Sheridan

SYST13416
Introduction to Linux
Operating

File System

FILE SYSTEMS

Objectives

- File systems
- Storage devices
- Device file system
- Partition layout
- Common Device File systems (FAT, FAT32, NTFS, EXT2, EXT3, JFS, ReiserFS)

File Systems

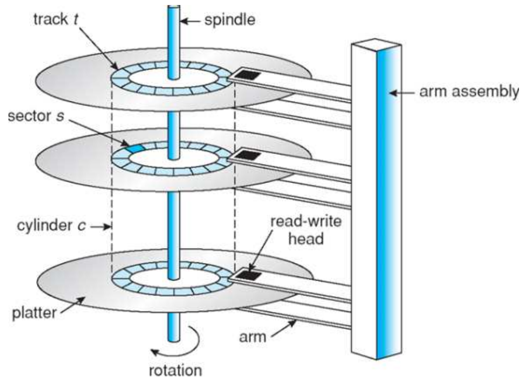
- **The file system** is the most visible aspect of an operating system for most users, by providing the mechanism to store and access data and programs.

- It provides:

   1. a collection of **files**, each storing related data, and

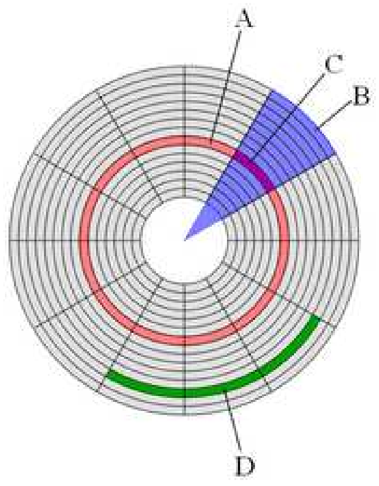
   2. a **directory** structure, which organizes and provides information about all the files in the system.

- File systems live on **storage devices**.



Storage Devices

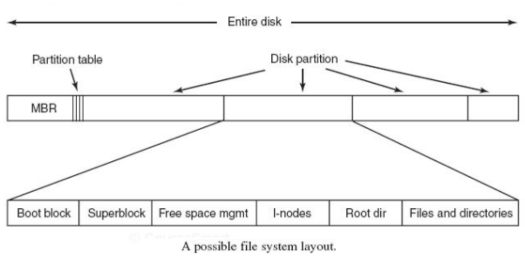
- The surface of **magnetic tapes** and **disks** are coated with a material that can be magnetized by a **write head**, and the stored magnetic field can be detected by a **read head**.
- Current tapes are capable of storing thousands of megabytes. Tapes, by nature, are **sequential storage media**, which means that they allow data to be written or read in one sequential stream from beginning to end. Should you desire access to a block of data in the middle of the tape, you must scan all preceding data on the tape to find the block of interest. Because of this, tapes are primarily **used for mass backup** of the data stored on large-capacity disk drives.
- A more convenient method of rapidly accessing stored data is provided by a **direct access storage device (DASD)**, where any one file or program can be written or read independently of its position on the storage medium. The most popular DASD in recent years has been the magnetic disk. A magnetic disk consists of either a single rigid platter or several platters that spin together on a common **spindle**.
- A **movable access arm** positions the **read/write heads** over, but not quite touching, the recordable surfaces.
- **Partition**: a section of the hard disk that holds a file system.
- **File system**: method to organize and store information on a system, which includes namespace, API, security model and implementation.



