CS 6V81-05: System Security and Malicious Code Analysis
File System Design Implementation

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

March 7th, 2012

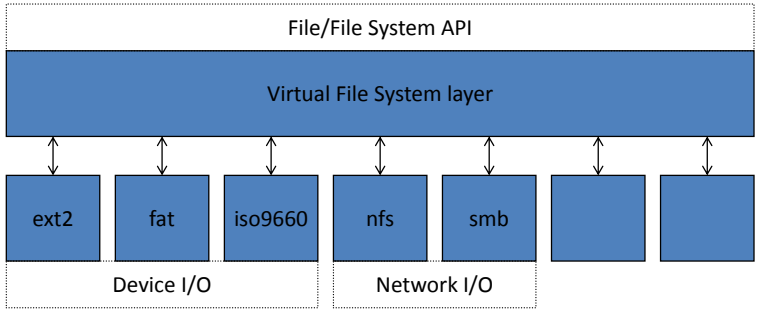
Outline

Outline

1. File System Design
2. File System: Disk Interface
3. File Carving: A Case Study
4. File System: Kernel Interface
5. Summary

File System Design

- A layered approach (remember "abstraction" + "layered approach")
  - Upper layer: virtual (logical) file system
  - Lower layer: specific file system modules



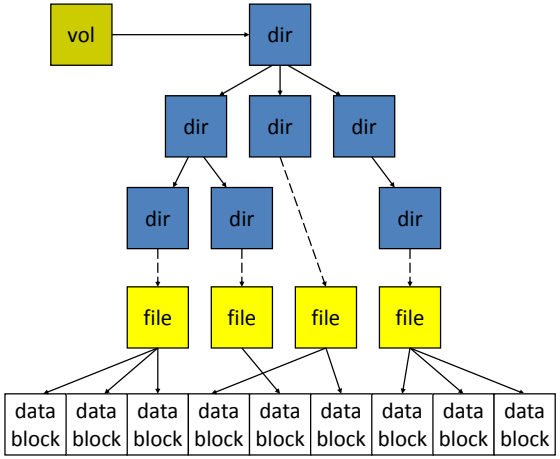
Virtual File System (VFS) Layer

- Purpose
  - Abstraction for all different file system implementations
- Functions
  - Provide the same file and file system interface
  - Manage all file and file system related data structures
  - Routines for efficient lookup, traverse the file system
  - Interact with specific file system modules

File System Basic Data Structures

- Volume Control Block (Unix: "superblock")
  - One per file system
  - Detail information about the file system
  - # of blocks, block size, free-block count/pointer, etc.
- File Control Block (Unix: "vnode" or "inode")
  - One per file
  - Detail information about the file
  - Permission, owner, size, data block locations, etc.
- Directory Node (Linux: "dentry")
  - One per directory entry (directory or file)
  - A tree data structure to encode the directory structure and tree layout
  - Pointer to file control block, parent, list of entries, etc.

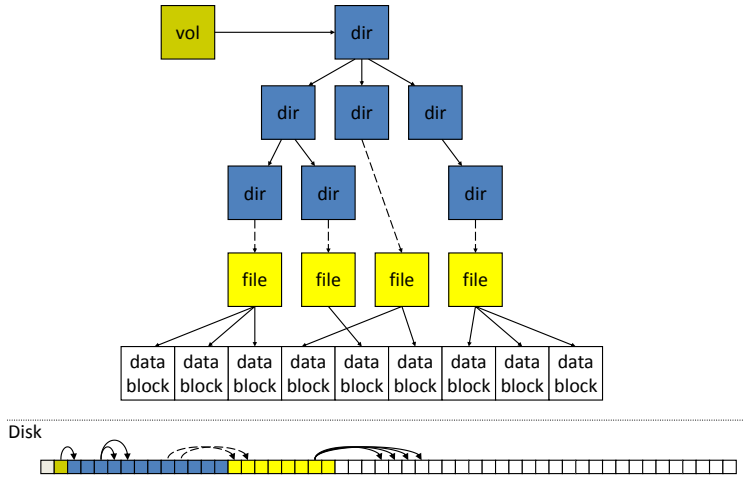
Abstract View



Where are the Data Structure Stored

- File system data structures
  - Volume control block (one per file system)
  - File control block (one per file)
  - Directory node (one per directory entry)
- Persistently stored on the secondary storage
  - In data block(s) allocated in the storage
- Loaded to memory when needed
  - Volume control block: in memory if file system is mounted
  - File control block: if the file is accessed
  - Directory node: during traversal of a file path

