File Management

- Stallings, Chapter 12 (Reading: 12.1, 12.7 – 12.9)
- Topics (quick overview)
  - Tree-structured file system, directories
  - Traditional Unix file management
  - Linux file management
  - Virtual file system (VFS)

Functions of File Management

- Provide a means to store data (files) on secondary memory (hard disk, SSD, flash card, CDROM, etc.)
- Need a structure to describe the location of files plus their attributes – this is the file system
- Other responsibilities:
  - Provide access control to files among users
  - Manage free space (blocks) on secondary storage
    - Allocate free blocks to files
    - Collect free blocks when files are deleted

Managing Records vs. Managing Bytes

- We can look at files from two perspectives
- From the application point of view:
  - Files are collections of (application-specific) records
  - E.g., insurance-related software and associated databases
- From the (general-purpose) operating system view:
  - files are sequences of bytes on secondary storage
  - operating system keeps track of the structure on disk
  - operating system implements application requests to read and manipulate files
- Much of Chapter 12 deals with the former. We'll focus on the OS role.

Application vs. OS roles

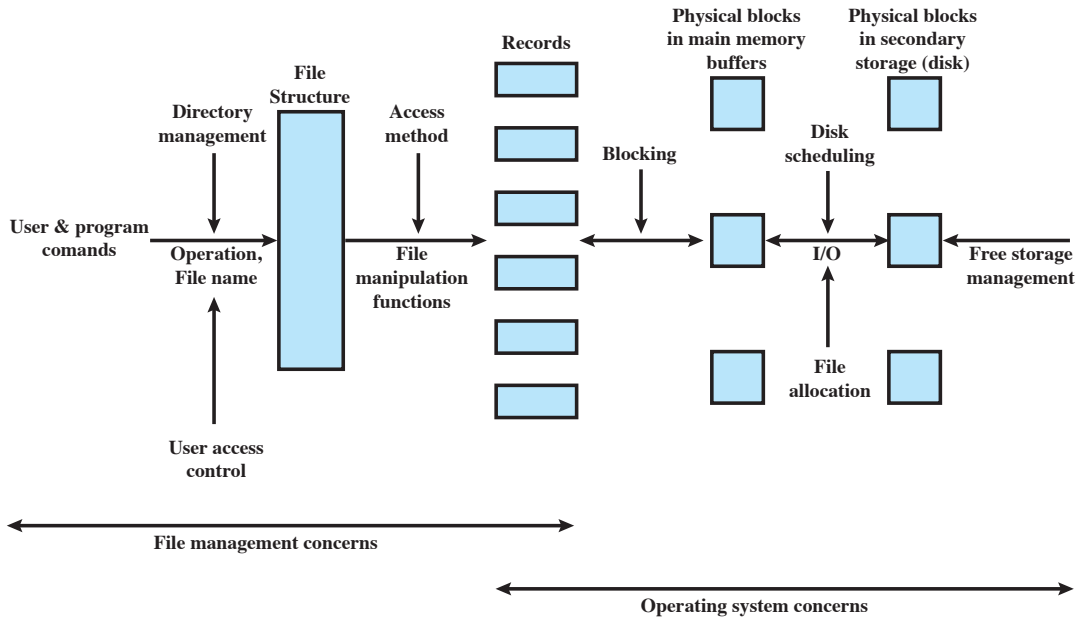


Figure 12.2 Elements of File Management

File Systems

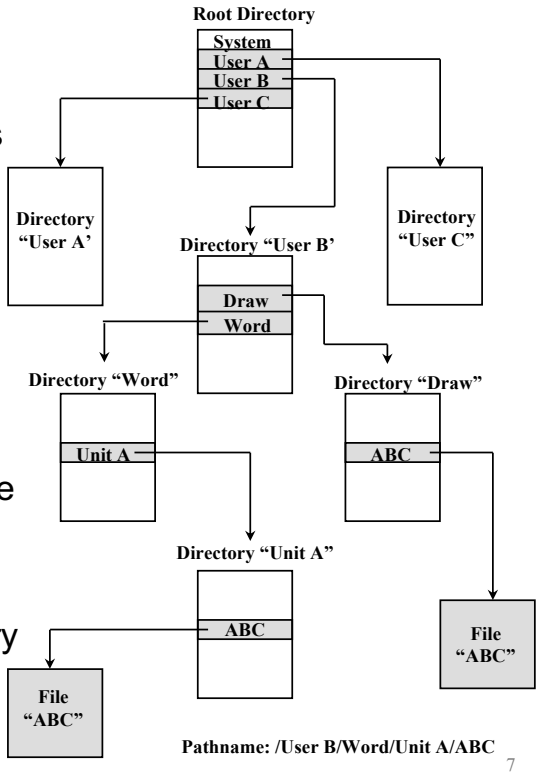
- A computer system might be dedicated to a particular business or scientific application
- Therefore, the system provides (application-level) support for specific database types needed by the application software
- But, to the underlying general-purpose OS, those databases are just collections of files
- The organization and structure of files visible to, and maintained by, the operating system, is referred to as a file system (or filesystem).