(A) Track
(B) Geometrical sector
(C) Track sector
(D) Cluster

Device file system

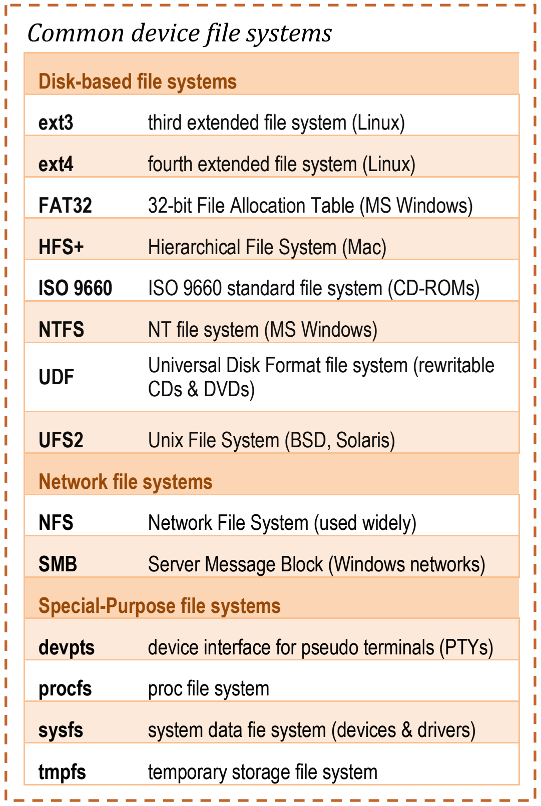
- **Unix file system** is the large, all-inclusive structure that contains every file and every directory in the entire system.
- **Device file system** is a smaller, individual file system that resides on the various storage devices.
- The UNIX file system is created by connecting the smaller device file systems into one large structure.
- Each device uses a file system appropriate for that type of device. A **partition** on a hard drive **uses a file system suitable** for a hard drive; a CD-ROM uses a file system suitable for CD-ROMs, and so on. The details involved in reading and writing data vary significantly depending on the type of device.



A possible file system layout.

Partition layout

- Each **partition** is divided into **tracks**, **cylinders**, and **sectors**, when the partition is formatted.
- Sectors are numbered 0 to 24, as an example. Tracts are number from the inside out, 0 to 7.
- Data is stored in a **block** (track number, sector number) and each **section** may have a fixed size. The inner blocks are the same size as the outer blocks.
- **Sector 0** of the disk is called **Master Boot Record (MRB)** and is used to boot the operating system.
- The **partition table** which gives the starting and ending address of each partition is stored at the end of the MBR.
- When the computer is booted, the BIOS reads and executes the MBR, locates the active partition, reads in its first block, called the **boot block** and executes it.
- Each disk has one MBR but may contain many partitions and each partition contains a boot block.
- Although the layout of a disk partition varies from file system to file system, the following elements are significant:
  - **Boot block**: code that initiates the boot process
  - **Super block**: contains all key parameters, such as magic number (file system type), number of blocks, and so on.
  - **Free space management**: usually a bitmap of free space of linked links, etc.
  - **Inode**: an array of data structures, one per file, telling all about the file.
  - **Directory**: a pointer to the root directory and pointers to all directories on the disk.



Common device file systems

Disk-based file systems	
ext3	third extended file system (Linux)
ext4	fourth extended file system (Linux)
FAT32	32-bit File Allocation Table (MS Windows)
HFS+	Hierarchical File System (Mac)
ISO 9660	ISO 9660 standard file system (CD-ROMs)
NTFS	NT file system (MS Windows)
UDF	Universal Disk Format file system (rewritable CDs & DVDs)
UFS2	Unix File System (BSD, Solaris)
Network file systems	
NFS	Network File System (used widely)
SMB	Server Message Block (Windows networks)
Special-Purpose file systems	
devpts	device interface for pseudo terminals (PTYs)
procfs	proc file system
sysfs	system data fie system (devices & drivers)
tmpfs	temporary storage file system

Common device file systems

FAT—File Allocation Table

- A typical cluster size is 2,048 bytes, 4,096 bytes, or 8,192 bytes
- File allocation table entries are 16 bits in length
- DOS 5.0 and later versions provide for support of hard disks up to 2 gigabytes with the 16-bit FAT entry limit by supporting separate FATs for up to four partitions

FAT32

- File allocation table entries are 32 bits in length
- The largest size hard disk that can be supported is two terabytes
- More efficient than FAT

NTFS--NT file system or New Technology File System

- Used by Windows NT/2000 operating system
- NTFS offers a number of improvements over FAT in terms of performance, extendibility, and security.
- Information about the file's clusters and other data is stored with each cluster, not just a governing table (such as FAT does)
- The sizes of clusters range from 512 bytes to 64 kilobytes

EXT2--Second Extended File System

- Popular Linux file system, designed by Stephen Tweedie.
- Max disk size: 4 terabytes; Max file size: 2 GB, Max file name: 255 characters

EXT3--Third Extended File System

- All features of EXT3 with added Journaling feature

Journaling File System (JFS)

- By logging changes made between system writes to the file system, a journaling file system ensures the availability of data in the event of a disk crash
- In the event of a system crash, a journaling file system promises faster reboot times, because not every file needs to be examined by fsck, the Unix file system checking utility

ReiserFS

- designed and implemented by a team at Namesys led by Hans Reiser, offers better disk space utilization, better disk access performance, fast crash recover, and reliability through data journaling.

