File System
Code Walkthrough

File System

- An organization of data and metadata on a storage device
- Data expected to be retained after a program terminates by providing efficient procedures to:
  - store,
  - retrieve, and
  - update data as well as
  - manage the available space on the device(s)
- Vague and broad definition at the very least

File System Types

- Abstract descriptions of the way data is organized in a files system of that type, like FAT16 or ext2

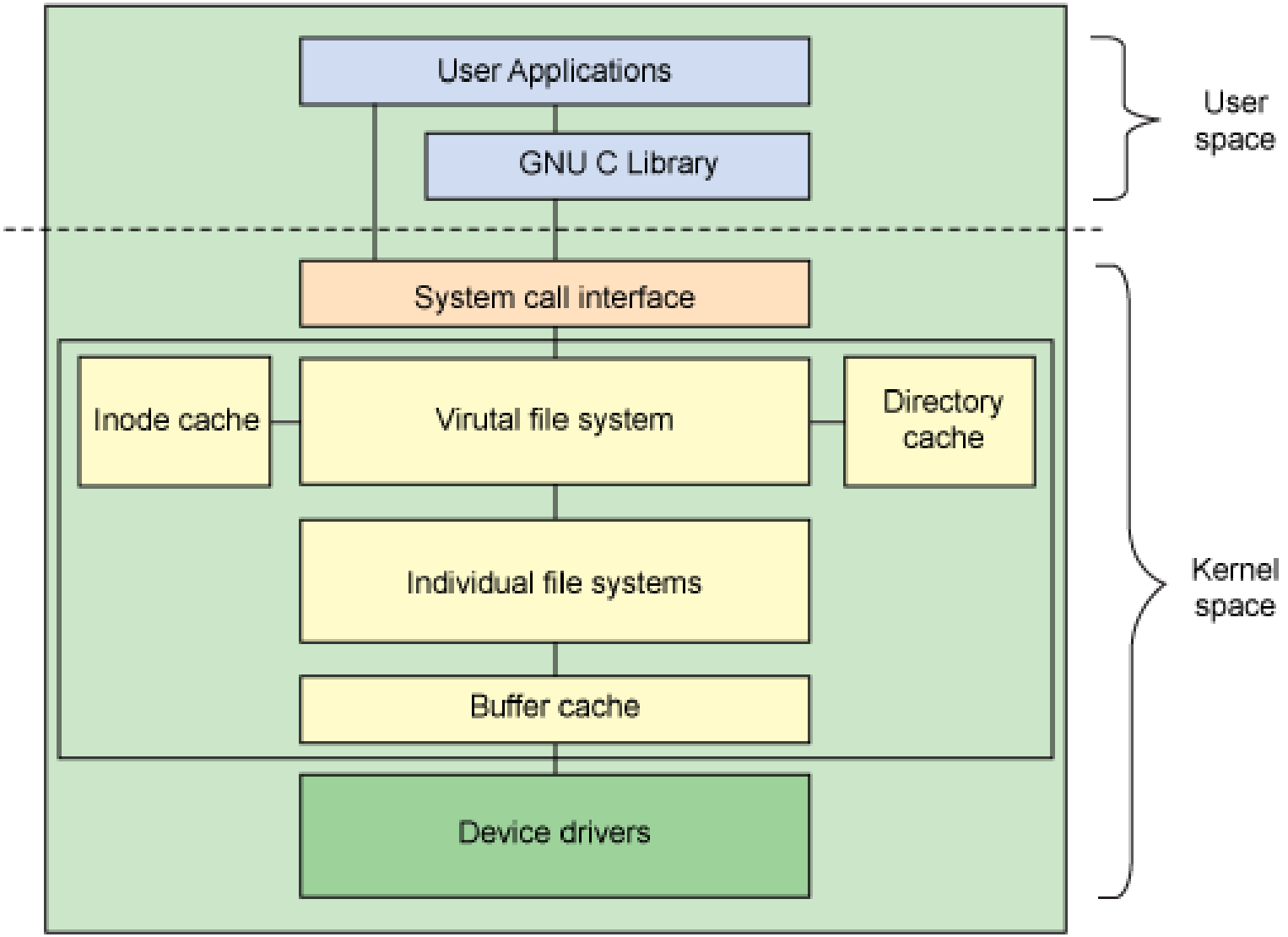
- Each has its own way of implementing the low level organization and storage of data