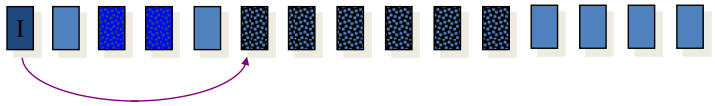
Storage View



File Allocation

- How to allocate data blocks to each file
- Allocation methods
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation
- Metrics
  - Efficiency: e.g., storage utilization (external fragmentation)
  - Performance: e.g., access speed

Contiguous Allocation

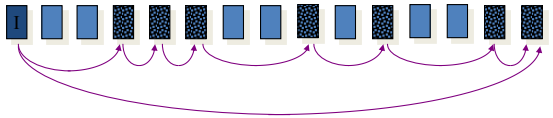


- File header specifies starting block & length
  - Placement/Allocation policies
  - First-fit, best-fit, ...

Pros+
- Best file read performance
- Efficient sequential & random access

Cons-
- Fragmentation!
- Problems with file growth
  – Pre-allocation?
  – On-demand allocation?

Linked Allocation

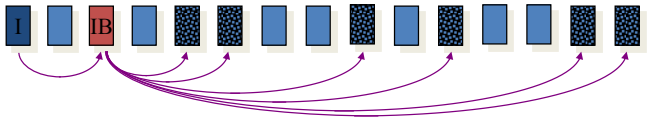


- Files stored as a linked list of blocks
- File header contains a pointer to the first and last file blocks

Pros+
- Easy to create, grow & shrink files
- No fragmentation

Cons-
- Impossible to do true random access
- Reliability
  – Break one link in the chain and. . .

Indexed Allocation



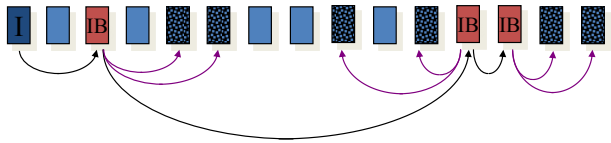
- Create a non-data block for each file called the index block
  - A list of pointers to file blocks
- File header contains the index block

Pros+
- Easy to create, grow & shrink files
- No fragmentation
- Supports direct access

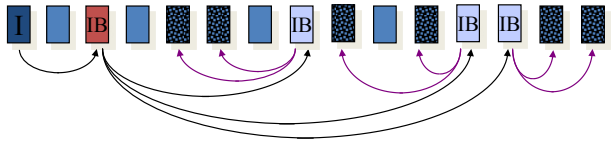
Cons-
- Overhead of storing index when files are small
- How to handle large files?

Indexed Allocation for Large Files

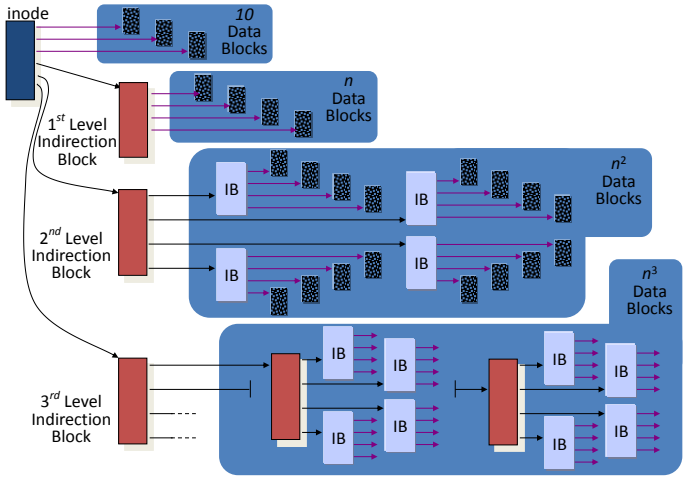
Linked index blocks (IB+IB+...)



