A (small) part of the *Linux VFS* module of the *kaiwanTECH* Linux Internals training programme.

Referenced kernel ver: 2.6.30

Once extracted, see the
`fs/fat`
folder.

Tip:
For ease of code browsing, do 'make tags' (or 'ctags -R') in the root folder of the kernel soure tree.

```
cd fs/fat
```

Superblock Setup

In *namei_msdos.c*:

```
...
static struct file_system_type msdos_fs_type = {
      .owner          = THIS_MODULE,
      .name           = "msdos",
      .get_sb         = msdos_get_sb,
      .kill_sb        = kill_block_super,
      .fs_flags       = FS_REQUIRES_DEV,
};

static int __init init_msdos_fs(void)
{
      return register_filesystem(&msdos_fs_type);
}
...
```

So, the routine invoked upon mounting is *msdos_get_sb* :

Following the call chain in *msdos_get_sb* (see the flow diagram), control hits VFS *fill_super* routine, which implicitly invokes the *msdos_fill_super* routine (function pointer passed via a parameter). This is the MSDOS-filesystem-specific routine to initialize the superblock.
This invokes the *fat_fill_super* routine which actually allocates memory for and initializes the MSDOS superblock structure – struct msdos_sb_info ; this is the data structure that MSDOS FS uses to map it's filesystem superblock into the kernel's VFS struct super_block structure. This routine reads from disk the MSDOS superblock, parses mount options, makes validity checks on the filesystem superblock, and finally intializes the structure.

```
...
        sb->s_magic = MSDOS_SUPER_MAGIC;
        sb->s_op = &fat_sops;
        sb->s_export_op = &fat_export_ops;
        sbi->dir_ops = fs_dir_inode_ops;
                              << sbi is the  msdos_sb_info structure >>
...
        bh = sb_bread(sb, 0);
        if (bh == NULL) {
                printk(KERN_ERR "FAT: unable to read boot sector\n");
                goto out_fail;
```

```
        }

        b = (struct fat_boot_sector *) bh->b_data;
...

        sbi->cluster_size = sb->s_blocksize * sbi->sec_per_clus;
        sbi->cluster_bits = ffs(sbi->cluster_size) - 1;
        sbi->fats = b->fats;
        sbi->fat_bits = 0;              /* Don't know yet */
        sbi->fat_start = le16_to_cpu(b->reserved);
        sbi->fat_length = le16_to_cpu(b->fat_length);
        sbi->root_cluster = 0;
        sbi->free_clusters = -1;        /* Don't know yet */
        sbi->free_clus_valid = 0;
        sbi->prev_free = FAT_START_ENT;

        if (!sbi->fat_length && b->fat32_length) {
                struct fat_boot_fsinfo *fsinfo;
                struct buffer_head *fsinfo_bh;

                /* Must be FAT32 */
                sbi->fat_bits = 32;
                sbi->fat_length = le32_to_cpu(b->fat32_length);
                sbi->root_cluster = le32_to_cpu(b->root_cluster);
...
```

<< To understand the underlying MSDOS filesystem design, see this link >>

It then sets up the root inode as well (including getting the superblock's s_root field to point to the root inode.

```
...
        root_inode = new_inode(sb);
        if (!root_inode)
                goto out_fail;
        root_inode->i_ino = MSDOS_ROOT_INO;
        root_inode->i_version = 1;
        error = fat_read_root(root_inode);
        if (error < 0)
                goto out_fail;
        error = -ENOMEM;
        insert_inode_hash(root_inode);
        sb->s_root = d_alloc_root(root_inode);
...
```

Inodes Setup

The inode represents any kind of file object. However, the VFS distinguishes between inode operations to be enacted on a directory object versus those to be enacted on a regular file (I/O) object.

So we have two 'inode_operations' structures that the filesystem implements – one for directory operations – creation, deletion, lookup, rename, etc – anything that operates directly on a “directory” (think “.” file) object, and one 'inode_operations' structure for actual file IO.

The MSDOS/FAT Directory Inode Operations

Whenever a new file is created, the filesystem has to allocate and initialize an inode. The 'create' method of the file_operations structure, therefore, is setup to point to a method to do this for the particular filesystem implementation.