The Linux FS

- Swiss Army knife of operating systems
- Supports a large number of file systems, from journaling to clustering to cryptographic

- Layered Architecture
- Separates user interface from FS implementation from the drivers that manipulate storage devices

The Linux FS


User Applications
GNU C Library
User space
System call interface
Inode cache
Virutal file system
Directory cache
Individual file systems
Buffer cache
Device drivers
Kernel space

User Space

- User Space contains applications and GNU C Library
- glibc provides UI for FS calls : open, read, write, close
- The Syscall interface acts a switch, funneling systems calls from user space to the appropriate endpoints in kernel space

Virtual File System

- Primary interface to underlying FS
- Exports a set of interfaces
- Abstracts them to the individual FSs
- Keeps track of the currently supported FS, as well as those currently mounted
- List of supported FS :

```
cat /proc/filesystems
```

"On a UNIX system, everything is a file; if something is not a file, it is a process."

Directories
Special files / I/O devices
Links
sockets
Named pipes

Neat interface!

File System in Linux

- A module implementing a FS type must announce its presence so that it can be used :

```
register_filesystem(struct file_system_type)
```

- Tasks:
  - To have a name
  - To know how it is mounted
  - To know how to lookup files
  - To know how to find file contents (R/W)

struct file_system_type

```
struct file_system_type{
      const char *name;                        //name of FS
      int fs_flags;                          // bitmap flags
      struct super_block *get_sb;              // for mount
      void (*kill_sb);                        // for unmount
      struct module *owner;               // module that owns
      struct file_system_type *next;             // next FS
      struct list_head fs_supers;  // head of list of
all superblocks of this type
}
```

fs/filesystems.c: static struct file_system_type *file_systems;

Mounting

- Mount syscall attaches a FS to the file hierarchy at some indicated point
- Needed:
  - A device to carry FS (disk, CDROM, ..)
  - A directory where FS on that device must be attached
  - A FS type

- `$ mount -t ext2 /dev/loop0 /mnt/point1`

struct vfsmount

mount.h

- Describes a mount
- Provides the file systems that are currently mounted

gourav:$ df –h – to see the mount point of partitions

struct vfsmount

```
struct file_system_type{
      struct list_head mnt_hash;
      struct vfsmount *mnt_parent;
      struct dentry *mnt_mountpoint;
      struct dentry *mnt_root;
      struct super_block *mnt_sb;
      struct list_head mnt_mounts;
      struct list_head mnt_child;
      atomic_t mnt_count;
      int mnt_flags;
      char *mnt_devname;
      struct list_head mnt_list;
}
```

struct vfsmount

- `mnt_hash` : vfsmounts live in a hash headed by `mount_hashtable[]`
- `mnt_parent` : vfsmount for parent
- `mnt_mouthpoint` : dentry for mouthpoint
- `mnt_root` : dentry for root of mounted tree
- `mnt_sb` : superblock of mounted FS
- `mnt_mounts, mnt_child` : The field mnt_mounts of a struct vfsmount is the head of a cyclic list of all submounts (mounts on top of some path relative to the present mount). The remaining links of this cyclic list are stored in the mnt_childfields of its submounting vfsmounts.
- `mnt_count` : Keep track of users of this structure
- `mnt_flags` : The mount flags, like MNT_NODEV, MNT_NOEXEC, MNT_NOSUID.
- `mnt_devname` : Name used in /proc/mounts.
- `mnt_list` : There was a global cyclic list vfsmntlist containing all mounts, used only to create the contents of/proc/mounts. This list is ordered by the order in which the mounts were done, so that one can do the umounts in reverse order. The field mnt_list contains the pointers for this cyclic list.