Directories (a.k.a. Folders)

- Since the early days of computing and file systems has been the concept of a **directory**
  - Provides mapping between file names and the files themselves
  - Contains (or points to) information about files
    - attributes
    - location
    - ownership
- Important: A directory is itself a **file**!
- Therefore, directories can contain directories
- Result is a tree-like structure

Tree-Structured File System

- Master (root) directory with user and system directories underneath it
- Each directory may have subdirectories and files as entries
- Files can be located by following a path from the root directory down various branches
  - this is the pathname for the file
  - E.g., /usr/local/bin/gcc
- Can have several files with the same file name as long as they have unique path names
- Current directory is the working directory
- Files are referenced relative to the working directory



A Word about File Access Control…

- Many types of access control have been studied and implemented
- In some systems, a complex access control method might be applied to a database by the application
- But ultimately those techniques must be implemented using mechanisms supported by the operating system. Examples:
  - traditional Unix user ID bit-oriented permissions
  - capabilities – like keys that can be passed around/modified
  - access control lists – list of users/permissions tied to file
  - role-based security – roles associated with job function(s)

Traditional Unix permission (POSIX standard)

- Structure:
  - type, owner-rwx, group-rwx, others-rwx
  - rwx = read, write, execute
- Types:
  - '-' regular file, 'd' directory, 'c' char-dev, 'b' blk-dev
  - 'l' symbolic link, 'p' pipe, 's' socket
- Unix groups
  - a user might belong to several groups
  - can use 'newgrp' command to create new shell with different group

Setuid and Setgid bits

- In some cases, an executable invoked by a user needs to access files that are protected.
- Setuid bit on an executable file enables that program to execute with effective uid of the owner of the file (e.g., root), not the user who invoked the command
- Examples:
  - traditional passwd command edits /etc/passwd directly
    - (modern smbpasswd does not work this way so is not setuid)
  - ping command tests whether a network host is reachable
    - needs access to a protected type of socket (raw)

```
<651 arctic:~ >ls -l /bin/ping
-rwsr-xr-x 1 root root 36136 Apr 12  2011 /bin/ping
```

Sticky bit

- Historically, the sticky bit on a popular executable meant that the program would remain in swap space, even after last use.
- Nowadays, it mainly is used for directories.
- If the sticky bit is set on a directory, then for a given file in that directory, only the owner of the file, owner of the directory, or root, can delete or rename the file
- Example: /tmp is writeable by all users, but one user should not be able to delete the file of another user.

```
<633 ned:~ >ls -ld /tmp
drwxrwxrwt 15 root root 360 Apr 11 22:17 /tmp
```


chmod command

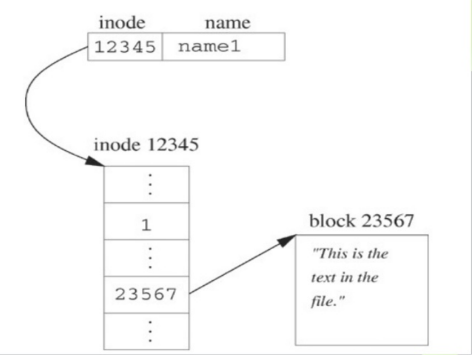
- Used to change the permissions on a Unix/Linux file
- Absolute modes: (see man page for symbolic modes)

```
     4000    (the set-user-ID-on-execution bit) Executable files with this bit set
             will run with effective uid set to the uid of the file owner.  Directo-
             ries with the set-user-id bit set will force all files and sub-directo-
             ries created in them to be owned by the directory owner and not by the
             uid of the creating process, if the underlying file system supports this
             feature: see chmod(2) and the suiddir option to mount(8).
     2000    (the set-group-ID-on-execution bit) Executable files with this bit set
             will run with effective gid set to the gid of the file owner.
     1000    (the sticky bit) See chmod(2) and sticky(8).
     0400    Allow read by owner.
     0200    Allow write by owner.
     0100    For files, allow execution by owner.  For directories, allow the owner
             to search in the directory.
     0040    Allow read by group members.
     0020    Allow write by group members.
     0010    For files, allow execution by group members.  For directories, allow
             group members to search in the directory.
     0004    Allow read by others.
     0002    Allow write by others.
     0001    For files, allow execution by others.  For directories allow others to
             search in the directory.
:
```

Directories

- Directories are simply a particular type of file
- Contains (filename,inode) pairs
- Inode is a structure, which **resides on disk**, containing meta-information about a file
  - type, permissions, owner, timestamps, location on disk, etc.

File Name	Inode #
⋮	⋮
"a"	97612
"b"	97612
⋮	⋮



Traditional Unix File Management

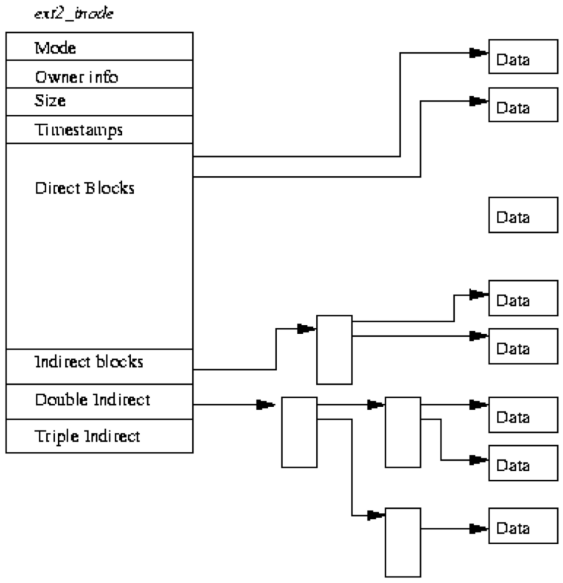
- Files are streams of bytes
- Types of files
  - regular - contents entered by user or program
  - directory - contains list of file names and “pointers” to inodes of the files in the directory
  - symbolic link – contains path of target file
    - Compare to hard link (later)
  - device file - used to access drivers of peripheral device
  - socket – provides access to network protocols
  - named pipe – little used mechanism for interprocess communication

Inodes

- Each file is represented by an inode
  - **inodes reside on disk as part of the file system**
  - copies of (recently referenced) inodes are also in memory
  - inode defines ownership, permissions, status of the file
  - file type: plain (regular), directory, symbolic link, character device, block device, socket, etc.
- Mapping path name to inode is responsibility of kernel
  - follow path name and read inodes/directories along the way
  - in Linux, a special directory cache is used for just this purpose
- Example:
  - /user/mckinley/courses/cse410/slides/4-2-files.pptx
  - how many directories in this path?