UNIX FILE SYSTEM

Objectives

- Unix File System: Namespaces, API, security model, implementation
- Filesystem hierarchy system (FHS)
- Root (/) file system
- Making a file system accessible (mount)
- File sharing: using a link (a pointer to another file or subdirectory)
- Where to use hard links
- Creating hard links and symbolic links (ln)
- Deleting symbolic links

Unix File System

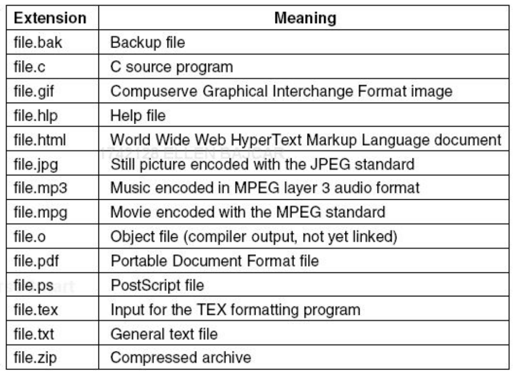
- A **file system** is the method that the operating system uses for organizing and storing your information on the system. Other types of objects are managed as well, including processes, serial ports, and inter-process communication channels. Generally, such objects are mapped into the file system namespace.
- The file system can be thought of as having four main components:
  - ✓ a **namespace**: a way of naming things and organizing them in a hierarchy;
  - ✓ an **API**: a set of system calls for navigating and manipulating objects;
  - ✓ a **security model**: a scheme for protecting, hiding, and sharing things ;
  - ✓ an **implementation**: software that ties the logical model to actual hardware.

Namespace – a way of naming things

- Many operating systems support **two-part file names**, with the two parts separated by a period, as in *prog.c*. The part following the **period** is called the **file extension** and usually indicates something about the file.
- Linux is **case sensitive**. For example, readme.txt and README.txt are seen as two different files.

Namespace – a way of organizing things

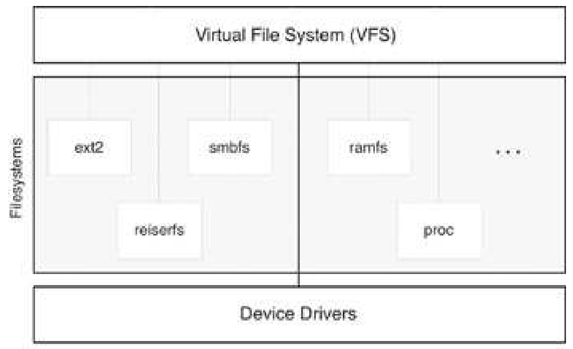
- The key to all operations is that the data has some kind of structure. Consider a filing cabinet that consists of multiple drawers, with each drawer holding a certain set of contents: project-related documents in one drawer, employment records in another. Sometimes drawers have compartments that allow different kinds of things to be stored together.
- These are all structures that determine where things go, that are used when it comes to sorting the papers that will be placed within, and that determine where things can be found.
  - ✓ **Directory**: a file that acts as a holder for other files and directories.
  - ✓ **File**: a collection of data that is stored on a disk and that can be manipulated as a single unit by its name.
  - ✓ **Directory tree**: includes a directory and all of its contents and its sub-directories.
  - ✓ **Parent directory**: a file that contains another directory.

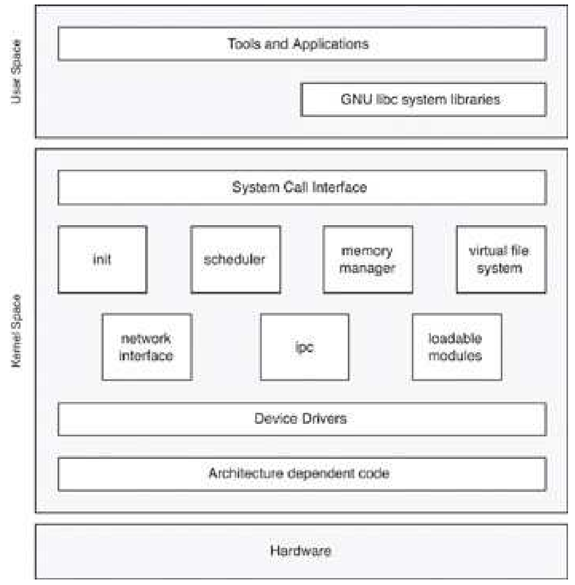


Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Some typical file extensions.