Mount Points of partitions

```
freddy:~> df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda8             496M  183M  288M  39% /
/dev/hda1             124M  8.4M  109M   8% /boot
/dev/hda5              19G   15G  2.7G  85% /opt
/dev/hda6             7.0G  5.4G  1.2G  81% /usr
/dev/hda7             3.7G  2.7G  867M  77% /var
fs1:/home             8.9G  3.7G  4.7G  44% /.automount/
fs1/root/home
```

/dev dontains references to all the CPU peripheral hardware, which are represented as special files

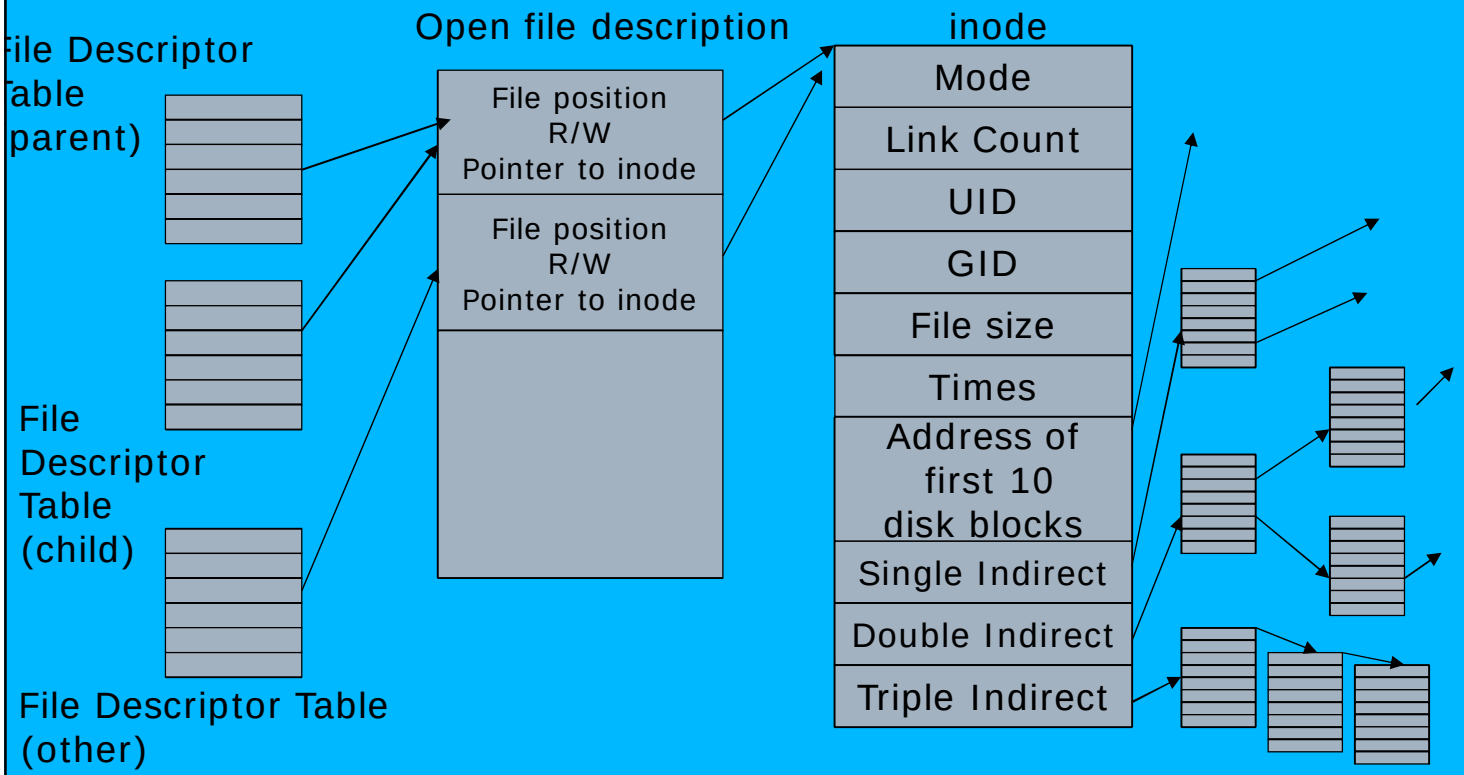
Moving on to the structures in the Linux VFS

