Linux System Administration

http://training.gbdirect.co.uk/

tel: 0870 200 7273

Overview

Contents

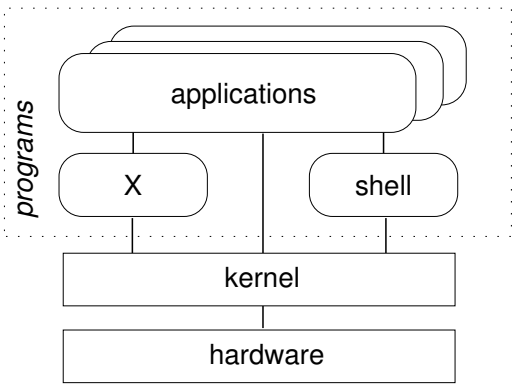
Module 1

Introduction

1.1 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

1.2 Unix System Architecture



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

1.3 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

1.4 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 important parts come from the **GNU** project
    - Free Software Foundation's project to make a free Unix
    - 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

1.5 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

1.6 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

1.7 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

1.8 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
```


1.9 Files

- Data can be stored in a **file**
- Each file has a **filename**
  - A label referring 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

1.10 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

1.11 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

1.12 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

1.13 Unix Command Feedback

- Typically, succesful commands do not give any output
- Messages are displayed in the case of errors
- The `rm` command 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

1.14 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

1.15 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'

1.16 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

1.17 Exercises

1. a. Log in.
   b. Log out.
   c. Log in again. Open a terminal window, to start a shell.
   d. Exit from the shell; the terminal window will close.
   e. Start another shell. Enter each of the following commands in turn.
      - `date`
      - `whoami`
      - `hostname`
      - `uname`

- `uptime`

2. **a.** Use the `ls` command to see if you have any files.

   b. 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.

   c. Use `ls` again to verify that the new file exists.

   d. Display the contents of the file.

   e. Display the file again, but use the cursor keys to execute the same command again without having to retype it.

3. **a.** Create a second file. Call it *secret-of-the-universe*, and put in whatever content you deem appropriate.

   b. Check its creation with `ls`.

   c. 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.

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

   a. Copy *secret-of-the-universe* to a new file called *answer.txt*. Use `Tab` to avoid typing the existing file's name in full.

   b. Now copy *hello.txt* to *answer.txt*. What's happened now?

   c. Delete the original file, *hello.txt*.

   d. Rename *answer.txt* to *message*.

   e. Try asking `rm` to delete a file called *missing*. What happens?

   f. Try copying *secret-of-the-universe* again, but don't specify a filename to which to copy. What happens now?

Module 2

Getting Started

2.1 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 slashes (`/`)
- If a path refers to a directory it can end in /
  - Usually an extra slash at the end of a path makes no difference

2.2 Examples of Absolute Paths

- An **absolute path** starts at the root of the directory hierarchy, and names directories under it:

  `/etc/hostname`
  - Meaning the file called *hostname* in the directory *etc* in the root directory
- We can use `ls` to list files in a specific directory by specifying the absolute path:

  `$ ls /usr/share/doc/`

2.3 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

2.4 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

2.5 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

2.6 Special Dot Directories

- Every directory contains two special filenames which help making relative paths:
  - The directory `..` points to the parent directory
    - `ls ..` will list the files in the parent directory
  - 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`

2.7 Using Dot Directories in Paths

- The special `..` and `.` directories can be used in paths just like any other directory name:

```
$ cd ../other-dir/
```

  - Meaning “the directory *other-dir* in the parent directory of the current directory”
- It is common to see `..` used to ‘go back’ several directories from the current directory:

```
$ ls ../../../../far-away-directory/
```

- The `.` directory is most commonly used on its own, to mean “the current directory”

2.8 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 considered ‘hidden’
- Make `ls` display all files, even the hidden ones, by giving it the `-a` (all) option:

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

- Hidden files are often used for configuration files
  - Usually found in a user’s home directory
- You can still read hidden files — they just don’t get listed by `ls` by default

2.9 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
```

2.10 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

2.11 Running Programs

- Programs under Linux are files, stored in directories like */bin* and */usr/bin*
  - Run them from the shell, simply by typing their name
- Many programs take options, which are added after their name and prefixed with -
- For example, the `-l` option to `ls` gives more information, including the size of files and the date they were last modified:

```
$ 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

2.12 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 (such as `*`), put single quotes around them:

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

2.13 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 it accepts
  - Other information, such as the name of the author

2.14 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 *
```

  - (All the files, that is, except the hidden ones)

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

2.15 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
```

2.16 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 into 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

2.17 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

2.18 Exercises

1. **a.** Use the `pwd` command to find out what directory you are in.
   b. If you are not in your home directory (*/home/USERNAME*) then use `cd` without any arguments to go there, and do `pwd` again.
   c. Use `cd` to visit the root directory, and list the files there. You should see *home* among the list.
   d. 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).
   e. Change into your home directory to confirm that you have gotten back to where you started.
2. **a.** Create a text file in your home directory called *shakespeare*, containing the following text:

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

   b. Rename it to *sonnet-18.txt*.
   c. Make a new directory in your home directory, called *poetry*.
   d. Move the poem file into the new directory.
   e. 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.
   f. Find a text editor program and use it to display and edit the sonnet.
3. **a.** From your home directory, list the files in the directory */usr/share*.
   b. 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 *doc*.
   c. Next list the files in the parent directory, and the directory above that.
   d. Try the following command, and make sure you understand the result:

   ```
   $ echo ~
   ```

   e. 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.

4. **a.** Use the `hostname` command, with no options, to print the hostname of the machine you are using.

   b. 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.

   c. Use the `locate` command to find files whose name contains the text 'hostname'. Which of the filenames printed contain the actual `hostname` program itself? Try running it by entering the program's absolute path to check that you really have found it.

5. **a.** 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)

   b. Use quoting to make `echo` print out an actual `*` symbol.

   c. Augment the *poetry* directory you created earlier with another file, *sonnet-29.txt*:

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

   d. Use the `cat` command to display both of the poems, using a wildcard.

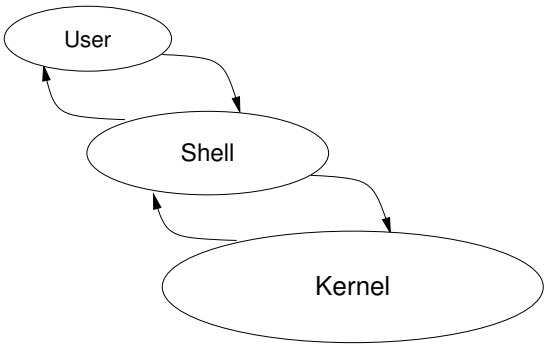
   e. Finally, use the `rm` command to delete the *poetry* directory and the poems in it.

Module 3

Work Effectively on the Unix Command Line

3.1 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 or open a terminal



3.2 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
- Although this course concentrates on Bash, the shell `tcsh` is also popular
  - Based on the design of the older C Shell (`csh`)

3.3 Shell Commands

- Shell 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
- For several reasons, some commands are built into the shell itself
  - Called **builtins**
  - Only a small number of commands are builtins, most are separate programs

3.4 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 `-`

3.5 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 `--`

3.6 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
```

3.7 Setting Shell Variables

- **Shell variables** can be used to store temporary values
- Set a shell variable's value as follows:

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

  - The double quotes are needed because the value contains a space
  - Easiest to put them in all the time
- Print out the value of a shell variable with the `echo` command:

```
$ echo $files
```

  - The dollar (`$`) tells the shell to insert the variable's value into the command line
- Use the `set` command (with no arguments) to list all the shell variables

3.8 Environment Variables

- Shell variables are private to the shell
- A special type of shell variables called **environment variables** are passed to programs run from the shell
- A program's **environment** is the set of environment variables it can access
  - In Bash, use `export` to export a shell variable into the environment:

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

  - Or combine those into one line:

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

- The `env` command lists environment variables

3.9 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
    ```

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

3.10 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`

3.11 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 built-in 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

3.12 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
```

3.13 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
- A command of the form `^`*`string`*`^`*`replacement`*`^` replaces the first occurrence of *`string`* with *`replacement`* in the previous command, and runs it:

```
$ echo $HOTSNAME

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


3.14 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')

3.15 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 succeeded:

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

3.16 Repeating Commands with `for`

- Commands can be repeated several times using `for`
  - Structure: `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

3.17 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
- Line breaks in the output are converted to spaces
- Another example: use `vi` to edit the last of the files found:

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

3.18 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

3.19 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
- Structure: `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

3.20 `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'

3.21 `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

3.22 Exercises

1. **a.** Use the `df` command to display the amount of used and available space on your hard drive.

   b. 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.

   c. 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` command.

2. **a.** Try `ls` with the `-a` and `-A` options. What is the difference between them?

   b. 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.

   c. 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.

   d. 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?

3. **a.** 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.

   b. 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.

   c. Find out what the `find` criterion `-iname` does.

Module 4

Process Text Streams Using Text Processing Filters

4.1 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

4.2 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

4.3 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
```

4.4 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`)

4.5 Counting Words and Lines with `wc`

- `wc` counts 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
```


4.6 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

4.7 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

4.8 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 omitted
  - The first character or field is numbered 1, not 0
- Example: select usernames of logged in users:

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


4.9 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:

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

4.10 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
```

4.11 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
```


4.12 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
```

4.13 Numbering Lines of a File with `nl` or `cat`

- Display the input with line numbers against each line
- 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

4.14 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`

4.15 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
```

4.16 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_
```

4.17 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
```

4.18 Reversing Files with `tac`

- 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
```


4.19 Translating Sets of Characters with `tr`

- `tr` translates one set of characters to another
- Usage: `tr` *`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)

4.20 `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
```

4.21 Modifying Files with `sed`

- `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
```

4.22 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

4.23 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
```

4.24 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

4.25 Exercises

1. **a.** 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.)

   b. Arrange for the list of usernames in `who`'s output to be sorted, and remove any duplicates.

   c. 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`?

   d. 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?

   e. Use `nl` to number the lines in the output of the previous question.