- In **MS-DOS**, file names are one to eight characters, plus an optional extension of one to three characters.
- In **Unix**, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *homepage.html.zip*, where .html indicates a web page in HTML and .zip indicates that the file, *homepage.html*, has bee compressed using the zi program.





Filesystem Hierarchy Standard (FHS)

- **The Filesystem Hierarchy Standard** is a document that specifies the layout directories on a Linux system. The FHS was devised to provide a common layout to simplify distribution-independent software development—so that stuff is in generally the same place across Linux distributions. The FHS specifies the following directory tree (taken directly from the FHS specification:

/	(the root directory)
/boot	(static files of the boot loader)
/dev	(device files)
/etc	(host-specific system configuration)
/lib	(essential shared libraries and kernel modules)
/mnt	(mount point for mounting a filesystem temporarily)
/opt	(add-on application software packages)
/sbin	(essential system binaries)
/tmp	(temporary files)
/usr	(secondary hierarchy)
/var	(variable data)

The Root (/) File system

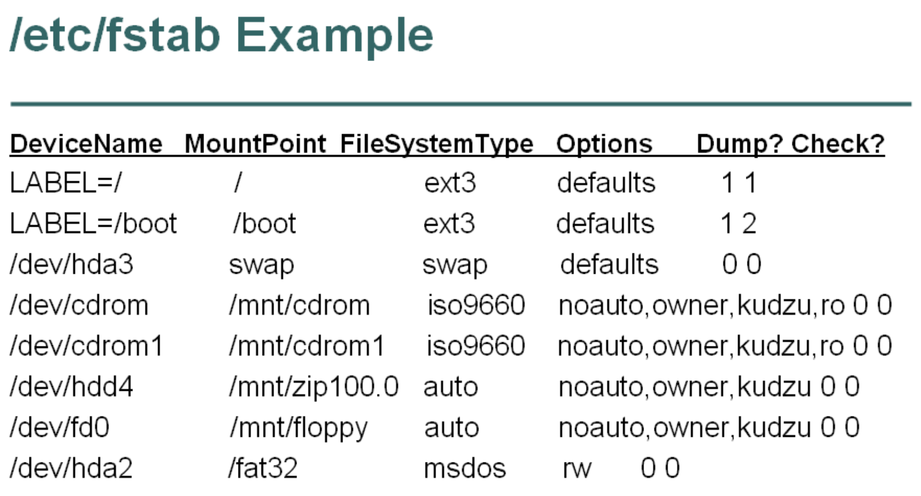
- ***The root filesystem,*** denoted by the symbol slash / contains the files you need to boot the system. It is the only filesystem automatically mounted in single-user mode. The root directory generally consists only of subdirectories and the kernel executable image file. Other files reside in the subdirectories as described in the FHS above.

Making a File System Accessible

- Mounting a File system: **a file system** must be mounted to a directory to be accessible.

  `mount`

- Permanently mounted file systems have an entry in the **/etc/fstab** file
- You require administrative privileges to modify /etc/fstab file



/etc/fstab Example

DeviceName	MountPoint	FileSystemType	Options	Dump? Check?
LABEL=/	/	ext3	defaults	1 1
LABEL=/boot	/boot	ext3	defaults	1 2
/dev/hda3	swap	swap	defaults	0 0
/dev/cdrom	/mnt/cdrom	iso9660	noauto,owner,kudzu,ro	0 0
/dev/cdrom1	/mnt/cdrom1	iso9660	noauto,owner,kudzu,ro	0 0
/dev/hdd4	/mnt/zip100.0	auto	noauto,owner,kudzu	0 0
/dev/fd0	/mnt/floppy	auto	noauto,owner,kudzu	0 0
/dev/hda2	/fat32	msdos	rw	0 0

Where to use Hard Links

Hard links can be useful in the following situations:

1. Consider that you have written a number of programs that read file foo.txt in $HOME/input_files. Later, you reorganize your directory structure and moved foo.txt to $HOME/data instead. What happens to all the programs that look for foo.txt at the original location? Simple, just link foo.txt to the directory input_files. With this link available, your existing programs will continue to find foo.txt in the input_files directory.
2. Links provide some protection against accidental deletion, especially when they exist in different directories. Referring to the previous application, even though there's only a single file foo.txt on disk, you have effectively made a backup of this file. If you inadvertently delete input_files/foo.txt, one link will still be available in data/foo.txt; your file is not gone yet.
3. Because of links, we don't need to maintain two programs as two separate disk files if there is very little difference between them. A file's name is available to a C program (as argv[0]_ and to a shell script (as $0). A single file with two links can have its program logic make it behave in two different ways depending on the name by which it is called.