Designed keeping in ext2 FS in mind

Structures

- Linux views all FSs from the perspective of a common set of objects
  - **Superblock** : Describes and maintains state for FS
  - **Inode** : Every object managed within FS (file/directory). Includes metadata
  - **Dentry** : Translates between names and inodes (used for caching)
  - **File** : Represents an open file (state for open file, write offset, etc.)

UNIX file structure implementation



Superblock

- Gives global information on a FS
  - Device on which it lives
  - Its block size
  - Its Type
  - The dentry of the root of the FS
  - The methods it has



struct super_block

```
struct super_block{
      dev_t s_dev;
      unsigned long s_blocksize;
      struct file_system_type *s_type;
      struct super_operations *s_op;
      struct dentry *s_root;
}
struct super_operations{
      alloc_inode(), destroy_inode(), read_inode(),
      write_inode(), ...
}
```



Superblock

- Dentry of the root directory gives the inode of the root directory
  - `sb->s_op->read_inode(inode)` reads inode from disk
  - `inode->i_op->lookup()` finds names in root dir.
- Each SB is on six lists, with links through the fields `s_list`, `s_dirty`, `s_io`, `s_anon`, `s_files`, `s_instances`.



Inode (Index Nodes)

- Kernel keeps track of files using in-core inodes derived by the low-level FS from on-disk inodes.
- An (in-core) inode contains the metadata of a file: its serial number, its protection (mode), its owner, its size, the dates of last access, creation and last modification, etc.
- It also points to the superblock of the filesystem the file is in, the methods for this file, and the dentries (names) for this file.

struct inode

```
struct inode{
     unsigned long i_ino;
     umode_t i_mode;
     struct qstr d_name;
     struct super_block *d_sb;
     struct dentry_operations *d_op;
     struct list_head d_subdirs;
}
```

Each dentry is on five lists, with links through the fields `d_hash`, `d_lru`, `d_child`, `d_subdirs`, `d_alias`.

Inode Lists

- **The dentry list**
- All dentries belonging to this inode (names for this file) are collected in a list headed by the inode fieldi_dentry with links in the dentry fields d_alias. This list is protected by the spinlock dcache_lock.
- **The hash list**
- All inodes live in a hash table, with hash collision chains through the field i_hash of the inode. These lists are protected by the spinlock inode_lock. The appropriate head is found by a hash function; it will be an element of the inode_hashtable[] array when the inode belongs to a superblock, oranon_hash_chain if not.
- **i_list**
- Inodes are collected into lists that use the i_list field as link field. The lists are protected by the spinlockinode_lock. An inode is either unused, and then on the chain with head inode_unused, or in use but not dirty, and then on the chain with head inode_in_use, or dirty, and then on one of the per-superblock lists with heads s_dirty or s_io.
- **i_devices**
- Inodes belonging to a given block device are collected into a list headed by the bd_inodes field of the block device, with links in the inode i_devices fields. The list is protected by the bdev_lock spinlock. It is used to set the i_bdev field to NULL and to reset i_mapping when the block device goes away.

