CST8177

Files, directories, links, and lots more

Files and directories are both the same sort of thing, collections of records (or lines of text) or binary data. The modern approach is hard-sectored or fixed-size physical blocks.

Directories are simply a special case of files, containing information about the files that are "in" it. You can see much of this from the **stat** command, but we're more accustomed to **ls**:

```
System Prompt$ ls -il sample-file
308527 -rw-rw-r--. 1 user1 user1 1249 2010-
01-10 21:28 sample-file
System Prompt$ ls -ild ~
302513 drwx------. 75 user1 user1 4096 2010-
09-12 16:19 /home/sample-dir
```

The **-i** option of **ls** shows the inode number, the basis of the organization of many filesystems.

Sample file stat

```
System Prompt$ stat sample-file
  File: `sample-file'
  Size: 1249            Blocks: 16        ...
...           IO Block: 4096   regular file
Device: 805h/2053d      Inode: 308527     ...
...                           Links: 1
Access: (0664/-rw-rw-r--)  Uid: (  500/   ...
...         user1)   Gid: (  500/ user1)
Access: 2010-09-10 14:20:09.000000000 -0400
Modify: 2010-01-10 21:28:25.000000000 -0500
Change: 2010-05-08 08:51:20.000000000 -0400
```

Sample directory stat

```
System Prompt$ stat ~
  File: `/home/sample-dir'
  Size: 4096            Blocks: 16      ...
...              IO Block: 4096   directory
Device: 805h/2053d      Inode: 302513   ...
...                              Links: 75
Access: (0700/drwx------)  Uid: (  500/ ...
...         user1)   Gid: (  500/ user1)
Access: 2010-09-12 17:01:08.000000000 -0400
Modify: 2010-09-12 16:19:29.000000000 -0400
Change: 2010-09-12 16:19:29.000000000 -0400
```

File Naming Conventions

- upper and lower case letters, can be mixed
- digits
- underscore, dash, and period (aka dot or full stop)
- up to 255 chars on modern versions
- case sensitive: **`demo.pdf`**, **`Demo.pdf`**, and **`demo.PDF`** are not the same file
- a filename starting with a period ("`.`") designates a special file (sometimes called a "hidden file"); use the **`-a`** option of **`ls`** to list
- The filename "`.`" alone designates the current directory, whatever that happens to be ...
- While "`..`" designates the parent of the current directory

more File Naming Conventions

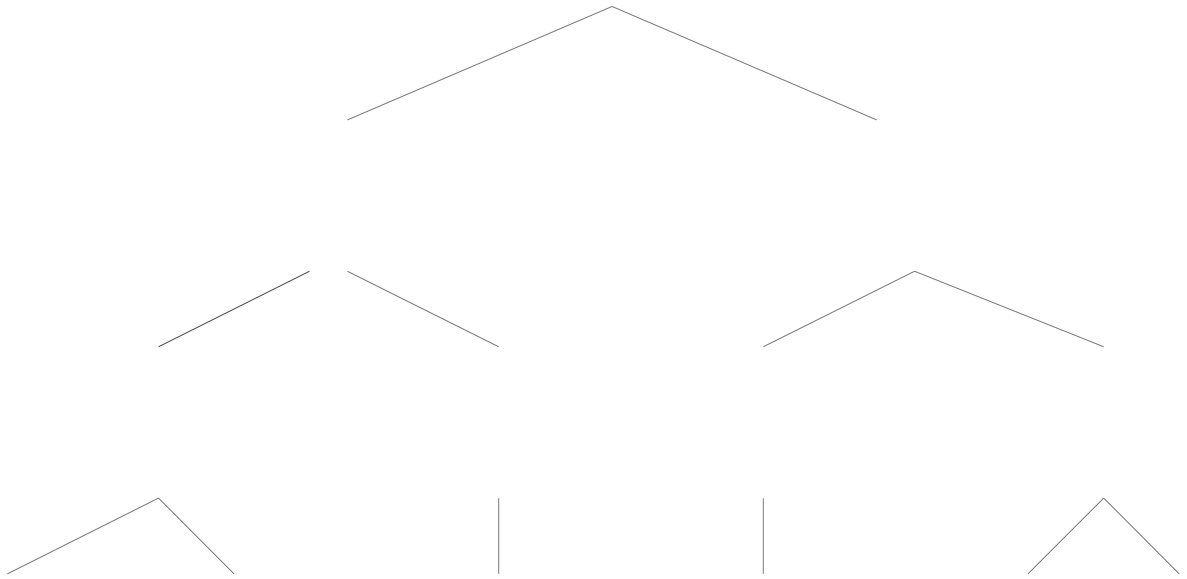
- "extensions" are purely a convention for humans;
  - unlike Windows, they mean nothing to Linux
  - certain applications may choose to use particular conventions; **`Python`** with **`.py`** and **`gcc`** with its special extensions, like **`.c`** and **`.o`**
- Pathnames similar to Microsoft (but different!)
- directories in pathnames are separated by "**/**" (and **<u>never</u>** "\"; backslash has a different meaning)
- "**`pwd`**" prints the (current) working directory
- "**`cd`**" changes to a new directory
- <u>relative</u> pathnames start with "**.**", "**..**", or a directory name.
- <u>absolute</u> pathnames start at the root ("**/**") directory.

