Linux System Administration

Linux Professional Institute (LPI) Certification Level 1 Exam 101

<gbdirect>

Summary Contents

Detailed Contents

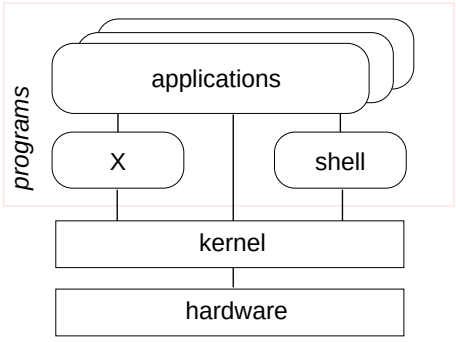
Module 1

Introduction

1. Unix and Linux

- Linux is based on Unix
  - Unix philosophy
  - Unix commands
  - Unix standards and conventions
- There is some variation between Unix operating systems
  - Especially regarding system administration
  - Often Linux-specific things in these areas

2. Unix System Architecture



- The shell and the window environment are programs
- Programs' only access to hardware is via the kernel

3. Unix Philosophy

- Multi-user
  - A **user** needs an **account** to use a computer
  - Each user must **log in**
  - Complete separation of different users' files and configuration settings
- Small components
  - Each component should perform a single task
  - Multiple components can be combined and chained together for more complex tasks
  - An individual component can be subsituted for another, without affecting other components

4. What is Linux?

- Linux kernel
  - Developed by Linus Torvalds
  - Strictly speaking, ‘Linux’ is just the kernel
- Associated utilities
  - Standard tools found on (nearly) all Linux systems
  - Many GNU utilities
    - Written by the Free Software Foundation
    - Some claim the OS as a whole should be ‘GNU/Linux’
- Linux distributions
  - Kernel plus utilities plus other tools, packaged up for end users
  - Generally with installation program
  - Distributors include: Red Hat, Debian, SuSE, Mandrake

5. Using a Linux System

- Login prompt displayed
  - When Linux first loads after booting the computer
  - After another user has logged out
- Need to enter a **username** and **password**
- The login prompt may be graphical or simple text
- If text, logging in will present a **shell**
- If graphical, logging in will present a **desktop**
  - Some combination of mousing and keystrokes will make a **terminal window** appear
  - A shell runs in the terminal window

6. Linux Command Line

- The shell is where commands are invoked
- A command is typed at a **shell prompt**
  - Prompt usually ends in a dollar sign ($)
- After typing a command press `Enter` to invoke it
  - The shell will try to obey the command
  - Another prompt will appear
- Example:

```
$ date
Thu Jun 14 12:28:05 BST 2001
$
```

  - The dollar represents the prompt in this course – do not type it

7. Logging Out

- To exit from the shell, use the `exit` command
- Pressing `Ctrl+D` at the shell prompt will also quit the shell
- Quitting all programs should log you out
  - If in a text-only single-shell environment, exiting the shell should be sufficient
  - In a window environment, the window manager should have a log out command for this purpose
- After logging out, a new login prompt should be displayed

8. Command Syntax

- Most commands take **parameters**
  - Some commands *require* them
  - Parameters are also known as **arguments**
  - For example, `echo` simply displays its arguments:

```
$ echo

$ echo Hello there
Hello there
```

- Commands are case-sensitive
  - Usually lower-case

```
$ echo whisper
whisper
$ ECHO SHOUT
bash: ECHO: command not found
```

9. Files

- Data can be stored in a **file**
- Each file has a **filename**
  - A label used to refer to a particular file
  - Permitted characters include letters, digits, hyphens (`-`), underscores (`_`), and dots (`.`)
  - Case-sensitive – *NewsCrew.mov* is a different file from *NewScrew.mov*
- The `ls` command lists the names of files

10. Creating Files with `cat`

- There are many ways of creating a file
- One of the simplest is with the `cat` command:

```
$ cat > shopping_list
cucumber
bread
yoghurts
fish fingers
```

- Note the greater-than sign (>) – this is necessary to create the file
- The text typed is written to a file with the specified name
- Press `Ctrl+D` after a line-break to denote the end of the file
  - The next shell prompt is displayed
- `ls` demonstrates the existence of the new file

11. Displaying Files' Contents with `cat`

- There are many ways of viewing the contents of a file
- One of the simplest is with the `cat` command:

```
$ cat shopping_list
cucumber
bread
yoghurts
fish fingers
```

- Note that no greater-than sign is used
- The text in the file is displayed immediately:
  - Starting on the line after the command
  - Before the next shell prompt

12. Deleting Files with `rm`

- To delete a file, use the `rm` ('remove') command
- Simply pass the name of the file to be deleted as an argument:

```
$ rm shopping_list
```

- The file and its contents are removed
  - There is no recycle bin
  - There is no 'unrm' command
- The `ls` command can be used to confirm the deletion

13. Unix Command Feedback

- Typically, succesful commands do not give any output
- Messages are displayed in the case of errors
- The `rm` is typical
  - If it manages to delete the specified file, it does so silently
  - There is no ‘File shopping_list has been removed’ message
  - But if the command fails for whatever reason, a message is displayed
- The silence can be be off-putting for beginners
- It is standard behaviour, and doesn’t take long to get used to

14. Copying and Renaming Files with `cp` and `mv`

- To copy the contents of a file into another file, use the `cp` command:

```
$ cp CV.pdf old-CV.pdf
```

- To rename a file use the `mv` (‘move’) command:

```
$ mv commitee_minutes.txt committee_minutes.txt
```

  - Similar to using `cp` then `rm`
- For both commands, the existing name is specified as the first argument and the new name as the second
  - If a file with the new name already exists, it is overwritten

15. Filename Completion

- The shell can making typing filenames easier
- Once an unambiguous prefix has been typed, pressing `Tab` will automatically ‘type’ the rest
- For example, after typing this:

```
$ rm sho
```

  pressing `Tab` may turn it into this:

```
$ rm shopping_list
```

- This also works with command names
  - For example, `da` may be completed to `date` if no other commands start ‘da’

16. Command History

- Often it is desired to repeat a previously-executed command
- The shell keeps a **command history** for this purpose
  - Use the `Up` and `Down` cursor keys to scroll through the list of previous commands
  - Press `Enter` to execute the displayed command
- Commands can also be edited before being run
  - Particularly useful for fixing a typo in the previous command
  - The `Left` and `Right` cursor keys navigate across a command
  - Extra characters can be typed at any point
  - `Backspace` deletes characters to the left of the cursor
  - `Del` and `Ctrl+D` delete characters to the right
    - Take care not to log out by holding down `Ctrl+D` too long

17. Exercise 1

1. Log in.
2. Log out.
3. Log in again. Open a terminal window, to start a shell.
4. Exit from the shell; the terminal window will close.
5. Start another shell. Enter each of the following commands in turn. The dollar ($) represents the prompt – do not type it.
   - `$ date`
   - `$ whoami`
   - `$ hostname`
   - `$ uname`
   - `$ uptime`

18. Exercise 2

1. Use the `ls` command to see if you have any files.
2. Create a new file using the `cat` command as follows:

   ```
   $ cat > hello.txt
   Hello world!
   This is a text file.
   ```

   Press `Enter` at the end of the last line, then `Ctrl+D` to denote the end of the file.
3. Use `ls` again to verify that the new file exists.
4. Display the contents of the file.
5. Display the file again, but use the cursor keys to execute the same command again without having to retype it.

19. Exercise 3

1. Create a second file. Call it *secret-of-the-universe*, and put in whatever content you deem appropriate.
2. Check its creation with `ls`.
3. Display the contents of this file. Minimise the typing needed to do this:
   - Scroll back through the command history to the command you used to create the file.
   - Change that command to display *secret-of-the-universe* instead of creating it.

20. Exercise 4

After each of the following steps, use `ls` and `cat` to verify what has happened.

1. Copy *secret-of-the-universe* to a new file called *answer.txt*. Use `Tab` to avoid typing the existing file's name in full.
2. Now copy *hello.txt* to *answer.txt*. What's happened now?
3. Delete the original file, *hello.txt*.
4. Rename *answer.txt* to *message*.
5. Try asking `rm` to delete a file called *missing*. What happens?
6. Try copying *secret-of-the-universe* again, but don't specify a filename to which to copy. What happens now?

Module 2

Getting Started

1. Files and Directories

- A **directory** is a collection of files and/or other directories
  - Because a directory can contain other directories, we get a directory **hierarchy**
- The ‘top level’ of the hierarchy is the **root directory**
- Files and directories can be named by a **path**
  - Shows programs how to find their way to the file
  - The root directory is referred to as /
  - Other directories are referred to by name, and their names are separated by /
- If a path refers to a directory it can end in /
  - Usually an extra slash at the end of a path makes no difference

2. Examples of Absolute Paths

- An **absolute path** starts at the root of the directory hierarchy, and names directories under it
- In the root directory is a directory called *bin*, which contains a file called *ls*:

```
/bin/ls
```

- The following example will run the `ls` command, by specifying the absolute path to it:

```
$ /bin/ls
```

- We can use `ls` to list files in a specific directory by specifying the absolute path:

```
$ ls /usr/share/doc/
```

3. Current Directory

- Your shell has a **current directory** – the directory in which you are currently working
- Commands like `ls` use the current directory if none is specified
- Use the `pwd` (print working directory) command to see what your current directory is:

```
$ pwd
/home/fred
```

- Change the current directory with `cd`:

```
$ cd /mnt/cdrom
$ pwd
/mnt/cdrom
```

- Use `cd` without specifying a path to get back to your home directory

4. Making and Deleting Directories

- The `mkdir` command makes new, empty, directories
- For example, to make a directory for storing company accounts:

```
$ mkdir Accounts
```

- To delete an empty directory, use `rmdir`:

```
$ rmdir OldAccounts
```

- Use `rm` with the `-r` (recursive) option to delete directories and all the files they contain:

```
$ rm -r OldAccounts
```

- Be careful – `rm` can be a dangerous tool if misused

5. Relative Paths

- Paths don't have to start from the root directory
  - A path which doesn't start with `/` is a **relative path**
  - It is relative to some other directory, usually the current directory
- For example, the following sets of directory changes both end up in the same directory:

```
$ cd /usr/share/doc
```

```
$ cd /
$ cd usr
$ cd share/doc
```

- Relative paths specify files inside directories in the same way as absolute ones

6. Special Dot Directories

- Every directory contains two special filenames which help making relative paths:
  - The directory `..` points to the parent directory, so to list the files in the directory which contains the current directory, use `ls ..`
  - For example, if we start from */home/fred*:

```
$ cd ..
$ pwd
/home
$ cd ..
$ pwd
/
```

- The special directory `.` points to the directory it is in
  - So `./foo` is the same file as `foo`

7. Hidden Files

- The special `.` and `..` directories don't show up when you do `ls`
  - They are **hidden files**
- Simple rule: files whose names start with `.` are hidden
- Make `ls` display even the hidden files, by giving the `-a` (all) option:

```
$ ls -a
.    ..    .bashrc    .profile
```

- Hidden files are often used for configuration files
- You can still read hidden files – they just don't get listed by `ls` by default

8. Paths to Home Directories

- The symbol `~` (tilde) is an abbreviation for your home directory
  - So for user 'fred', the following are equivalent:

```
$ cd /home/fred/documents/
$ cd ~/documents/
```

- The `~` is **expanded** by the shell, so programs only see the complete path
- You can get the paths to other users' home directories using `~`, for example:

```
$ cat ~alice/notes.txt
```

- The following are all the same for user 'fred':

```
$ cd
$ cd ~
$ cd /home/fred
```

9. Looking for Files in the System

- The command `locate` lists files which contain the text you give
- For example, to find files whose name contains the word 'mkdir':

```
$ locate mkdir
/usr/man/man1/mkdir.1.gz
/usr/man/man2/mkdir.2.gz
/bin/mkdir
...
```

