Operating Systems

13. File Systems

Paul Krzyzanowski

Rutgers University

Spring 2015

Terminology

What's a file system?

- Traditionally
  - A way to manage variable-size persistent data
    - Organize, store, retrieve, delete information
  - Random access
    - Arbitrary files can be accessed by name
    - Arbitrary parts of a file can be accessed
  - File systems are implemented on top of block devices

- More abstract
  - A way to access information by name
    - Devices
    - System configuration, process info, random numbers

Terms

- Disk
  - Non-volatile block-addressable storage.
- Block = sector
  - Smallest chunk of I/O on a disk
  - Common block sizes = 512 or 4096 (4K) bytes
    E.g., WD Black Series 4TB drive has 7,814,037,168 512-byte sectors
- Partition
  - Set of contiguous blocks on a disk. A disk has ≥ 1 partitions
- Volume
  - Disk, disks, or partition that contains a file system
  - A volume may span disks

More terms

- Track
  - Blocks are stored on concentric *tracks* on a disk
- Cylinder
  - The set of all blocks on one track
    (obsolete now since we don't know what's where)
- Seek
  - The movement of a disk head from track to track

File Terms

- File
  - A unit of data managed by the file system
- Data: (Contents)
  -
  - Unstructured (byte stream) or structured (records)
- Name
  - A textual name that identifies the file

File Terms

- Metadata
  - Information about the file (creation time, permissions, length of file data, location of file data, etc.)
- Attribute
  - A form of metadata – a textual name and associated value (e.g., source URL, author of document, checksum)
- Directory (folder)
  - A container for file names
  - Directories within directories provide a hierarchical name system

File System Terms

- Superblock
  - Area on the volume that contains key file system information
- inode (file control block)
  - A structure that stores a file's metadata and location of file data
- Cluster
  - Logical block size used in the file system that is equivalent to *N* blocks
- Extent
  - Group of contiguous clusters identified by a starting block number and a block count

Design Choices

Namespace

Flat, hierarchical, or other?

Multiple volumes

Explicit device identification (A:, B:, C:, D:)

or integrate into one namespace?

File types

Unstructured (byte streams)

or structured (e.g., indexed files)?

File system types

Support one type of file system

or multiple types (iso9660, NTFS, ext3)?

Metadata

What kind of attributes should the file system have?

Implementation

How is the data laid out on the disk?

Working with the Operating System

File System Operations

Formatting

- Formatting
  - Low-level formatting
    Identify sectors, CRC regions on the disk
    Done at manufacturing time; user can reinitialize disk
  - Partitioning
    Divide a disk into one or more regions
    Each can hold a separate file system
  - High-level formatting
    Initialize a file system for use
- Initializing a file system
  - Initialize size of volume
  - Determine where various data structures live:
    - Free block bitmaps, inode lists, data blocks
    - Initialize structures to show an empty file system

Mounting

- Make file system available for use
- *mount* system call
  - Pass the file system type, block device & mount point
- Steps
  - Access the raw disk (block device)
  - Read superblock and file system metadata (free block bitmaps, root directory, etc.)
  - Check to see if the file system was properly unmounted (clean?)
    - If not, validate the structure of the file system
  - Prepare in-memory data structures to access the volume
    - In-memory version of the superblock
    - References to the root directory
    - Free block bitmaps
  - Mark the superblock as “dirty”

Unmounting

- Ensure there are no processes with open files in the file system
- Remove file system from the OS name space
- Flush all in-memory file system state to disk
- Mark the superblock as “clean” (unmount took place)