still more File Naming Conventions

- Your **`HOME`** directory is always at "`~`" (tilde). That is, "~" as the start of a pathname represents the current user's home directory. Does it count as absolute or relative?
- there are no 'drives' (**`A:`** or **`C:`**) as in Microsoft OSes. Instead there is a single tree-structure of directories, with **mount points** for storage devices (like **`/media/disk`** for a USB stick emulating a hard drive).
- To execute a program from your current directory, preface it with "`./`" so it can be found (this has nothing to do with the slash-dot ("`/.`") web site.
- If you prefer, you can add "`./`" to your **`PATH`**:

```
PATH=$PATH:./
```

A Sample Directory Tree



- A sample <u>absolute</u> path could be

  `/def/def/jhi/some.file.or.another`

- A sample relative path (from where?) could be:

  `../jkl/str/another.file`

- Or if your **`HOME`** directory is **`/abc`**:

  `~/ghi/mno/my.own.file`

- Is **~** a relative or an absolute path? Why?
- Every file and every directory (what is the difference?) has a user and a group associated with it.
- These are stored as the uid (user id) and the gid (group id), where each is an integer number.

Manipulating Files and Directories

`mkdir dirname`

Creates a new directory.

`rmdir dirname`

Remove a directory (only if empty)

`rm [options] filenames`

Remove file (with **`-rf`**, also directories, empty or not). The most useful options are **`-rf`**, which can also be quite dangerous. The **`r`** causes **`rm`** to recurse Through all directories, and the **`f`** (force) does not ask permission before removing.

`cd [directory]`

Change to directory; with no directory given, it defaults to current user's home directory.

`pwd`

Lists the full path of the current directory.

`cat filenames`

Catenate files to **`stdout`**/screen with no formatting; list a text file's content (see also **`tac`**).

`date [options]`

Set or display the system date and time; see the **`man`** page for details to use this.

`cp [options] source destination`

Copy files and directories. The **`-a`** option (archive) is particularly useful for directories

`mv source destination`

Move files and directories: essentially copy plus delete. **`mv`** is also used to rename a file in place.

`find [options]`

Searches the filesystem according to what you need. It can be quite complex, and at the same time extremely useful. We'll have to go over this together in some detail. See also **`xargs`**.

`whereis [options] command`

Lists the paths for binary, source, man page for a command for certain well-known directories.

`locate [options] name`

Reads a database usually updated daily for files. It does not check if files found still exist and never reports on newer files.

`more [options] filenames`

Like **`cat`**, but pauses at the end of every screenful (page). Now often replaced by **`less`**.

`df [options] [filesystem-list]`

Reports filesystem disk usage; see **`man`** page.

