update the offset and continue
    continue;
  }
  wait_on_buffer (bh);          For synchronisation purposes
  if (!buffer_uptodate(bh))   If the buffer is not up-to-date, we had a read error. Run away!
    break;
```

```
  de = (struct ext2_dir_entry_2 *) bh->b_data;
  dlimit = bh->b_data + sb->s_blocksize;
  while ((char *) de < dlimit) {  Look for the directory entry in question
    int de_len;

    if ((char *) de + namelen <= dlimit &&    Found a match...
        ext2_match (namelen, name, de)) {
      if (!ext2_check_dir_entry("ext2_find_entry",   ...do a full check, just to be sure
                                dir, de, bh, offset))
        goto failure;          If the full check failed, we free bh_use   and return NULL
      for (i = 0; i < NAMEI_RA_SIZE; ++i) {  Free bh_use
        if (bh_use[i] != bh)
          brelse (bh_use[i]);
      }
      *res_dir = de;           This is the directory entry that we want
      return bh;               Return the buffer
    }
. . . . . . . . . . .
  else                         If we still need to read more blocks, do so
    bh = ext2_getblk (dir, block + NAMEI_RA_SIZE, 0, &err);
  bh_use[block % NAMEI_RA_SIZE] = bh;
  if (bh && !buffer_uptodate(bh))
    bh_read[toread++] = bh;
  }
```

When `ext2_find_entry()` returns to `ext2_unlink()`, we have our file in an `ext2_dir_entry_2` struct. The next step is to hand it off to `fs/ext2/namei.c:ext2_delete_entry()`:

```
static int ext2_delete_entry (struct inode * dir,
                              struct ext2_dir_entry_2 * de_del,
                              struct buffer_head * bh)
{
  struct ext2_dir_entry_2 * de, * pde;
  int i;

  i = 0;
  pde = NULL;
  de = (struct ext2_dir_entry_2 *) bh->b_data;   Grab the dir_entry struct from the buffer
  while (i < bh->b_size) {      Look for the file that we want to delete
 . . . . . . . . . . .          Sanity checks. Surprised?
    if (de == de_del)  {        Found it!
      if (pde)                  If there is a previous dir_entry...
 . . . . . . . . . . .          ...do something (I care not to say what)
```

```
      else                                Otherwise, if there is no previous dir_entry...
        de->inode = 0;                    ...toss this inode
      dir->i_version = ++event;
      mark_buffer_dirty_inode(bh, dir);   Mark the buffer as having a dirty inode
      if (IS_SYNC(dir)) {                 If we need to keep this data synchronised...
        ll_rw_block (WRITE, 1, &bh);      ...write the buffer out to disk
        wait_on_buffer (bh);
      }
      return 0;
    }
    i += le16_to_cpu(de->rec_len);
    pde = de;
    de = (struct ext2_dir_entry_2 *) ((char *) de + le16_to_cpu(de->rec_len));
  }
  return -ENOENT;
}
```

You may have noticed that nothing is actually deleted; the meta-data is merely updated to omit the inode of the file that was "deleted". This lazy way of doing things has advantages and disadvantages. An advantage is that we can usually recover an accidentally deleted file:

```
: jmglov@delyana; sudo mount -o loop /tmp/test-01.fs /tmp/mnt/test-01/
: jmglov@delyana; echo 'test for echo...' >/tmp/mnt/test-01/foobar
: jmglov@delyana; sudo umount /tmp/mnt/test-01/
: jmglov@delyana; strings -t o /tmp/test-01.fs
   2170 test-01
   2360 }$L}
 512040 lost+found
 512064 foobar
 544000 test for echo...
40002170 test-01
40002360 }$L}
: jmglov@delyana; sudo mount -o loop /tmp/test-01.fs /tmp/mnt/test-01/
: jmglov@delyana; rm /tmp/mnt/test-01/foobar
: jmglov@delyana; sudo umount /tmp/mnt/test-01/
: jmglov@delyana; strings -t o /tmp/test-01.fs
   2170 test-01
   2360 }$L}
 512040 lost+found
 512064 foobar
 544000 test for echo...
40002170 test-01
40002360 }$L}
```

As you can see, the data in the file and even the filename itself are still existent on the disk, in their previous locations. This means that recovering the data in the file is rather trivial, and “undeleting” the file, i.e. restoring its inode to its proper place in the directory from which it was deleted, is possible. See the Linux Ext2fs Undeletion mini-HOWTO:

`http://www.praeclarus.demon.co.uk/tech/e2-undel/html/howto.html`

or be a real man and use a hex editor. The bad news for you 5cr1p7 X1dd13Z (or privacy advocates) in the audience is that deleting a file is not going to wipe that incriminating (or personal) data from your hard drive. When you get rid of a hard drive that contained pr0n, erm, sensitive data, you probably want to do something like:

```
i=0
while [ \$i -lt 10  ]; do
  dd if=/dev/zero of=/dev/hdaN bs=1k
  let ‘‘i+=1’’
done
dd if=/dev/urandom of=/dev/hdaN bs=1k
```

Of course, the hardest of the hard-core recommend using `/dev/urandom` about 10 times before zeroing out the drive. Maybe that’s being a bit too paranoid...

Reading, Writing, and Seeking

In the mother of all cop-outs, I now claim that since `f_op->read`, `f_op->write`, and `f_op->llseek` resolve to `generic_file_read()`, `generic_file_write()`, and `generic_file_llseek()` for Ext2, they are of no interest to us...

Of course, this is a rather blatant lie, but I do not have time to talk about them, and they ***are*** actually generic. All three functions, and most of the functions that they call to do their dirty work, live in `mm/filemap.c`. Maybe Denis wants to tell you about this stuff when he talks about Ext3? ;)

References

1. Daniel P Bovet and Marco Cesati; Understanding the Linux Kernel, 2nd Edition
2. Linux 2.4.5 kernel source
3. Jack Daniels, Jim Beam, and Sam Adams; *"Wow! This code actually makes sense!"*