File System Validation

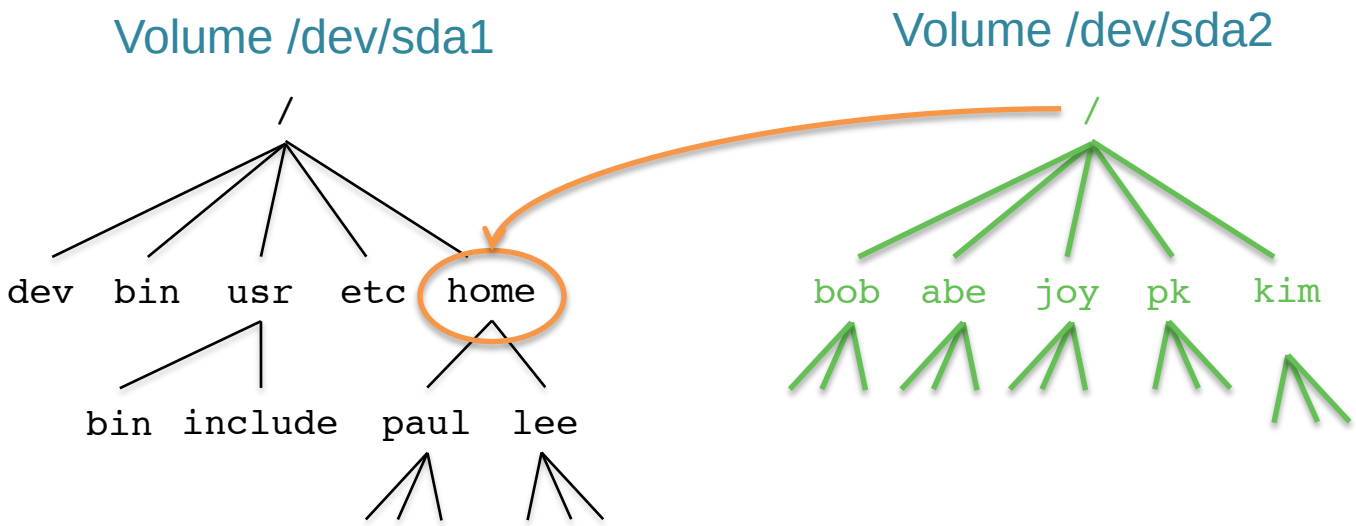
- OS performs file system operations in memory first
  - Block I/O goes to the buffer cache
- Not all blocks might be written to the disk if the system shuts down, crashes, or the volume is removed
  - This can leave the file system in an inconsistent state
- File system check program (e.g., *fsck* on POSIX systems)

File System Validation: checks (example)

- check each file size and its list of blocks
  - File size matches list of blocks allocated to the file
- check pathnames
  - Directory entries point to valid inodes
  - Proper parent-child links and no loops
- check connectivity
  - Unreferenced files and directories
- check reference counts (link counts in inode)
  - Link counts & duplicate blocks in files and directories
- check free block bitmaps
  - blocks marked free should really be free
  - free counts should reflect bitmap data

Mounting: building up a name space

- Combine multiple file systems into a single hierarchical name space
- The mounted file system overlays (& hides) anything in the file system under that mount point
- Looking up a pathname may involve traversing multiple mount points

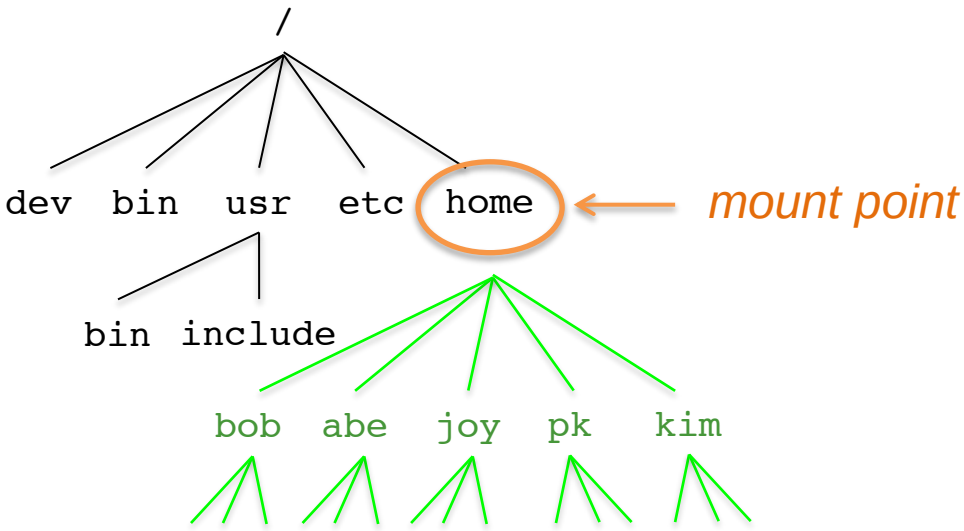



```
mount -t ext4 /dev/sda2 /home
```