- `locate` is useful for finding files when you don't know exactly what they will be called, or where they are stored
- For many users, graphical tools make it easier to navigate the filesystem
  - Also make file management simpler

10. Running Programs

- Programs under Linux are files, stored in directories like */bin* and */usr/bin*
- Programs are run from the shell, simply by typing their name
- Many programs take options, which are added after their name and prefixed with `-`, for example:

```
$ ls
Accounts  notes.txt  report.txt
$ ls -l
drwxrwxr-x    2 fred   users     4096 Jan 21 10:57 Accounts
-rw-rw-r--    1 fred   users      345 Jan 21 10:57 notes.txt
-rw-r--r--    1 fred   users     3255 Jan 21 10:57 report.txt
```

- Many programs accept filenames after the options
  - Specify multiple files by separating them with spaces

11. Specifying Multiple Files

- Most programs can be given a list of files
  - For example, to delete several files at once:

    ```
    $ rm oldnotes.txt tmp.txt stuff.doc
    ```

  - To make several directories in one go:

    ```
    $ mkdir Accounts Reports
    ```

- The original use of `cat` was to join multiple files together
  - For example, to list two files, one after another:

    ```
    $ cat notes.txt morenotes.txt
    ```

- If a filename contains spaces, or characters which are interpreted by the shell (e.g., `*`, `~`), put single quotes around them:

```
$ rm 'Beatles - Strawberry Fields.mp3'
$ cat '* important notes.txt *'
```

12. Finding Documentation for Programs

- Use the `man` command to read the manual for a program
- The manual for a program is called its **man page**
  - Other things, like file formats and library functions also have man pages
- To read a man page, specify the name of the program to `man`:

```
$ man mkdir
```

- To quit from the man page viewer press `q`
- Man pages for programs usually have the following information:
  - A description of what it does
  - A list of options which it accepts
  - Other information, such as the name of the author

13. Specifying Files with Wildcards

- Use the * wildcard to specify multiple filenames to a program:

```
$ ls -l *.txt
-rw-rw-r--    1 fred   users      108 Nov 16 13:06 report.txt
-rw-rw-r--    1 fred   users      345 Jan 18 08:56 notes.txt
```

- The shell expands the wildcard, and passes the full list of files to the program
- Just using * on its own will expand to all the files in the current directory:

```
$ rm *
```

- Names with wildcards in are called **globs**, and the process of expanding them is called **globbing**

14. Chaining Programs Together

- The `who` command lists the users currently logged in
- The `wc` command counts bytes, words, and lines in its input
- We combine them to count how many users are logged in:

```
$ who | wc -l
```

- The | symbol makes a **pipe** between the two programs
  - The output of `who` is fed into `wc`
- The `-l` option makes `wc` print only the number of lines
- Another example, to join all the text files together and count the words, lines and characters in the result:

```
$ cat *.txt | wc
```

15. Graphical and Text Interfaces

- Most modern desktop Linux systems provide a **graphical user interface** (GUI)
- Linux systems use the X window system to provide graphics
  - X is just another program, not built-in to Linux
  - Usually X is started automatically when the computer boots
- Linux can be used without a GUI, just using a command line
- Use `Ctrl+Alt+F1` to switch to a text console – logging in works as it does in X
  - Use `Ctrl+Alt+F2`, `Ctrl+Alt+F3`, etc., to switch between virtual terminals – usually about 6 are provided
  - Use `Ctrl+Alt+F7`, or whatever is after the virtual terminals, to switch back to X

16. Text Editors

- Text editors are for editing plain text files
  - Don't provide advanced formatting, like word processors
  - Extremely important – manipulating text is Unix's *raison d'être*
- The most popular editors are Emacs and Vim, both of which are very sophisticated, but take time to learn
- Simpler editors include `nano`, `pico`, `kedit` and `gnotepad`
- Some programs run a text editor for you
  - They use the `$EDITOR` variable to decide which editor to use
  - Usually it is set to `vi`, but it can be changed
  - Another example of the component philosophy

17. Exercise 1

1. Use the `pwd` command to find out what directory you are in.
2. If you are not in your home directory (*/home/USERNAME*) then use `cd` without any arguments to go there, and do `pwd` again.
3. Use `cd` to visit the root directory, and list the files there. You should see *home* among the list.
4. Change into the directory called *home* and again list the files present. There should be one directory for each user, including the user you are logged in as (you can use `whoami` to check that).
5. Change into your home directory to confirm that you have gotten back to where you started.

18. Exercise 2

1. Create a text file in your home directory called *shakespear*, containing the following text:

```
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate
```

2. Rename it to *sonnet-18.txt*.
3. Make a new directory in your home directory, called *poetry*.
4. Move the poem file into the new directory.
5. Try to find a graphical directory-browsing program, and find your home directory with it. You should also be able to use it to explore some of the system directories.
6. Find a text editor program and use it to display and edit the sonnet.

19. Exercise 3

1. From your home directory, list the files in the directory */usr/share*.
2. Change to that directory, and use `pwd` to check that you are in the right place. List the files in the current directory again, and then list the files in the directory called *docs*.
3. Next list the files in the parent directory, and the directory above that.
4. Try the following command, and make sure you understand the result: `echo ~`
5. Use `cat` to display the contents of a text file which resides in your home directory (create one if you haven't already), using the `~/` syntax to refer to it. It shouldn't matter what your current directory is when you run the command.

20. Exercise 4

1. Use the `hostname` command, with no options, to print the hostname of the machine you are using.
2. Use `man` to display some documentation on the `hostname` command. Find out how to make it print the IP address of the machine instead of the hostname. You will need to scroll down the manpage to the 'Options' section.
3. Use the `locate` command to find files whose name contains the text 'hostname'. Which of the filenames printed contain the actual `hostname` program itself?

21. Exercise 5

1. The * wildcard on its own is expanded by the shell to a list of all the files in the current directory. Use the `echo` command to see the result (but make sure you are in a directory with a few files or directories first)
2. Use quoting to make `echo` print out an actual * symbol.
3. If you created a *poetry* directory earlier, augment it with another file, *sonnet-29.txt*:

```
When in disgrace with Fortune and men's eyes,
I all alone beweep my outcast state,
```

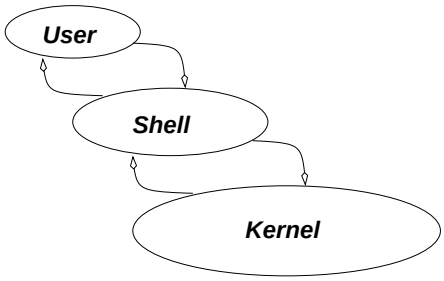
4. Use the `cat` command to display both of the poems, using a wildcard.
5. Finally, use the `rm` command to delete the *poetry* directory and the poems in it.

Module 3

Work Effectively on the Unix Command Line

1. Shells

- A **shell** provides an interface between the user and the operating system kernel
- Either a **command interpreter** or a graphical user interface
- Traditional Unix shells are **command-line interfaces** (CLIs)
- Usually started automatically when you log in



2. The Bash Shell

- Linux's most popular command interpreter is called `bash`
  - The **Bourne-Again Shell**
  - More sophisticated than the original `sh` by Steve Bourne
  - Can be run as `sh`, as a replacement for the original Unix shell
- Gives you a **prompt** and waits for a **command** to be entered
  - Traditionally, the prompt ends in $
  - Press Enter to run a command
- Although we concentrate on Bash, the shell `tcsh` is also popular
  - Based on the design of the older C Shell (`csh`)

3. Navigating the Filesystem

- Use the shell's builtin command `cd` to visit a directory
- The shell's `pwd` built-in command prints the current directory
  - These **builtins** are part of the shell
- For example:

```
$ pwd
/home/jeff
$ cd email
$ pwd
/home/jeff/email
```

- The commands entered consist of words
  - Separated by spaces (whitespace)
  - The first word is the command to run
  - Subsequent words are options or arguments to the command

4. Command-Line Arguments

- The words after the command name are passed to a command as a list of **arguments**
- Most commands group these words into two categories:
  - **Options**, usually starting with one or two hyphens
  - **Filenames**, directories, etc., on which to operate
- The options usually come first, but for most commands they do not need to
- There is a special option '`--`' which indicates the end of the options
  - Nothing after the double hyphen is treated as an option, even if it starts with `-`

5. Syntax of Command-Line Options

- Most Unix commands have a consistent syntax for options:
  - Single letter options start with a hyphen, e.g., `-B`
  - Less cryptic options are whole words or phrases, and start with two hyphens, for example `--ignore-backups`
- Some options themselves take arguments
  - Usually the argument is the next word: `sort -o` *`output_file`*
- A few programs use different styles of command-line options
  - For example, long options (not single letters) sometimes start with a single `-`, rather than `--`

6. Examples of Command-Line Options

- List all the files in the current directory:

```
ls
```

- List the files in the 'long format' (giving more information):

```
ls -l
```

- List full information about some specific files:

```
ls -l notes.txt report.txt
```

- List full information about all the *.txt* files:

```
ls -l *.txt
```

- List all files in long format, even the hidden ones:

```
ls -l -a
ls -la
```

7. Where Programs are Found

- The location of a program can be specified explicitly:
  - `./sample` runs the `sample` program in the current directory
  - `/bin/ls` runs the `ls` command in the */bin* directory
- Otherwise, the shell looks in standard places for the program
  - The variable called `$PATH` lists the directories to search in
  - Directory names are separated by colon, for example:

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin
```

  - Running `whoami` will run */bin/whoami* or */usr/bin/whoami* or */usr/local/bin/whoami* (whichever is found first)

8. Setting Shell Variables

- **Shell variables** can be used to store temporary values
- Print out the value of a shell variable with the `echo` command:

```
echo $files
```

- Set a shell variable's value as follows:

```
files="notes.txt report.txt"
```

  Note that the $ *must* be omitted when setting a variable
- Shell variables are private to the shell, but **environment variables** are passed to programs run from the shell
  - In Bash, use `export` to put a shell variable in the environment:

```
files="notes.txt report.txt"
export files
```

- New values can be built from old ones: `PATH="$PATH:/sbin"`

9. Bash Configuration Variables

- Some variables contain information which Bash itself uses
  - The variable called `$PS1` (*Prompt String 1*) specifies how to display the shell prompt
- Use the `echo` command with a $ sign before a varable name to see its value, e.g.