So, we see in *fs/fat/namei_msdos.c* :

```
static const struct inode_operations msdos_dir_inode_operations = {
        .create         = msdos_create,
        .lookup         = msdos_lookup,
        .unlink         = msdos_unlink,
        .mkdir          = msdos_mkdir,
        .rmdir          = msdos_rmdir,
        .rename         = msdos_rename,
        .setattr        = fat_setattr,
        .getattr        = fat_getattr,
};

static int msdos_fill_super(struct super_block *sb, void *data, int silent)
{
        int res;

        res = fat_fill_super(sb, data, silent, &msdos_dir_inode_operations, 0);
...
```

fs/fat/inode.c:

```
...
/*
 * Read the super block of an MS-DOS FS.
 */
int fat_fill_super(struct super_block *sb, void *data, int silent,
                   const struct inode_operations *fs_dir_inode_ops, int isvfat)
{
...
      sbi->dir_ops = fs_dir_inode_ops;
...
```

If a userspace process attempts to create a new file on an MSDOS filesystem, the kernel VFS ultimately switches the request to the fs_dir_inode_ops function, in this case, the create method which is *msdos_create*.

The MSDOS/FAT File Inode Operations

In *fs/fat/inode*.c :

The *fat_fill_inode* routine [1] is the one responsible for initializing the inode, for both a directory object as well as a non-directory object.

```
...
/* doesn't deal with root inode */
static int fat_fill_inode(struct inode *inode, struct msdos_dir_entry *de)
```

```
{
        struct msdos_sb_info *sbi = MSDOS_SB(inode->i_sb);
...
        if ((de->attr & ATTR_DIR) && !IS_FREE(de->name)) {
                inode->i_generation &= ~1;
                inode->i_mode = fat_make_mode(sbi, de->attr, S_IRWXUGO);
                inode->i_op = sbi->dir_ops;
                        << sbi->dir_ops is the same structure we saw above, viz,
                           the msdos_dir_inode_operations structure. >>
                inode->i_fop = &fat_dir_operations;
...
    } else { /* not a directory */
                inode->i_generation |= 1;
                inode->i_mode = fat_make_mode(sbi, de->attr,
                        ((sbi->options.showexec && !is_exec(de->name + 8))
                         ? S_IRUGO|S_IWUGO : S_IRWXUGO));
                MSDOS_I(inode)->i_start = le16_to_cpu(de->start);
                if (sbi->fat_bits == 32)
                    MSDOS_I(inode)->i_start |= (le16_to_cpu(de->starthi) << 16);

                MSDOS_I(inode)->i_logstart = MSDOS_I(inode)->i_start;
                inode->i_size = le32_to_cpu(de->size);
                inode->i_op = &fat_file_inode_operations;
                inode->i_fop = &fat_file_operations;
                inode->i_mapping->a_ops = &fat_aops;
...
```

Furthermore, as we can see above, the inode has two operation pointers:

i_op: for the inode methods operating on the inode object itself, and

i_fop: for the methods that operate on the open file object that the inode represents.

In *fs/fat/file.c* :

```
...
const struct inode_operations fat_file_inode_operations = {
        .truncate       = fat_truncate,
        .setattr        = fat_setattr,
        .getattr        = fat_getattr,
};
```

In *fs/fat/file.c* :

```
...
const struct file_operations fat_file_operations = {
        .llseek         = generic_file_llseek,
        .read           = do_sync_read,
        .write          = do_sync_write,
        .aio_read       = generic_file_aio_read,
        .aio_write      = generic_file_aio_write,
        .mmap           = generic_file_mmap,
        .release        = fat_file_release,
        .ioctl          = fat_generic_ioctl,
        .fsync          = file_fsync,
        .splice_read    = generic_file_splice_read,
};
...
```

These will be invoked via the usual VFS route (filp->f_op->*foo*), where *foo* is the method – system call - invoked from the userspace process (or thread).

In fact, we can see from the above implementation, that the MSDOS filesystem (and indeed all the

FAT variants – MSDOS (FAT12), FAT16, FAT32 (VFAT)), invoke the generic VFS methods for read, write, lseek, mmap and aio_[read|write].

VFS component	*Corr. MSDOS/FAT component*	*Macro to access it*
`struct super_block`	`struct msdos_sb_info`	`MSDOS_SB`
`struct inode`	`struct msdos_inode_info`	`MSDOS_I`

[1] When does the *fat_fill_inode* routine get invoked?
The *fat_build_inode* function invokes it. So what invokes *fat_build_inode* ?
Cscope can provide us with an answer (output below, on the 2.6.30 kernel):

```
Functions calling this function: fat_build_inode

  File          Function       Line

0 inode.c       fat_get_parent 752 inode = fat_build_inode(sb, de, i_pos);

1 namei_msdos.c msdos_lookup   220 inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);

2 namei_msdos.c msdos_create   306 inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);

3 namei_msdos.c msdos_mkdir    394 inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);

4 namei_vfat.c  vfat_lookup    732 inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);

5 namei_vfat.c  vfat_create    788 inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);

6 namei_vfat.c  vfat_mkdir     882 inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);
```