`/home` becomes a mount point for `/dev/sda2`

Mounting: building up a name space

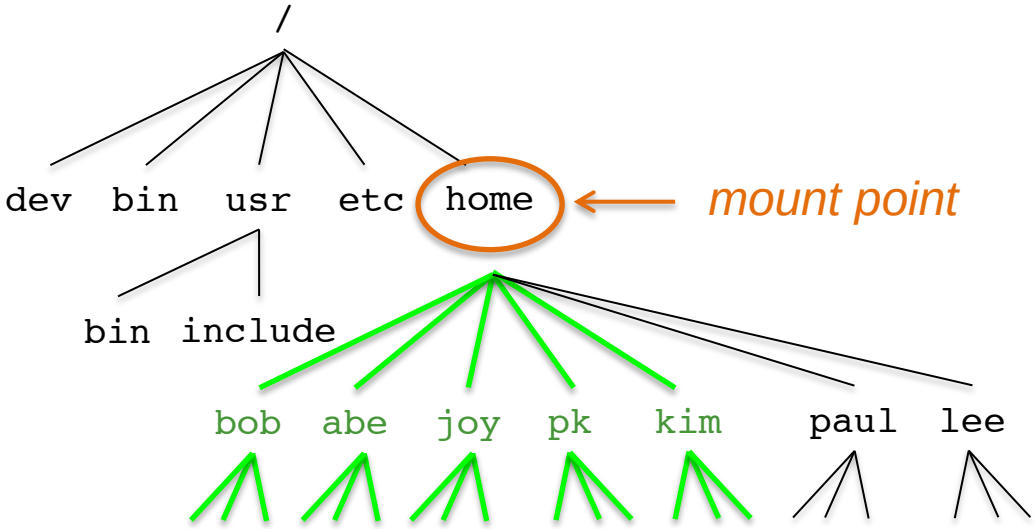
Volume /dev/sda1



`/home/paul` and `/home/lee` are no longer visible

Union mounts

Mounted file system merges the existing namespace



Considerations:

- Search path (what if two names are the same in the file systems)?
- Where to write?

Create a file

- Create an inode to hold info (metadata) about the file
  - Initialize timestamps
  - Set permissions/modes
  - Set size = 0
- Add a directory entry for the new inode
  - Directory entry = set of { filename, inode #}
  - Use current directory or pathname specified by filename

Create a directory

- A directory is just like a file
  - Contents = set of {name, inode} pairs

- Steps
  - Create a new inode (& initialize)
  - Initialize contents to contain
    - A directory entry to the parent (name = "`..`")
    - A directory entry to itself (name = "`.`") – on POSIX systems

Links to files

- Symbolic link
  - A file's contents contain a link to another file or directory

```
ln -s current_file new_file
```

  - If you delete `current_file`, then `new_file` will have a *broken link*

- Hard link (alias)
  - A new directory entry is created for the same inode.
  - Inodes contain link counts
  - A file is deleted when the link count = 0

```
ln current_file new_file
```

Open a file

Steps

- Lookup: scan one or more directories to find the name
  - *namei*: name to inode lookup
  - Pathname traversal
  - Mount point traversal
- Get info & verify access
  - Read the inode (from the directory entry)
  - Check access permissions & ownership
  - Allocate in-memory structure to store info about open file
- Return a file handle (file descriptor)
  - Index into an *open files* table for the process
  - The process uses the file descriptor for operations on this file

Write to a file

- OS keeps track of current read/write offset in an open file (seek pointer)
  - Can be modified (*lseek* system call)
- Steps
  - If the file is going to grow because of the write:
    - Allocate extra disk blocks (if needed) – update free block bitmap
  - Read file data if not writing on a block boundary
  - Write one or more blocks of data from memory to disk
  - Update file size
  - Update the current file offset in memory
- Writes are usually buffered in memory & delayed to optimize performance
  - Buffer cache

Read from a file

Steps

1. Check size of file to ensure no read past end of file

2. Identify one or more blocks that need to be read
   - Information is in inode, usually cached in memory

3. May need to read additional blocks to get the block map to find the desired block numbers

4. Increment the current file offset by the amount that was read

Delete a file

- Remove the file from its directory entry
  - This stops other programs from opening it – they won't see it
- If there are no programs with open references to the file AND there are no hard links to the file
  - Mark all the blocks used by the file as free
  - Mark the inode used by the file as free
  - Check this condition when closing a file (or exiting a process)
- This allows processes to continue accessing a file even after it was deleted

Rename a file

- If source & destination directories are the same
  - Check that old and new names are different
- If the source is a directory (rename a directory)
  - Check that destination is not its subdirectory – *avoid loops*
- If the destination name exists
  - If it's a file then delete the destination file
- Either
  - Link the destination name into the destination directory
  - Link the source file name to the destination file name
- Delete the source file name

Read a directory

- Directories are like files but contain a set of {*name, inode*} tuples

- The file system implementation parses the storage structure
  - You don't have to deal with list vs. B+ tree formats

- Operations:
  - *opendir*: open a directory for reading
  - *readdir*: iterate through the contents of the directory
  - *closedir*: close a directory entry