Dentries (Directory Entries)

- A file may have several names, and a layer of dentries that represent pathnames speed up the lookup operation.
- They encode the FS tree structure.
- Main parts include:
  - Inode that belongs to it (if any)
  - Final part of pathname
  - Name of containing directory

struct dentry

```
struct dentry{
        struct inode *d_inode;
        struct dentry *d_parent;
        struct qstr d_name;
        struct super_block *d_sb;
        struct dentry_operations *d_op;
        struct list_head d_subdirs;
}
struct qstr{
        const unsigned char *name;
        unsigned int len;
        unsigned int hash;
}
```

```
struct
dentry_operations{
        d_revalidate();
        d_hash();
        d_compare();
        d_delete();
        d_release();
        d_iput();
}
```

Each dentry is on five lists, with links through the fields `d_hash`, `d_lru`, `d_child`, `d_subdirs`, `d_alias`.

Value of a dentry

- Concatenation of name of its parent `d_parent`, a slash character, and its own name `d_name`
- Pathname of the root of the FS (with `dentry->d_parent == dentry`) is "/", and this is also its `d_name`
- A dentry is called *negative* if it does not have an associated inode i.e. if it is a name only.
- Although a dentry represents a pathname, there may be several dentries for the same pathname, namely when overmounting has taken place. Such dentries have different inodes.
- The converse, an inode with several dentries, can also occur.

Dentry Lists

- **d_hash:**

  Dentries are used to speed up the lookup operation. A hash table dentry_hashtable is used, with an index that is a hash of the name and the parent. The hash collision chain has links through the dentry fieldsd_hash. This chain is protected by the spinlock dcache_lock.
- **d_lru:**

  All unused dentries are collected in a list dentry_unused with links in the dentry fields d_lru. This list is protected by the spinlock dcache_lock.
- **d_child, d_subdirs:**

  All subdirectories of a given directory are collected in a list headed by the dentry field d_subdirs with links in the dentry fields d_child. These lists are protected by the spinlock dcache_lock.
- **d_alias:**

  All dentries belonging to the same inode are collected in a list headed by the inode field i_dentry with links in the dentry fields d_alias. This list is protected by the spinlock dcache_lock.

Files

- Represent open files, an inode along with a current (reading/writing) offset.
- The offset can be set by `lseek()` syscall.
- Instead of a pointer to the inode we have a pointer to the dentry - that means that the name used to open a file is known

struct file

```
struct file{
     struct dentry *f_dentry;
     struct vfsmount *f_vfsmnt;
     struct file_operations *f_op;
     mode_t f_mode;
     loff_t f_pos;
     struct fown_struct f_owner;
     unsigned int f_uid, f_gid;
     unsigned long f_version;
}
```

Each file is on two lists, with links through the fields `f_list`, `f_ep_links`

File Lists

- **f_list**

  It is the list of all files belonging to a given superblock. There is a second use: the tty driver collects all files that are opened instances of a tty in a list headed by tty->tty_files with links through the file field f_list. Conversely, these files point back at the tty via their field private_data.