Example Inode Structure

- Ext2 file system (for many years the Linux default)
- Since then, ext3 and ext4 have been added
- Mode field describes permissions as we have seen: -rwxrwxrwx bits
- Owner uid
- Size
- Timestamps?
- Where is the filename?



Directories, inodes and files

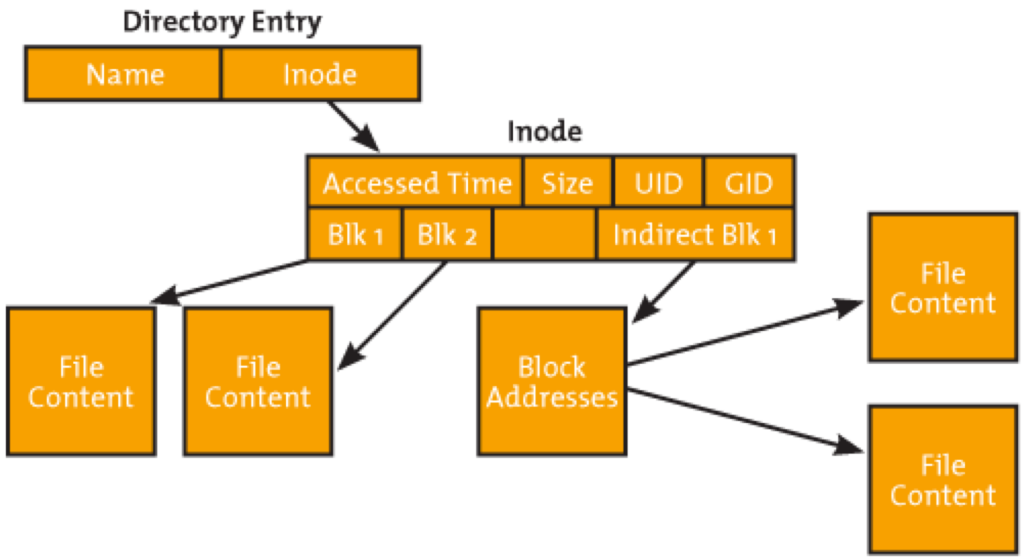


FIGURE 1 RELATIONSHIP BETWEEN THE DIRECTORY ENTRY, AN INODE, AND BLOCKS OF AN ALLOCATED FILE

Accessing files/directories in Linux

- #include <sys/types.h>
- #include <dirent.h>
- opendir(const char *dirname) opens the directory located at dirname.
- readdir(DIR dir obj) returns a pointer to the next directory entry of type (struct dirent) or NULL when there are no more entries.
- closedir(DIR dir obj) closes and frees the directory object.

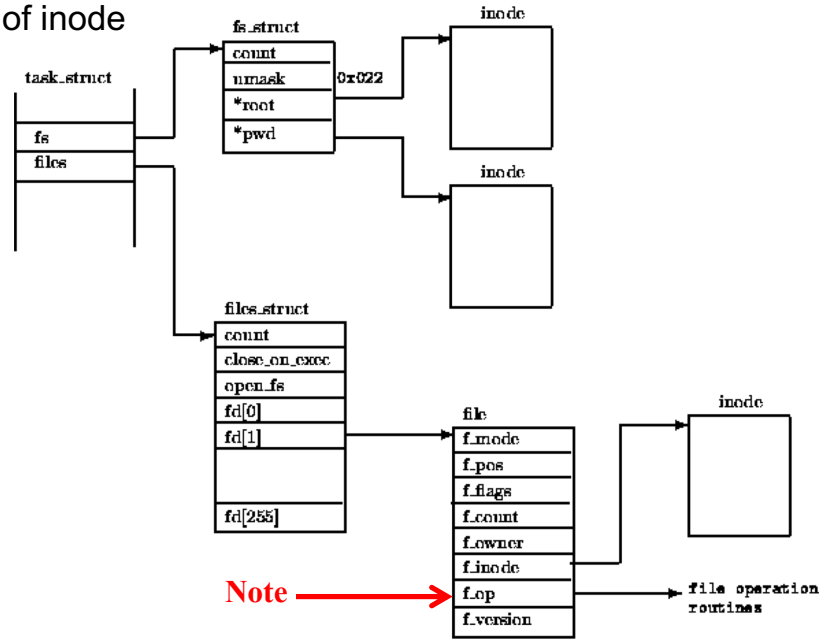
Directory Entry (in Linux)

```
struct dirent {
    ino_t       d_ino;        /* inode number */
    off_t       d_off;        /* offset to the next dirent */
    unsigned short d_reclen;  /* length of this record */
    unsigned char  d_type;    /* type of file; not supported
                                 by all file system types */
    char           d_name[256];  /* filename */
};
```

- BTW, why do we have to read one at at time? Why not just read the whole directory at once?