2. **a.** 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
   ```

   b. 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`.

   c. 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.

   d. 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.

3. **a.** 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.

b. 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:

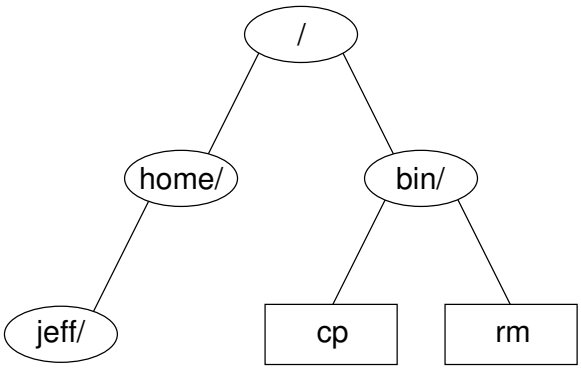
```
$ chmod a+rx my_ls
```


Module 5

Perform Basic File Management

5.1 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 `/`



5.2 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/share/dict/words
/home/jeff/recipe
```

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

5.3 File Extensions

- It's common to put an **extension**, beginning with a dot, on the end of a filename
- The extension can indicate 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 just a convention
  - The kernel just treats them as a normal part of the name
  - A few programs use extensions to determine the type of a file

5.4 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

5.5 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

5.6 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

5.7 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

5.8 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/
```


5.9 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/
```

5.10 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/*
```

5.11 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

5.12 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
```

5.13 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
```

5.14 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
```


5.15 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
```

5.16 Exercises

1. a. Use `cd` to go to your home directory, and create a new directory there called *dog*.
   b. Create another directory within that one called *cat*, and another within that called *mouse*.
   c. Remove all three directories. You can either remove them one at a time, or all at once.
   d. If you can delete directories with `rm -r`, what is the point of using `rmdir` for empty directories?
   e. Try creating the *dog/cat/mouse* directory structure with a single command.
2. a. Copy the file */etc/passwd* to your home directory, and then use `cat` to see what's in it.
   b. Rename it to *users* using the `mv` command.
   c. Make a directory called *programs* and copy everything from */bin* into it.
   d. Delete all the files in the *programs* directory.
   e. Delete the empty *programs* directory and the *users* file.
3. a. The `touch` command can be used to create new empty files. Try that now, picking a name for the new file:

   ```
   $ touch baked-beans
   ```

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

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

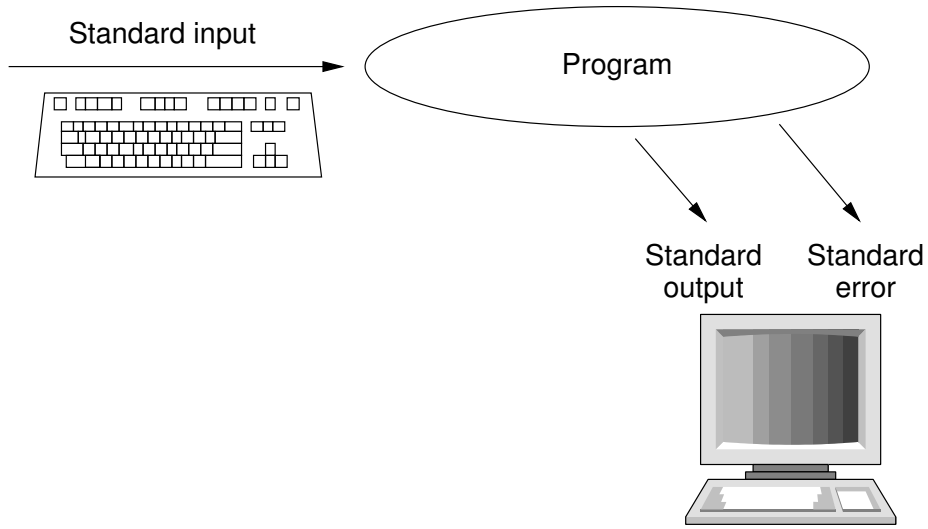
   c. Wait for a minute, and then try the previous two steps again, and see what changes. What happens when we don't specify a time to `touch`?
   d. Try setting the timestamp on the file to a value in the future.
   e. When you're finished with it, delete the file.

Module 6

Use Unix Streams, Pipes and Redirects

6.1 Standard Files

- Processes are connected to three standard files



- Many programs open other files as well

6.2 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

6.3 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

6.4 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

6.5 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
```


6.6 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
```

6.7 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

6.8 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

6.9 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

6.10 Running Programs with `xargs`

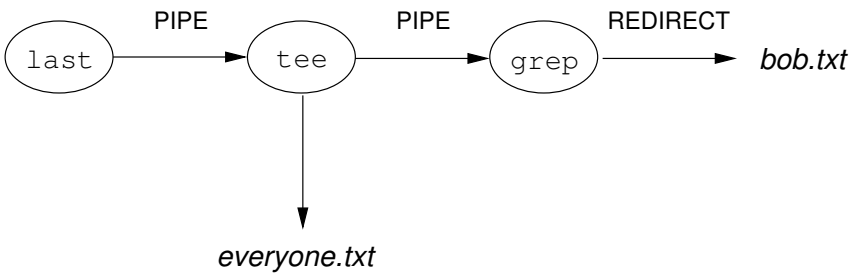
- `xargs` reads pieces of text and runs another program with them as its arguments
  - Usually its input is a list of filenames to give to a file processing program
- Syntax: `xargs` *`command`* `[`*`initial args`*`]`
- Use `-l` *n* to use *n* items each time the command is run
  - The default is 1
- `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
  ```

6.11 `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
  ```



6.12 Exercises

1. **a.** Try the example on the 'Pipes' slide, using `rev` to reverse some text.

   b. Try replacing the `echo` command with some other commands which produce output (e.g., `whoami`).

   c. What happens when you replace `rev` with `cat`? You might like to try running `cat` with no arguments and entering some text.

2. **a.** 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.

   b. 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

7.1 Searching Files with `grep`

- `grep` prints lines from files which match a pattern
- For example, to find an entry 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` rev*e*rses the test, printing lines which don't match

7.2 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

7.3 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

7.4 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'

7.5 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 `+`

7.6 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

7.7 Further Reading

- `man 7 regex`
- *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 Sed user manual (1978), http://www.urc.bl.ac.yu/manuals/progunix/sed.txt

7.8 Exercises

1. **a.** Use `grep` to find information about the HTTP protocol in the file */etc/services*.

   b. 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`.

   c. Add another use of `grep -v` to your pipeline to remove blank lines (which match the pattern `^$`).

   d. 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

8.1 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`

8.2 `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

8.3 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

8.4 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

8.5 Exercises

1. a. Start a process by running `man bash` and suspend it with `Ctrl+Z`.
   b. Run `xclock` in the background, using `&`.
   c. Use `jobs` to list the backgrounded and stopped processes.
   d. Use the `fg` command to bring `man` into the foreground, and quit from it as normal.
   e. Use `fg` to foreground `xclock`, and terminate it with `Ctrl+C`.
   f. 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

9.1 What is a Process?

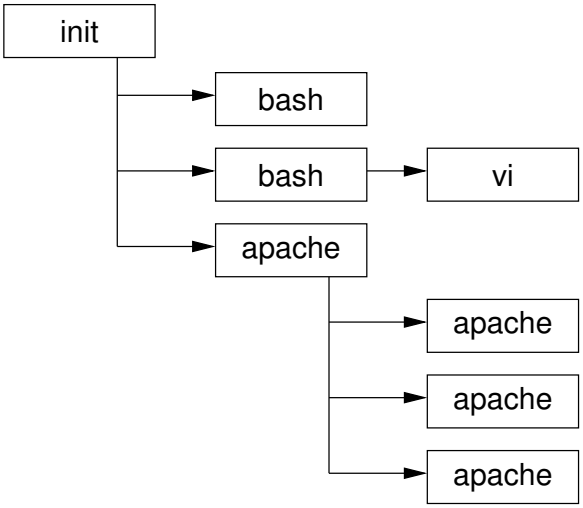
- The kernel considers each program running on your system to be a **process**
- A process ‘lives’ as it executes, with a lifetime that may be short or long
- A process is said to ‘die’ when it terminates
- The kernel identifies each process by a number known as a process id, or **pid**
- The kernel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

9.2 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

9.3 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)

9.4 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

9.5 `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*

9.6 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

9.7 `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

9.8 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

9.9 `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

9.10 `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)

9.11 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

9.12 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.

9.13 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

9.14 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

9.15 Exercises

1. **a.** Use `top` to show the processes running on your machine.

b. Make `top` sort by memory usage, so that the most memory-hungry processes appear at the top.

c. Restrict the display to show only processes owned by you.

d. Try killing one of your processes (make sure it's nothing important).

e. Display a list of all the processes running on the machine using `ps` (displaying the full command line for them).

f. Get the same listing as a tree, using both `ps` and `pstree`.

g. Have `ps` sort the output by system time used.

Module 10

Modify Process Execution Priorities

10.1 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`

10.2 `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 &
```

10.3 `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
```

10.4 Exercises

1. **a.** 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 &
```

   b. Use `ps -l` to check the script's nice level

   c. 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`

   d. Try using `nice` or `renice` to make a process' niceness less than 0

Module 11

Advanced Shell Usage

11.1 More About Quoting

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

11.2 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

11.3 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
```

11.4 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 "\*/"
\*/
```

11.5 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.\""
```

11.6 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

11.7 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
```

11.8 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

11.9 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: `.[^.]*`

11.10 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

11.11 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

11.12 Exercises

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

Module 12

Filesystem Concepts

12.1 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*’)

12.2 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

12.3 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

12.4 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

13.1 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

13.2 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:

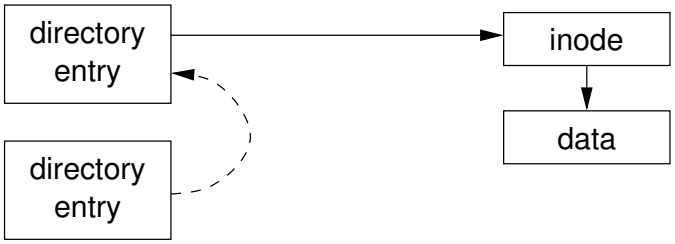
```
$ ln -s real-file file-link
$ ln -s real-dir dir-link
$ ls -l file-link dir-link
lrwxrwxrwx  1 bob  bob   9 Jan 11 15:22 file-link -> real-file
lrwxrwxrwx  1 bob  bob   8 Jan 11 15:22 dir-link -> real-dir
```

13.3 Hard Links

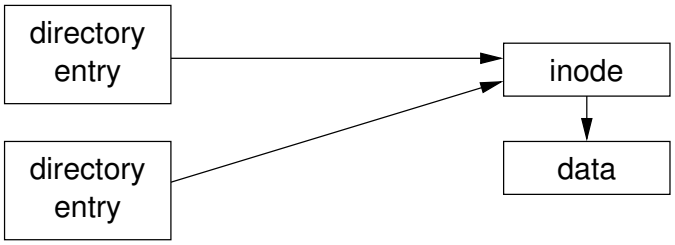
- Where symlinks refer to other files by name, a **hard link** refers to another file by inode number
  - An inode is the data structure that describes a file on disk
  - It contains information about the file, including its type (file/directory/device), modification time, permissions, etc.
- A directory 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

13.4 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:



13.5 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

13.6 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

13.7 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`

13.8 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!

13.9 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

13.10 Exercises

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

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

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

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

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


Module 14

Manage File Ownership

14.1 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

14.2 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
```

14.3 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

14.4 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

14.5 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/
```


14.6 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/
```

14.7 Exercises

1. **a.** Find out who owns the file */bin/ls* and who owns your home directory (in */home*).
   b. Log on as root, and create an empty file with `touch`. The user and group owners should be 'root' — check with `ls`.
   c. Change the owner of the file to be 'users'.
   d. Change the group owner to be any non-root user.
   e. Change both of the owners back to being 'root' with a single command.

Module 15

Use File Permissions to Control Access to Files

15.1 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

15.2 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

15.3 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')

15.4 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

15.5 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

15.6 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

15.7 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 *a*ll users (`a`) on the file *new-program*

15.8 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

15.9 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

15.10 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
```

15.11 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
```

15.12 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

15.13 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
```

15.14 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)

15.15 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
```


15.16 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

15.17 Exercises

1. **a.** Find out what permissions are set on your home directory (as a normal user). Can other users access files inside it?

   b. 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.

   c. Check the permissions on */bin* and */bin/ls* and satisfy yourself that they are reasonable.

   d. Check the permissions available on */etc/passwd* and */etc/shadow*.

   e. 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

16.1 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

16.2 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

16.3 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!

16.4 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

16.5 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

16.6 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`

16.7 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

Module 17

Control Filesystem Mounting and Unmounting

17.1 Mounting Filesystems

- As far as many parts of a Linux system are concerned, a partition contains entirely arbitrary data
- When installing, you set things up so that a partition contains a filesystem — a way of organising data into files and directories
- One filesystem is made the **root filesystem**: the root directory on that filesystem becomes the directory named /
- Other filesystems can be **mounted**: the root directory of that filesystem is grafted onto a directory of the root filesystem
  - This arranges for every file in every mounted filesystem to be accessible from a single unified name space
- The directory grafted onto is called the **mount point**

17.2 Mounting a Filesystem: `mount`

- 'Important' filesystems are mounted at boot-up; other filesystems can be mounted or unmounted at any time
- The `mount` command mounts a filesystem
  - You usually need to have root permission to mount a filesystem
- `mount` makes it easy to mount filesystems configured by the system administrator
- For example, many systems are configured so that