- **The event poll list**

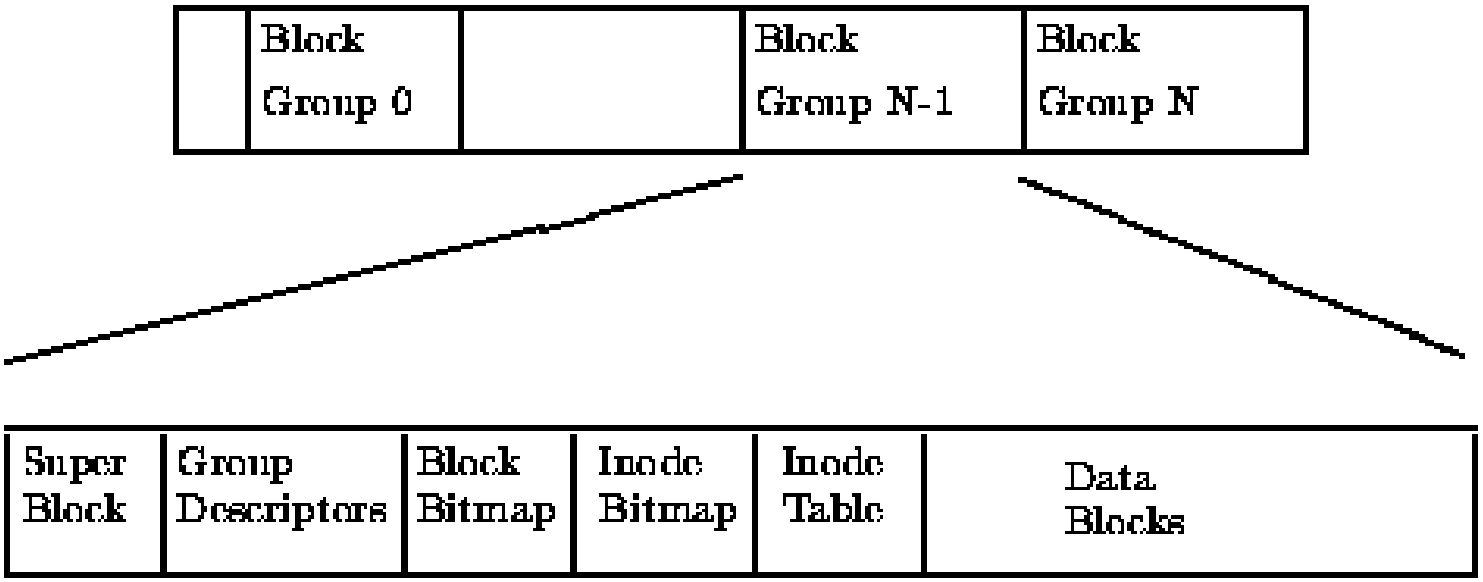
  All event poll items belonging to a given file are collected in a list with head f_ep_links, protected by the file field f_ep_lock.

An example File System that resides below the Linux VFS

ext2 file system type

- The space in ext2 is split up into blocks.
- These blocks are grouped into block groups. There are typically thousands of blocks on a large file system. Data for any given file is typically contained within a single block group where possible. This is done to minimize the number of disk seeks when reading large amounts of contiguous data.
- Each block group contains a copy of the superblock and block group descriptor table, and all block groups contain a block bitmap, an inode bitmap, an inode table and finally the actual data blocks.
- The group descriptor stores the location of the block bitmap, inode bitmap and the start of the inode table for every block group. These, in turn, are stored in a group descriptor table.

ext2


Block
Group 0
Block
Group N-1
Block
Group N
Super
Block
Group
Descriptors
Block
Bitmap
Inode
Bitmap
Inode
Table
Data
Blocks