Multilevel index blocks (IB*IB*...)



Multi-level Indexed Allocation in UNIX



Multi-level Indexed Allocation in UNIX

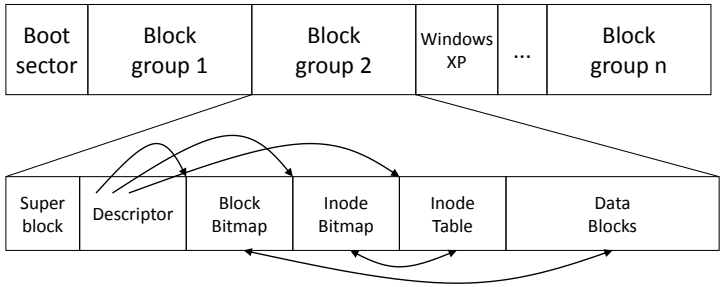
- File header contains 13 pointers
  - 10 pointes to data blocks;
  - 11th pointer $\rightarrow$ indirect block;
  - 12th pointer $\rightarrow$ doubly-indirect block;
  - 13th pointer $\rightarrow$ triply-indirect block
- Implications
  - Upper limit on file size
  - Blocks are allocated dynamically, files can easily expand
  - Small files are cheap
  - Allocate indirect blocks only for large files, and large files require a lot of seek to access indirect blocks

Free-Space List: Bit Vector

- Represent the list of free blocks as a bit vector:
- 11111111111111100111010101110111 1...
- If bit i = 0 then block i is free, otherwise it is allocated

Outline

Ext2 Disk Layout



• The size of each segment should be a multiple of blocks.

struct ext2_super_block

```
struct ext2_super_block {
     [0] __le32 s_inodes_count;
     [4] __le32 s_blocks_count;
     [8] __le32 s_r_blocks_count;
    [12] __le32 s_free_blocks_count;
    [16] __le32 s_free_inodes_count;
    [20] __le32 s_first_data_block;
    [24] __le32 s_log_block_size;
    [28] __le32 s_log_frag_size;
    [32] __le32 s_blocks_per_group;
    [36] __le32 s_frags_per_group;
    [40] __le32 s_inodes_per_group;
    [44] __le32 s_mtime;
    [48] __le32 s_wtime;
    [52] __le16 s_mnt_count;
```

struct ext2_super_block

```
struct ext2_super_block {
    [54] __le16 s_max_mnt_count;
    [56] __le16 s_magic;
    [58] __le16 s_state;
    [60] __le16 s_errors;
    [62] __le16 s_minor_rev_level;
    [64] __le32 s_lastcheck;
    [68] __le32 s_checkinterval;
    [72] __le32 s_creator_os;
    [76] __le32 s_rev_level;
    [80] __le16 s_def_resuid;
    [82] __le16 s_def_resgid;
    [84] __le32 s_first_ino;
    [88] __le16 s_inode_size;
    [90] __le16 s_block_group_nr;
    [92] __le32 s_feature_compat;
    [96] __le32 s_feature_incompat;
   [100] __le32 s_feature_ro_compat;
   [104] __u8 s_uuid[16];
   [120] char s_volume_name[16];
   [136] char s_last_mounted[64];
```


struct ext2_super_block

```
struct ext2_super_block {
   [200] __le32 s_algorithm_usage_bitmap;
   [204] __u8 s_prealloc_blocks;
   [205] __u8 s_prealloc_dir_blocks;
   [206] __u16 s_padding1;
   [208] __u8 s_journal_uuid[16];
   [224] __u32 s_journal_inum;
   [228] __u32 s_journal_dev;
   [232] __u32 s_last_orphan;
   [236] __u32 s_hash_seed[4];
   [252] __u8 s_def_hash_version;
   [253] __u8 s_reserved_char_pad;
   [254] __u16 s_reserved_word_pad;
   [256] __le32 s_default_mount_opts;
   [260] __le32 s_first_meta_bg;
   [264] __u32 s_reserved[190];
}
SIZE: 1024
```


struct ext2_group_desc

```
struct ext2_group_desc {
   [0] __le32 bg_block_bitmap;
   [4] __le32 bg_inode_bitmap;
   [8] __le32 bg_inode_table;
  [12] __le16 bg_free_blocks_count;
  [14] __le16 bg_free_inodes_count;
  [16] __le16 bg_used_dirs_count;
  [18] __le16 bg_pad;
  [20] __le32 bg_reserved[3];
}
SIZE: 32
```