`du [options] dir-list`

Calculates disk space usage for files and directories or just a total if the **`-s`** option (summary) is used. May seem slow since it reads all directories recursively.

`whoami`

Displays the effective user ID (particularly useful when switching from one user to another)

`who [options]`

Displays information about users who are currently logged into the system, locally or remotely.

`vi [options] filename`

Standard command-line editor with multiple options and extensive capabilities. Always available in any level on any Linux/Unix system.

`shutdown [option] when [message]`

Stops all processes and safely brings the system down, often for **`-h`** (halt) or **`-r`** (reboot). The <u>when</u> argument is often "**`now`**", meaning right away, or **`+m`** for an <u>m</u> minutes delay.

`exit`

Terminate the current shell process. Often used to Leave a script. Can also be used to exit a shell instead of Control-D (**`^D`**, **`stdin`** end-of-file).

`telinit runlevel`

Change the current runlevel. Must be **`root`**.

`runlevel`

Display the current and former init-process runlevel or **`N`** if no former runlevel exists.

`head` and **`tail [options] filename`**

Lists the start or end of a file; see **`tail -f`**.

`dmesg`

Displays the critical content of logs for the last boot sequence.

`su [user] -`

Superuser command, to change userid. The **`-`** or **`-l`** option emulates login.

touch file-list

Updates the time and date stamp. If the file does not exist, an empty one is created.

`echo some text here > filename`

The **`echo`** built-in displays its arguments on **`stdout`**. By redirection, a non-empty file can be created or replaced. With **`>>`**, the arguments are appended to an existing file or behave like **`>`**.

`ln [option] source destination`

Creates links between files. If the **`-s`** (symbolic) option is present, a soft link is created. Otherwise a hard link is created within the current filesystem (only!).

Directories can be linked only by soft links.

Rights and File attributes

- access permission information is maintained for all files and directories on the filesystem by Linux
- commands such as file allow Linux to determine (or at least guess) what file type any file on the system might be.

Example of directory listing:

```
 Type rights      links       owner     group       ...
                 size       Date/Time modified     ...
                                                filename
   drwxr-xr-x      2    root    root      ...
...           1024    Jul    9  10:10   files

   -rwxr-xr--        0    guest guest    ...
...           5096    Apr 10  09:15   list.txt
```

Rights (permissions)
- 10 characters with file type as the first letter
- access modes (remaining letters)

Link count
- number of links to this file or directory

User-owner Login Name
- user who owns the file/directory
- based on owner UID

User-owner Group Name
- group who owns the file/directory
- based on owner GID

File Size
- size (in bytes or K) of the file/directory

Date/Time Modified
- date and time when last created / modified / saved

File Name
- actual file/directory name

File Types

Linux recognizes and identifies several file types, which is coded into the first letter of the first field of information about the file:

-	(**dash**) a regular file
b	block device special file
c	character device special file
d	a directory
l	a symbolic (soft) link
p	a named pipe or FIFO
s	socket special filename

File Access Privileges

- In Linux, 3 types of access permissions or privileges can be associated with a file
  - **`read`** (`r`) grants rights to read a file
  - **`write`** (**`w`**) grants rights to create, write to, or remove a file
  - **`execute`** (**`x`**) grants rights to run the file (to execute the file as a command)
- All 3 permissions can then be applied to each of 3 types of file users
  - **User** owner/creator of the file
  - **Group** group to which user must belong to gain associated rights
  - **Others** Anyone else not **User** or not a part of **Group** (we used to call it Rest-of-world)

File Access Privileges

In Linux, rights are typically referred to by their octal equivalent, as seen below. This can then be translated into an appropriate value for each of the 3 user types

- values of 100’s for **User**
- values of 10’s for **Group**
- values of 1’s for **Other**