Under the Hood: Linux kernel data structures

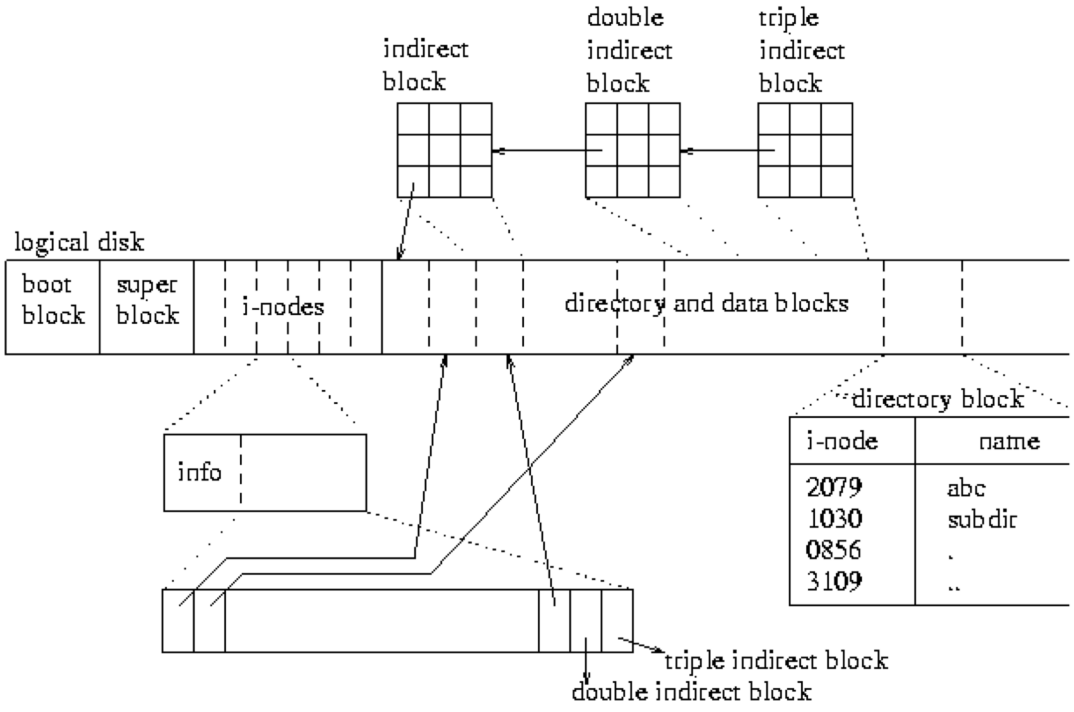
- Files are opened by applications
  - returned file descriptor indexes into table of open files
  - entry there points to file structure table entry, which in turn points to in-memory copy of inode



Traditional Unix File System Layout

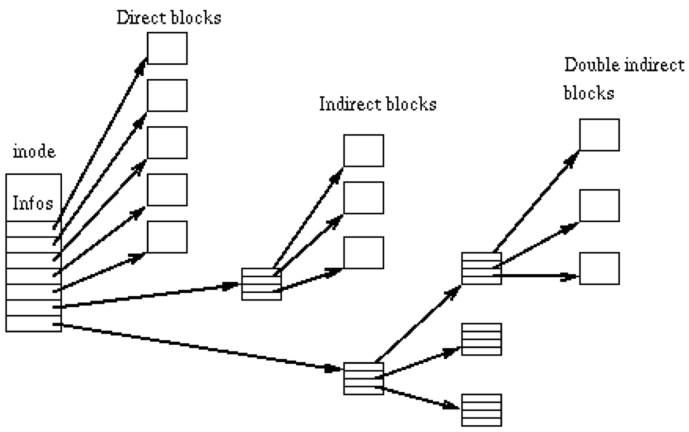
- Contents of file system
  - Boot block (bootstrap code, typically first disk sector)
  - Superblock: describes state of file system
    - size of file system, number of free blocks
    - list of free blocks, index of next free block
    - size of inode list, number of free inodes
    - list of free inodes (cache), index of next free inode
  - Inode list: includes the root inode
    - linear list of inodes, each has type field - 0 (available), 1 (used)
  - Data blocks (sectors on disk): each block can belong to one file
    - From 1950s to 2000s, most disk sectors (blocks) were 512 bytes
    - Newer disks use 4096-byte sectors

Traditional layout



Direct and Indirect Blocks

- Inode contains
  - Metadata (size, device ID, owner, type, permissions, etc.)
  - 12 direct pointers, 1 indirect, 1 double indirect, 1 triple indirect pointers



- Maximum file size using 4KB blocks?

Performance problems with this design?

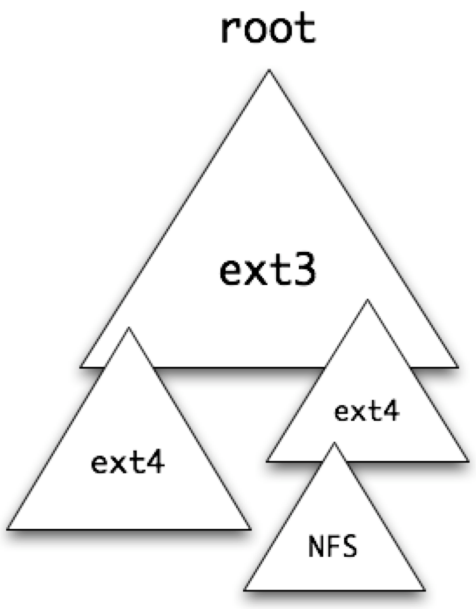
- free list eventually becomes distributed randomly over the disk as files are created and destroyed
  - hence, files are widely distributed over disk
- also, files in same directory are not necessarily near one another
- inodes are not near their respective files

Mounting a File System

- Files are not accessed directly through a device driver (via a device file) but rather through the tree-structured file system
- Therefore, we need to map the root of the tree to a physical location on disk – this is called mounting
- Root file system – contains files needed to boot
- Other file systems (laid out in disk partitions) can be **mounted** at various branches (directories) of an existing file system
  - The directory is called a mount point
  - It is "covered up" while the new file system is mounted

File system mounting

- Imagine adding and removing whole branch structures in a tree
- File systems can be different types



Mounting (cont.)