  ```
  $ mount /mnt/cdrom
  ```

  will mount the contents of the machine's CD-ROM drive under the directory */mnt/cdrom*

17.3 Mounting Other Filesystems

- `mount /dev/sdb3 /mnt/extra` mounts the filesystem stored in the `/dev/sdb3` device on the mount point */mnt/extra*
- You may occasionally need to specify the filesystem type explicitly:

```
# mount -t vfat /dev/hdd1 /mnt/windows
```

  - Allowable filesystem types are listed in the `mount(8)` manpage
- To see a list of the filesystems currently mounted, run `mount` without any options

17.4 Unmounting a Filesystem: `umount`

- A filesystem can be unmounted with `umount`
  - Note the spelling!
- `umount /mnt/extra` unmounts whatever is on the */mnt/extra* mount point
- `umount /dev/sdb3` unmounts the filesystem in the `/dev/sdb3` device, wherever it is mounted
- You normally need to have root permission to unmount a filesystem
- It's also impossible to unmount a 'busy' filesystem
  - A filesystem is busy if a process has a file on it open
  - Or if a process has a directory within it as its current directory

17.5 Configuring `mount`: */etc/fstab*

- The */etc/fstab* file contains information about filesystems that are known to the system administrator
  - Specifying a filesystem in */etc/fstab* makes it possible to use its mount point as the only argument to `mount`
- */etc/fstab* also configures which filesystems should be mounted at boot-up
- Each line in */etc/fstab* describes one filesystem
- Six columns on each line

17.6 Sample */etc/fstab*

- A sample */etc/fstab* file:

```
# device    mount-point  type     options          (dump)  pass-no
/dev/hda3   /            ext2     defaults              1        1
/dev/hda1   /boot        ext2     defaults              1        2
/dev/hda5   /usr         ext2     defaults              1        2
/dev/hdb1   /usr/local   ext2     defaults              1        2
/dev/hdb2   /home        ext2     defaults              1        2
none        /proc        proc     defaults              0        0
/dev/scd0   /mnt/cdrom   iso9660  noauto,users,ro       0        0
/dev/fd0    /mnt/floppy  auto     noauto,users          0        0
```


17.7 Filesystem Types

- The most common filesystem types are:

Type	Usage
`ext2`	The standard Linux filesystem
`iso9660`	The filesystem used on CD-ROMs
`proc`	Not a real filesystem, so uses `none` as the device. Used as a way for the kernel to report system information to user processes
`vfat`	The filesystem used by Windows 95
`auto`	Not a real filesystem type. Used as a way of asking the `mount` command to probe for various filesystem types, particularly for removable media

- Networked filesystems include `nfs` (Unix-specific) and `smbfs` (Windows or Samba)
- Other, less common types exist; see `mount(8)`

17.8 Mount Options

- Comma-separated options in */etc/fstab*
- Alternatively, use comma-separated options with `-o` on the `mount` command line
- Common mount options:

Option	Description
`noauto`	In */etc/fstab*, prevents the filesystem being mounted at bootup. Useful for removable media
`ro`	Mount the filesystem read-only
`users`	Let non-root users mount and unmount this filesystem
`user`	Like `users`, but non-root users can only unmount filesystems that they themselves mounted

- Other less common mount options exist, as well as many options for individual filesystem types — see `mount(8)`

17.9 Other Columns in */etc/fstab*

- The fifth column is called `dump`
  - Used by the `dump` and `restore` backup utilities
  - Few people use those tools
  - Just use `1` for normal filesystems, and `0` for removable filesystems
- The sixth column is called `pass-no`
  - Controls the order in which automatically-mounted filesystems are checked by `fsck`
  - Use `1` for the root filesystem
  - Use `0` for filesystems that aren't mounted at boot-up
  - Use `2` for other filesystems

17.10 Mounting a File

- Using **loop devices**, Linux can mount a filesystem stored in a normal file, instead of a disk
- Useful for testing images of CD-ROMs before burning them to disk
- For example, to create a filesystem of roughly floppy-disk size:

```
# dd if=/dev/zero of=disk.img bs=1024 count=1400
# mke2fs -F disk.img
```

- To mount the file so that its contents is accessible through */mnt/disk*:

```
# mount -o loop disk.img /mnt/disk
```

17.11 Exercises

1. **a.** Use `mount` to find out which filesystems are mounted.

   b. Check the */etc/fstab* file to see whether the floppy drive is configured properly, and find out what its mount point is set to.

   c. Mount a floppy disk at the default mount point.

   d. Copy a file onto the floppy disk. Does Linux write it immediately? Unmount the floppy to ensure that everything on it is properly written, and it is safe to remove.

   e. Try the commands on the last slide to mount a file, and try copying some files into it. Try using the `df` command to see how much space is available in the file. Unmount */mnt/disk* as you would any other filesystem.

Module 18

Maintain the Integrity of Filesystems

18.1 Filesystem Concepts

- The files stored on a disk partition are organised into a **filesystem**
- There are several filesystem types; the common Linux one is called **ext2**
- A filesystem contains a fixed number of **inodes**
  - 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 file name refers to an inode, not to the file directly
  - This allows **hard links**: many file names referring to the same inode

18.2 Potential Problems

- Over time, an active filesystem can develop problems:
  - It can fill up, causing individual programs or even the entire system to fail
  - It can become corrupted, perhaps due to a power failure or a system crash
  - It can run out of space for inodes, so no new files or directories can be created
- Monitoring and checking filesystems regularly can help prevent and correct problems like these

18.3 Monitoring Space: `df`

- Run `df` with no arguments to get a listing of free space on all mounted filesystems
- Usually better to use the `-h` option, which displays space in human-readable units:

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda8             248M   52M  183M  22% /
/dev/hda1              15M  5.6M  9.1M  38% /boot
/dev/hda6              13G  5.0G  7.4G  41% /home
/dev/hda5              13G  4.6G  7.8G  37% /usr
/dev/hda7             248M  125M  110M  53% /var
```

- The `Use%` column shows what percentage of the filesystem is in use
- You can give `df` directories as extra arguments to make it show space on the filesystems those directories are mounted on

18.4 Monitoring Inodes: `df`

- Filesystems rarely run out of inodes, but it would be possible if the filesystem contains many small files
- Run `df -i` to get information on inode usage on all mounted filesystems:

```
$ df -i
Filesystem            Inodes   IUsed   IFree IUse% Mounted on
/dev/hda8              65736    8411   57325   13% /
/dev/hda1               4160      30    4130    1% /boot
/dev/hda6            1733312  169727 1563585   10% /home
/dev/hda5            1733312  138626 1594686    8% /usr
/dev/hda7              65736    1324   64412    2% /var
```

- In this example, every filesystem has used a smaller percentage of its inodes (`IUse%`) than of its file space
  - This is a good sign!

18.5 Monitoring Disk Usage: du

- `df` shows a summary of the *free* space on a partition
- `du`, on the other hand, shows information about disk space *used* in a directory tree
- Takes one or more directories on the command line:

```
$ du /usr/share/vim
2156    /usr/share/vim/vim58/doc
2460    /usr/share/vim/vim58/syntax
36      /usr/share/vim/vim58/tutor
16      /usr/share/vim/vim58/macros/hanoi
16      /usr/share/vim/vim58/macros/life
40      /usr/share/vim/vim58/macros/maze
20      /usr/share/vim/vim58/macros/urm
156     /usr/share/vim/vim58/macros
100     /usr/share/vim/vim58/tools
5036    /usr/share/vim/vim58
5040    /usr/share/vim
```

18.6 du Options

Option	Description
`-a`	Show all files, not just directories
`-c`	Print a cumulative total for all directories named on the command line
`-h`	Print disk usage in human-readable units
`-s`	Print only a summary for each directory named on the command line
`-S`	Make the size reported for a directory be the size of only the files in that directory, not the total including the sizes of its subdirectories

18.7 Finding and Repairing Filesystem Corruption: fsck

- Sometimes filesystems do become corrupted
  - Perhaps there was a power failure
  - Or maybe your kernel version has a bug in it
- The `fsck` program checks the integrity of a filesystem
  - And can make repairs if necessary
- Actually has two main parts:
  - A ‘driver program’, `fsck`, which handles any filesystem type
  - One ‘backend program’ for each specific filesystem type
- The backend program for ext2 is `e2fsck`, but it is always invoked through `fsck`

18.8 Running `fsck`

- `fsck` is normally run at system startup
  - So it gets run automatically if the system was shut down uncleanly
- It can also be run manually:

```
# fsck /dev/sdb3
```

  - Interactively asks whether to fix problems as they are found
- Use `-f` to force checking the filesystem, even if `fsck` thinks it was cleanly umounted
- Use `-y` to automatically answer 'yes' to any question
- Usually a bad idea to run `fsck` on a mounted filesystem!

18.9 Exercises

1. a. Check the free disk space on the computer.

   b. Display just the usage information for the partition that contains */usr/*. Display this in human-readable units.

   c. Look at the free space and inodes of the partition of */var/tmp* first. Then run these commands:

```
$ mkdir /var/tmp/foo
$ seq -f '/var/tmp/foo/bar-%04.f' 0 2000 | xargs touch
```

   What has happened? Look at the free space and inodes again.

   Remove the files when you have finished.

2. Go into the */var/* directory. Run each of the following commands as root, and explain the difference in their output:

   a. `# du`

   b. `# du -h`

   c. `# du -h *`

   d. `# du -hs`

   e. `# du -hs *`

   f. `# du -hsS *`

   g. `# du -hsc *`

   h. `# du -bsc *`

Module 19

Find System Files and Place Files in the Correct Location

19.1 Unix Filesystem Layout

- Many common directory names are abbreviated versions of real words
- Traditional structure which has developed over many years
  - Most system files have their proper place
  - Programs rely on them being in the correct place
  - Users familiar with Unix directory structure can find their way around any Unix or Linux system
- But a user's home directory can be structured however they want

19.2 The Filesystem Hierarchy Standard

- Started as an attempt to standardise Linux filesystem layouts
  - Called the FSSTND when the first version was published in 1994
- Widely accepted by distributors
  - But few Linux systems are 100% compliant yet
- Intended to avoid fragmentation of Linux distributions
- Renamed to the **File Hierarchy Standard**, or **FHS**
- Now intended to apply to all Unix-like operating systems