User			Group			Other		
R	W	X	R	W	X	R	W	X
400	200	100	40	20	10	4	2	1

List of all octal values 0 to 7

r	*w*	*x*	*Octal Value*	*Meaning*
0	0	0	0	No permission
0	0	1	1	Execute-only permission
0	1	0	2	Write-only permission
0	1	1	3	Write and execute permissions
1	0	0	4	Read-only permission
1	0	1	5	Read and execute permissions
1	1	0	6	Read and write permissions
1	1	1	7	Read, write and execute permissions

Directory Access Privileges

The same three types of access permissions or privileges are associated with a directory, but with some differences:

- **read** (**r**) rights to read the directory
- **write** (**w**) rights to create or remove in the directory
- **execute** (**x**) rights to access the directory

All three permissions can then be applied to each of three types of directory users as before.

- **User** owner/creator of the file
- **Group** group to which user must belong
- **Others** everyone else (Rest-of-world)

Linux File Permissions

Three special access bits on files for extra control.
They can be combined as needed.

SUID - Set User ID bit

- Allows commands to change the “effective user ID” to the one identified by the file's UID. That is, commands runs as UID rather than as the actual user.

```
chmod 4xxx file-list

chmod u+s file-list
```

Linux File Permissions

SGID - Set Group ID bit

- Allows commands to change the “effective group ID” to the one identified by the GID. Thus commands runs as GID rather than as the user’s current group, much like suid/UID.

```
chmod 2xxx file-list

chmod g+s file-list
```

Linux File Permissions

sticky bit (restricted deletion flag)

- Must be the user-owner of a directory to be able to set the sticky bit for it.

```
chmod 1xxx dir-list

chmod +t dir-list
```

The sticky bit prevents unprivileged users from removing or renaming a file in the directory unless they are the owner of the file or the directory; this is commonly found only on world-writable directories like **`/tmp`**.

Linux File Permissions

What permissions a user will have is determined:

- If the user the <u>owner</u> of the file and/or directory, then the **<u>user</u>** rights are used.
- If the user is <u>not</u> the owner but is a part of the group owning the file and/or directory, then the **<u>group</u>** rights are used.
- If the user is neither the owner nor a part of the group owning the file, then the **<u>other</u>** rights are used.

- Once the access rights level is determined, the actual rights (**`rwx`**) are then identified and used for any command or process the user wishes to implement on the file and/or directory.

Linux File Permissions

- The filesystem also uses the **umask** for default rights assigned to created files.
  - **umask** - display current UMASK
  - **umask xyz** - sets new UMASK to octal permissions, where **x=user**, **y=group** and **z=other** permissions as usual.
- When a new file or directory is created by a user, the system sets its access privileges based on the user's **umask.**
  - file access = default permissions – umask
  - It's actually a NAND, not a subtraction.
- Default access permissions are always:
  - **777** for a directory or executable file
  - **666** for any other files

Linux File Permissions

- It is important for the Linux file system manager to maintain the information for each file and directory, including
  - ownership of files and directories
  - access rights on files and directories
  - The 3 timestamps seen in **`stat`**
  - Location and sequence of each data block
- The information is maintained within the file system information (**inodes**) on the hard disk
- This information is used to determine every file system action.

Linux Basic Admin Tools

`chown owner[:group] files`

- Change ownership of files and directories (available for **`root`** only)

Examples:

`chown guest:guest file1 dir2`

- change ownership of **`file1`** and **`dir2`** to user **`guest`** and group **`guest`**

`chown guest dir2`

- change ownership of **`dir2`** to user **`guest`** but leave the group the same

`chown :guest file1`

- change ownership of **`file1`** to group **`guest`** but leave the user the same (use **`chgrp`** instead)

Linux Basic Admin Tools

`chmod permissions files`

- Explicitlly change file access permissions

Examples:

`chmod +x file1`

- changes **`file1`** to have executable rights for user/group/other

`chmod u+r,g-w,o-rw file2`

- changes **`file2`** to add read rights for user, remove write rights for group and remove both read and write rights for others

`chmod 550 dir2`

- changes **`dir2`** to have only read and execute rights for user and group but no rights for other