ext2 inode

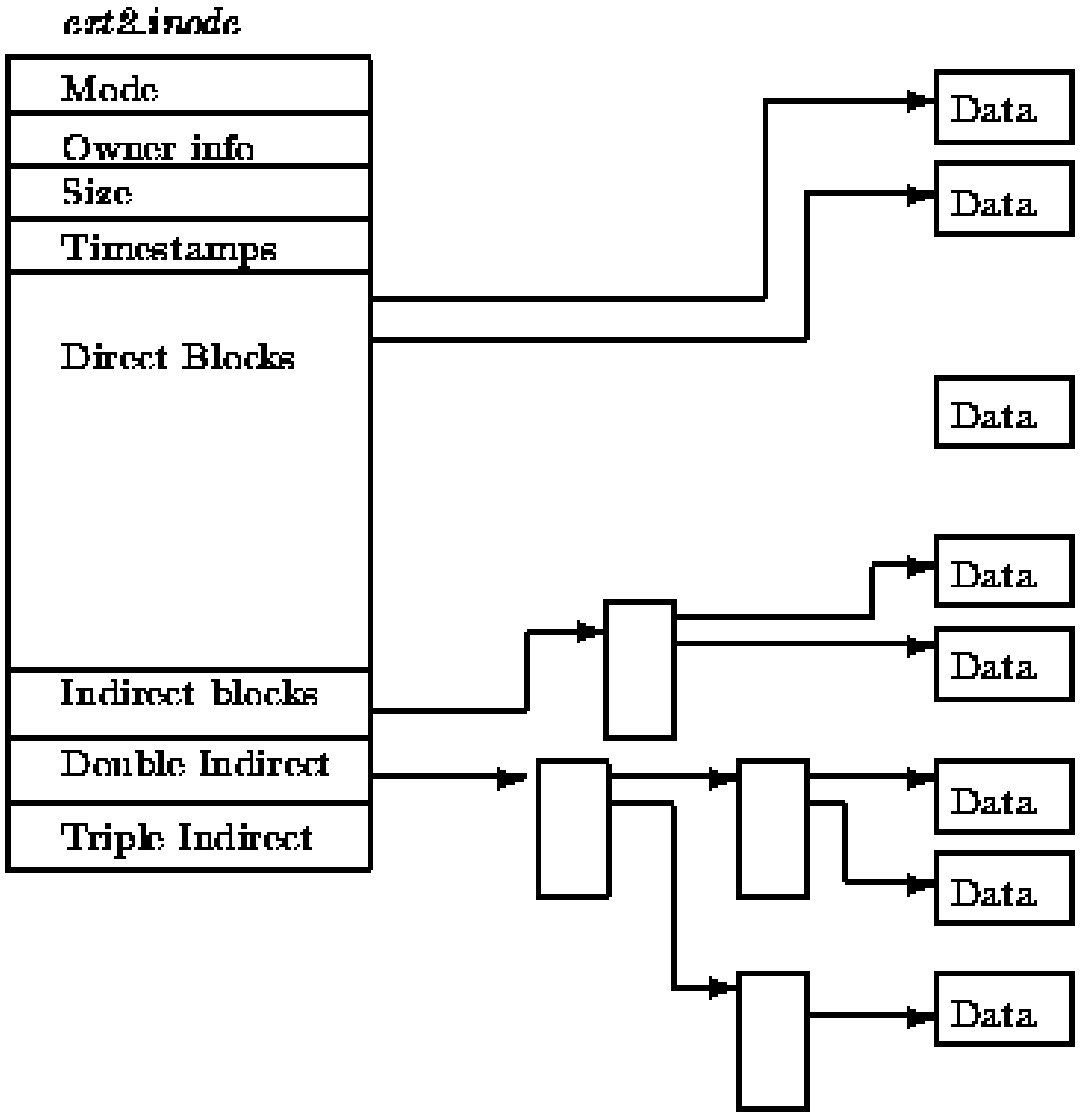
Every file and directory in the file system is described by one and only one inode.

Mode: Type of inode and the permissions that users have to it. For EXT2, an inode can describe one of file, directory, symbolic link, block device, character device or FIFO.

Owner Information: The user and group identifiers of the owners of this file or directory. This allows the file system to correctly allow the right sort of accesses,

Size: The size of the file in bytes, modified,

Datablocks: Pointers to the blocks that contain the data that this inode is describing.



Other ext2 structures

- **The EXT2 Superblock**

  The Superblock contains a description of the basic size and shape of this file system. The information within it allows the file system manager to use and maintain the file system. Usually only the Superblock in Block Group 0 is read when the file system is mounted but each Block Group contains a duplicate copy in case of file system corruption. Amongst other information it holds the:

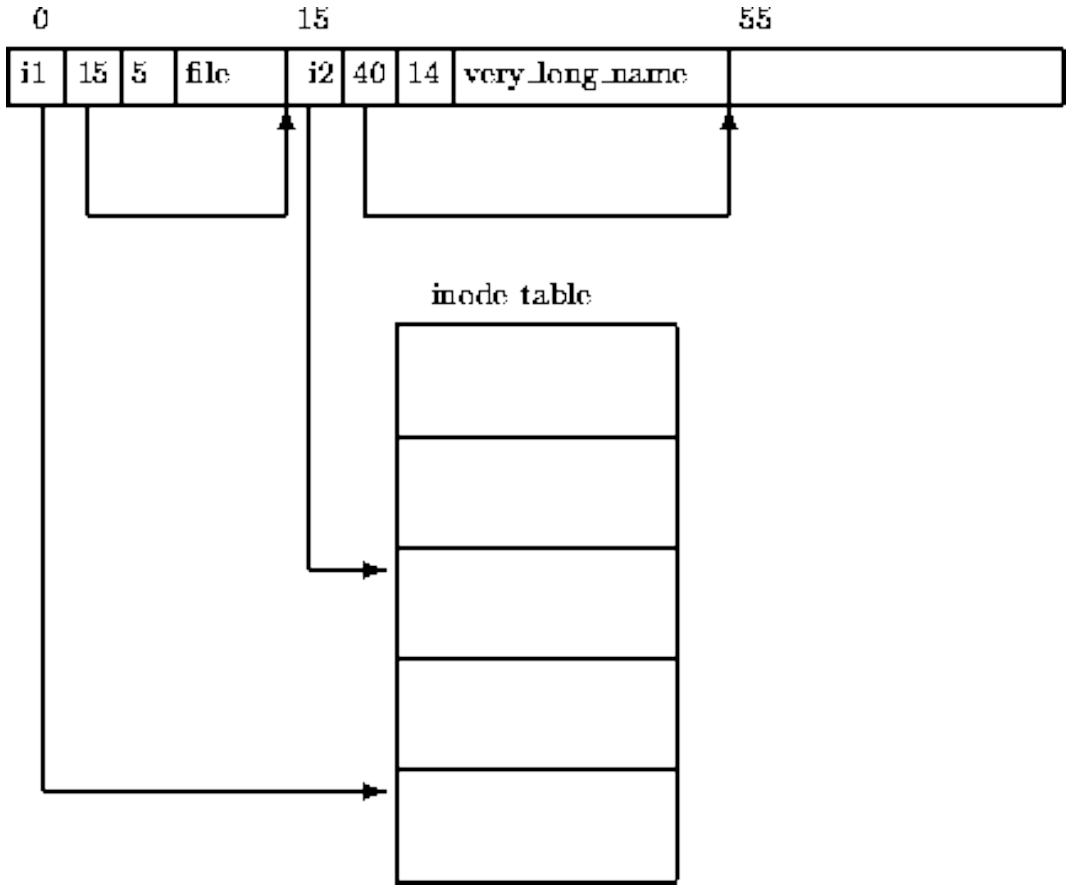
- **The EXT2 Group Descriptor**

  Each Block Group has a data structure describing it. Like the Superblock, all the group descriptors for all of the Block Groups are duplicated in each Block Group in case of file system corruption.

ext2 directories

Directories are special files that are used to create and hold access paths to the files in the file system.
Inode: The inode for this directory entry. This is an index into the array of inodes held in the Inode Table of the Block Group.
Name length: The length of this directory entry in bytes,
Name: The name of this directory entry.

The first two entries for every directory are always the standard **.** and **..** entries meaning *this directory* and *the parent directory* respectively.



Some more examples

- Disk file systems: FAT(File Allocation Table), NTFS (New Technology File System), HFS (Hierarchical File System), ext[2-4] (Extended File System)
- Optical discs: ISO 9660 (Compact Disc File System), UDF (Universal Disc Format)
- Flash file systems: FTL (Flash Translation Layer)

Sources

- http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/
- http://www.win.tue.nl/~aeb/linux/lk/lk-8.html