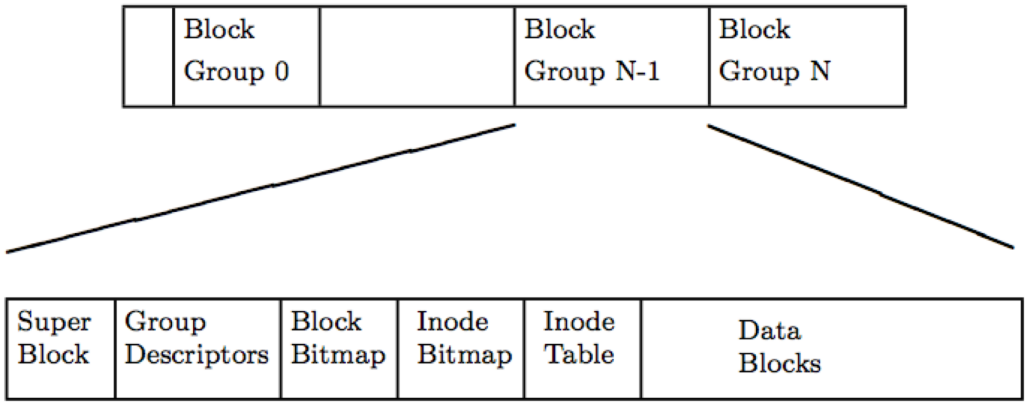
- File systems to be mounted at boot time, including the root file system (mounted on '/') are listed in a file (/etc/fstab in Linux)
- Other file systems can be mounted as the system executes
  - Manually:
    e.g., `mount –t type device mount_point`
  - `mount -t iso9660 -o ro /dev/cdrom /mnt`
  - `mount –t ext3 /dev/sda6 /var`
  - `umount dir  (to unmount)`
  - Automatically (e.g., CDROM, flash drive)

Example File Systems

- Linux
  - ext2 (derived from Berkeley Fast File System)
  - ext3, ext4, XFS, JFS, …
- OS X
  - HFS Plus (inherited from original MacOS)
  - UFS (Unix File System – basically FFS)
- Windows
  - FAT (File Allocation Table – traditional)
  - NTFS – default for Windows NT family
  - exFAT– optimized for flash drives (lower overhead than NTFS)

Linux EXT2 File System

- Designed to address problems with traditional Unix FS
- Heavily influenced by BSD Fast File Sytem (FFS)
- Logical partitions divided into block groups
- Superblock replicated on each block group for fault tolerance



EXT2 Groups

- Group descriptor describes a block group; all group descriptors are replicated in each group
- Superblock also replicated in each group
- Contents of group descriptor
  - data block bitmap for allocation/deallocation
  - inode bitmap
  - first block of inode table
  - free block count
  - free inode count
  - …
- EXT2 tries to allocate new blocks for a file in 8-block chunks in same block group

Ext2 Superblock

- Read into memory when file system is mounted
- Each block group contains a duplicate in case of corruption
- Contents
  - magic number (0xEF53) indicating this is ext2 file system
  - mount count and maximum mount count. Purpose?
  - block group number - which group holds this copy
  - block size (e.g., 4096 bytes)
  - blocks per group
  - number of free blocks in fs
  - number of free inodes in fs
  - inode number of first inode in fs (directory entry for '/')