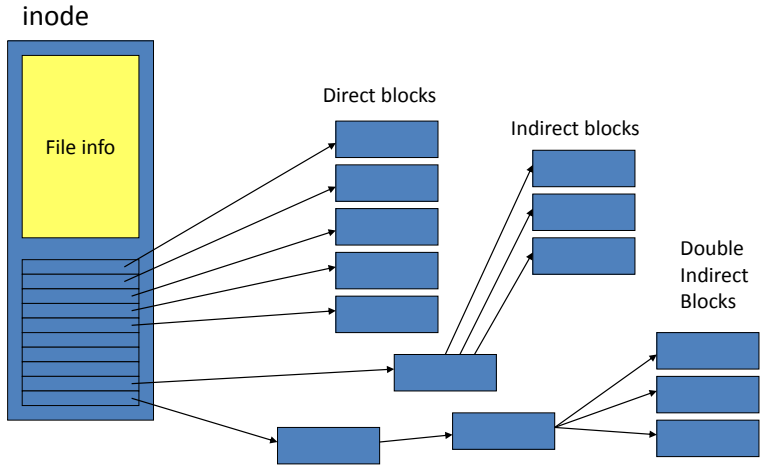

struct ext2_inode

```
struct ext2_inode {
    [0] __le16 i_mode;
    [2] __le16 i_uid;
    [4] __le32 i_size;
    [8] __le32 i_atime;
   [12] __le32 i_ctime;
   [16] __le32 i_mtime;
   [20] __le32 i_dtime;
   [24] __le16 i_gid;
   [26] __le16 i_links_count;
   [28] __le32 i_blocks;
```

struct ext2_inode

```
    [32] __le32 i_flags;
         union {
             struct {...} linux1;
             struct {...} hurd1;
             struct {...} masix1;
    [36] } osd1;
    [40] __le32 i_block[15];
   [100] __le32 i_generation;
   [104] __le32 i_file_acl;
   [108] __le32 i_dir_acl;
   [112] __le32 i_faddr;
         union {
             struct {...} linux2;
             struct {...} hurd2;
             struct {...} masix2;
   [116] } osd2;
}
SIZE: 128
```


Inode diagram



Outline

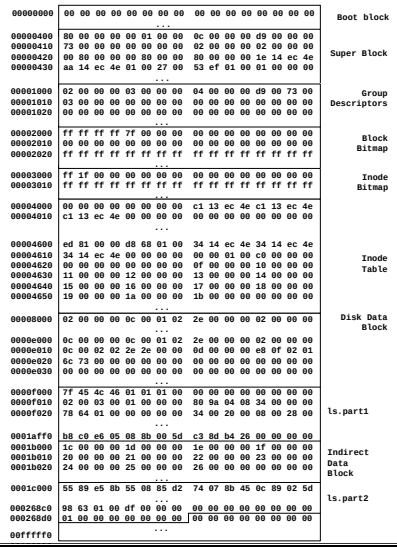
Background

A 1M disk, and formatted it using EXT2 with block size 4K → 256 blocks (4K each) and 128 `inodes` for this image. Thus, the block bit map of this disk has 32 bytes in length (256 bits), and the `inode` bit map has 16 bytes in length (128 bits).

We then copied `/bin/ls` (92,376 bytes → 23 data blocks needed) to it, and took a snapshot (`dd`) of the raw disk image. The data dump of the snapshot is shown in next slides.

After that, we removed the `ls` file, and took another snapshot of the raw disk image. Next we compared the two images and annotated the difference using the EXT2 disk data management structures, in particular, `ext2_super_block`, `ext2_group_desc`, `ext2_dir _entry`, and `ext2_inode`.