```
$ echo $PS1
[\u@\h \W]\$
```

- The special characters `\u`, `\h` and `\W` represent shell variables containing, respectively, your user/login name, machine's hostname and current working directory, i.e.,
  - `$USER`, `$HOSTNAME`, `$PWD`

10. Using History

- Previously executed commands can be edited with the `Up` or `Ctrl+P` keys
- This allows old commands to be executed again without re-entering
- Bash stores a **history** of old commands in memory
  - Use the builtin command `history` to display the lines remembered
  - History is stored between sessions in the file *~/.bash_history*
- Bash uses the `readline` library to read input from the user
  - Allows Emacs-like editing of the command line
  - `Left` and `Right` cursor keys, and `Delete` work as expected

11. Reusing History Items

- Previous commands can be used to build new commands, using **history expansion**
- Use `!!` to refer to the previous command, for example:

```
$ rm index.html
$ echo !!
echo rm index.html
rm index.html
```

- More often useful is `!`*`string`*, which inserts the most recent command which started with *`string`*
  - Useful for repeating particular commands without modification:

```
$ ls *.txt
notes.txt  report.txt
$ !ls
ls *.txt
notes.txt  report.txt
```

12. Retrieving Arguments from the History

- The event designator `!$` refers to the last argument of the previous command:

```
$ ls -l long_file_name.html
-rw-r--r-- 1 jeff  users   11170 Oct 31 10:47 long_file_name.html
$ rm !$
rm long_file_name.html
```

- Similarly, `!^` refers to the first argument
- The pattern `^`*`string`*`^`*`replacement`*`^` replaces the first occurrence of *`string`* with *`replacement`* in the previous command, and runs it:

```
$ echo $HOTSNAME

$ ^TS^ST^
echo $HOSTNAME
tiger
```

13. Summary of Bash Editing Keys

- These are the basic editing commands, by default:
  - `Right` – move cursor to the right
  - `Left` – move cursor to the left
  - `Up` – previous history line
  - `Down` – next history line
  - `Ctrl+A` – move to start of line
  - `Ctrl+E` – move to end of line
  - `Ctrl+D` – delete current character
- There are alternative keys, as for the Emacs editor, which can be more comfortable to use than the cursor keys
- There are other, less often used keys, which are documented in the `bash` man page (section 'Readline')

14. Combining Commands on One Line

- You can write multiple commands on one line by separating them with `;`
- Useful when the first command might take a long time:

```
time-consuming-program; ls
```

- Alternatively, use `&&` to arrange for subsequent commands to run only if earlier ones failed:

```
time-consuming-potentially-failing-program && ls
```

15. Repeating Commands with `for`

- Run multiple commands on one line by separating them with `;`
- The same commands can be repeated using `for`
  - Syntax: `for` *`varname`* `in` *`list`*`;` `do` *`commands...`*`;` `done`
- For example, to rename all *.txt* files to *.txt.old*:

```
$ for file in *.txt;
> do
>   mv -v $file $file.old;
> done
barbie.txt -> barbie.txt.old
food.txt -> food.txt.old
quirks.txt -> quirks.txt.old
```

- The command above could also be written on a single line
- The value of the variable can be expanded by prefixing it with a dollar sign (e.g., `$file`)

16. Command Substitution

- **Command substitution** allows the output of one command to be used as arguments to another
- For example, use the `locate` command to find all files called *manual.html* and print information about them with `ls`:

```
ls -l $(locate manual.html)
ls -l ‘locate manual.html‘
```

- The punctuation marks on the second form are opening single quote characters, called **backticks**
  - The `$()` form is usually preferred, but backticks are widely used
- Newlines are stripped from the output before the substitution
- Another example: use `vi` to edit the last of the files found:

```
vi $(locate manual.html | tail -1)
```

17. Finding Files with `locate`

- The `locate` command is a simple and fast way to find files
- For example, to find files relating to the email program `mutt`:

```
locate mutt
```

- The `locate` command searches a database of filenames
  - The database needs to be updated regularly
  - Usually this is done automatically with `cron`
  - But `locate` will not find files created since the last update
- The `-i` option makes the search case-insensitive
- `-r` treats the pattern as a regular expression, rather than a simple string

18. Finding Files More Flexibly: `find`

- `locate` only finds files by name
- `find` can find files by any combination of a wide number of criteria, including name
- Syntax: `find` *`directories criteria`*
- Simplest possible example: `find .`
- Finding files with a simple criterion:

```
$ find . -name manual.html
```

  Looks for files under the current directory whose name is *manual.html*
- The *criteria* always begin with a single hyphen, even though they have long names

19. `find` Criteria

- `find` accepts many different criteria; two of the most useful are:
  - `-name` *`pattern`*: selects files whose name matches the shell-style wildcard *pattern*
  - `-type d`, `-type f`: select directories or plain files, respectively
- You can have complex selections involving ‘and’, ‘or’, and ‘not’

20. `find` Actions: Executing Programs

- `find` lets you specify an action for each file found; the default action is simply to print out the name
  - You can alternatively write that explicitly as `-print`
- Other actions include executing a program; for example, to delete all files whose name starts with *manual*:

```
find . -name 'manual*' -exec rm '{}' ';'
```

- The command `rm '{}'` is run for each file, with `'{}'` replaced by the filename
- The `{}` and `;` are required by `find`, but must be quoted to protect them from the shell

21. Exercise 1

1. Use the `df` command to display the amount of used and available space on your hard drive.
2. Check the man page for `df`, and use it to find an option to the command which will display the free space in a more human-friendly form. Try both the single-letter and long-style options.
3. Run the shell, `bash`, and see what happens. Remember that you were already running it to start with. Try leaving the shell you have started with the `exit` or `logout` commands.

22. Exercise 2

1. Try `ls` with the `-a` and `-A` options. What is the difference between them?
2. Write a `for` loop which goes through all the files in a directory and prints out their names with `echo`. If you write the whole thing on one line, then it will be easy to repeat it using the command line history.
3. Change the loop so that it goes through the names of the people in the room (which needn't be the names of files) and print greetings to them.
4. Of course, a simpler way to print a list of filenames is `echo *`. Why might this be useful, when we usually use the `ls` command?

23. Exercise 3

1. Use the `find` command to list all the files and directories under your home directory. Try the `-type d` and `-type f` criteria to show just files and just directories.
2. Use `locate` to find files whose name contains the string '`bashbug`'. Try the same search with `find`, looking over all files on the system. You'll need to use the `*` wildcard at the end of the pattern to match files with extensions.
3. Find out what the `find` criterion `-iname` does.

Module 4

Process Text Streams Using Text Processing Filters

1. Working with Text Files

- Unix-like systems are designed to manipulate text very well
- The same techniques can be used with plain text, or text-based formats
  - Most Unix configuration files are plain text
- Text is usually in the **ASCII** character set
  - Non-English text might use the ISO-8859 character sets
  - Unicode is better, but unfortunately many Linux command-line utilities don't (directly) support it yet

2. Lines of Text

- Text files are naturally divided into lines
- In Linux, a line ends in a **line feed** character
  - Character number 10, hexadecimal `0x0A`
- Other operating systems use different combinations
  - Windows and DOS use a carriage return followed by a line feed
  - Macintosh systems use only a carriage return
  - Programs are available to convert between the various formats

3. Filtering Text and Piping

- The Unix philosophy: use small programs, and link them together as needed
- Each tool should be good at one specific job
- Join programs together with **pipes**
  - Indicated with the pipe character: |
  - The first program prints text to its **standard output**
  - That gets fed into the second program's **standard input**
- For example, to connect the output of `echo` to the input of `wc`:

```
$ echo "count these words, boy" | wc
```

4. Displaying Files with `less`

- If a file is too long to fit in the terminal, display it with `less`:

```
$ less README
```

- `less` also makes it easy to clear the terminal of other things, so is useful even for small files
- Often used on the end of a pipe line, especially when it is not known how long the output will be:

```
$ wc *.txt | less
```

- Doesn't choke on strange characters, so it won't mess up your terminal (unlike `cat`)

5. Concatenating Files with `cat`

- The `cat` filter concatenates the contents of all the files named on its command line
- More commonly used with just one file to quickly send its contents to the screen:

```
$ cat /etc/resolv.conf
domain gbdirect.co.uk
nameserver 127.0.0.1
nameserver 192.168.100.12
```

6. Counting Words and Lines with `wc`

- Count characters, words and lines in a file
- If used with multiple files, outputs counts for each file, and a combined total
- Options:
  - `-c` output character count
  - `-l` output line count
  - `-w` output word count
  - Default is `-clw`
- Examples: display word count for *essay.txt*:

```
$ wc -w essay.txt
```

- Display the total number of lines in several text files:

```
$ wc -l *.txt
```

7. Sorting Lines of Text with `sort`

- The `sort` filter reads lines of text and prints them sorted into order
- For example, to sort a list of words into dictionary order:

```
$ sort words > sorted-words
```

- The `-f` option makes the sorting **case-insensitive**
- The `-n` option sorts numerically, rather than lexicographically

8. Removing Duplicate Lines with `uniq`

- Use `uniq` to find unique lines in a file
  - Removes *consecutive* duplicate lines
  - Usually give it sorted input, to remove all duplicates
- Example: find out how many unique words are in a dictionary:

```
$ sort /usr/dict/words | uniq | wc -w
```

- `sort` has a `-u` option to do this, without using a separate program:

```
$ sort -u /usr/dict/words | wc -w
```

- `sort | uniq` can do more than `sort -u`, though:
  - `uniq -c` counts how many times each line appeared
  - `uniq -u` prints only unique lines
  - `uniq -d` prints only duplicated lines

9. Selecting Parts of Lines with `cut`

- Used to select columns or fields from each line of input
- Select a range of
  - Characters, with `-c`
  - Fields, with `-f`
- Field separator specified with `-d` (defaults to tab)
- A range is written as start and end position: e.g., `3-5`
  - Either can be missed out
  - The first byte, character or field is numbered 1, not 0
- Example: select usernames of logged in users:

```
$ who | cut -d" " -f1 | sort -u
```

10. Expanding Tabs to Spaces with `expand`

- Used to replace tabs with spaces in files
- Tab size (maximum number of spaces for each tab) can be set with `-t` *`number`*
  - Default tab size is 8
- To only change tabs at the beginning of lines, use `-i`
- Example: change all tabs in *foo.txt* to three spaces, display it to the screen (both of these are the same):

```
$ expand -t 3 foo.txt
$ expand -3 foo.txt
```

11. Using `fmt` to Format Text Files

- Arranges words nicely into lines of consistent length
- Use `-u` to convert to uniform spacing
  - One space between words, two between sentences
- Use `-w` *`width`* to set the maximum line width in characters
  - Defaults to 75
- Example: change the line length of *notes.txt* to a maximum of 70 characters, and display it on the screen:

```
$ fmt -w 70 notes.txt | less
```

12. Reading the Start of a File with `head`

- Prints the top of its input, and discards the rest
- Set the number of lines to print with `-n` *`lines`* or `-`*`lines`*
  - Defaults to ten lines
- View the headers of a HTML document called *homepage.html*:

```
$ head homepage.html
```

- Print the first line of a text file (two alternatives):

```
$ head -n 1 notes.txt
$ head -1 notes.txt
```

13. Reading the End of a File with `tail`

- Similar to `head`, but prints lines at the end of a file
- The `-f` option watches the file forever
  - Continually updates the display as new entries are appended to the end of the file
  - Kill it with `Ctrl+C`
- The option `-n` is the same as in `head` (number of lines to print)
- Example: monitor HTTP requests on a webserver:

```
$ tail -f /var/log/httpd/access_log
```

14. Numbering Lines of a File with `nl`

- Add a line number to each line of the input
- There are options to finely control the formating
- By default, blank lines aren't numbered
  - The option `-ba` numbers every line
  - `cat -n` also numbers lines, including blank ones

15. Dumping Bytes of Binary Data with `od`

- Prints the numeric values of the bytes in a file
- Useful for studying files with non-text characters
- By default, prints two byte words in octal
- Specify an alternative with the `-t` option
  - Give a letter to indicate base: `o` for octal, `x` for hexadecimal, `u` for unsigned decimal, etc.
  - Can be followed by the number of bytes per word
  - Add `z` to show ASCII equivalents alongside the numbers
  - A useful format is given by `od -t x1z` – hexadecimal, one byte words, with ASCII
- Alternatives to `od` include `xxd` and `hexdump`

16. Paginating Text Files with `pr`

- Convert a text file into paginated text, with headers and page fills
- Rarely useful for modern printers
- Options:
  - `-d` double spaced output
  - `-h` *`header`* change from the default header to *`header`*
  - `-l` *`lines`* change the default lines on a page from 66 to *`lines`*
  - `-o` *`width`* set ('offset') the left margin to *`width`*
- Example:

```
$ pr -h "My Thesis" thesis.txt | lpr
```

17. Dividing Files into Chunks with `split`

- Splits files into equal sized segments
- Syntax: `split [`*`options`*`] [`*`input`*`] [`*`output-prefix`*`]`
- Use `-l` *n* to split a file into *n*-line chunks
- Use `-b` *n* to split into chunks of *n* bytes each
- Output files are named using the specified output name with *aa*, *ab*, *ac*, etc., added to the end of the prefix
- Example: Split *essay.txt* into 30 line files, and save the output to files *short_aa*, *short_ab*, etc:

```
$ split -l 30 essay.txt short_
```

18. Using `split` to Span Disks

- If a file is too big to fit on a single floppy, Zip or CD-ROM disk, it can be split into small enough chunks
- Use the `-b` option, and with the `k` and `m` sufixes to give the chunk size in kilobytes or megabytes
- For example, to split the file *database.tar.gz* into pieces small enough to fit on Zip disks:

```
$ split -b 90m database.tar.gz zip-
```

- Use `cat` to put the pieces back together:

```
$ cat zip-* > database.tar.gz
```

19. `tac`: Backwards `cat`

- Similar to `cat`, but in reverse
- Prints the last line of the input first, the penultimate line second, and so on
- Example: show a list of logins and logouts, but with the most recent events at the end:

```
$ last | tac
```

20. Translating Sets of Characters with `tr`

- Translate one set of characters to another
- Usage: `tr [`*`options`*`]` *`start-set end-set`*
- Replaces all characters in *`start-set`* with the corresponding characters in *`end-set`*
- Cannot accept a file as an argument, but uses the standard input and output
- Options:
  - `-d` deletes characters in *`start-set`* instead of translating them
  - `-s` replaces sequences of identical characters with just one (squeezes them)

21. `tr` Examples

- Replace all uppercase characters in *input-file* with lowercase characters (two alternatives):

```
$ cat input-file | tr A-Z a-z
$ tr A-Z a-z < input-file
```

- Delete all occurrences of `z` in *story.txt*:

```
$ cat story.txt | tr -d z
```

- Run together each sequence of repeated `f` characters in *lullaby.txt* to with just one `f`:

```
$ tr -s f < lullaby.txt
```

22. sed – the Stream Editor

- `sed` uses a simple script to process each line of a file
- Specify the script file with `-f` *`filename`*
- Or give individual commands with `-e` *`command`*
- For example, if you have a script called *spelling.sed* which corrects your most common mistakes, you can feed a file through it:

```
$ sed -f spelling.sed < report.txt > corrected.txt
```

23. Substituting with sed

- Use the `s/`*`pattern`*`/`*`replacement`*`/` command to substitute text matching the *`pattern`* with the *`replacement`*
  - Add the `/g` modifier to replace every occurrence on each line, rather than just the first one
- For example, replace ‘thru’ with ‘through’:

```
$ sed -e 's/thru/through/g' input-file > output-file
```

- `sed` has more complicated facilities which allow commands to be executed conditionally
  - Can be used as a very basic (but unpleasantly difficult!) programming language

24. Put Files Side-by-Side with paste

- `paste` takes lines from two or more files and puts them in columns of the output
- Use `-d` *`char`* to set the delimiter between fields in the output
  - The default is tab
  - Giving `-d` more than one character sets different delimiters between each pair of columns
- Example: assign passwords to users, separating them with a colon:

```
$ paste -d: usernames passwords > .htpasswd
```

25. Performing Database Joins with `join`

- Does a database-style 'inner join' on two tables, stored in text files
- The `-t` option sets the field delimiter
  - By default, fields are separated by any number of spaces or tabs
- Example: show details of suppliers and their products:

```
$ join suppliers.txt products.txt | less
```

- The input files must be sorted!
- This command is rarely used – databases have this facility built in

26. Exercise 1

1. Type in the example on the `cut` slide to display a list of users logged in. (Try just `who` on its own first to see what is happening.)
2. Arrange for the list of usernames in `who`'s output to be sorted, and remove any duplicates.
3. Try the command `last` to display a record of login sessions, and then try reversing it with `tac`. Which is more useful? What if you pipe the output into `less`?
4. Use `sed` to correct the misspelling 'enviroment' to 'environment'. Use it on a test file, containing a few lines of text, to check it. Does it work if the misspelling occurs more than once on the same line?
5. Use `nl` to number the lines in the output of the previous question.

27. Exercise 2

1. Try making an empty file and using `tail -f` to monitor it. Then add lines to it from a different terminal, using a command like this:

```
$ echo "testing" >>filename
```

2. Once you have written some lines into your file, use `tr` to display it with all occurances of the letters `A`–`F` changed to the numbers `0`–`5`.
3. Try looking at the binary for the `ls` command (*/bin/ls*) with `less`. You can use the `-f` option to force it to display the file, even though it isn't text.
4. Try viewing the same binary with `od`. Try it in its default mode, as well as with the options shown on the slide for outputting in hexadecimal.

28. Exercise 3

1. Use the `split` command to split the binary of the `ls` command into 1Kb chunks. You might want to create a directory especially for the split files, so that it can all be easily deleted later.
2. Put your split `ls` command back together again, and run it to make sure it still works. You will have to make sure you are running the new copy of it, for example `./my_ls`, and make sure that the program is marked as 'executable' to run it, with the following command:

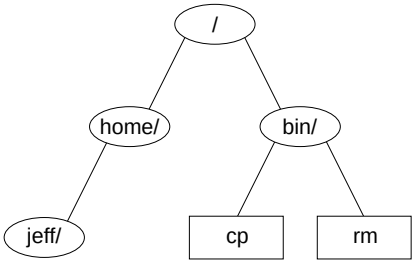
```
$ chmod a+rx my_ls
```

Module 5

Perform Basic File Management

1. Filesystem Objects

- A **file** is a place to store data: a possibly-empty sequence of bytes
- A **directory** is a collection of files and other directories
- Directories are organized in a hierarchy, with the **root directory** at the top
  - The root directory is referred to as /



2. Directory and File Names

- Files and directories are organized into a **filesystem**
- Refer to files in directories and sub-directories by separating their names with /, for example:

```
/bin/ls
/usr/dict/words
/home/jeff/recipe
```

- Paths to files either start at / (absolute) or from some 'current' directory

3. File Extensions

- It's common to put an **extension**, beginning with a dot, on the end of a filename
- The extension indicates the type of the file:

.txt	Text file
.gif	Graphics Interchange Format image
.jpg	Joint Photographic Experts Group image
.mp3	MPEG-2 Layer 3 audio
.gz	Compressed file
.tar	Unix 'tape archive' file
.tar.gz, *.tgz*	Compressed archive file

- On Unix and Linux, file extensions are entirely a matter of convention – the operating system itself ignores them
  - Only a few specific programs use extensions to guess what type a file is

4. Visiting Directories with cd

- Use `cd` to change the **current working directory**
  - If no argument is given, it changes to your home directory
- Use `pwd` to find out what the current directory is
- For example:

```
$ cd /usr/src
$ pwd
/usr/src
```

- To switch back to the previous directory: `cd -`

5. Going Back to Previous Directories

- The `pushd` command takes you to another directory, like `cd`
  - But also saves the current directory, so that you can go back later
- For example, to visit Fred's home directory, and then go back to where you started from:

```
$ pushd ~fred
$ cd Work
$ ls
...
$ popd
```

- `popd` takes you back to the directory where you last did `pushd`
- `dirs` will list the directories you can pop back to

6. Filename Completion

- Modern shells help you type the names of files and directories by completing partial names
- Type the start of the name (enough to make it unambiguous) and press `Tab`
- For an ambiguous name (there are several possible completions), the shell can list the options:
  - For Bash, type `Tab` twice in succession
  - For C shells, type `Ctrl+D`
- Both of these shells will automatically escape spaces and special characters in the filenames

7. Wildcard Patterns

- Give commands multiple files by specifying patterns
- Use the symbol * to match any part of a filename:

```
$ ls *.txt
accounts.txt  letter.txt  report.txt
```

- Just * produces the names of all files in the current directory
- The wildcard ? matches exactly one character:

```
$ rm -v data.?
removing data.1
removing data.2
removing data.3
```

- Note: wildcards are turned into filenames by the shell, so the program you pass them to can't tell that those names came from wildcard expansion

8. Copying Files with cp

- Syntax: `cp [options] source-file destination-file`
- Copy multiple files into a directory: `cp files directory`
- Common options:
  - `-f`, force overwriting of destination files
  - `-i`, interactively prompt before overwriting files
  - `-a`, archive, copy the contents of directories recursively

9. Examples of cp

- Copy */etc/smb.conf* to the current directory:

```
$ cp /etc/smb.conf .
```

- Create an identical copy of a directory called *work*, and call it *work-backup*:

```
$ cp -a work work-backup
```

- Copy all the GIF and JPEG images in the current directory into *images*:

```
$ cp *.gif *.jpeg images/
```

10. Moving Files with `mv`

- `mv` can rename files or directories, or move them to different directories
- It is equivalent to copying and then deleting
  - But is usually much faster
- Options:
  - `-f`, force overwrite, even if target already exists
  - `-i`, ask user interactively before overwriting files
- For example, to rename *poetry.txt* to *poems.txt*:

```
$ mv poetry.txt poems.txt
```

- To move everything in the current directory somewhere else:

```
$ mv * ~/old-stuff/
```

11. Deleting Files with `rm`

- `rm` deletes ('removes') the specified files
- You must have write permission for the directory the file is in to remove it
- Use carefully if you are logged in as root!
- Options:
  - `-f`, delete write-protected files without prompting
  - `-i`, interactive – ask the user before deleting files
  - `-r`, recursively delete files and directories
- For example, clean out everything in */tmp*, without prompting to delete each file:

```
$ rm -rf /tmp/*
```

12. Deleting Files with Peculiar Names

- Some files have names which make them hard to delete
- Files that begin with a minus sign:

```
$ rm ./-filename
$ rm -- -filename
```

- Files that contain peculiar characters – perhaps characters that you can't actually type on your keyboard:
  - Write a wildcard pattern that matches *only* the name you want to delete:

    ```
    $ rm -i ./name-with-funny-characters*
    ```

  - The `./` forces it to be in the current directory
  - Using the `-i` option to `rm` makes sure that you won't delete anything else by accident

13. Making Directories with `mkdir`

- Syntax: `mkdir` *`directory-names`*
- Options:
  - `-p`, create intervening parent directories if they don't already exist
  - `-m` *`mode`*, set the access permissions to *`mode`*
- For example, create a directory called *mystuff* in your home directory with permissions so that only you can write, but eveyone can read it:

```
$ mkdir -m 755 ~/mystuff
```

- Create a directory tree in */tmp* using one command with three subdirectories called *one*, *two* and *three*:

```
$ mkdir -p /tmp/one/two/three
```

14. Removing Directories with `rmdir`

- `rmdir` deletes empty directories, so the files inside must be deleted first
- For example, to delete the *images* directory:

```
$ rm images/*
$ rmdir images
```

- For non-empty directories, use `rm -r` *`directory`*
- The `-p` option to `rmdir` removes the complete path, if there are no other files and directories in it
  - These commands are equivalent:

```
$ rmdir -p a/b/c
$ rmdir a/b/c  a/b  a
```

15. Identifying Types of Files

- The data in files comes in various different formats (executable programs, text files, etc.)
- The `file` command will try to identify the type of a file:

```
$ file /bin/bash
/bin/bash: ELF 32-bit LSB executable, Intel 80386, version 1,
dynamically linked (uses shared libs), stripped
```

- It also provides extra information about some types of file
- Useful to find out whether a program is actually a script:

```
$ file /usr/bin/zless
/usr/bin/zless: Bourne shell script text
```

- If `file` doesn't know about a specific format, it will guess:

```
$ file /etc/passwd
/etc/passwd: ASCII text
```

16. Changing Timestamps with `touch`

- Changes the **access** and **modification** times of files
- Creates files that didn't already exist
- Options:
  - `-a`, change only the access time
  - `-m`, change only the modification time
  - `-t [`*`YYYY`*`]`*`MMDDhhmm`*`[.`*`ss`*`]`, set the timestamp of the file to the specified date and time
  - GNU `touch` has a `-d` option, which accepts times in a more flexible format
- For example, change the time stamp on *homework* to January 20 2001, 5:59p.m.