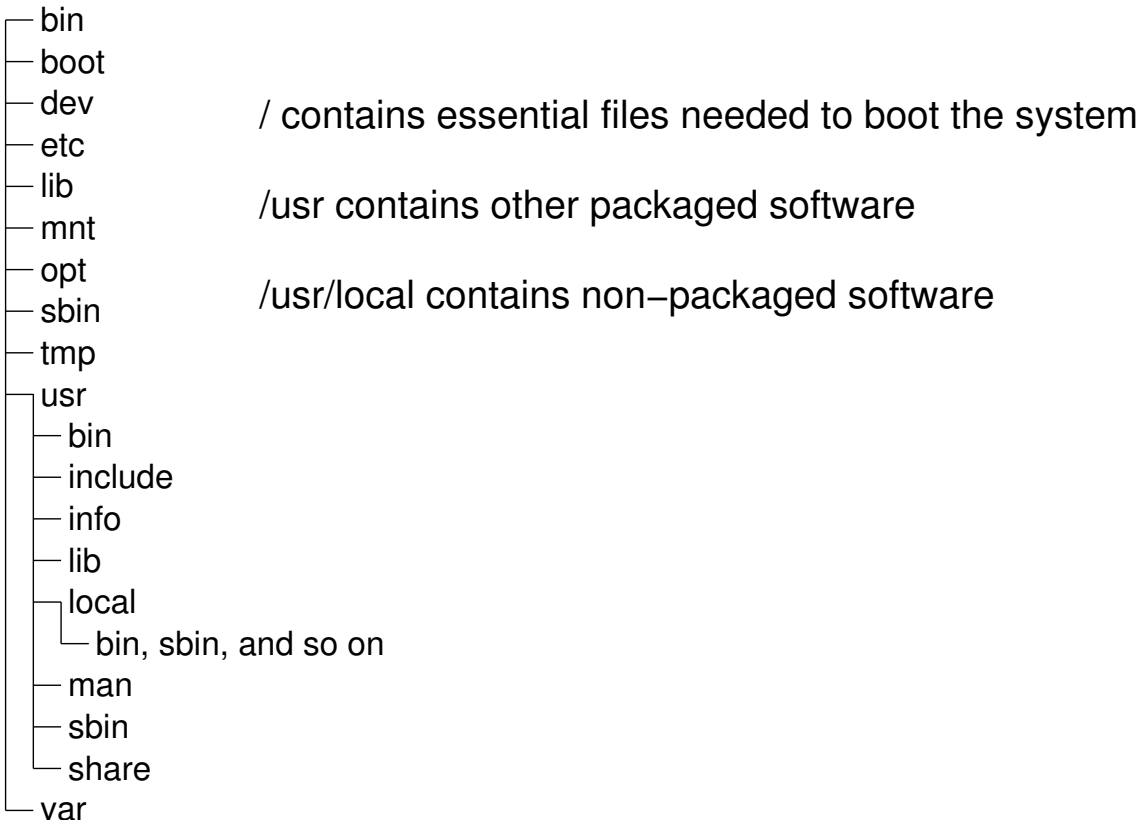
19.3 Shareable and Non-Shareable Data

- Some files can be shared between multiple computers, using networked filesystems such as NFS
  - This can save space, although cheap hard drives mean that this is not so important now
  - More importantly, it can help to centralise administration for a network
- Usually programs, email and home directories are all shareable
- Log files and machine-specific configuration files are not shareable

19.4 Static and Dynamic Data

- Some files hardly ever need to be changed, while others change all the time
- It can help to store static files separately from those which regularly change:
  - The static files can be on a partition mounted read-only (such as a CD-ROM)
- Programs and libraries are usually static (except when new software is installed)
- Home directories and status files are usually more variable

19.5 Overview of the FHS



19.6 FHS: Installed Software

- Programs are usually found in the *bin* and *sbin* directories
  - These are found in /, */usr* and */usr/local*
- *sbin* is used for programs likely to be useful to system administrators rather than to general users (mail dæmon, web server, etc.)
- These directories are named after **binaries**
  - Most programs in them are binaries (compiled programs), although some are human-readable scripts
- Libraries are stored in directories called *lib*, found in the same places as *bin*
  - These directories should be listed in */etc/ld.so.conf*

19.7 FHS: Other Directories Under */usr*

- */usr/include* contains header files used by C/C++ programs
- */usr/X11R6* contains files used by the X Window system, including programs, libraries, configuration files and documentation
- */usr/local* is where software is installed when it is compiled from source code rather than installed as a package
- */usr/share* contains files which are not specific to the architecture of a machine, e.g., fonts and icons
  - Theoretically could be shared between different types of machine over a network
- */usr/src* often contains the source code for the Linux kernel
  - Usually kept in a directory such as *linux-2.2.20*, with a symbolic link to it called *linux*

19.8 FHS: Directories Under */var*

- */var/run* contains pid files (process-id files for currently-running dæmon programs)
  - Also contains *utmp*, a record of user logins
- */var/mail* or */var/spool/mail* is where each user's email is queued up until it is deleted or saved
- */var/log* contains logs produced by various programs, including `syslog`
- */var/cache* contains data generated by programs which is cached to save time
  - Cached data can be regenerated if deleted

19.9 FHS: Other Directories

- */etc* contains configuration files
- */mnt* is used to mount other filesystems temporarily
  - For example, floppy discs are mounted on */mnt/floppy*
- */boot* contains files used by LILO to boot the system
- */dev* contains device files, which provide access to hardware devices such as disk drives or serial ports
- */tmp* is used by many programs for temporary files
- */opt* can contain packages of software from third parties (i.e., not in the native package management format)

19.10 FHS: Other Directories

- */proc* provides access to information from the kernel, particularly about running processes
- */home* contains directories which belong to each user
  - Use `echo ~` to find out where your home directory is
- */root* is the home directory of the `root` user

19.11 Finding Programs with `which`

- Searches for programs which can be run
- Looks in the same directories as the shell
  - Determined by the `$PATH` environment variable
  - Use `echo $PATH` to see what directories are searched
- For example, to find out where `gnumeric` is:

```
$ which gnumeric
```

- This is useful if different versions of the same program are installed in different places

19.12 The `type` Built-in Command

- `type` is like `which`, but is built into the shell
  - It tells you about shell aliases and functions
  - Not available in the C Shell
- `type -p` is just like `which`
- `type -a` shows all the commands of the name given
  - Useful for detecting duplicate programs, or aliases which are hiding real programs
- See `help type` for full details

19.13 Checking for Shell Builtins with `type`

- Some commands are built into the shell
  - Examples include `cd`, `test`, `pwd` and `ulimit`
- The *Bash* shell has a builtin called `type` which reports on whether a command is a builtin
- For example, to see whether the `test` command will run a shell builtin, or a real program:

```
$ type test
```

- The example shows that `test` will run a shell builtin, even though there is a real program with the same name
- `type` will also identify shell aliases and functions

19.14 Updating the `locate` Database

- Use the `updatedb` program to refresh the database of files used by `locate`
- Modern versions are configured by giving options to `updatedb`
  - `-e` provides a list of directories which will not be searched
  - `-f` gives the names of filesystem types to miss out
  - See the manpage for full details
- `updatedb` is usually run by `cron` every night
  - Look in */etc/cron.daily* for the script which runs it

19.15 *updatedb.conf*

- Older versions of GNU `updatedb` used the configuration file */etc/updatedb.conf*
  - For compatibility, some modern versions still read it
- The configuration is done by setting environment variables
- For example, to ignore certain filesystems:

```
$ PRUNEPATHS="/tmp /usr/tmp /var/tmp /mnt /var/spool"
$ export PRUNEPATHS
```

- The `$PRUNEFS` variable lists the names for filesystems which should be ignored (e.g., `nfs`, `iso9660`, etc.)
- These variables are equivalent to the `-e` and `-f` options

19.16 `whatis`

- `whatis` finds manpages with the given name and lists them
  - Usually only useful when the name of a command is already known
- For example, to find manpages about `bash`:

```
$ whatis bash
```

- The database searched by `whatis` is updated with the `makewhatis` command
  - This should be run when new manpages are installed
  - Debian instead has */etc/cron.daily/man-db*, which also expunges old cached man pages

19.17 Finding Manpages with `apropos`

- The `apropos` command is similar to `whatis`
  - The difference is that any word in the title line of a manpage can match the word given
- `apropos` *`word`* is identical to `man -k` *`word`*
- For example, to find commands relating to directories:

```
$ apropos directories
$ man -k directories
```

- `apropos` also uses the database built by `makewhatis`

19.18 Web Resources

- The FHS — http://www.pathname.com/fhs/

19.19 Exercises

1. a. Find out whether the `ls` command runs a program directly, or is a shell alias or function.
   b. Locate the binary of the `traceroute` program.
   c. Use `whatis` to find out what the `watch` command does.
   d. Use `apropos` to find programs for editing the partition table of disks.
   e. See if the Linux installation you are using has an *updatedb.conf*, and look at the current configuration if it has.
   f. Log on as root and update the `locate` database with the `updatedb` command.

Module 20

Set and View Disk Quotas

20.1 What are Quotas?

- Quotas are a way of limiting the amount of disk space that users may take up
- Some organisations (perhaps those with untrusted external users) absolutely need to ensure that:
  - No user can prevent other users from using a reasonable amount of disk space
  - No user can impede the correct functioning of the system
- Some organisations don't need to worry about this — their users can be trusted not to abuse the system
- Unfortunately, quota management is unnecessarily hard on Linux
  - Could user education avoid the need for quotas?
  - Disk space is cheap!

20.2 Hard and Soft Limits

- Quotas have **hard limits** and **soft limits**
- A user can exceed the soft limit without retribution
  - But only for a certain period of time — the **grace period**
  - The user is also warned that the soft limit has been exceeded
- A hard limit may never be exceeded
- If a user tries to exceed a hard limit (or an expired soft limit), the attempt fails
  - The program gets the same error message it would if the filesystem itself had run out of disk space
- Grace periods are set per-filesystem

20.3 Per-User and Per-Group Quotas

- Most quotas are set per-user
  - Each user has his or her own soft limit and hard limit
- Quotas can also be set per-group
  - A group can be given a soft limit and hard limit
- Group quotas apply to all users in a group
- If a group hard limit has been reached, no user in the group may use more space
  - Including users who have not yet reached their individual quota

20.4 Block and Inode Limits

- Quotas can be set for **blocks**
  - Limits the amount of data space that may be used
- Quotas can also be set for **inodes**
  - Limits the number of files that may be created

20.5 Displaying Quota Limits: `quota`

- The `quota` command displays quota limits
- Specifying a username or the name of a group will show information about their quotas:

```
# quota fred
```

- The `-v` option will show full information about all quotas, even where there are no limits

20.6 Options in */etc/fstab*

- The options in */etc/fstab* specify which filesystems should have quota enabled
  - Add the option `usrquota` to enable user quotas
  - Use `grpquota` to enable group quotas
  - Either or both can be used for each filesystem:

```
/dev/hda1  /            ext2  defaults
/dev/hdb1  /home        ext2  defaults,usrquota
/dev/hdb2  /work/shared ext2  defaults,usrquota,grpquota
```

- The filesystems with quota enabled should have files called `quota.user` and `quota.group` in their root directories
- The following commands will create them:

```
# touch /partition/quota.{user,group}
# chmod 600 /partition/quota.{user,group}
```


20.7 Enabling Quota: `quotaon`

- `quotaon` turns on quota support
  - Can only be done by root
  - Support must be compiled into the kernel, but this is done by default on all modern distributions
- `quotaoff` disables quota support
- For example, to turn on quota on all filesystems:

```
# quotaon -av
```

- Quota can be turned on or off for individual filesystems

20.8 Changing Quota Limits: `setquota`

- Command line program to alter quota limits for a user or group
- Specify the name of a user or group with `-u` *`username`* or `-g` *`groupname`*
- Specify the filesystem to alter after the `-u` or `-g` option
- Finally, the limits to set must be specified in the following order:
  - Soft limit for blocks
  - Hard limit for blocks
  - Soft limit for inodes
  - Hard limit for inodes
- Setting any limit to 0 will remove that limit

20.9 `edquota`

- `edquota` allows quotas to be edited interactively, in a text editor
  - The file in the text editor will be a temporary file
  - `edquota` will read it back in when the editor terminates
- Use the `-g` option to edit group quotas
- Some versions of Red Hat have a bug where you need to delete an extraneous space before the time unit when doing `edquota -t`

20.10 `repquota`

- The `repquota` command prints information about the quota limits assigned to each user
  - Also shows actual number of blocks and inodes used
- Use the `-a` option for information on all filesystems, or specify the filesystem on the command line
- Use `-g` to show group quotas
- Use `-v` for more complete information