Raw Data Dump of a 1M EXT2 Disk Image



Differential Reverse Engineering: Before and After

Field Name	Offset	Size	Old Value	New Value
free_blocks_count	0x040c	4	d9 00 00 00	f1 00 00 00
free_inodes_count	0x0410	4	73 00 00 00	74 00 00 00
superblock.mtime	0x042c	4	1e 14 ec 4e	f6 14 ec 4e
superblock.wtime	0x0430	4	aa 14 ec 4e	ff 14 ec 4e
free_blocks_count	0x100c	2	d9 00	f1 00
free_inodes_count	0x1003	2	73 00	74 00
group[0].block_bitmap	0x2000	32	ff ff ff ff 7f 00	ff 7f 00 00 .. 00
group[0].inode_bitmap	0x3000	16	ff 1f 00 .. 00	ff 0f 00 .. 00
inode[11].ctime	0x458c	4	34 14 ec 4e	fa 14 ec 4e
inode[11].mtime	0x4590	4	34 14 ec 4e	fa 14 ec 4e
inode[12].ctime	0x460c	4	34 14 ec 4e	fa 14 ec 4e
inode[12].dtime	0x4614	4	00 00 00 00	fa 14 ec 4e
inode[12].links_count	0x461a	4	01 00 00 00	00 00 00 00
group[0].data(dir_entry.inode)	0xe018	4	0d 00 00 00	00 00 00 00

Table: The Value Differences Between Our Two Disk Snapshots

ext2_super_block

ext2_super_block

`ext2_super_block`: There are four fields changed: `free_blocks_count` field (at disk offset `0x040c`) gets increased from `d9` to `f1` (217 to 241, exactly 24 blocks), `free_inodes_count` field (at disk offset `0x0410`) gets increased from `73` to `74`, and `mtime` and `wtime` field (at disk offset `0x042c` and `0x0430`) gets changed as well because of the different access time. Note that at offset `0x438` in the disk layout, there is a magic number `53 ef` for EXT2 file system.

ext2_group_desc,block_bitmap,inode_bitmap

ext2_group_desc

`ext2_group_desc`: Only field `free_blocks_count` (at disk offset `0x100c`) gets increased from `d9` to `f1`, and field `free_inodes_count` (at disk offset `0x100e`) gets increased from `73` to `74`.

block_bitmap

`block_bitmap`: There are 24 bits (from 16th to 39th bit) that are changed from 1 (used) to 0 (free), indicating that there is an additional 24 free blocks. Note that our `ls` binary code occupies 23 data blocks.

inode_bitmap

`inode_bitmap`: There is only one bit (at the 13th bit) that is changed from 1 (used) to 0 (free), indicating that there is a free `inode` available.

ext2_inode

ext2_inode

`ext2_inode`: There are two (12th and 13th) `inodes` involved. For the 12th `inode` that is at offset 11 in the `inode_table`, only its `ctime` and `mtime` are changed, indicating the different creation and modification time. Note that the 12th `inode` is related to the directory entry, which has a data pointer pointing to a data block `0e 00 00 00`, which is actually at address `0xe000` because each block is 4K. For the 13th `inode` that is at offset 12 in the `inode_table`, its `ctime` and `dtime` gets changed indicating when the file was deleted. Also, its `links_count` field decreased to 0.

ext2_dir_entry

ext2_dir_entry

`ext2_dir_entry`: This particular data structure is actually in a data block because it stores the filename and the corresponding `inode` of our `ls` file. Note that only the `inode` number field gets changed from `0d 00 00 00` to `00 00 00 00`. Originally, this entry pointed to a file at `inode 0x0d` (the 13th `inode`). After the deletion, it points to `inode` 0.

Observation

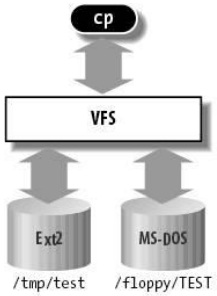
When deleting a file, the actual file content is not erased. Instead, in addition to timing related fields such as `ctime` and `dtime`, only the `links_count` field in the file content `inode` and the `inode` number in the corresponding directory entry are changed.