Links

- **The Linux system does not store the filename in the node so that a file can have multiple file names.** When this happens, we say the file has more than one link. We can then access the file by any of its links.
- The ln command links a file, providing it with an alias and increasing the link count by one. This count is maintained in the inode. The ln command can create both a hard and a soft link and has a syntax similar to the one used by cp command.

Creating Hard Links

- Before you create a hard link, examine the listing of the objects you wish to link. For example, consider the following listing:

```
254414 --rw-r--r-- 1 romeo users 5 Dec 24 09:38 date.sh
```

- Note that the first number displayed is the file's inode. To display a long listing that includes the inode, use the options l and i.

```
ls –li date.sh
```

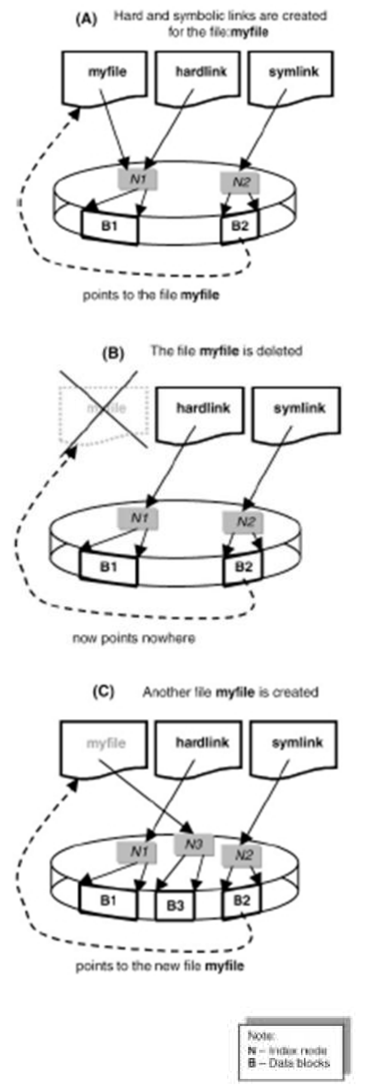
- To create a hard link to date.sh with who.sh:

```
ln date.sh who.sh
```

- Prior to the invocation of ln, the current directory had an entry containing date.sh and its inode number 254414. After the invocation, the kernel performed two tasks:
  - It added an entry to the directory for the filename who.sh, but having the same inode number
  - It also updated the link count in the inode from one to two
- Note that there is actually one file and that we cannot refer to them as two files, but only as two filenames. If you create one more link:

```
ln who.sh ps.sh
```

  another directory entry will be created and the link count would be incremented to three.


(A) Hard and symbolic links are created for the file:myfile
myfile
hardlink
symlink
N1
N2
B1
B2
points to the file myfile
(B) The file myfile is deleted
myfile
hardlink
symlink
N1
N2
B1
B2
now points nowhere
(C) Another file myfile is created
myfile
hardlink
symlink
N1
N3
N2
B1
B3
B2
points to the new file myfile
Note:
N – Index node
B – Data blocks

Creating Symbolic link

- Imagine that hundred files in the directory input_files have been moved to the directory data as part of the reorganization process mentioned earlier. To ensure that all programs still see the files in their original location, we could hard-link these files to the new data directory, but that would mean adding a hundred entries to the directory. It's here that one encounters two limitations of hard links:
  - You cannot link a file across two file systems
  - You cannot link a directory, not even within the same file system
- A symbolic link overcomes both problems. Until now, we have divided files into three categories: ordinary files, directories, and devices. The symbolic link is the fourth file type. Consider the following listing:

```
lrwxrwxrwx 1 romeo users 17 Aug 11 00:49 hex.c -> c_progs/hexdump.c
```

- A symbolic link is identified by the l (el) as the file type and the pointer notation -> that follows the filename. The ln command creates symbolic links when the –s option is used. We can create a symbolic link to date.sh

```
ln -s date.sh data.sym
```

- The long listing resulting from

```
ls -li date.sh data.sym
```

  generates a different display:

```
254414 -rw-r--r-- 2 romeo users 5 Dec 24 09:38 date.sh
254311 lrwxrwxrwx 2 romeo users 7 Dec 29 10:52 date.sym->date.sh
```

- Here, date.sym is a symbolic link to date.sh. Unlike a hard link, a symbolic link is a separate file with its own inode number. The date.sym simply contains the pathname date.sh. Note that the size 7 refers to the name that contains 7 characters.
- The two files are not identical; only date.sh contains the actual contents. A command like

```
cat date.sym
```

follow the symbolic link and display the file the link points to.