Module 21

Boot the System

21.1 Boot Loaders

- When Linux boots, the kernel is loaded into memory by a **boot loader**
- Passes parameters to the Linux kernel
- Allows one of several operating systems to be loaded
  - Multiple versions of the Linux kernel
  - **Dual-booting** with Windows and other OSes
- The most popular boot loader is LILO (the Linux loader)
  - Full user documentation provided
    - Look for the a directory called something like */usr/share/doc/lilo/* or */usr/doc/lilo-0.21/*
    - The user guide is in a file called *user.ps* or *User_Guide.ps*

21.2 LILO

- LILO runs when the system is booted
- The `lilo` command configures how LILO will next run
- The file */etc/lilo.conf* specifies the configuration to be set by the `lilo` command
  - Need to run the `lilo` command for changes to have affect
  - Manual page *lilo.conf(5)*
- *lilo.conf* has options in the form *`name`*`=`*`value`*
- Options for specifix OSes are indented
  - Linux kernels to install are introduced with `image=`
  - Other OSes are introduced with `other=`
- Other options are generic, or are defaults for the OSes

21.3 Sample *lilo.conf* File

```
boot = /dev/hda       # put loader on the MBR
root = /dev/hda1      # device to mount as /

delay = 40            # 4 second delay
compact               # may make booting faster
read-only             # needed to allow root to be fscked

image = /vmlinuz-2.2.20    # stable kernel (default because it's 1st)
   label = linux-2.2.20
   alias = linux           # shorter label
   vga = ask               # let us choose the console size

image = /vmlinuz-2.5.1     # cutting edge kernel
   label = linux-2.5.1

other = /dev/hda3     # Windows is installed on a different partition
   label = windows
   table = /dev/hda
```

21.4 Selecting What to Boot

- When LILO runs it displays the prompt `LILO:`
  - If only some of the letters appear, the boot process failed at some point
- It waits the specified delay for something to start being typed
- Load a particular kernel or OS by entering its label or alias
  - Just press `Enter` for the default
  - Pressing `Tab` lists the available labels
  - Some versions of LILO present a menu to select from with the cursor keys
- If no key has been pressed by the end of the delay, the first kernel or OS is loaded

21.5 Other Ways of Starting Linux

- Grub — complex boot loader which includes a shell and support for accessing filesystems
- LoadLin — a Dos program which can start Linux from within Dos
  - Occasionally used to start Linux after a Dos driver has configured some hardware

21.6 Specifying Kernel Parameters

- Linux kernels take parameters which affect how they run
- Parameters can be specified at boot time:
  - At the LILO prompt
  - After the image label

    ```
    LILO: linux-2.2.20 root=/dev/hda3
    ```

    - Specifies the root filesystem
- Details of parameters are in *BootPrompt-HOWTO*

21.7 Specifying Kernel Parameters in *lilo.conf*

- Kernel parameters can also be specified in *lilo.conf*
  - Sensible to test first at the LILO prompt
- Common parameters have *lilo.conf* option names
- Any parameter can be set with the `append` option

```
image = /vmlinuz-2.2.0
  label = linux-2.2.20
  root = /dev/hda3
  append = "hdc=ide-scsi"
```

21.8 Useful Kernel Parameters

- `root=`*`device`* — set the filesystem to mount as root
- `ro` and `rw` — mount the root filesystem read-only or read-write, respectively
  - Usually this should be `read-only` in *lilo.conf*, to allow `fsck`s
- `nfsroot=`*`server...`* — use a network filesystem as root (e.g., in a diskless workstation)
- `init=`*`program`* — the name of the first program the kernel will run, which is usually */sbin/init*
  - Can be set to */bin/sh* if starting with `init` is broken
- There are many other parameters to configure specific hardware devices

21.9 Boot Messages

- When the kernel starts up it prints a flurry of information
- This can often be useful in finding problems
- A log of this information is kept in */var/log/dmesg*
- The `dmesg` command can print the most recent messages
  - This can show problems which have occurred since boot
- After boot, most log messages are handled by `syslog`

21.10 Kernel Modules

- Many features of the Linux kernel can be built as modules
  - Can be loaded when needed, and unloaded later
  - Compiled modules are stored under */lib/modules/*
- These commands manage modules:
  - `lsmod` — lists currently loaded modules
  - `rmmod` — removes an unused module
  - `insmod` — loads a single module
  - `modprobe` — loads a module, and any other modules it needs
- The file */etc/modules.conf* configures these commands
  - */etc/conf.modules* on some systems
  - Has a manpage, *modules.conf(5)*

21.11 Exercises

1. a. Look at the compiled modules available on the system
   b. List the currently-loaded modules.
   c. Load the `parport` module, and check that it's worked
   d. Unload the `parport` module, and check again
   e. Try unloading a module currently in use. What happens?
2. a. Reboot the computer. You can do this safely by quitting all programs, logging out, then pressing `Ctrl+Alt+Del`. When the LILO prompt appears, list the available options. Load the default.
   b. Reboot again. This time pass the parameter `init=/bin/sh` to the kernel. What happens?
      - Which directory are you in?
      - What's the output of the `hostname` command?
      - Can you create a new file?

      Exit the shell and reboot again
3. Make a backup of *lilo.conf*, then add a new section at the end of the original:
   - Copy the options for the default kernel.
   - Change the label to "shell" (and remove any aliases).
   - Set the first program run by the kernel to `/bin/sh`.

   Make the change live, then reboot and test it.

   Put things back afterwards.

Module 22

Change Runlevels and Shutdown or Reboot System

22.1 Understanding Runlevels

- A Linux system runs in one of several **runlevels** — modes providing different features and levels of functionality
- Linux systems normally have seven runlevels, numbered from 0–6:
  - Three are mandatory (0 = halt, 6 = reboot, 1 = single-user)
  - Four are user-defined (2–5)
- No consensus between administrators or distributions about how to organise the user-defined runlevels
  - Some rely (partly) on runlevels to define which major subsystems are running
  - Others prefer the flexibility of starting and stopping subsystems individually, without changing runlevel
  - In every common distribution, there is at least one user-defined runlevel which has the same services as another

22.2 Typical Runlevels

Runlevel	Description
0	A ‘transitional’ run-level, used to tell the system to shut itself down safely. Once the system has shut down, it needs a manual reboot to reactivate.
1	Single-user mode, used for maintenance. Users may not log in, and many services (usually including all networking facilities) are disabled. There is only one terminal active, on which `root` is automatically logged in.
2–5	Multi-user modes. Some systems make all of these identical. Others disable networking (or NFS file-sharing) in runlevel 2, and/or enable a graphical login in runlevel 5 (but not in other runlevels).
6	A ‘transitional’ run-level, used to tell the system to reboot.

22.3 Single-User Mode and `sulogin`

- Many Linux distributions use a program called `sulogin` to restrict access to single-user mode
- `sulogin` is run when the system enters single-user mode
- It requires the `root` password to be typed on the console before switching to single-user mode
  - If the password is not typed, `sulogin` returns the system to the normal runlevel
- Why is `sulogin` needed?
  - Untrusted users may have access to the system's keyboard during bootup
  - In many configurations, this would enable them to boot the system up in single-user mode

22.4 Shutting Down and Restarting the System

- To safely shut down the system, run the `halt` command as root
  - This is a safe shutdown: it stops all services, disables all network interfaces, and unmounts all filesystems
- To safely reboot the system, run `reboot` as root
  - Most systems also let you hit `Ctrl+Alt+Del` on the console
- Alternatively, the `shutdown` command allows you to schedule the power-down or reboot, to give users warning to save their work
  - Halt at 6pm:

    ```
    # shutdown -h 18:00
    ```

  - Reboot thirty minutes from now:

    ```
    # shutdown -r +30
    ```

22.5 Setting the Default Runlevel

- The system's default runlevel on bootup is configured in */etc/inittab*
- To configure a default runlevel of 3, */etc/inittab* should contain the line:

  ```
  id:3:initdefault
  ```

- There should be only one `initdefault` line in */etc/inittab*

22.6 Selecting a Different Runlevel at Bootup

- Most bootloaders (including LILO) give you the ability to type in a **kernel command line**
- Naming a runlevel on the kernel command line selects that runlevel for use on system startup
- To start in single-user mode:

  ```
  linux 1
  ```

- To start in emergency mode:

  ```
  linux -b
  ```

  Emergency mode provides nothing beyond a shell to type into — useful for repairing serious system corruption

22.7 Determining the Current Runlevel

- The `runlevel` command prints the system's previous and current runlevels:

  ```
  $ /sbin/runlevel
  N 3
  ```

- If there is no previous runlevel (for example, if the runlevel hasn't been changed from the default), `N` is printed instead

22.8 Switching Runlevel

- The system has a process named `init`, with pid 1, which is the ultimate ancestor of *all* other processes
- `init` is responsible for controlling runlevels, so switching runlevels involves telling `init` to do something:
  - As root, run

    ```
    # telinit 1
    ```

    to switch into a given runlevel
- You can alternatively use `init` itself, with the same syntax:

  ```
  # init 5
  ```

- Obviously, changing runlevels should not be undertaken lightly
  - In particular, changing runlevel can terminate important system services, or affect users' ability to log in

22.9 Services in Each Runlevel: the *init.d* Directory

- */etc* contains an *init.d* directory, and an *rcN.d* directory for each runlevel *N*
  - Some distributions (notably Red Hat) put all these directories in */etc/rc.d*, not directly under */etc*
- *init.d* contains an **init script** for each service that can be started
- The *rcN.d* directories contain symbolic links to the init scripts
  - These symbolic links control which services are available in each runlevel

22.10 Symbolic Links in *rcN.d*

- Symbolic links in the *rcN.d* directory are either **start links** or **stop links**
  - Start links are named *SNNservice*, where *NN* is a number and *service* is the name of a service
  - Stop links are named *KNNservice*
- The start links for a runlevel directory indicate which services should be started when that runlevel is entered
- Correspondingly, the stop links indicate which services should be stopped when the runlevel is entered
- The `rc` shell script (usually */etc/rc.d/rc* or */etc/init.d/rc*) runs the relevant init script appropriately for start links and stop links

22.11 Starting or Stopping Individual Services

- You can also start or stop an individual service without changing runlevel
- An init script always takes an argument of `start` or `stop` to start or stop the relevant service
- For example, if the MySQL database server has an init script */etc/init.d/mysql*, you can start MySQL with

  ```
  # /etc/init.d/mysql start
  ```

  or stop it with

  ```
  # /etc/init.d/mysql stop
  ```

- Some init scripts also accept an argument of `restart` (stop and then re-start) or `reload` (reload the service's configuration file)

22.12 Exercises

1. a. Look in */etc/init.d* or */etc/rc.d/init.d* to see what services can be started by `init`.

   b. Try running the script for `crond`, and use it to stop the `cron` service, and then start it up again.

   c. Take a quick look at the program in a text editor (it's a small shell script) to get a rough idea of what it

does.

d. Look in the *rc3.d* directory to see what services are killed and started when switching to runlevel 3.

e. Use `telinit` to change to single-user mode.

f. Once in single-user mode, use `top` to see what processes are left running.

2. a. Reboot the machine by changing to runlevel 6.

b. When the `LILO` prompt appears, type `Tab` to see a list of operating systems to boot. Type the name of the one you want followed by a space and the number 1, to indicate that you want to boot straight into single-user mode.

c. Change back to runlevel 3.