In the absence of file system meta data such as the `inode` and directory entry, file carving is challenging. As evidenced from our example, we cannot recover the filename that is managed in the directory entry even though the directory entry is not corrupted (overwritten) because the `inode` pointer is cleared in the corresponding directory entry. That also explains why most file caving systems cannot recover filenames. Also, we have to deal with the fragmentation.

Outline

Virtual File System

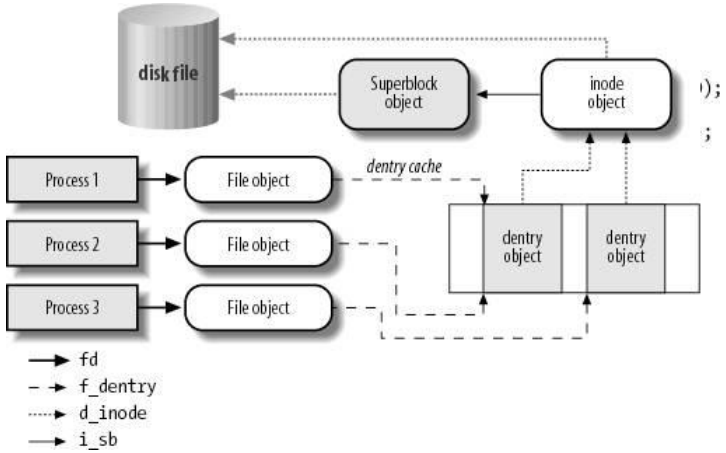
VFS is a kernel software layer that handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of file systems.




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

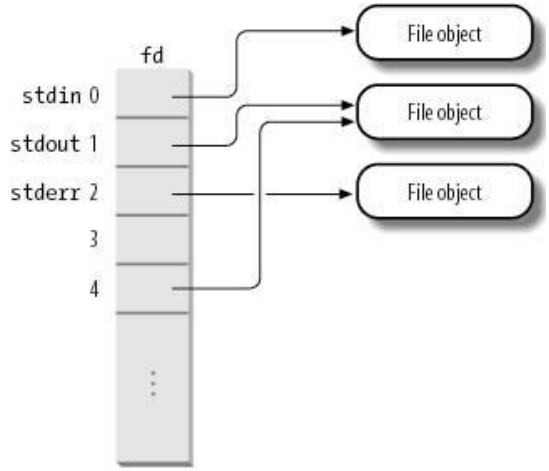
Interaction between processes and VFS objects



task_struct.files.fd_array

```
struct task_struct {
  [1120] struct fs_struct *fs;
  [1124] struct files_struct *files;
}
struct files_struct {
    [0] atomic_t count;
    [4] struct fdtable *fdt;
    [8] struct fdtable fdtab;
  [128] spinlock_t file_lock;
  [144] int next_fd;
  [148] struct embedded_fd_set close_on_exec_init;
  [152] struct embedded_fd_set open_fds_init;
  [156] struct file *fd_array[32];
}
struct fs_struct {
   [0] atomic_t count;
   [4] rwlock_t lock;
  [20] int umask;
  [24] struct dentry *root;
  [28] struct dentry *pwd;
  [32] struct dentry *altroot;
  [36] struct vfsmount *rootmnt;
  [40] struct vfsmount *pwdmnt;
```


fd_array



struct file

```
struct file {
        union {
            struct list_head fu_list;
            struct rcu_head fu_rcuhead;
     [0] } f_u;
     [8] struct dentry *f_dentry;
    [12] struct vfsmount *f_vfsmnt;
    [16] const struct file_operations *f_op;
    [20] atomic_t f_count;
    [24] unsigned int f_flags;
    [28] mode_t f_mode;
    [32] loff_t f_pos;
    [40] struct fown_struct f_owner;
    [76] unsigned int f_uid;
    [80] unsigned int f_gid;
    [84] struct file_ra_state f_ra;
   [124] long unsigned int f_version;
   [128] void *f_security;
   [132] void *private_data;
   [136] struct list_head f_ep_links;
   [144] spinlock_t f_ep_lock;
   [160] struct address_space *f_mapping;
}
```