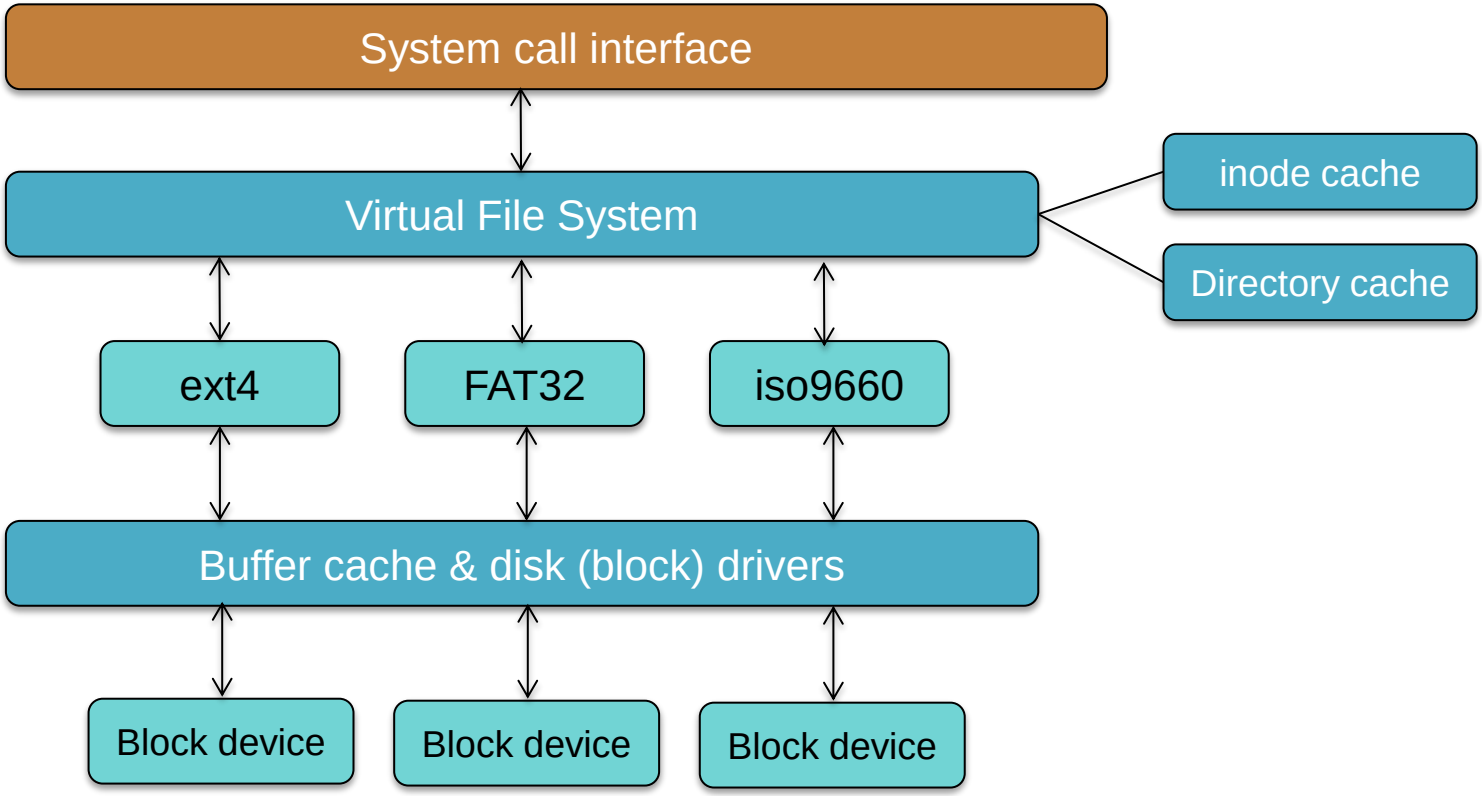
Read & Write metadata

- Read inode information
  - *stat* system call
- Write metadata: calls to change specific fields
  - *chown*: change owner
  - *chgrp*: change group
  - *chmod*: change permissions
  - *utime*: change access & modification times
- Extended attributes (name-value sets)
  - *listxattr*: list extended attributes
  - *getxattr*: get a value of given extended attribute
  - *setxattr*: set an extended attribute
  - *removexattr*: remove extended attribute

Operating System Interfaces for File Systems

Virtual File System (VFS) Interface

- Abstract interface for a file system object
- Each *real* file system interface exports a common interface



Keeping track of file system types

Like drivers, file systems can be built into the kernel or compiled as loadable modules (loaded at mount)