Module 23

Use and Manage Local System Documentation

23.1 Manual Pages

- Most Linux commands have an associated **manual page**
  - Often known as a **manpage**
- Viewed with the `man` command:

```
$ man ls
```

- Press `q` to quit and return to the shell prompt

23.2 Navigating Within Manual Pages

- `man` uses the `less` viewer
- Use the cursor keys for scrolling
- Other common keystrokes:

`Space`	jump down a page
`b`	jump back up a page
`/word`	search for the next occurrence of “word”
`n`	repeat the previous search
`g`	go to the top

- Help on all keystokes is available with `h`

23.3 Sections of a Manual Page

- Manpages have a traditional format
- Manpages for user commands tend to have some or all of these sections:
  - NAME — name and single-line reason for the command
  - SYNOPSIS — possible arguments
  - DESCRIPTION — fuller explanation of the command
  - OPTIONS
  - FILES — any files the command needs
  - ENVIRONMENT — pertinent environment variables
  - BUGS
  - AUTHOR
  - SEE ALSO

23.4 Sections of the Manual

- Each manpage is in a **section** of the manual
- User commands are in section 1
- Different sections can contain pages of the same name:
  - The ‘passwd’ page in section 1 describes the `passwd` command
  - The ‘passwd’ page in section 5 describes the */etc/passwd* file
  - These are often referred to as “passwd(1)” and “passwd(5)”
- A page can be requested from a particular section by specifying its number:
  - `man 1 passwd`
  - `man 5 passwd`

23.5 Manual Section Numbering

- Most commands are documented in section 1
  - This is the first place `man` looks
  - So the `passwd` command's manpage can also be viewed with:

    ```
    $ man passwd
    ```
- Other sections you may need:
  - Some system administration commands are in section 8
  - File formats are in section 5
  - Miscellany is in section 7
- A complete list of sections is in man(7)
- Each section has an introduction page called 'intro':

  ```
  $ man 8 intro
  ```

23.6 Determining Available Manpages with `whatis`

- The `whatis` command lists manpages with the specified name:

  ```
  $ whatis hostname
  hostname             (1)  - show or set the system's host name
  hostname             (7)  - host name resolution description
  ```
- Section number in brackets
- Single-line description from the NAME section
- Useful for quickly discovering what a command does
  - "What is `tac`?"

    ```
    $ whatis tac
    ```
- `man -f` is equivalent to `whatis`

23.7 Printing Manual Pages

- Manpages can be printed out in a nicely-formatted way:

  ```
  $ man -t head > head.ps
  ```
  - Formats the manpage for `head` as PostScript and writes it to *head.ps* in the current directory
- Alternatively, send the PostScript directly to a printer:

  ```
  $ man -t head | lpr
  ```


23.8 Searching for Manpages with `apropos`

- To search for pages with a NAME section matching a particular keyword, use `apropos`:

```
$ apropos gif
gif2tiff        (1)  - create a TIFF file from a GIF87 format image file
giftopnm        (1)  - convert a GIF file into a portable anymap
ppmtogif        (1)  - convert a portable pixmap into a GIF file
Data::Dumper    (3)  - stringified perl data structures, suitable for both printing and eval
```

- Can't restrict the search to a particular section
  - But can `grep` the output:

```
$ apropos gif | grep '(1)'
```

- `man -k` is equivalent to `apropos`

23.9 Displaying All Manpages of a Particular Name with `man -a`

- To display all pages which have a particular name, regardless of their section, use `man -a`:

```
$ man -a hostname
```

  - Displays hostname(1)
  - Waits for you to quit
  - Displays hostname(8)

23.10 Searching the Content of All Manpages with `man -K`

- It is possible to search through the textual content of the entire manual
- `man -K` (note upper-case) does this:

```
$ man -K printer
```

  - Filename of each matching page is displayed in turn
  - Prompt for choosing whether to display it
- Not particularly useful
  - Many false matches obscuring the data you want
  - Slow to search so much text
  - Tedious to respond to each prompt

23.11 Finding the Right Manual Page

- Sometimes commands' documentation are not quite where expected
- Related commands can be grouped together on one page:
  - These commands all display the same page:

```
$ man gzip
$ man gunzip
$ man zcat
```

  - Can be misleading if you look up one command and start reading the description of another

23.12 Help on Shell Builtins

- Shell built-in commands are documented in shells' manpages:
  - cd(1) refers the reader to bash(1)
  - echo(1) relates to `/bin/echo`, but in most shells `echo` is a separate built-in command
- The bash(1) manual page has details, but is too big
  - See the section 'SHELL BUILTIN COMMANDS'
- For brief explanations of builtin functions, use `help`:

```
$ help help
help: help [-s] [pattern ...]
Display helpful information about builtin commands.  If PATTERN is
...
```

- Run `help` without arguments to get a list of builtin commands

23.13 Location of Manual Pages

- Manpages are stored in the filesystem
- You can use `man` to find the locations of a given manpage
  - Use the `-a` and`-w` options to show the locations of all manpages with a given name:

```
$ man -aw passwd
/usr/man/man1/passwd.1.gz
/usr/man/man5/passwd.5.gz
```

- Common locations for manpages include */usr/man* and */usr/share/man*
  - Locally-installed packages often put manpages under */usr/local/man*

23.14 Info Pages

- GNU have a rival documentation system called **info**
  - GNU utilities have info pages
  - Often duplicating man pages
  - But some GNU utilities have half-hearted man pages
  - A few other programs use info too
- An info page is viewed with the `info` command:

```
$ info cat
$ info ls
```

- Use `q` to quit and return to the shell
- Emacs has a better info viewer built in, and there is an alternative, slicker viewer available, called `pinfo`

23.15 Navigating Within Info Pages

- Scroll with the cursor keys, `PgUp` and `PgDn`
- An info page may be split into **nodes**
  - For example, the `ls` page has separate nodes covering file selection, formatting, and sorting
  - Hyperlinks between nodes start with stars
  - Node navigation keystrokes:

`Tab`	jump to next hyperlink
`Enter`	follow hyperlink
`l`	return to your previous location
`n`	go to the following (‘Next’) node
`p`	go to the preceding (‘Prev’) node
`u`	go to the parent (‘Up’) node

  - The ‘Next’, ‘Prev’, and ‘Up’ destinations are shown at the top.

23.16 Documentation in */usr/share/doc/*

- Some programs’ main (or only) documentation is not available as man or info pages
- */usr/share/doc/* contains other formats of documentation
  - Usually plain text
  - Sometimes HTML
  - Subdirectory per package, such as */usr/share/doc/grep-2.4/*
- On many systems (particularly older ones) */usr/doc* is used instead of */usr/share/doc*
- On really awkward systems both directories exist, and contain different documentation!

23.17 Contents of */usr/share/doc*

- Documentation in */usr/share/doc* is often information only relevant to system administration of a package, not users of it:
  - Installation instructions, licence, change log
- Sometimes more user-friendly documentation than elsewhere
  - For example */usr/share/doc/ImageMagick-4.2.9/ImageMagick.html*
  - HTML help is more common for interactive applications, and very rare for traditional Unix commands
  - Programs ported from other platforms often have documentation in */usr/share/doc/* rather than man pages

23.18 Interrogating Commands for Help

- Some commands have no external documentation, but have an option to display some help:

```
$ netscape -help
```

- Others do have documentation elsewhere but have an option to display a usage summary:

```
$ vim -h
```

- GNU utilities all have a `--help` option for this:

```
$ grep --help
```

- Discovering which, if any, option does this can often only be found by trial and error

23.19 Finding Documentation

- Unfortunately some luck is required for finding documentation
- With time you pick up the hang of what is likely to be documented where
- The `locate` command can be useful for finding all files related to a particular command.
- Web search engines can sometimes be the fastest way of searching for documentation
  - Many places have the entire manual pages hosted on the web, which Google *et al* have conveniently indexed

23.20 Exercises

1. a. Use `man man` to open the man page which details how to use the `man` command itself
   b. Press the `h` (help) key to see a summary of commands and keystrokes
   c. Find out how to do the following things:
      i. Move to the start and the end of the man page
      ii. Move up and down the text one screen at a time

iii. Move up and down one line at a time

iv. Search for a pattern in the text

v. Repeat a previous search with a single keypress

vi. Move to a specific line, by number

2. a. From the man page for `man`, find out what commands to type to do the following:

i. Get a list of manual pages about the 'whatis' database

ii. Get a list of manual pages from section 1 whose descriptions contain the word 'print'

iii. Search for man pages containing the string 'cdrom' (but why is this a problem?)

b. Practice using `man` to find out about things which interest you, or try some of these examples:

i. Bitmap image formats like JPEG, GIF, XPM and PNG

ii. Communications concepts like modems, serial connections, telnet, PCMCIA and PPP

iii. Filesystems like NFS, ext2, FAT, vfat, msdos and Samba

3. a. Take a quick look at the documentation for the `tar` command, in both the man page and the Info documentation. How are they different?

b. What happens when you run `info` without specifying which document to view?

Module 24

Find Linux Documentation on the Internet

24.1 The Linux Documentation Project

- The **Linux Documentation Project**, or **LDP**, promotes and develops Linux documentation
- http://www.linuxdoc.org/
  - Many mirrors
  - http://www.mirror.ac.uk/sites/www.linuxdoc.org/
  - http://www.doc-linux.co.uk/LDP/
- The LDP is “working on developing free, high quality documentation for the GNU/Linux operating system”
- “The overall goal of the LDP is to collaborate in all of the issues of Linux documentation”

24.2 HOWTOs

- A **HOWTO** is a document describe how to do something
- HOWTOs cover a wide range of topics
  - General overviews of topic areas, such as *Printing-HOWTO*
  - Detailed instruction of very specific tasks, such as *Bash-Prompt-HOWTO*
  - Information for particular groups of users, such as *Belgian-HOWTO*
- Various authors
- Varying quality

24.3 Obtaining HOWTOs

- HOWTOs are written in a special mark-up language which enables them easily to be produced in several formats:
  - HTML
  - Plain text
  - PostScript
  - PDF ('Acrobat')
- Some formats may be installed in */usr/share/doc/HOWTOs/*
- They are all on the LDP website:
  - Good: They are all clearly dated, so you can see how recent their advice is
  - Bad: A number of them haven't been updated for several years

24.4 Vendor- and Application-Specific Web Sites

- Unsurprisingly, particular programs often have their own web presence:
  - *Less*: http://www.greenwoodsoftware.com/less/
  - *The Gimp*: http://www.gimp.org/
- Linux distributions also have their own websites:
  - Debian: http://www.uk.debian.org/
  - Red Hat: http://www.redhat.com/
  - SuSE: http://www.suse.co.uk/
  - Mandrake: http://www.linux-mandrake.com/
- May have specific mailing lists or web-based forums

24.5 Usenet Newsgroups

- There are many usenet newsgroups related to Linux
- The international groups are divided by topic:
  - comp.os.linux.setup
  - comp.os.linux.help
  - comp.os.linux.hardware
- Search the archives for answers to questions:
  - http://groups.google.co.uk/advanced_group_search
  - Formerly DejaNews, or just Deja, but now hosted by Google
  - May find the answer much faster
  - It irritates people to encounter well-answered questions being asked many times
- Not always the most friendly places for beginners

24.6 FAQs

- There have been several attempts to document questions frequently asked about Linux, with their answers
- Many newsgroups have their own FAQs:
  - http://www.faqs.org/faqs/by-newsgroup/
- The LDP have a frequently-updated Linux FAQ:
  - http://www.linuxdoc.org/FAQ/Linux-FAQ/
- Again, search these before asking for help.