struct dentry

```
struct dentry {
     [0] atomic_t d_count;
     [4] unsigned int d_flags;
     [8] spinlock_t d_lock;
    [24] struct inode *d_inode;
    [28] struct hlist_node d_hash;
    [36] struct dentry *d_parent;
    [40] struct qstr d_name;
    [52] struct list_head d_lru;
         union {
             struct list_head d_child;
             struct rcu_head d_rcu;
    [60] } d_u;
    [68] struct list_head d_subdirs;
    [76] struct list_head d_alias;
    [84] long unsigned int d_time;
    [88] struct dentry_operations *d_op;
    [92] struct super_block *d_sb;
    [96] void *d_fsdata;
   [100] struct dcookie_struct *d_cookie;
   [104] int d_mounted;
   [108] unsigned char d_iname[36];
}
```


struct inode

```
struct inode {
     [0] struct hlist_node i_hash;
     [8] struct list_head i_list;
    [16] struct list_head i_sb_list;
    [24] struct list_head i_dentry;
    [32] long unsigned int i_ino;
    [36] atomic_t i_count;
    [40] umode_t i_mode;
    [44] unsigned int i_nlink;
    [48] uid_t i_uid;
    [52] gid_t i_gid;
    [56] dev_t i_rdev;
    [60] loff_t i_size;
    [68] struct timespec i_atime;
    [76] struct timespec i_mtime;
    [84] struct timespec i_ctime;
    [92] unsigned int i_blkbits;
    [96] long unsigned int i_blksize;
   [100] long unsigned int i_version;
   [104] blkcnt_t i_blocks;
```


struct inode

```
struct inode {
   [112] short unsigned int i_bytes;
   [116] spinlock_t i_lock;
   [132] struct mutex i_mutex;
   [160] struct rw_semaphore i_alloc_sem;
   [188] struct inode_operations *i_op;
   [192] const struct file_operations *i_fop;
   [196] struct super_block *i_sb;
   [200] struct file_lock *i_flock;
   [204] struct address_space *i_mapping;
   [208] struct address_space i_data;
   [328] struct dquot *i_dquot[2];
   [336] struct list_head i_devices;
   [344] struct pipe_inode_info *i_pipe;
   [348] struct block_device *i_bdev;
   [352] struct cdev *i_cdev;
   [356] int i_cindex;
   [360] __u32 i_generation;
   [364] long unsigned int i_dnotify_mask;
   [368] struct dnotify_struct *i_dnotify;
   [372] struct list_head inotify_watches;
   [380] struct mutex inotify_mutex;
```

struct inode

```
struct inode {
  [408] long unsigned int i_state;
  [412] long unsigned int dirtied_when;
  [416] unsigned int i_flags;
  [420] atomic_t i_writecount;
  [424] void *i_security;
        union {
            void *generic_ip;
  [428] } u;
  [432] seqcount_t i_size_seqcount;
}
SIZE: 436
```


struct super_block

```
struct super_block {
    [0] struct list_head s_list;
    [8] dev_t s_dev;
   [12] long unsigned int s_blocksize;
   [16] unsigned char s_blocksize_bits;
   [17] unsigned char s_dirt;
   [20] long long unsigned int s_maxbytes;
   [28] struct file_system_type *s_type;
   [32] struct super_operations *s_op;
   [36] struct dquot_operations *dq_op;
   [40] struct quotactl_ops *s_qcop;
   [44] struct export_operations *s_export_op;
   [48] long unsigned int s_flags;
   [52] long unsigned int s_magic;
   [56] struct dentry *s_root;
   [60] struct rw_semaphore s_umount;
   [88] struct mutex s_lock;
  [116] int s_count;
```