- Each file system registers itself with VFS
- Kernel maintains a list of file systems

```
struct file_system_type {
        const char *name;     name of file system type
        int fs_flags;         requires device, fs handles moves, kernel-only mount, …
        struct super_block *(*get_sb)(struct file_system_type *,
                int, char *, void *, struct vfsmount *); set up superblock
        void (*kill_sb) (struct super_block *);     call to clean up at unmount
        struct module *owner;     module that owns this
        struct file_system_type *next;     next file system type in list
        struct list_head fs_supers;     list of all superblocks of this type
        struct lock_class_key s_lock_key;     used for lock validation (optional)
        struct lock_class_key s_umount_key; used for lock validation (optional)
};
```

Keeping track of mounted file systems

- Before mounting a file system, first check if we know the file system type: look through the `file_systems` list
  - If not found, the kernel daemon will load the file system module

```
/lib/modules/3.13.0-46-generic/kernel/fs/ntfs/ntfs.ko
/lib/modules/3.13.0-46-generic/kernel/fs/hfsplus/hfsplus.ko
/lib/modules/3.13.0-46-generic/kernel/fs/jffs2/jffs2.ko
/lib/modules/3.13.0-46-generic/kernel/fs/minix/minix.ko
…
```

- The kernel keeps a linked list of mounted file systems:

```
current->namespace->list
```

- Check that the mount point is a directory and nothing is already mounted there

VFS: Common set of objects

- Superblock: Describes the file system
  - Block size, max file size, mount point
  - One per mounted file system
- inode: represents a single file
  - Unique identifier for every object (file) in a specific file system
  - File systems have methods to translate a name to an inode
  - VFS inode defines all the operations possible on it
- dentry: directory entries & contents
  - Name of file/directory, child dentries, parent
  - Directory entries: translations of names to inodes
- file: represents an open file
  - VFS keeps state: mode, read/write offset, etc.

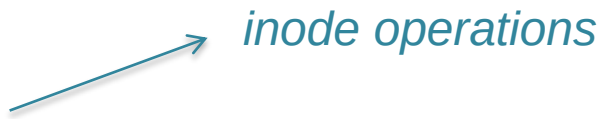
VFS superblock

- Structure that represents info about the file system
- Includes
  - File system name
  - Size
  - State
  - Reference to the block device
  - List of operations for managing inodes within the file system:
    - *alloc_inode*, *destroy_inode*, *read_inode*, *write_inode*, *sync_fs*, …

VFS inode

- Uniquely identifies a file in a file system
- Access metadata (attributes) of the file (except name)

```
struct inode {
        unsigned long i_ino;
        umode_t i_mode;
        uid_t i_uid;
        gid_t i_gid;
        kdev_t i_rdev;
        loff_t i_size;
        struct timespec i_atime;
        struct timespec i_ctime;
        struct timespec i_mtime;
        struct super_block *i_sb;
        struct inode_operations *i_op;
        struct address_space *i_mapping;
        struct list_head i_dentry;
        ...
}
```



VFS inode operations

Functions that operate on file & directory *names and attributes*

```
struct inode_operations {
        int (*create) (struct inode *, struct dentry *, int);
        struct dentry * (*lookup) (struct inode *, struct dentry *);
        int (*link) (struct dentry *, struct inode *, struct dentry *);
        int (*unlink) (struct inode *, struct dentry *);
        int (*symlink) (struct inode *, struct dentry *, const char *);
        int (*mkdir) (struct inode *, struct dentry *, int);
        int (*rmdir) (struct inode *, struct dentry *);
        int (*mknod) (struct inode *, struct dentry *, int, dev_t);
        int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char *,int);
        int (*follow_link) (struct dentry *, struct nameidata *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *, const void *, size_t, int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
};
```

VFS File operations

Functions that operate on file & directory *data*

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                        unsigned long, unsigned long);
};
```

VFS File operations

Not all functions need to be implemented!

Example: The same file_operations are used for a character device driver

```
struct file_operations mydriver_fops = {
    .owner = MYFS_MODULE;
    .open = myfs_open;          /* allocate resources */
    .read = myfs_read_file;
    .write = myfs_write_file;
    .release = myfs_release;  /* release resources */
    /* llseek, readdir, poll, mmap, readv, etc. not implemented */
};

register_filesystem(&myfs_type)
```

```
static struct file_system_type myfs_type = {
  .owner   = THIS_MODULE;
  .name    = "myfs";
  .get_sb = myfs_get_super,
  .kill_sb = myfs_kill;
};
```

The End