24.7 Local Help

- More localized forums are often friendlier
- UK newsgroup:
  - news:uk.comp.os.linux
  - http://www.ucolfaq.lug.org.uk/
- UK Linux users mailing lists:
  - http://www.ukuug.org/sigs/linux/newsgroups.shtml
- Many areas have their own Linux user groups
  - http://www.lug.org.uk/lugs/
  - Meetings
  - Local mailing lists

Module 25

Tune the User Environment and System Environment Variables

25.1 Configuration Files

- Many programs, including shells, read configuration files
- Files which apply to only one user are stored in that user's home directory, usually as **hidden files**
  - Use `ls -A` in your home directory to find them
  - Hidden files have names which start with '`.`'
  - Often such files have names ending in `rc`, for 'run commands', for example the Vim editor uses *.vimrc*
- Sometimes whole directories of configuration information are present in a home directory, for example *.kde* and *.gnome*

25.2 Shell Configuration Files

- Bash reads *~/.bashrc* whenever it starts as an interactive shell
  - That file often sources a global file, for example:

```
if [ -f /etc/bashrc ]; then
   . /etc/bashrc
fi
```

- Bash also reads a profile file if it is a login shell
  - First it reads the global configuration from */etc/profile*
  - Then one of *~/.bash_profile*, *~/.bash_login* or *~/.profile*
- Login shells also source *~/.bash_logout* when the user exits

25.3 Changing Environment Variables

- The value of an environment variable can be set on the command line or in a configuration file as follows:

```
export VARIABLE=VALUE
```

- To see the current value of a variable: `echo $VARIABLE`
- The shell searches for programs to run in a list of directories in the variable `$PATH`, which are separated by the '`:`' character
  - If you want to run programs which aren't in */bin* or */usr/bin* then you might want to add them to your `$PATH`, for example:

```
export PATH="$PATH:/usr/local/bin:/usr/games"
```

- Some other variables, such as `$INFOPATH`, use the same convention

25.4 Changing the Prompt

- Setting `PS1` will change the shell prompt, for example:

```
export PS1='\w\$ '
```

- Characters preceded by `\` are given special interpretations, for example:
  - `\t` and `\d` display the time and date in the prompt
  - `\w` or `\W` show the current working directory
  - `\$` shows up as either `$` or `#`, depending on whether you are a normal user or root
  - `\u` displays your username
  - `\h` displays the hostname of the machine
- `PS2` is an alternative prompt, displayed when bash needs more input before it can run a complete command

25.5 Shell Aliases

- It is often useful to have bash aliases for commands like `ls` and `ls -l`, perhaps adding options:

```
alias "l=ls --color=auto -F"
alias "ll=l -l"
```

- The `alias` command with no arguments will show a list of currently defined aliases
- To show what one particular alias is set to, pass the name to `alias` without setting it to anything:

```
alias l
```


25.6 Setting Up Home Directories for New Accounts

- When a new user account is created, a new home directory is also made
- Each new home directory is populated with a skeleton set of configuration files
  - These are copied from */etc/skel* by the `useradd` command
- Setting up these files with useful defaults can make life easier for new users
- Linux distributions usually have a simple */etc/skel* directory with a few files in

25.7 Exercises

1. **a.** Use the shell builtin `alias` to get a list of the aliases currently defined.
   b. Define a new alias for changing to the parent directory (i.e., `cd ..`). For example, you could call it `up`.
   c. Edit your `.bashrc` file to add the alias to it permanently, so that the `alias` command is run whenever a shell starts.
   d. Login as root and look in the directory */etc/skel* to find out what configuration files a new user will get.
   e. Create a text file called *.signature*, which is the signature appended to emails you send.
2. **a.** Use `echo` to print the current value of the `$PS1` environment variable.
   b. Try setting it to something different. You might like to try putting some of the special `\` sequences in, but remember to use single quotes, so that the backslashes are not interpreted by the shell.
   c. Decide how you would like your prompt to appear, and edit your *.bashrc* file to set `PS1` every time you start a shell.

Module 26

Configure and Use System Log Files

26.1 `syslog`

- Many events that occur on a Linux system should be logged for administrative purposes
- Linux has a facility called `syslog` that allows any service or part of the system to log such events
- `syslog` can be configured to log different events to different places
  - Events can be selected based on severity ('level') and/or on the service that encountered the event ('facility')
  - Messages can go to files, to the system console, or to a centralised `syslog` server running on another machine

26.2 */etc/syslog.conf*

- `syslog`'s configuration is in */etc/syslog.conf*; each line looks like:

  ```
  facility.level    destination
  ```

- The *facility* is the creator of the message — one of `auth`, `authpriv`, `cron`, `daemon`, `kern`, `lpr`, `mail`, `news`, `syslog`, `user`, or `local0` through `local7`
- The *level* is a severity threshold beyond which messages will be logged — one of (from lowest to highest): `debug`, `info`, `notice`, `warning`, `err`, `crit`, `alert`, `emerg`
- The `destination` indicates where messages selected by the *facility* and *level* will be sent
  - Normally the name of a log file (under */var/log*), or */dev/console* to send messages to the system console

26.3 Sample */etc/syslog.conf*

```
# Log all kernel messages to the console.
# Logging much else clutters up the screen.
kern.*                                  /dev/console

# Log anything (except mail) of level info or higher.
# Don't log private authentication messages!
# Notice that we separate message selectors with a semicolon.
# Note the use of severity "none" to exclude a facility.
*.info;mail.none;news.none;authpriv.none    /var/log/messages

# The authpriv file has restricted access.
authpriv.*                              /var/log/secure

# Log all the mail messages in one place.
mail.*                                  /var/log/maillog
```

26.4 Reconfiguring `syslog`

- If you change */etc/syslog.conf*, you need to tell `syslog` to re-read the configuration
- Accomplished by sending the `syslogd` process a `SIGHUP` signal
- The process id to send the signal to can be found with the `pidof` command:

  ```
  # kill -HUP $(pidof /sbin/syslogd)
  ```

- Alternatively, use the `killall` command to kill the `syslogd` process by name:

  ```
  # killall -HUP /sbin/syslogd
  ```

26.5 Examining Logs: `less` and `grep`

- You sometimes need to manually scan log files for notable activity
- Since logs are plain text, you can use standard text-processing tools like to examine them
- To review the entire contents of a log file:

  ```
  # less /var/log/messages
  ```

  Note: you may need to be `root` to do this
- To look for messages on a certain topic:

  ```
  # grep -i sshd /var/log/messages
  ```

  Looks for messages from `sshd`, the Secure Shell server

26.6 Examining Logs in Real Time: `tail`

- It is sometimes useful to keep an eye on new messages arriving in a log file
- The `-f` option to `tail` will watch the file forever:

```
# tail -f /var/log/messages
```

- Continually updates the display as new messages are appended to the end of the file
  - Kill it with `Ctrl+C` when you're done

26.7 Log Rotation

- `syslog` will normally allow log files to grow without bound
  - Until you run out of disk space...
- The solution is to use **log rotation**: a scheme whereby existing log files are periodically renamed and ultimately deleted
  - But `syslog` continues to write messages into the file with the 'correct' name
- Most Linux systems come with a program called `logrotate`, which should be run daily by `cron`
- `logrotate` can be configured with */etc/logrotate.conf* to perform rotation on any or all log files
  - You can choose for each file how often it is rotated and how many old logs are kept

26.8 Sample */etc/logrotate.conf*

```
# Gzip rotated files by default
compress

# Keep 5 weeks' worth, and restart syslogd after rotating
/var/log/messages {
   rotate 5
   weekly
   postrotate
      killall -HUP /sbin/syslogd
   endscript
}

# Keep 1 month's worth.  Specify ownership and permissions of
# the new file.
/var/log/wtmp {
   rotate 1
   monthly
   create 0664 root utmp
}
```


26.9 Exercises

1. **a.** Log on to your machine as root and use `less` to browse through */var/log/messages*.

   b. Start monitoring the file for additions using `tail`.

   c. In another terminal, logged on as a normal user, try using `su` to change to root (find out what is written to the logs when correct and incorrect passwords are given to `su`).

   d. Look at the configuration file for `logrotate` to find out how */var/log/messages* is rotated (some systems will have the configuration in a file in */etc/logrotate.d*.

Module 27

Automate and Schedule System Administration Tasks

27.1 Running Commands in the Future

- There is sometimes a need for commands not to be run immediately, but scheduled to run later
- One-off commands:
  - "At 10:00 tomorrow, e-mail me this reminder message."
  - These are known as **at commands**
- Regularly repeating commands:
  - "Every night, rebuild the database used by the `locate` command."
  - These are known as **cron jobs**

27.2 At Commands

- At commands are defined using `at`:

```
$ at 16:30
at> pstree > processes
at> <EOT>
```

- The time the command should run is given as a parameter to `at`
- `at` then prompts for the command itself
  - Command(s) exactly as they would be typed in the shell
  - Press `Ctrl+D` to finish
- The **at dæmon** will run the command at the specified time
  - In this example, the output of running `pstree` at 16:30 will be saved in the file *processes*

27.3 Commands Run by the At Dæmon

- A command executed by the at dæmon:
  - Has the permissions of its owner
  - Runs in the directory it was set up
  - Has the environment in which it was set up
  - Does not run in a terminal
- Output from the command:
  - Cannot be included in a terminal window
  - Will be mailed to its owner

27.4 At Command Specification

- A command may be specified on standard input instead of interactively
- From a file:

```
$ at 16:30 < monitor_processes.sh
```

  - The commands contained in the file *monitor_processes.sh* are run at 16:30

27.5 Opening Windows from At Commands

- The `$DISPLAY` environment variable is not provided in at commands' environments
- This needs to be set for an at command to be able to open windows
- Discover the current value and type it in again:

```
$ echo $DISPLAY
beehive:0
$ at 11:00
at> DISPLAY=beehive:0 xclock &
at> <EOT>
```

- Use interpolation to embed it in the command:

```
$ echo "DISPLAY=$DISPLAY clock &" | at 11:00
```


27.6 At Command Date & Time Specification

- Unadorned times are in the next 24 hours:

```
$ at 09:30
```

- Tomorrow can be specified explictly:

```
$ at 17:00 tomorrow
```

- A specific date can be used:

```
$ at 11:00 Nov 11
$ at 00:30 16.04.06
```

- Relative times can be specified in minutes, hours, days, or weeks:

```
$ at now + 45 minutes
$ at 16:00 + 3 days
```

27.7 Managing At Commands

- `atq` lists any pending at commands:

```
$ atq
38      2002-01-16 11:00 a
```

  - The number at the start of each line identifies that at command

- A particular at command can be displayed with `at -c`:

```
$ at -c 38
#!/bin/sh
umask 2
cd /home/simon || { echo 'Bad directory' >&2; exit 1 }
echo 'Check the download has completed.'
```

  - Real at commands include the environment too

- Remove an at command with `atrm`:

```
$ atrm 38
```

27.8 Simple Cron Job Specification

- The simplest method for specifying cron jobs is to save each job as a separate file in an appropriate directory:
  - */etc/cron.daily/* is for jobs to be run daily
  - Once a day, each file in that directory is run
  - The files are typically shell scripts
  - There are equivalent directories for monthly, weekly, and possibly hourly jobs
  - Each job is run with root permissions
- Normally only root can set up cron jobs this way
- Any required environment variables must be set explicitly

27.9 More Complex Cron Job Specification

- Sometimes more control is needed:
  - To run jobs at a non-standard time
  - To run jobs as a user other than root
- The directory */etc/cron.d/* is for this purpose
- Each file in that directory must contain lines in a specific format:
  - When the command should run
  - For which user the command should be run
  - The command to be run
- Such a file is known as a **cron table** or **crontab**
  - Details are in *crontab(5)*
  - Easier to have one file per job

27.10 Crontab Format

- Blank lines are ignored
- Comments are lines starting with a hash (#)
- Environment variables can be set:

```
PATH=/usr/local/bin
```

- Example cron job specification

```
30 9 * * *  root  /usr/local/bin/check_logins
```

  - At 09:30
  - On all days
  - For the root user
  - Run the command `/usr/local/bin/check_logins`