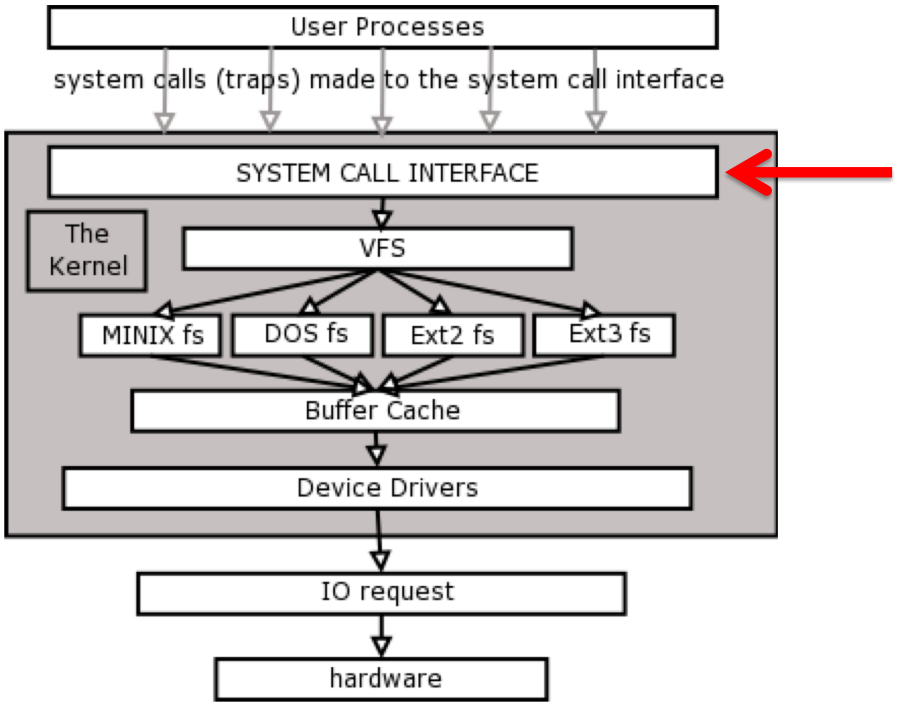
Ext3, Ext4

- Ext3 (2001) adds journaling
  - separate area of file system reserved to track all changes to the file system
  - greatly reduces chance of corruption when system crashes
  - many CSE file systems are ext3
- Ext4 (2008)
  - supports files up to 16 TB
  - some improvements in journaling, multiple block allocation

Linux Virtual File System (VFS)

- Linux (and other operating systems) support many different types of file systems simultaneously
  - ext, ext2, minix, msdos, vfat, …
- VFS hides details of particular file system from user programs, and from the rest of the kernel
- Inodes, superblocks, access routines of individual files systems are all mapped onto **generic** data structures and routines (latter exist only in memory, not on disk)

VFS structure



VFS Features

- Provides processes with transparent access to many types of local file systems and to remote file systems
- Maintains data structures that describe the whole (virtual) file system and the real, mounted file systems
- Maintains (VFS) superblocks and (VFS) inodes, just like real file systems, BUT these exist only in memory, not on disk, and are constructed based on information in their “real” counterparts
- VFS only maintains data structures for file systems currently mounted and files currently in use.

VFS Operation

- As each (mounted) file system is initialized at boot time, it registers itself with the VFS
- A file system can be either built into kernel or built as a loadable module.
- When a file system is mounted, VFS reads its superblock and maps this information onto a VFS superblock structure.
- VFS keeps a list of mounted file systems together with their superblocks.
- And how does this support multiple file system types?

VFS Operation (cont.)

- Each VFS superblock contains pointers to routines that perform **file-system-specific** functions, such as reading an inode.
- E.g., the `read_inode` routine in the kernel (fs/inode.c) is generic and just calls the operation in the superblock, which fills in a (VFS!) inode:

```
/*
 * This gets called with I_LOCK held: it needs
 * to read the inode and then unlock it
 */
static inline void read_inode(struct inode *inode, struct
                                super_block *sb)
{
        sb->s_op->read_inode(inode);
        unlock_inode(inode);
}
```

Key: level of indirection!!

VFS Inodes

- A VFS inode is a **kernel data structure** that is filled in when the (real) inode is read from disk.
- As with the superblock operations, the functions defining the operations on the inode are **filled in as the system executes** and depend on the file system type (`i_op` field)
- And we have already seen that operations on files (read, write, etc.) depend on the file type (`f_op` field of `file` structure)
- Besides different file system types, what else can the VFS hide?

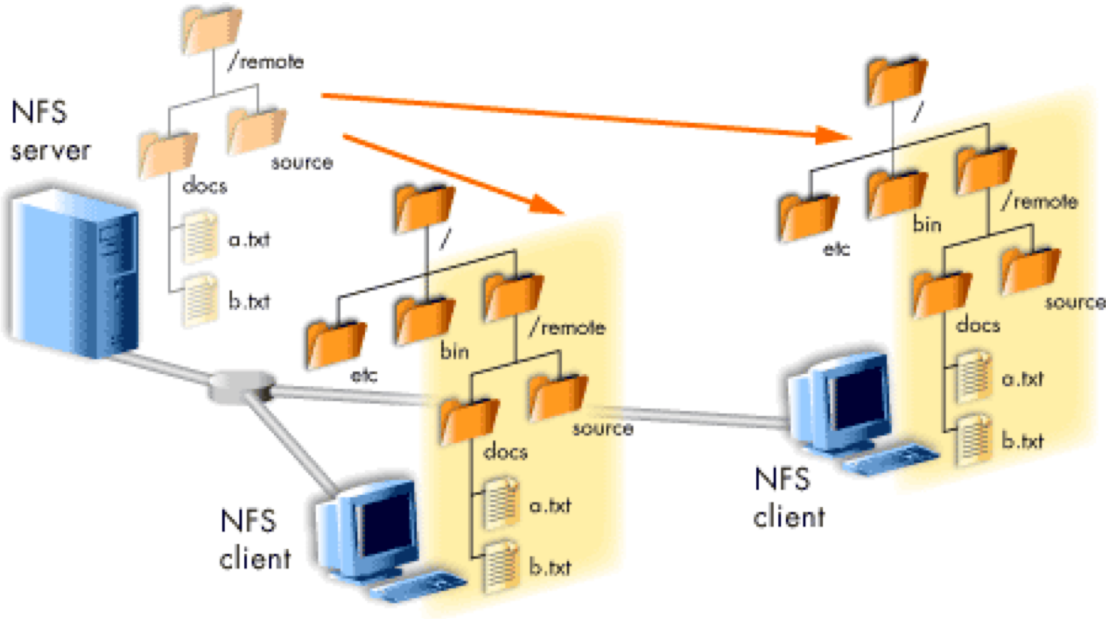
Distributed File Systems

- The basic idea is to make file systems available across a computer network.
- Advantages:
  - Management of user files can be centralized on a server
  - Reduce disk space requirements on individual computers
  - If implemented correctly, location of files is not known or relevant to the user
  - Can implement protocols for redundancy, high-availability transparently to the user
- Potential disadvantages:
  - inconsistencies, network delay, single-point of failure