- A symbolic link can also point to an absolute pathname, but to ensure portability, we often use a relative pathname:
  ```
  ln -s ../jscript/search.html search.html
  ```
- To link hundred files in the directory data, you can use symbolic link named input_files to connect the data:
  ```
  ln -s data input_files
  ```
- Being more flexible, a symbolic link is also known as a soft link or symlink. As for a hard link, the rm command removes a symbolic link even if it points to a directory.
- Symbolic links are used extensively. System files constantly change locations with version enhancements. Yet all programs still need to find the files where they originally were. Windows shortcuts are more like symbolic links.

Deleting Symbolic Links

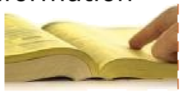
- Think twice before you delete the file or directory that a **symbolic link points to**. Removing date.sym won’t affect much because you can easily recreate the link. But if you remove date.sh, you will lose the file containing the data.
- A disaster could occur if you remove data instead of input_files symbolic link, resulting in loss of many files. In either cas, date.sym and input_files would point to a non-existent file. These links are known as dangling symbolic links.
- The pwd command shows you the path you used to get to the directory, which may be different than the actual directory you are in.

DIRECTORIES

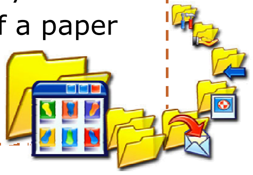
Objectives

- Understand Linux directory structure and operations
- File system navigation: pathname, current working directory, absolute path, relative path
- Navigating the directory tree
- Organizing your home directory

Although we often refer to a directory as **folder**, the analogy **is not a very good one**. A better analogy would be **a phone book or yellow pages**, which holds the information about people and companies, just like a **directory** holds information about **the file system objects**.



Folder is **a graphical representation of a directory**, originating from Windows where the standard graphical icon used to represent a directory is a folder (or picture of a paper file).



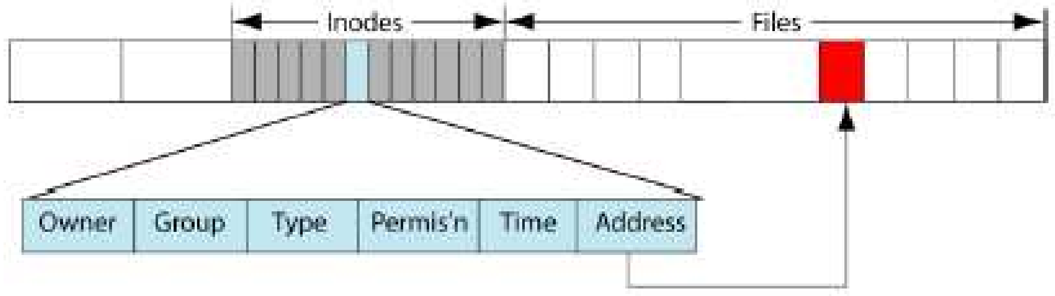
Directories and Subdirectories

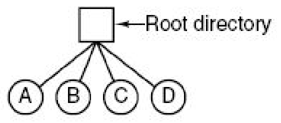
We use directories to organize files into a hierarchical tree-like system. To do so, we **collect files together into groups** and **store each group in its own directory**.

Since **directories are themselves files**, a directory can contain other directories, which create the hierarchy.

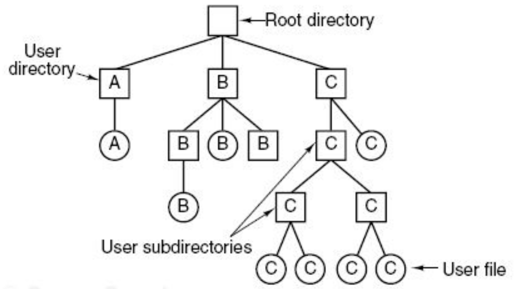
A directory does not hold the actual files, it merely contains the information Unix needs to locate the files.