struct super_block

```
struct super_block {
  [120] int s_syncing;
  [124] int s_need_sync_fs;
  [128] atomic_t s_active;
  [132] void *s_security;
  [136] struct xattr_handler **s_xattr;
  [140] struct list_head s_inodes;
  [148] struct list_head s_dirty;
  [156] struct list_head s_io;
  [164] struct hlist_head s_anon;
  [168] struct list_head s_files;
  [176] struct block_device *s_bdev;
  [180] struct list_head s_instances;
  [188] struct quota_info s_dquot;
  [364] int s_frozen;
  [368] wait_queue_head_t s_wait_unfrozen;
  [392] char s_id[32];
  [424] void *s_fs_info;
  [428] struct mutex s_vfs_rename_mutex;
  [456] u32 s_time_gran;
}
SIZE: 460
```


struct file_system_type

```
struct file_system_type {
   [0] const char *name;
   [4] int fs_flags;
   [8] int (*get_sb)(struct file_system_type *, int,
        const char *, void *, struct vfsmount *);
  [12] void (*kill_sb)(struct super_block *);
  [16] struct module *owner;
  [20] struct file_system_type *next;
  [24] struct list_head fs_supers;
  [32] struct lock_class_key s_lock_key;
  [32] struct lock_class_key s_umount_key;
}
SIZE: 32
```

pipefs, proc, sockfs, sysfs, rootfs, ext2, ext3, jfs...

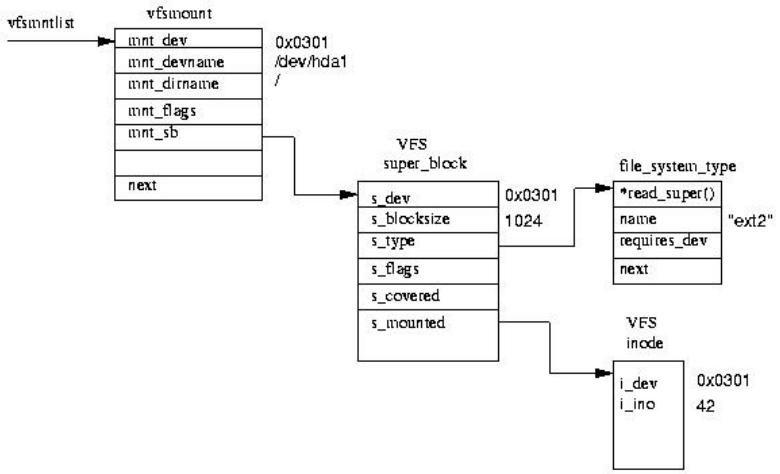
struct namespace

```
struct task_struct {
    [1128] struct namespace *namespace;
}
struct namespace {
   [0] atomic_t count;
   [4] struct vfsmount *root;
   [8] struct list_head list;
  [16] wait_queue_head_t poll;
  [40] int event;
}
SIZE: 44
```


struct vfsmount

```
struct vfsmount {
    [0] struct list_head mnt_hash;
    [8] struct vfsmount *mnt_parent;
   [12] struct dentry *mnt_mountpoint;
   [16] struct dentry *mnt_root;
   [20] struct super_block *mnt_sb;
   [24] struct list_head mnt_mounts;
   [32] struct list_head mnt_child;
   [40] atomic_t mnt_count;
   [44] int mnt_flags;
   [48] int mnt_expiry_mark;
   [52] char *mnt_devname;
   [56] struct list_head mnt_list;
   [64] struct list_head mnt_expire;
   [72] struct list_head mnt_share;
   [80] struct list_head mnt_slave_list;
   [88] struct list_head mnt_slave;
   [96] struct vfsmount *mnt_master;
  [100] struct namespace *mnt_namespace;
  [104] int mnt_pinned;
}
SIZE: 108
```


Navigated Mounted File System



Outline

File system

Design

- Meta-data (superblock, inode, dentry)
- File allocation, deallocation

Implementation

- Disk Interface (Disk data layout)
- Kernel Interface (VFS)

References

- http://en.wikipedia.org/wiki/File_system
- UNIX Operating System Design
- Understanding Linux Kernel (3rd edition)
- “Bin-Carver” paper in DFRWS 2012.
- Some slides are compiled from:
  - http://www.os-book.com/
  - http://www.stanford.edu/ ouster/cgi-bin/cs140-winter12/index.php
  - http://ppi.fudan.edu.cn/os/