Example Distributed File Systems

- Network File System (Sun Microsystems, 1984)
  - has become an Internet standard,
  - widely used on Linux and other platforms
- Andrew File System (Carnegie-Mellon, 1980s)
  - enhancements for security, authentication, consistency and scalability
  - follow-on Coda File System
- Distributed File System (Microsoft, circa 2000)
  - based on Samba network protocol, runs on Windows server
  - accessible from Linux and other clients

Network File System Example

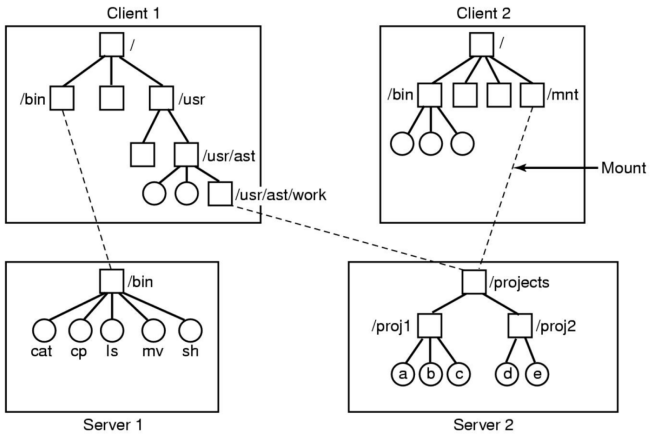


The NFS server exports the /remote filesystem to two NFS clients.

Another Example

- Different clients can mount remote directories in different locations in their local file systems

Network File System (1)



- Examples of remote mounted file systems
- Directories are shown as squares, files as circles

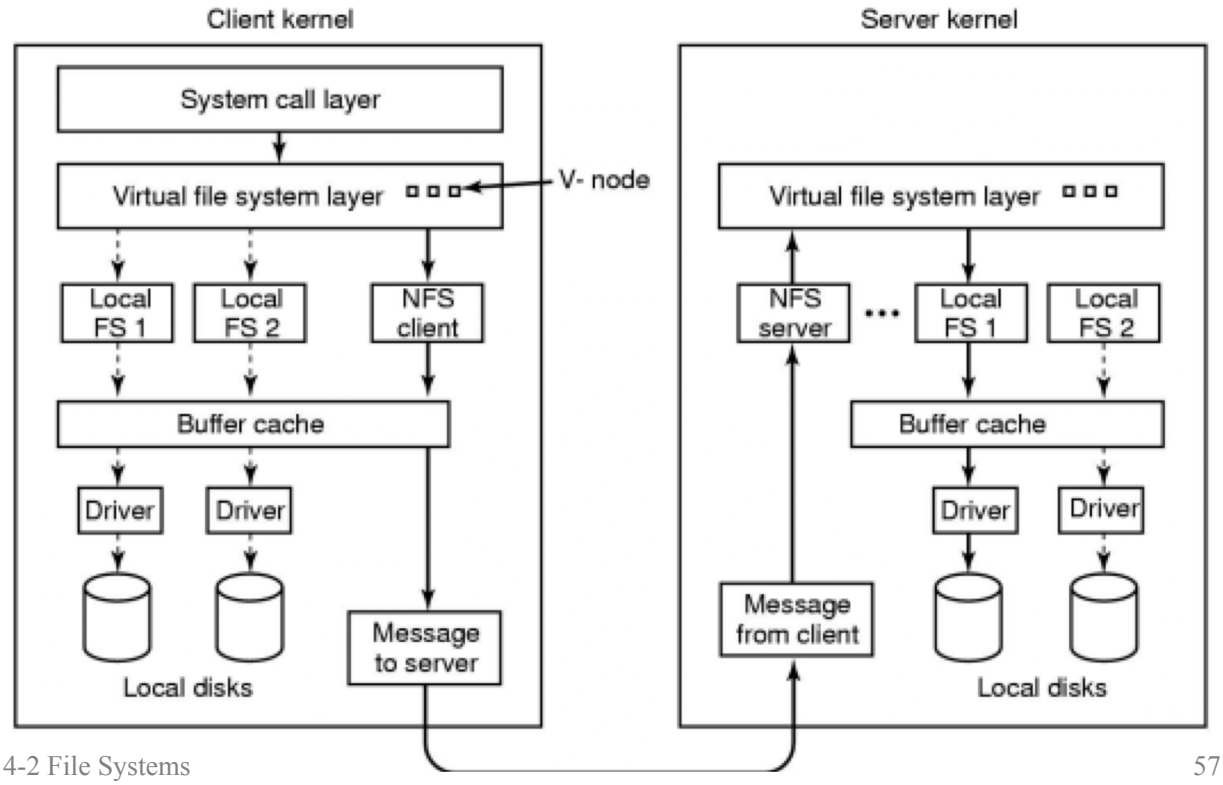
NFS Overview

- Based on underlying Remote Procedure Call (RPC) protocol
  - Also developed by Sun in early 1980s
  - Applications can access remote services via function calls
  - Predecessor to Java Remote Method Invocation (RMI)
- NFS enables the file system of a remote computer system to appear as if attached to the local computer
- An individual file system can be composed of one or more remotely mounted file system that appears as a single set of files to the user.
- Remote/Local is completely transparent to the user.