```
$ touch -t 200101201759 homework
```

17. Exercise 1

1. Just after you've logged in or opened a terminal window, what directory is your current directory? (The `pwd` command might be useful here.)
2. Change directory to */etc* and then */tmp*. Each time use `pwd` to check that you got to the right place, and `ls` to see what files are there.
3. From */tmp*, use `cd ..` and see where you end up. Then try it again from your working directory.
4. Find out what files are in */bin*, without changing your current directory. You should recognise the names of some of the programs stored there.
5. List the files in */usr/bin* and */usr/local/bin* (if it exists). Use the `Tab` key to save typing.

18. Exercise 2

1. Use `cd` to go to your home directory, and create a new directory there called *dog*.
2. Create another directory within that one called *cat*, and another within that called *mouse*.
3. Remove all three directories. You can either remove them one at a time, or all at once.
4. If you can delete directories with `rm -r`, what is the point of using `rmdir` for empty directories?
5. Try creating the *dog/cat/mouse* directory structure with a single command.

19. Exercise 3

1. Copy the file */etc/passwd* to your home directory, and then use `cat` to see what's in it.
2. Rename it to *users* using the `mv` command.
3. Make a directory called *programs* and copy everything from */bin* into it.
4. Delete all the files in the *programs* directory.
5. Delete the empty *programs* directory and the *users* file.

20. Exercise 4

1. The `touch` command can be used to create new empty files. Try that now, picking a name for the new file:

   ```
   $ touch baked-beans
   ```

2. Get details about the file using the `ls` command:

   ```
   $ ls -l baked-beans
   ```

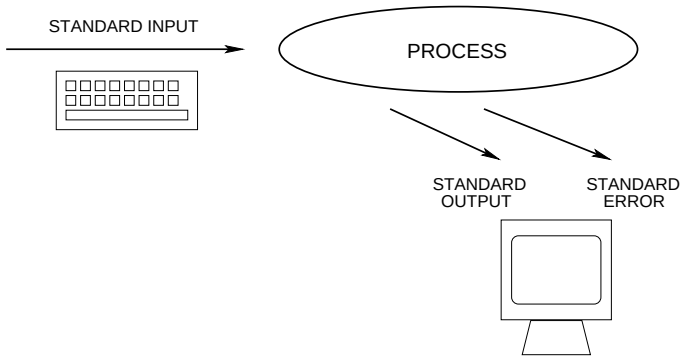
3. Wait for a minute, and then try the previous two steps again, and see what changes. What happens when we don't specify a time to `touch`?
4. Try setting the timestamp on the file to a value in the future.
5. When you're finished with it, delete the file.

Module 6

Use Unix Streams, Pipes and Redirects

1. Standard Files

- Processes are connected to three standard files



- Many programs open other files as well

2. Standard Input

- Programs can read data from their **standard input** file
- Abbreviated to **stdin**
- By default, this reads from the keyboard
- Characters typed into an interactive program (e.g., a text editor) go to stdin

3. Standard Output

- Programs can write data to their **standard output** file
- Abbreviated to **stdout**
- Used for a program's normal output
- By default this is printed on the terminal

4. Standard Error

- Programs can write data to their **standard error** output
- Standard error is similar to standard output, but used for error and warning messages
- Abbreviated to **stderr**
- Useful to separate program output from any program errors
- By default this is written to your terminal
  - So it gets ‘mixed in’ with the standard output

5. Pipes

- A **pipe** channels the output of one program to the input of another
  - Allows programs to be chained together
  - Programs in the chain run concurrently
- Use the vertical bar: |
  - Sometimes known as the ‘pipe’ character
- Programs don’t need to do anything special to use pipes
  - They read from stdin and write to stdout as normal
- For example, pipe the output of `echo` into the program `rev` (which reverses each line of its input):

```
$ echo Happy Birthday! | rev
!yadhtriB yppaH
```

6. Connecting Programs to Files

- **Redirection** connects a program to a named file
- The < symbol indicates the file to read input from:

```
$ wc < thesis.txt
```

  - The file specified becomes the program’s standard input
- The > symbol indicates the file to write output to:

```
$ who > users.txt
```

  - The program’s standard output goes into the file
  - If the file already exists, it is overwritten
- Both can be used at the same time:

```
$ filter < input-file > output-file
```

7. Appending to Files

- Use >> to append to a file:

```
$ date >> log.txt
```

  - Appends the standard output of the program to the end of an existing file
  - If the file doesn't already exist, it is created

8. Redirecting Multiple Files

- Open files have numbers, called **file descriptors**
- These can be used with redirection
- The three standard files always have the same numbers:

Name	Descriptor
Standard input	0
Standard output	1
Standard error	2

9. Redirection with File Descriptors

- Redirection normally works with stdin and stdout
- Specify different files by putting the file descriptor number before the redirection symbol:
  - To redirect the standard error to a file:

```
$ program 2> file
```

  - To combine standard error with standard output:

```
$ program > file 2>&1
```

  - To save both output streams:

```
$ program > stdout.txt 2> stderr.txt
```

- The descriptors 3–9 can be connected to normal files, and are mainly used in shell scripts

10. Running Programs with `xargs`

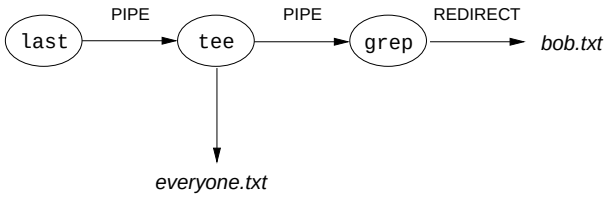
- `xargs` reads pieces of text and runs another program with them as its arguments
  - Usually its input is a list of filenames to give to a file processing program
- Syntax: `xargs` *`command`* [*`initial args`*]
- Use `-l` *n* to use *n* items each time the command is run
  - The default is 1
- `xargs` is very often used with input piped from `find`
- Example: if there are too many files in a directory to delete in one go, use `xargs` to delete them ten at a time:

```
$ find /tmp/rubbish/ | xargs -l10 rm -f
```

11. `tee`

- The `tee` program makes a 'T-junction' in a pipeline
- It copies data from stdin to stdout, and also to a file
- Like > and | combined
- For example, to save details of everyone's logins, and save Bob's logins in a separate file:

```
$ last | tee everyone.txt | grep bob > bob.txt
```



12. Exercise 1

1. Try the example on the 'Pipes' slide, using `rev` to reverse some text.
2. Try replacing the `echo` command with some other commands which produce output (e.g., `whoami`).
3. What happens when you replace `rev` with `cat`? You might like to try running `cat` with no arguments and entering some text.

13. Exercise 2

1. Run the command `ls --color` in a directory with a few files and directories. Some Linux distributions have `ls` set up to always use the `--color` option in normal circumstances, but in this case we will give it explicitly.
2. Try running the same command, but pipe the output into another program (e.g., `cat` or `less`). You should spot two differences in the output. `ls` detects whether its output is going straight to a terminal (to be viewed by a human directly) or into a pipe (to be read by another program).

Module 7

Search Text Files Using Regular Expressions

1. Searching Files with `grep`

- `grep` prints lines from files which match a pattern
- For example, to find the entries in the password file */etc/passwd* relating to the user 'nancy':

```
$ grep nancy /etc/passwd
```

- `grep` has a few useful options:
  - `-i` makes the matching case-insensitive
  - `-r` searches through files in specified directories, recursively
  - `-l` prints just the names of files which contain matching lines
  - `-c` prints the count of matches in each file
  - `-n` numbers the matching lines in the output
  - `-v` reverses the test, printing lines which don't match

2. Pattern Matching

- Use `grep` to find patterns, as well as simple strings
- Patterns are expressed as **regular expressions**
- Certain punctuation characters have special meanings
- For example this might be a better way to search for Nancy's entry in the password file:

```
$ grep '^nancy' /etc/passwd
```

  - The caret (`^`) anchors the pattern to the start of the line
- In the same way, $ acts as an **anchor** when it appears at the end of a string, making the pattern match only at the end of a line

3. Matching Repeated Patterns

- Some regexp special characters are also special to the shell, and so need to be protected with quotes or backslashes
- We can match a repeating pattern by adding a modifier:

```
$ grep -i 'continued\.*'
```

- Dot (`.`) on its own would match any character, so to match an actual dot we escape it with \
- The * modifier matches the preceding character zero or more times
- Similarly, the \+ modifier matches one or more times

4. Matching Alternative Patterns

- Multiple subpatterns can be provided as alternatives, separated with `\|`, for example:

```
$ grep 'fish\|chips\|pies' food.txt
```

- The previous command finds lines which match at least one of the words
- Use `\(...\)` to enforce precedence:

```
$ grep -i '\(cream\|fish\|birthday\) cakes' delicacies.txt
```

- Use square brackets to build a **character class**:

```
$ grep '[Jj]oe [Bb]loggs' staff.txt
```

- Any single character from the class matches; and ranges of characters can be expressed as '`a-z`'

5. Extended Regular Expression Syntax

- `egrep` runs `grep` in a different mode
  - Same as `grep -E`
- Special characters don't have to be marked with `\`
  - So `\+` is written `+`, `\(...\)` is written `(...)`, etc
  - In extended regexps, `\+` is a literal `+`

6. sed

- `sed` reads input lines, runs editing-style commands on them, and writes them to stdout
- `sed` uses regular expressions as patterns in substitutions
  - `sed` regular expressions use the same syntax as `grep`
- For example, to used `sed` to put # at the start of each line:

```
$ sed -e 's/^/#/' < input.txt > output.txt
```

- `sed` has simple substitution and translation facilities, but can also be used like a programming language

7. Further Reading

- *Sed and Awk*, 2nd edition, by Dale Dougherty and Arnold Robbins, 1997
- The Sed FAQ, http://www.dbnet.ece.ntua.gr/~george/sed/sedfaq.html
- The original user manual (1978), http://www.urc.bl.ac.yu/manuals/progunix/sed.txt

8. Exercise 1

1. Use `grep` to find information about the HTTP protocol in the file */etc/services*.
2. Usually this file contains some comments, starting with the '#' symbol. Use `grep` with the `-v` option to ignore lines starting with '#' and look at the rest of the file in `less`.
3. Add another use of `grep -v` to your pipeline to remove blank lines (which match the pattern `^$`).
4. Use `sed` (also in the same pipeline) to remove the information after the '/' symbol on each line, leaving just the names of the protocols and their port numbers.

Module 8

Job Control

1. Job Control

- Most shells offer **job control**
  - The ability to stop, restart, and background a running process
- The shell lets you put `&` on the end of a command line to start it in the **background**
- Or you can hit `Ctrl+Z` to **suspend** a running foreground job
- Suspended and backgrounded jobs are given numbers by the shell
- These numbers can be given to shell job-control built-in commands
- Job-control commands include `jobs`, `fg`, and `bg`

2. `jobs`

- The `jobs` builtin prints a listing of active jobs and their job numbers:

```
$ jobs
[1]-  Stopped                 vim index.html
[2]   Running                 netscape &
[3]+  Stopped                 man ls
```

- Job numbers are given in square brackets
  - But when you use them with other job-control builtins, you need to write them with percent signs, for example `%1`
- The jobs marked `+` and `-` may be accessed as `%+` or `%-` as well as by number
  - `%+` is the shell's idea of the **current job** – the most recently active job
  - `%-` is the *previous* current job

3. `fg`

- Brings a backgrounded job into the foreground
- Re-starts a suspended job, running it in the foreground
- `fg %1` will foreground job number 1
- `fg` with no arguments will operate on the current job

4. bg

- Re-starts a suspended job, running it in the background
- `bg %1` will background job number 1
- `bg` with no arguments will operate on the current job
- For example, after running `gv` and suspending it with `Ctrl+Z`, use `bg` to start it running again in the background

5. Exercise 1

1. Start a process by running `man bash` and suspend it with `Ctrl+Z`.
2. Run `xclock` in the background, using `&`.
3. Use `jobs` to list the backgrounded and stopped processes.
4. Use the `fg` command to bring `man` into the foreground, and quit from it as normal.
5. Use `fg` to foreground `xclock`, and terminate it with `Ctrl+C`.
6. Run `xclock` again, but this time without `&`. It should be running in the foreground (so you can't use the shell). Try suspending it with `Ctrl+Z` and see what happens. To properly put it into the background, use `bg`.

Module 9

Create, Monitor, and Kill Processes

1. What is a Process?

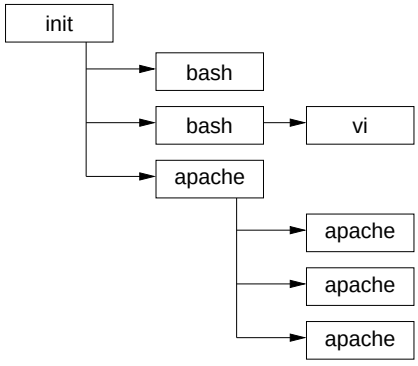
- The kernel considers each program running on your system to be a **process**
- A process 'lives' as it executes, with a lifetime that may be short or long
- A process is said to 'die' when it terminates
- The kernel identifies each process by a number known as a process id, or **pid**
- The kernel keeps track of various properties of each process

2. Process Properties

- A process has a user id (**uid**) and a group id (**gid**) which together specify what permissions it has
- A process has a parent process id (**ppid**) – the pid of the process which created it
  - The kernel starts an `init` process with pid 1 at boot-up
  - Every other process is a descendant of pid 1
- Each process has its own **working directory**, initially inherited from its parent process
- There is an **environment** for each process – a collection of named environment variables and their associated values
  - A process's environment is normally inherited from its parent process

3. Parent and Child Processes

- The `init` process is the ancestor of all other processes:



- (Apache starts many child processes so that they can serve HTTP requests at the same time)

4. Process Monitoring: ps

- The ps command gives a snapshot of the processes running on a system at a given moment in time
- Very flexible in what it shows, and how:
  - Normally shows a fairly brief summary of each process
  - Normally shows only processes which are both owned by the current user and attached to a terminal
- Unfortunately, it doesn't use standard option syntax
- Instead it uses a mixture of options with one of three syntaxes:
  - Traditional BSD ps: a single letter *with no hyphen*
  - Unix98 ps: a single letter preceded by a hyphen
  - GNU: a word or phrase preceded by two hyphens

5. ps Options

- ps has many options
- Some of the most commonly used are:

Option	Description
a	Show processes owned by other users
f	Display process ancestors in a tree-like format
u	Use the 'user' output format, showing user names and process start times
w	Use a wider output format. Normally each line of output is truncated; each use of the w option makes the 'window' wider
x	Include processes which have no controlling terminal
-e	Show information on *all* processes
-l	Use a 'long' output format
-f	Use a 'full' output format
-C *cmd*	Show only processes named *cmd*
-U *user*	Show only processes owned by *user*

6. Process Monitoring: pstree

- Displays a snapshot of running processes
- Always uses a tree-like display, like ps f
  - But by default shows only the name of each command
- Normally shows all processes
  - Specify a pid as an argument to show a specific process and its descendants
  - Specify a user name as an argument to show process trees owned by that user

7. `pstree` Options

Option	Description
`-a`	Display commands' arguments
`-c`	Don't compact identical subtrees
`-G`	Attempt to use terminal-specific line-drawing characters
`-h`	Highlight the ancestors of the current process
`-n`	Sort processes numerically by pid, rather than alphabetically by name
`-p`	Include pids in the output

8. Process Monitoring: `top`

- Shows full-screen, continuously-updated snapshots of process activity
  - Waits a short period of time between each snapshot to give the illusion of real-time monitoring
- Processes are displayed in descending order of how much processor time they're using
- Also displays system uptime, load average, CPU status, and memory information

9. `top` Command-Line Options

Option	Description
`-b`	Batch mode – send snapshots to standard output
`-n` *num*	Exit after displaying *num* snapshots
`-d` *delay*	Wait *delay* seconds between each snapshot
`-i`	Ignore idle processes
`-s`	Disable interactive commands which could be dangerous if run by the superuser

10. `top` Interactive Commands

Key	Behaviour
`q`	Quit the program
`Ctrl+L`	Repaint the screen
`h`	Show a help screen
`k`	Prompts for a pid and a signal, and sends that signal to that process
`n`	Prompts for the number of processes to show information; 0 (the default) means to show as many as will fit
`r`	Change the priority ('niceness') of a process
`s`	Change the number of seconds to delay between updates. The number may include fractions of a second (`0.5`, for example)

11. Signalling Processes

- A process can be sent a **signal** by the kernel or by another process
- Each signal is a very simple message:
  - A small whole number
  - With a mnemonic name
- Signal names are all-capitals, like `INT`
  - They are often written with `SIG` as part of the name: `SIGINT`
- Some signals are treated specially by the kernel; others have a conventional meaning
- There are about 30 signals available, not all of which are very useful

12. Common Signals for Interactive Use

- The command `kill -l` lists all signals
- The following are the most commonly used:

Name	Number	Meaning
`INT`	2	Interrupt – stop running. Sent by the kernel when you press `Ctrl+C` in a terminal.
`TERM`	15	"Please terminate." Used to ask a process to exit gracefully.
`KILL`	9	"Die!" Forces the process to stop running; it is given no opportunity to clean up after itself.
`TSTP`	18	Requests the process to stop itself temporarily. Sent by the kernel when you press `Ctrl+Z` in a terminal.
`HUP`	1	Hang up. Sent by the kernel when you log out, or disconnect a modem. Conventionally used by many dæmons as an instruction to re-read a configuration file.

13. Sending Signals: `kill`

- The `kill` command is used to send a signal to a process
  - Not just to terminate a running process!
- It is a normal executable command, but many shells also provide it as a built-in
- Use `kill -HUP` *`pid`* or `kill -s HUP` *`pid`* to send a `SIGHUP` to the process with that *pid*
- If you miss out the signal name, `kill` will send a `SIGTERM`
- You can specify more than one pid to signal all those processes

14. Sending Signals to Dæmons: `pidof`

- On Unix systems, long-lived processes that provide some service are often referred to as **dæmons**
- Dæmons typically have a configuration file (usually under */etc*) which affects their behaviour
- Many dæmons read their configuration file only at startup
- If the configuration changes, you have to explicitly tell the dæmon by sending it a `SIGHUP` signal
- You can sometimes use `pidof` to find the dæmon's pid; for example, to tell the `inetd` dæmon to reload its configuration, run:

  ```
  $ kill -HUP $(pidof /usr/sbin/inetd)
  ```

  as root

15. Exercise 1

1. Use `top` to show the processes running on your machine.
2. Make `top` sort by memory usage, so that the most memory-hungry processes appear at the top.
3. Restrict the display to show only processes owned by you.
4. Try killing one of your processes (make sure it's nothing important).
5. Display a list of all the processes running on the machine using `ps` (displaying the full command line for them).
6. Get the same listing as a tree, using both `ps` and `pstree`.
7. Have `ps` sort the output by system time used.

Module 10

Modify Process Execution Priorities

1. Concepts

- Not all tasks require the same amount of execution time
- Linux has the concept of **execution priority** to deal with this
- Process priority is dynamically altered by the kernel
- You can view the current priority by looking at `top` or `ps -l` and looking at the `PRI` column
- The priority can be biased using `nice`
  - The current bias can be seen in the `NI` column in `top`

2. `nice`

- Starts a program with a given priority bias
- Peculiar name: 'nicer' processes require fewer resources
- Niceness ranges from +19 (very nice) to −20 (not very nice)
- Non-root users can only specify values from 1 to 19; the root user can specify the full range of values
- Default niceness when using `nice` is 10
- To run a command at increased niceness (lower priority):

```
$ nice -10 long-running-command &
$ nice -n 10 long-running-command &
```

- To run a command at decreased niceness (higher priority):

```
$ nice --15 important-command &
$ nice -n -15 important-command &
```

3. `renice`

- `renice` changes the niceness of existing processes
- Non-root users are only permitted to increase a process's niceness
- To set the process with pid 2984 to the maximum niceness (lower priority):

```
$ renice 20 2984
```

  - The niceness is just a number: no extra `-` sign
- To set the process with pid 3598 to a lower niceness (higher priority):

```
$ renice -15 3598
```

- You can also change the niceness of all a user's processes:

```
$ renice 15 -u mikeb
```

4. Exercise 1

1. Create the following shell script, called `forever`, in your home directory:

```
#!/bin/sh
while [ 1 ]; do
   echo hello... >/dev/null;
done
```

   Make it executable and run it in the background as follows:

```
$ chmod a+rx forever
$ ./forever &
```

2. Use `ps -l` to check the script's nice level
3. Run the script with `nice` and give it a niceness of 15. Try running it alongside a less nice version, and see what the difference is in `top`
4. Try using `nice` or `renice` to make a process' niceness less than 0

Module 11

Advanced Shell Usage

1. More About Quoting

- The shell actually has *three* different quoting mechanisms:
  - Single quotes
  - Backslashes
  - Double quotes

2. Quoting: Single Quotes

- Putting single quotes round something protects it from special interpretation by the shell:

```
$ xmms 'Tom Lehrer - Poisoning Pigeons in the Park.mp3'
$ rm 'b*lls and whistles'
```

- But single quotes (obviously) don't protect single quotes themselves
  - So you can't quote something like `She said, "Don't go."` with only single quotes

3. Quoting: Backslashes

- You can put a backslash \ in front of any single character to turn off its special meaning:

```
$ echo M\&S
$ xmms Suzanne\ Vega\ -\ Tom\'s\ Diner.mp3
$ mail -s C:\\MSDOS.SYS windows-user@example.com
```

4. Quoting: Double Quotes

- Putting double quotes round something protects *most* things within it from interpretation by the shell
  - A dollar sign $ retains its special interpretation
  - As do backticks ‘ ‘
  - ! can’t be escaped in double quotes
- A backslash can be used within double quotes to selectively disable the special interpretation of $, ‘ and \:

```
$ mail -s "C:\\MSDOS.SYS" windows-user@example.com
$ echo "It cost $price US\$"
```

- Putting a backslash in front of anything else gives you *both* characters:

```
$ echo "\*/"
\*/
```

5. Quoting: Combining Quoting Mechanisms

- You can build up an argument for a command by combining several chunks of differently-quoted text
- Just put the chunks next to each other with no intervening whitespace:

```
$ echo "double-quoted"'.single-quoted.'unquoted
double-quoted.single-quoted.unquoted
$ echo 'She said, "Don'\''t go."'
She said, "Don't go."
```

- Rarely needed – the last example is probably better written as:

```
$ echo "She said, \"Don't go.\""
```

6. Recap: Specifying Files with Wildcards

- * in a glob pattern can stand for any sequence of characters:

```
$ ls -l *.txt
-rw-rw-r--    1 fred   users      108 Nov 16 13:06 report.txt
-rw-rw-r--    1 fred   users      345 Jan 18 08:56 notes.txt
```

- * on its own expands to all files in the current directory
- Glob expansion is done by the shell
  - So a program can’t tell when the user ran it with a glob as an argument

7. Globbing Files Within Directories

- You can use globs to get filenames within directories:

```
$ ls Accounts/199*.txt
Accounts/1997.txt  Accounts/1998.txt  Accounts/1999.txt
$ ls ../images/*.gif
../images/logo.gif  ../images/emblem.gif
```

- You can also use globs to expand names of intervening directories:

```
$ cd /usr/man && ls man*/lp*
man1/lpq.1.gz  man1/lprm.1.gz    man4/lp.4.gz   man8/lpd.8.gz
man1/lpr.1.gz  man1/lptest.1.gz  man8/lpc.8.gz
```

8. Globbing to Match a Single Character

- * matches any sequence of characters
- To match any single character, use ?:

```
$ ls ?ouse.txt
```

  Matches *mouse.txt* and *house.txt*, but not *grouse.txt*

- Can be useful for making sure that you only match files of at least a certain length:

```
$ rm ???*.txt
```

  Matches any file ending in *.txt* that has at least three characters before the dot

9. Globbing to Match Certain Characters

- Instead of matching any single character, we can arrange to match any of a given group of characters
- `*.[ch]` matches any file ending in *.c* or *.h*
- `*[0-9].txt` matches any text file with a digit before the dot
- You can use a caret as the first thing in the brackets to match any character that *isn't* one of the listed ones
- `[^a-z]*.jpg` matches any JPEG file that doesn't begin with a lower-case letter
- To match any hidden file except the . and .. directories: `.[^.]*`

10. Generating Filenames: {}

- You can use braces {} to generate filenames:

```
$ mkdir -p Accounts/200{1,2}
$ mkdir Accounts/200{1,2}/{0{1,2,3,4,5,6,7,8,9},1{0,1,2}}
```

- You could even combine those two lines:

```
$ mkdir -p Accounts/200{1,2}/{0{1,2,3,4,5,6,7,8,9},1{0,1,2}}
```

- Or combine brace expansion with quoting:

```
$ echo 'Hello '{world,Mum}\!
Hello world! Hello Mum!
```

- Braces can be used for generating any strings, not just filenames
- Distinctly different from ordinary glob expansion – the words generated don't need to be names of existing files or directories

11. Shell Programming

- The shell is designed to be both:
  - A convenient environment to type commands into
  - A simple programming language
- Any command that can be typed at the command line can be put into a file – and *vice versa*
- Programming features include variables, loops (including `for`), and even shell functions
- The Unix component approach makes it very easy to write shell scripts to perform fairly complex tasks
- Common application domains for shell scripting include:
  - Text processing
  - Automation of system administration tasks

12. Exercise 1

1. Print out the following message: `*** SALE $$$ ***`.
2. Try escaping the same string using single quotes, double quotes and backslashes.
3. Echo the message 'quoting isn't simple', escaping the spaces by putting single quotes around it.
4. Use the glob pattern `.[^.]*` to list all the hidden files in your home directory.
5. To find out what shells are available on your system, list the programs in */bin* whose names end in `sh`.
6. Use `[]` brackets to list all the files in */usr/bin* with names starting with `a`, `b` or `c`.

Module 12

Filesystem Concepts

1. Filesystems

- Some confusion surrounds the use of the term ‘filesystem’
- Commonly used to refer to two distinct concepts
  1. The hierarchy of directories and files which humans use to organise data on a system (‘unified filesystem’)
  2. The formatting system which the kernel uses to store blocks of data on physical media such as disks (‘filesystem *types*’)

2. The Unified Filesystem

- Unix and Linux systems have a **unified filesystem**
  - Any file, on any disk drive or network share, can be accessed through a name beginning with /
- The unified filesystem is made up of one or more **individual filesystems** (‘branches’ of the unified hierarchy)
  - Each individual filesystem has its own root
  - That root can be grafted onto any directory in the unified filesystem
  - The directory where an individual filesystem is grafted into the unified filesystem is the individual filesystem’s **mount point**
- An individual filesystem lives on a physical device (such as a disk drive), though not necessarily on the same computer

3. File Types

- Files directly contain data
- Directories provide a hierarchy of files: they can contain both files and other directories
- Files and directories are both **file types**
- Other file types exist, including **device special files**:
  - Device files provide a way of asking the kernel for access to a given physical device
  - The data that the device file seems to contain is actually the raw sequence of bytes or sectors on the device itself
  - Device files are by convention stored under the */dev* directory

4. Inodes and Directories

- An **inode** is the data structure that describes a file on an individual filesystem
- It contains information about the file, including its type (file/directory/device), size, modification time, permissions, etc.
- You can regard an inode as being the file itself
- The inodes within an individual filesystem are numbered
  - An inode number is sometimes called an ‘inum’
- Note that a file’s name is stored not in its inode, but in a directory
  - A directory is stored on disk as a list of file and directory names
  - Each name has an inode number associated with it
  - Separating names from inodes means that you can have multiple directory entries referring to the same file

Module 13

Create and Change Hard and Symbolic Links

1. Symbolic Links

- A **symbolic link** (or **symlink**) is a pseudo-file which behaves as an alternative name for some other file or directory
- The ‘contents’ of the symlink are the real name pointed to
- When you try to use a file name including a symlink, the kernel replaces the symlink component with its ‘contents’ and starts again
- Symlinks allow you to keep a file (or directory) in one place, but pretend it lives in another
  - For example, to ensure that an obsolete name continues to work for older software
  - Or to spread data from a single filesystem hierarchy over multiple disk partitions

2. Examining and Creating Symbolic Links

- `ls -l` shows where a symbolic link points to:

```
$ ls -l /usr/tmp
lrwxrwxrwx  1 root root 30 Sep 26  2000 /usr/tmp -> /var/tmp
```

- `ls` can also be made to list symlinks in a different colour to other files, or to suffix their names with ‘@’
- A symlink is created with the `ln -s` command
- Its syntax is similar to `cp` – the original name comes first, then the name you want to create:

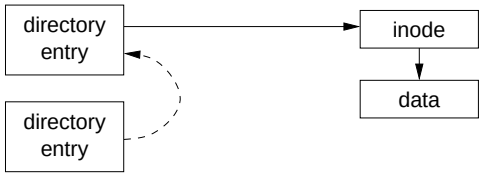
```
$ ln -s real-file file-link
$ ln -s real-dir dir-link
$ ls -l file-link dir-link
lrwxrwxrwx  1 bob  bob   9 Jan 11 15:22 file-link -> real-file
lrwxrwxrwx  1 bob  bob   8 Jan 11 15:22 dir-link -> real-dir
```

3. Hard Links

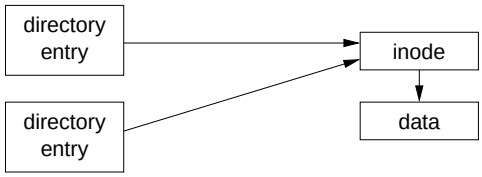
- Where symlinks refer to other files by name, a **hard link** refers to another file by inode number
  - An inode is the data structure that describes a file on disk
  - It contains information about the file, including its type (file/directory/device), modification time, permissions, etc.
- A directory entry contains a name and an inode number
  - So a file’s name is not considered to be part of the file itself
- You get a hard link when different directory entries on a filesystem refer to the same inode number

4. Symlinks and Hard Links Illustrated

- A symbolic link refers to filename, which in turn refers to an inode:



- A hard link is a normal directory entry, referring directly to an inode:



5. Comparing Symlinks and Hard Links

Symlinks	Hard links
Symlinks are distinctly different from normal files, so we can distinguish a symlink from the original it points to	Multiple hard-link style names for the same file are indistinguishable; the term ‘hard link’ is merely conventional
Symlinks can point to any type of file (normal file, directory, device file, symlink, etc.)	Hard links may not point to a directory (or, on some non-Linux systems, to a symlink)
Symlinks refer to names, so they can point to files on other filesystems	Hard links work by inode number, so they can only work within a single filesystem
Conversely, if you rename or delete the original file pointed to by a symlink, the symlink gets broken	Renaming or deleting the ‘original’ file pointed to by a hard link has no effect on the hard link
Symlinks may take up additional disk space (to store the name pointed to)	Hard links only need as much disk space as a directory entry

6. Examining and Creating Hard Links

- Use the `ln` command to create a hard link
- Don’t use the `-s` option when creating hard links
- As when creating symlinks, the order of the arguments to `ln` mimics `cp`:

```
$ ls -l *.dtd
-rw-r--r--    1 anna   anna    11170 Dec  9 14:11 module.dtd
$ ln module.dtd chapter.dtd
$ ls -l *.dtd
-rw-r--r--    2 anna   anna    11170 Dec  9 14:11 chapter.dtd
-rw-r--r--    2 anna   anna    11170 Dec  9 14:11 module.dtd
```

- Notice that the link count in the listing increases to 2
- The two names are now indistinguishable
  - Deleting or renaming one doesn’t affect the other

7. Preserving Links

- Commands that operate on files often take options to specify whether links are followed
- The `tar` command notices when two files it's archiving are hard links to each other, and stores that fact correctly
- By default `tar` also stores symlinks in archives
  - Use the `-h` option (`--dereference`) to instead store the file pointed to
- The `cp` command by default ignores both hard links and symlinks
  - Use the `-d` option (`--no-dereference`) to preserve all links
  - Use the `-R` option (`--recursive`) when copying recursively to ensure that symlinks are preserved
  - The `-a` option (`--archive`) implies both `-d` and `-R`

8. Finding Symbolic Links to a File

- The `find` command has a `-lname` option which searches for symbolic links containing some text:

```
$ find / -lname '*file' -printf '%p -> %l\n'
```

- This command prints the names and destinations of all symbolic links whose destination ends in `file`
- Be aware that running `find` over the entire filesystem is very disk-intensive!

9. Finding Hard Links to a File

- Hard links can be found by searching for directory entries with a given inode number
- First, identify the filesystem and inode number of the file you're interested in:

```
$ df module.dtd
Filesystem         1k-blocks      Used Available Use% Mounted on
/dev/sdb3           13647416   5241196   7712972  40% /home
$ ls -i module.dtd
 245713 module.dtd
```

- Then use `find`'s `-inum` option to look for directory entries in that filesystem with that inode number:

```
$ find /home -xdev -inum 245713
```

- The `-xdev` option prevents `find` from recursing down into other filesystems

10. Exercise 1

1. Make a temporary directory and change into it.
2. Make some test files as follows:

```
$ echo "oranges and lemons" > fruit
$ echo spuds > veg
```

3. Make a symbolic link called *starch* to the *veg* file.
4. Make a hard link called *citrus* to the appropriate file, and check that it has the same inode number.
5. Delete the original *fruit* file and check that *citrus* still contains the text.
6. Delete the original *veg* file and try to look at the contents of *starch*. Use `ls` to check the symlink.

11. Exercise 2

1. Try to see what the following loop does, and then create some *.htm* files and try it:

```
$ for htm in *.htm; do
>    ln -s $htm ${htm}l;
> done
```

2. Make a symlink called *dir* to a directory (such as */etc*).
3. Try the following commands to display the link and compare the results:

```
$ ls -l dir
$ ls -l dir/
```

Module 14

Manage File Ownership

1. Users and Groups

- Anyone using a Linux computer is a **user**
- The system keeps track of different users, by username
  - Security features allow different users to have different privileges
- Users can belong to **groups**, allowing security to be managed for collections of people with different requirements
- Use `su` to switch to a different user
  - Quicker than logging off and back on again
- `su` prompts you for the user's password:

```
$ su - bob
Password:
```

  The `-` option makes `su` behave as if you've logged in as that user

2. The Superuser: Root

- Every Linux system has a user called 'root'
- The root user is all-powerful
  - Can access any files
- The root user account should only be used for system administration, such as installing software
- When logged in as root, the shell prompt usually ends in `#`
- Usually best to use `su` for working as root:

```
$ whoami
fred
$ su -
Password:
# whoami
root
```

3. Changing File Ownership with chown

- The `chown` command changes the ownership of files or directories
- Simple usage:

```
# chown aaronc logfile.txt
```

- Makes *logfile.txt* be owned by the user `aaronc`
- Specify any number of files or directories
- Only the superuser can change the ownership of a file
  - This is a security feature – quotas, set-uid

4. Changing File Group Ownership with `chgrp`

- The `chgrp` command changes the group ownership of files or directories
- Simple usage:

```
# chgrp staff report.txt
```

- Makes `staff` be the group owner of the file *logfile.txt*
- As for `chown`, specify any number of files or directories
- The superuser may change the group ownership of any file to any group
- The owner of a file may change its group ownership
  - But only to a group of which the owner is a member

5. Changing the Ownership of a Directory and Its Contents

- A common requirement is to change the ownership of a directory and its contents
- Both `chown` and `chgrp` accept a `-R` option:

```
# chgrp -R staff shared-directory
```

- Mnemonic: ‘recursive’
- Changes the group ownership of *shared-directory* to `staff`
  - And its contents
  - And its subdirectories, recursively
- Changing user ownership (superuser only):

```
# chown -R root /usr/local/share/misc/
```

6. Changing Ownership and Group Ownership Simultaneously

- The `chown` command can change the user-owner and group-owner of a file simultaneously:

```
# chown aaronc:www-docs public_html/interesting.html
```

- Changes the user owner to `aaronc` and the group owner to `www-docs`
- Can use the `-R` option as normal
- A dot (`.`) may be used instead of a colon:

```
# chown -R aaronc.www-docs /www/intranet/people/aaronc/
```

7. Exercise 1

1. Find out who owns the file */bin/ls* and who owns your home directory (in */home*).
2. Log on as root, and create an empty file with `touch`. The user and group owners should be ‘root’ – check with `ls`.
3. Change the owner of the file to be ‘users’.
4. Change the group owner to be any non-root user.
5. Change both of the owners back to being ‘root’ with a single command.

Module 15

Use File Permissions to Control Access to Files

1. Basic Concepts: Permissions on Files

- Three types of permissions on files, each denoted by a letter
- A permission represents an action that can be done on the file:

Permission	Letter	Description
Read	`r`	Permission to read the data stored in the file
Write	`w`	Permission to write new data to the file, to truncate the file, or to overwrite existing data
Execute	`x`	Permission to attempt to execute the contents of the file as a program

- Occasionally referred to as ‘permission bits’
- Note that for scripts, you need both execute permission *and* read permission
  - The script interpreter (which runs with your permissions) needs to be able to read the script from the file

2. Basic Concepts: Permissions on Directories

- The `r`, `w`, `x` permissions also have a meaning for directories
- The meanings for directories are slightly different:

Permission	Letter	Description
Read	`r`	Permission to get a listing of the directory
Write	`w`	Permission to create, delete, or rename files (or subdirectories) within the directory
Execute	`x`	Permission to change to the directory, or to use the directory as an intermediate part of a path to a file

- The difference between read and execute on directories is specious – having one but not the other is almost never what you want

3. Basic Concepts: Permissions for Different Groups of People

- As well as having different types of permission, we can apply different sets of permissions to different sets of people
- A file (or directory) has an **owner** and a **group owner**
- The `r`, `w`, `x` permissions are specified separately for the owner, for the group owner, and for everyone else (the ‘world’)

4. Examining Permissions: `ls -l`

- The `ls -l` command allows you to look at the permissions on a file:

```
$ ls -l
drwxr-x---    9 aaronc   staff      4096 Oct 12 12:57 accounts
-rw-rw-r--    1 aaronc   staff     11170 Dec  9 14:11 report.txt
```

- The third and fourth columns are the owner and group-owner
- The first column is the permissions:
  - One character for the file type: `d` for directories, `-` for plain files
  - Three characters of `rwx` permissions for the owner (or a dash if the permission isn't available)
  - Three characters of `rwx` permissions for the group owner
  - Three characters of `rwx` permissions for everyone else

5. Preserving Permissions When Copying Files

- By default, the `cp` command makes no attempt to preserve permissions (and other attributes like timestamps)
- You can use the `-p` option to preserve permissions and timestamps:

```
$ cp -p important.txt important.txt.orig
```

- Alternatively, the `-a` option preserves all information possible, including permissions and timestamps

6. How Permissions are Applied

- If you own a file, the per-owner permissions apply to you
- Otherwise, if you are in the group that group-owns the file, the per-group permissions apply to you
- If neither of those is the case, the for-everyone-else permissions apply to you

7. Changing File and Directory Permissions: chmod

- The `chmod` command changes the permissions of a file or directory
  - A file's permissions may be changed only by its owner or by the superuser
- `chmod` takes an argument describing the new permissions
  - Can be specified in many flexible (but correspondingly complex) ways
- Simple example:

```
$ chmod a+x new-program
```

adds (+) executable permission (`x`) for all users (`a`) on the file *new-program*

8. Specifying Permissions for chmod

- Permissions can be set using letters in the following format:

```
[ugoa][+=-][rwxX]
```

- The first letters indicate who to set permissions for:
  - `u` for the file's owner, `g` for the group owner, `o` for other users, or `a` for all users
- = sets permissions for files, + adds permissions to those already set, and - removes permissions
- The final letters indicate which of the `r`, `w`, `x` permissions to set
  - Or use capital `X` to set the `x` permission, but only for directories and already-executable files

9. Changing the Permissions of a Directory and Its Contents

- A common requirement is to change the permissions of a directory and its contents
- `chmod` accepts a `-R` option:

```
$ chmod -R g+rwX,o+rX public-directory
```

- Mnemonic: 'recursive'
- Adds `rwx` permissions on *public-directory* for the group owner, and adds `rx` permissions on it for everyone else
  - And any subdirectories, recursively
  - Any any contained executable files
  - Contained non-executable files have `rw` permissions added for the group owner, and `r` permission for everyone else

10. Special Directory Permissions: ‘Sticky’

- The */tmp* directory must be world-writable, so that anyone may create temporary files within it
- But that would normally mean that anyone may delete *any* files within it – obviously a security hole
- A directory may have ‘sticky’ permission:
  - Only a file’s owner may delete it from a sticky directory
- Expressed with a `t` (mnemonic: *t*emporary directory) in a listing:

```
$ ls -l -d /tmp
drwxrwxrwt   30 root     root      11264 Dec 21 09:35 /tmp
```

- Enable ‘sticky’ permission with:

```
# chmod +t /data/tmp
```

11. Special Directory Permissions: Setgid

- If a directory is **setgid** (‘set group-id’), files created within it acquire the group ownership of the directory
  - And directories created within it acquire both the group ownership *and* setgid permission
- Useful for a shared directory where all users working on its files are in a given group
- Expressed with an `s` in ‘group’ position in a listing:

```
$ ls -l -d /data/projects
drwxrwsr-x   16 root   staff    4096 Oct 19 13:14 /data/projects
```

- Enable setgid with:

```
# chmod g+s /data/projects
```

12. Special File Permissions: Setgid

- Setgid permission may also be applied to executable files
- A process run from a setgid file acquires the group id of the file
- Note: Linux doesn’t directly allow scripts to be setgid – only compiled programs
- Useful if you want a program to be able to (for example) edit some files that have a given group owner
  - Without letting individual users access those files directly

13. Special File Permissions: Setuid

- Files may also have a **setuid** ('set user-id') permission
- Equivalent to setgid: a process run from a setuid file acquires the user id of the file
- As with setgid, Linux doesn't allow scripts to be setuid
- Expressed with an `s` in 'user' position in a listing:

```
$ ls -l /usr/bin/passwd
-r-s--x--x   1 root   root   12244 Feb  7  2000 /usr/bin/passwd
```

- Enable setuid with:

```
# chmod u+s /usr/local/bin/program
```

14. Displaying Unusual Permissions

- Use `ls -l` to display file permissions
  - Setuid and Setgid permissions are shown by an `s` in the user and group execute positions
  - The sticky bit is shown by a `t` in the 'other' execute position
- The letters `s` and `t` cover up the execute bits
  - But you can still tell whether the execute bits are set
  - Lowercase `s` or `t` indicates that execute is enabled (i.e., there is an `x` behind the letter)
  - Uppercase `S` or `T` indicates that execute is disabled (there is a `-` behind the letter)

15. Permissions as Numbers

- Sometimes you will find numbers referring to sets of permissions
- Calculate the number by adding one or more of the following together:

4000	Setuid	40	Readable by group owner
2000	Setgid	20	Writable by group owner
1000	'Sticky'	10	Executable by group owner
400	Readable by owner	4	Readable by anyone
200	Writable by owner	2	Writable by anyone
100	Executable by owner	1	Executable by anyone

- You may use numerical permissions with `chmod`:

```
$ chmod 664 *.txt
```

is equivalent to:

```
$ chmod ug=rw,o=r *.txt
```

16. Default Permissions: `umask`

- The `umask` command allows you to affect the default permissions on files and directories you create:

```
$ umask 002
```

- The argument is calculated by adding together the numeric values for the `rwx` permissions you *don't* want on new files and directories
  - This example has just 2 – avoid world-writable, but turn everything else on
- Other common `umask` values:
  - `022` – avoid world- and group-writable, allow everything else
  - `027` – avoid group-writable, and allow no permissions for anyone else
- You normally want to put a call to `umask` in your shell's startup file

17. Exercise 1

1. Find out what permissions are set on your home directory (as a normal user). Can other users access files inside it?
2. If your home directory is only accessible to you, then change the permissions to allow other people to read files inside it, otherwise change it so that they can't.
3. Check the permissions on */bin* and */bin/ls* and satisfy yourself that they are reasonable.
4. Check the permissions available on */etc/passwd* and */etc/shadow*.
5. Write one command which would allow people to browse through your home directory and any subdirectories inside it and read all the files.

Module 16

Create Partitions and Filesystems

1. Concepts: Disks and Partitions

- A hard disk provides a single large storage space
- Usually split into **partitions**
  - Information about partitions is stored in the **partition table**
  - Linux defaults to using partition tables compatible with *Microsoft Windows*
  - For compatibility with *Windows*, at most four primary partitions can be made
  - But they can be **extended partitions**, which can themselves be split into smaller **logical partitions**
    - Extended partitions have their own partition table to store information about logical partitions

2. Disk Naming

- The device files for IDE hard drives are */dev/hda* to */dev/hdd*
  - *hda* and *hdb* are the drives on the first IDE channel, *hdc* and *hdd* the ones on the second channel
  - The first drive on each channel is the IDE 'master', and the second is the IDE 'slave'
- Primary partitions are numbered from 1–4
- Logical partitions are numbered from 5
- The devices */dev/hda*, etc., refer to whole hard disks, not partitions
  - Add the partition number to refer to a specific partition
  - For example, */dev/hda1* is the first partition on the first IDE disk
- SCSI disks are named */dev/sda*, */dev/sdb*, etc

3. Using `fdisk`

- The `fdisk` command is used to create, delete and change the partitions on a disk
- Give `fdisk` the name of the disk to edit, for example:

```
# fdisk /dev/hda
```

- `fdisk` reads one-letter commands from the user
  - Type `m` to get a list of commands
  - Use `p` to show what partitions currently exist
  - Use `q` to quit without altering anything
  - Use `w` to quit and write the changes
    - Use with caution, and triple-check what you're doing!

4. Making New Partitions

- Create new partitions with the `n` command
  - Choose whether to make a primary, extended or logical partition
  - Choose which number to assign it
- `fdisk` asks where to put the start and end of the partition
  - The default values make the partition as big as possible
  - The desired size can be specified in megabytes, e.g., `+250M`
- Changes to the partition table are only written when the `w` command is given

5. Changing Partition Types

- Each partition has a type code, which is a number
- The `fdisk` command `l` shows a list of known types
- The command `t` changes the type of an existing partition
  - Enter the type code at the prompt
- Linux partitions are usually of type ‘Linux native’ (type 83)
- Other operating systems might use other types of partition, many of which can be understood by Linux

6. Making Filesystems with `mkfs`

- The `mkfs` command initializes a filesystem on a new partition
  - Warning: any old data on the partition will be lost
  - For example, to make an `ext2` filesystem on `/dev/hda2`:

    ```
    # mkfs -t ext2 -c /dev/hda2
    ```

  - `-t` sets the filesystem type to make, and `-c` checks for bad blocks on the disk
- `mkfs` uses other programs to make specific types of filesystem, such as `mke2fs` and `mkdosfs`

7. Useful Websites

- Tutorial on making partitions – http://www.linuxnewbie.org/nhf/intel/installation/fdisk_nhf/Fdisk.html
- Linux Partition HOWTO – http://www.linuxdoc.org/HOWTO/mini/Partition/
- Table of `fdisk` commands and partition types – http://wwwinfo.cern.ch/pdp/as/linux/fdisk/index.html