- Directories are files that are lists of simple files and other directories ;
- Directory file consists of an array of directory entries:
  `inode number + file name`
- The **inode** number is four (4) bytes long and is an index value for the array of index nodes on the disk. An **index node** (inode) contains file attributes.





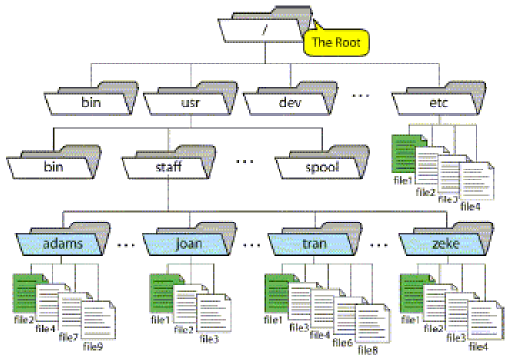
A single-level directory system containing four files.



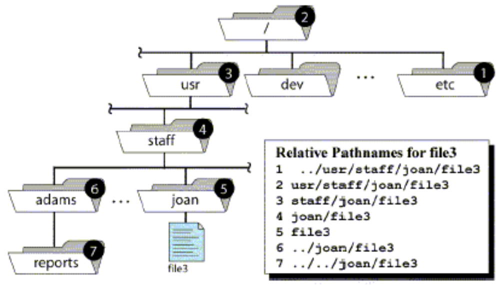
A hierarchical directory system.

Directory Operations

- The allowed system calls for managing directories exhibit more variation from system to system than system calls for files. To give an impression of what they are and how they work, here is a sample:
  1. **Create**. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system or by the mkdir program.
  2. **Delete**. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot be deleted.
  3. **Opendir**. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
  4. **Closedir**. When a directory has been read, it should be closed to free up internal table space.
  5. **Readdir**. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
  6. **Rename**. In many respects, directories are just like files and can be renamed the same way files can be.
  7. **Link**. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path



Example: Relative Pathnames for file3



Relative Pathnames for file3

```
1  ../usr/staff/joan/file3
2 usr/staff/joan/file3
3 staff/joan/file3
4 joan/file3
5 file3
6 ../joan/file3
7 ../../joan/file3
```

File System Navigation

Pathname

- A pathname is a location of any specific file or command.
- A full pathname contains all the branch directories between the root and the specific file.

Current working Directory

- This is where the calling process is at the time of the call; it can be obtained through use of **pwd** command from the shell or getcwd() from within a C program.

Absolute pathname

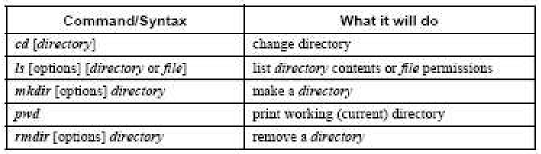
- Declares the path starting at the root directory and declaring each directory on the path. The pathname
  /etc/passwd
  is absolute.

Relative pathname

- Declares the path in relation to your current position. It does not contain / as the first character. For example, to read the same passwd file by specifying
  passwd
  the current working directory must be **/etc**.

1.

Absolute Pathnames	***Relative Pathnames***
Start at the root directory	Start at your current directory
Always start with a slash /	Never start with a slash
The absolute pathname to some object (file, etc.) is always the same.	The relative pathname to an object depends on your current directory.



Command/Syntax	What it will do
cd [*directory*]	change directory
ls [options] [*directory* or *file*]	list *directory* contents or *file* permissions
mkdir [options] *directory*	make a *directory*
pwd	print working (current) directory
rmdir [options] *directory*	remove a *directory*

Note that creating objects outside your home directory requires you to have administrative privileges.

Navigating the Directory Tree

- To find out the directory pathname to your current location, use the command
  `pwd`
- To list the names of the objects in your current directory
  `ls`
- To list long listing of the current directory ("el es dash el", note they are not ones)
  `ls –l`
- To change to the parent directory of your current directory
  `cd ..`
- To change to the root directory of the Linux file system
  `cd /`
- To change to your user's home directory
  `cd ~`
- To change to your user's home directory (no options, no arguments)
  `cd`

Organizing Your Home Directory

- To create a new directory, use the command
  `mkdir dirName`
- To remove an empty directory
  `rmdir emptyDir`
- To remove a directory and its content, including all subdirectories
  `rm –r anyDir`

Contents