Role of VFS in NFS

- A key enabler is the Virtual File System
  - Already provides infrastructure to hide different types of file systems
  - A remote file system is just another type…
- (Note: In some operating systems that support VFS, the in-memory inode is referred to as a vnode)
- Linux has retained the name inode, although for implementations of AFS vnode is used.
- Regardless of name, the basic idea is same:
  - common data structure for meta-information about files from different types of file systems, including remote ones

Role of VFS



Client-Server Protocol

- User processes act as clients in accessing files on a remote NFS server.
  - A client's access to a NFS file is translated into an RPC request and sent over the network
  - On the server are multiple (kernel) threads that handle requests from clients
  - Returned response is translated to look just like the file is local
    - e.g., a read or write goes through buffer cache
    - calls such as chmod return results just as if the file were local
- An important key is the fact that VFS inode points to ops functions based on the type of filesystem.

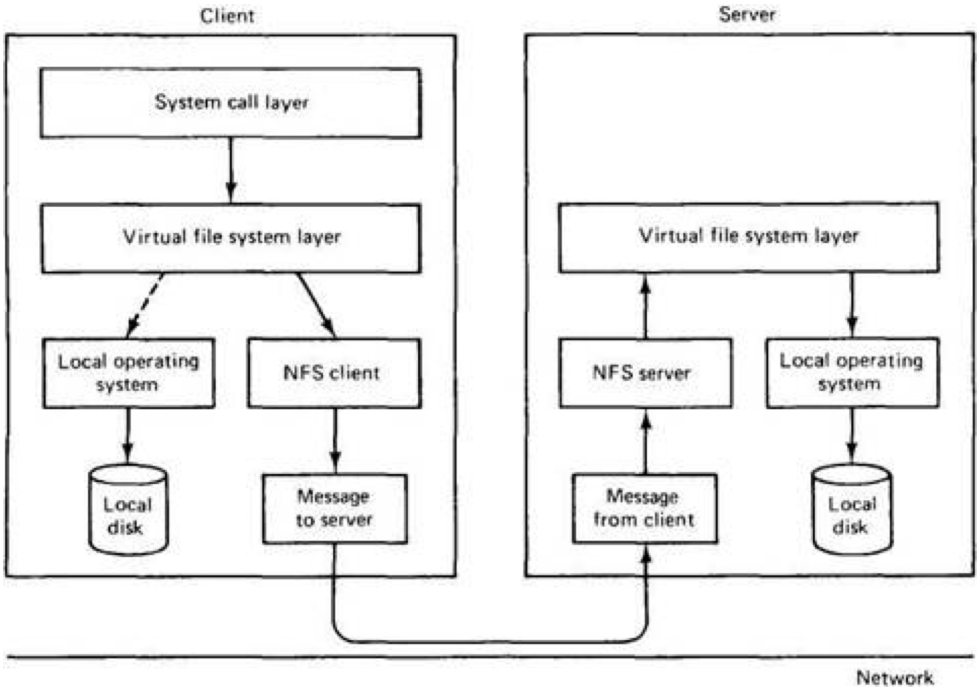
NFS is Stateless

- NFS does not maintain state at the server
  - all requests are independent of one another
  - therefore all requests need to completely describe the desired operation
  - E.g., a write operation must include the file “handle,” offset, length of write operation (in a local write, the current state incl. write position of the file is maintained by the kernel)
- Another consequence:
  - NFS requests are idempotent. What does this mean?
- Why is NFS stateless?

NFS and the TCP/IP protocols

- Until version 4, NFS was based on underlying UDP protocol
  - UDP is a best-effort, unreliable service
  - well suited for a stateless protocol
- In version 4, communication uses TCP
  - TCP is a reliable transport level protocol
  - But TCP involves maintaining state at both ends of connection.
- Why use TCP instead of UDP?

Illustration of NFS Client/Server Protocol



Example: Trace of read() system call

1. User process invokes read() with file descriptor, fd
2. VFS maps fd to in-memory inode (vnode), which indicates fs type is NFS (now the kernel code executing is considered an NFS client)
3. Client checks local buffer (page) cache for the data. If found, copy data to user space as usual.
4. If not found, client composes an RPC read request and send to server. Additional (asynchronous) threads are scheduled to issue read-ahead requests.
5. Server receives RPC request, assigns a kernel server thread to handle it.
6. Server thread initiates disk I/O (if necessary)
   a. Reads “through” server buffer cache. If data is there, can respond immediately.
   b. If reading from diskthread sleeps waiting for response, awakened when I/O is complete.
7. Server thread sends back RPC response (ack) with data to client machine.
8. Client thread is awakened, stores data in buffer cache, copies requested data to user space, returns from system call.

Other NFS Issues

- Further details of NFS are beyond the scope of CSE 410 and involve issues of networking and distributed computing
  - buffer cache consistency, file locking, crash recovery, automounting of NFS file systems, and so on…
- If you are interested, here are a couple of resources:
  - http://docstore.mik.ua/orelly/networking_2ndEd/nfs
  - Tanenbaum, Distributed Systems: Principles and Paradigms, 2nd Ed. (2007)
  - many more, just google “network file system”

Summary

- The key issue in file system design is to provide applications with access to disk that looks much like access to memory in terms of performance
- File systems have not changed drastically over the years, although a number of innovations have been introduced to improve
  - their performance (FFS, EXT2, EXT3, etc.)
  - their generality and flexibility (VFS)
  - their accessibility (NFS, Andrew, etc.)
- The Linux kernel and other modern operating systems take advantage of these innovations.