27.11 Crontab Date & Time Specification

- Order of the date and time fields:
  - Minute (0–59)
  - Hour (0–23)
  - Day of the month (1–31)
  - Month (1–12)
  - Day of the week (0–7; 0 and 7 are Sunday)
- Note: Fields *almost* in ascending order
- The command is run when the fields match the current time
- A field containing an asterisk (`*`) always matches
- Three-letter abbreviations can be used for month and day names

```
# Run every Friday night at 17:30:
30 17 * * Fri root /usr/local/bin/weekly-backup
```

27.12 More Complex Crontab Dates & Times

- A list of alternative values for a field are specified by commas:

```
# Run at :15 and :45 past each hour:
15,45 * * * * httpd /usr/local/bin/generate-stats-page
```

- A range is specified with a hyphen:

```
# Run every half hour 09:15-17:45 Mon-Fri:
15,45 9-17 * * 1-5 root /usr/local/bin/check-faxes
```

- Numbers rather than names must be used for months and days in lists and ranges
- A step through a range is specified with a slash:

```
# Run every two hours 08:30-18:30 Mon-Fri:
30 8-18/2 * * 1-5 root /usr/local/bin/check-faxes
```

27.13 */etc/crontab*

- The */etc/crontab* file is an older way of specifying cron jobs
- Each job in that file is like a file from */etc/cron.d/*
- Having many unrelated cron jobs in a single file is much harder to manage
- This file may be the mechanism by which your system runs the contents of */etc/cron.daily/* and friends
- There is no need to use this file for anything else

27.14 User Crontabs

- Sometimes non-root users need to set up cron jobs
- Each user has a crontab file
  - This is not edited directly
  - The `crontab` command manipulates it
  - Use `crontab -e` option to edit the crontab
    - The editor in the `$EDITOR` variable is invoked for this
  - Use `crontab -l` to display the crontab
- The format is very similar to that of */etc/rc.d/* crontabs
  - But there is no username field
  - All commands run as the owner of the crontab

27.15 Cron Job Output

- Cron jobs do not run in a terminal window
- Generally they are administrative tasks designed not to produce any output when run successfully
- Any output that is generated by a cron job is mailed:
  - The recipient can be specified in the `$MAILTO` environment variable
  - Otherwise mail is sent to the job's owner
  - Jobs in */etc/cron.daily et al* are owned by root

27.16 At Command and Cron Job Permissions

- Non-root users can be prohibited from having crontabs
  - If */etc/cron.allow* exists then only users listed in it may have a crontab
  - If it doesn't exist but */etc/cron.deny* does, then users not listed in the latter may have a crontab
  - If neither exist, then all users may have crontabs
- Permissions for running at commands are similar:
  - The files */etc/at.allow* and */etc/at.deny* are analogous
  - If neither file exists then no users may run at commands
  - If only */etc/at.deny* exists but is empty then all users may run at commands

27.17 Exercises

1. a. Set up a command which in a couple of minutes' time will write the disk usage of the partition containing */home/* to *~/home.use*.

   b. Change to your home directory. Set up a command which in ten minutes' time will write the disk usage of the partition containing the current directory to a file.

   Repeat the above for two more directories in other partitions, writing to separate files.

   Before the time is up, display the list of pending commands.

   Examine each one. How do they differ?

2. a. Set up a command which will mail you a reminder message in a few minutes. Remember that output from a job run by the at dæmon will be mailed to its owner, so there is no need to invoke an explicit mail command.

   Check that the mail arrives. *Mutt* is a simple mail reader you can use to do this.

   b. Make the `xeyes` command start in a minute's time, capable of opening window's on your screen. Remember that any error messages from a command failing to run will be mailed to you.

   c. Make a clock appear on your screen at four o'clock this afternoon.

3. a. Create a file containing a command which will delete all files in *~/tmp/* when it is run. Make that script be run every hour.

   b. Set up a script that once a minute appends the following data to a file:

   - The current date and time
   - A count of the number of processes currently running

   Monitor that file, and start and stop programs to affect the number reported. When you've finished, deactivate the script.

   c. As a non-privileged user, set up two scripts:

   - In even-numbered minutes, write the time the system has been running to a particular file.
   - In odd-numbered minutes, delete that file.

4. Set up scripts to run as a non-privileged user at the following times; you won't be able to test most of them, but it gives you practice at getting used to the order of the fields. Add the jobs one at a time, so that your crontab will be parsed to check for errors after each one.

   - At 09:25 every Sunday in December
   - At 21:30 on the 1st and 15th of every Month
   - Every three hours at weekends
   - Half past every hour during weekdays
   - The first day of every quarter
   - The evening of the first and third Thursday of each month
   - At 17:30 every other day

Module 28

Maintain an Effective Data Backup Strategy

28.1 Reasons for Backup

- Disks fail
- Bugs in software can cause corruption
- Configuration mistakes by administrator
- Accidental deletion or overwriting (e.g., with `rm`, `mv` or `cp`)
- Malicious deletion or virus attack
- Theft of machines with hard drives in
- Fire, or other disasters which can destroy hardware

28.2 Backup Media

- Traditionally, backups have been made onto tapes
  - Can store lots of data on reasonably cheap tapes
- Copying to a different hard disk
  - There is a risk of losing the backup along with the original
  - Better if on a remote computer
- CD writers can be used to store backups on CDs
  - Convenient for long-term storage
  - Handy to remove to remote locations

28.3 Types of Backup

- **Full backup** — includes everything of importance
  - Might not include system files which are from the install CD
  - Can include a lot of files, many of which hardly ever change
- **Differential backup** — only includes changes since last *full* backup
  - Nightly backup only needs to include files changed since the last full backup
  - Recovery requires the full backup on which it was based
- **Incremental backup** — only includes changes since last backup
  - Nightly backup only includes files changed in the last 24 hours
  - Recovery requires the last full backup and a complete sequence of incremental backups after that

28.4 Backup Strategy

- The backup schedule should be regular and well known by those who rely on it
  - It must be decided what to backup and what can be left out
- Typically a full backup is done once a week or once a month
  - Daily changes are recorded in a differential or incremental backup each night
- Large sites might have more than these two levels to their strategy
- Monthly tapes might be kept for a long time, in case a really old file becomes important

28.5 Archiving Files with `tar`

- `tar` can package up files for distribution or backup
  - Originally for “tape archive” manipulation
  - Files can be stored anywhere
- Encapsulates many files in a single file
  - Known as a **tar archive** or a **tarball**
- Has unusual command-line option syntax
  - Common options are given as single letters
  - No hyphen is needed
- `tar` must be given exactly one action option
  - Indicates which operation to perform
  - Must be the first option

28.6 Creating Archives with `tar`

- Use the `c` option to create an archive
- For example, to create an archive called *docs.tar.gz* containing everything in the *documents* directory:

```
$ tar czf docs.tar.gz documents
```

  - `f` specifies the archive's filename
    - Must be followed directly by the filename
    - Common to use *.tar* extension
    - Any subsequent options require a hyphen
  - The `z` option compresses the archive with `gzip`
    - *.tar.gz* extension used to indicate compression
    - *.tgz* extension also popular
  - The list of files and directories to archive follows the options.

28.7 Listing the Files in `tar` Archives

- To check that a `tar` file has been made correctly, use the `t` operation (for 'list'):

```
$ tar tzf docs.tar.gz
```

- The `z` and `f` options work as for the `c` operation
- To show more information about files, add the `v` (for 'verbose') option
  - Shows information similar to `ls -l`
  - Can also be specified with `c` to list filenames as they are added

28.8 Extracting Files from `tar` Archives

- Use the `x` operation to extract files from an archive:

```
$ tar xzvf docs.tar.gz
```

- The `v` option lists the files as they are extracted
- To extract individual files, list them on the command line:

```
$ tar xzvf docs.tar.gz documents/phone-numbers.txt
```

- Other useful options:
  - `k` (`--keep-old-files`) will not overwrite any existing files, only extracting missing ones
  - `p` (`--preserve-permissions`) will set extracted files to have the permissions they had when archived

28.9 Device Files for Accessing Tapes

- Under Linux, tape drives are accessed through several groups of device files
- Each device group has number, with the first drive numbered 0
- These are the most commonly used devices:
  - */dev/st0* — SCSI tape drive, which will be automatically rewound after each operation
  - */dev/nst0* — the same drive, but with no automatic rewinding
  - */dev/ft0* — floppy tape drive
  - */dev/nft0* — the same without rewinding
  - */dev/ht0* — ATAPI tape drive
  - */dev/nht0* — the same without rewinding

28.10 Using `tar` for Backups

- Tape drive devices can be read and written directly by `tar`
- To write a backup of */home* to the first SCSI tape drive:

```
# tar cvf /dev/st0 /home
```

- We haven't used compression (the `z` option)
  - This might make the backup slower, at least on less powerful machines
  - Compressing the whole archive would make it much less resilient against corruption
- In the example the auto-rewinding device is used, so the tape will be rewound after `tar` is finished, and the archive can be extracted:

```
# tar xvf /dev/st0 /tmp/restored-home
```

28.11 Controlling Tape Drives with `mt`

- `mt` can move tapes backwards and forwards, and perform other operations on them
- Usage: `mt [-f device] command [count]`
- The `-f` option sets the tape device to use (e.g., */dev/st0*)
  - The default is usually */dev/tape*, which should be a symlink to a non-rewinding device, like */dev/nst0*
- These are some of the more common commands:
  - `fsf`, `bsfm` — move forwards and backwards one (or *`count`*) files
  - `eod` — go to the end of the valid data
  - `rewind` — go to the start of the tape
  - `offline` — eject the tape

28.12 Deciding What to Backup

- Being selective about what is included in the backups can drastically reduce the time and space taken
- For example, */bin*, */sbin*, */lib* and */usr* could be restored from an installation CD
  - But it might still be worth backing them up to make restoration simpler
- The things which are most likely to be important in backups are:
  - */home*
  - The CVS repository, or other places where project work is stored
  - Some directories under */var* (particularly email)

28.13 What Not to Backup

- Some other areas which shouldn't be backed up are:
  - */tmp* — usually doesn't contain anything of lasting value
  - */proc* — automatically generated by the kernel
  - */dev* — if using devfs this is also generated automatically
  - */mnt* — some media mounted here, like CD ROMS, typically aren't backed up
  - Filesystems mounted remotely whose backup is taken care of elsewhere

28.14 Scripting Backup

- It is common to have a script to perform backups each night
  - Might perform different types of backup, e.g., a full backup on Saturday night and a differential one on other nights
- Such a script can be run with `cron`, making backup automatic
- Example scripts are readily available on the WWW

28.15 Other Backup Software

- `cpio` — alternative archiving program
- `afio` — similar to `cpio`, but allows files to be compressed individually
  - Compressed archives can be made which are resilient to corruption
- `dump` and `restore` — access the filesystem directly, rather than through the kernel
  - Software is specific to filesystem types (versions available for ext2/ext3)
  - Can preserve things which `tar`, etc., cannot
    - File access times (`tar` has to access the file to archive it)
    - Special files which archiver programs might not understand

28.16 Exercises

1. a. Create a single file in your home directory containing a backup of all the contents of */etc/*.
   b. Create another archive with the same contents, but compressed to save disk space. Compare the sizes of the two archives.
   c. List the contents of each of your archives.
   d. Extract the contents of one of the archives into a directory under your home directory.
   e. Create a new subdirectory, and extract a single file from the archive into it.
      - Your current directory must be the one into which to extract.
      - You will need specfiy the path of the file to be extracted, but without a leading slash.
2. With an archive of */etc/*, extracted under your home directory:
   a. Modify at least two files in your extracted copy. Re-extract one of them from the archive, losing your changes in that one file but preserving your changes elsewhere.
   b. Delete some files in your extracted copy. Make tar discover which these are and re-extract them from the archive, without clobbering changes made to other files.
3. a. Produce a list of the names of all files under */home/* which have been modified in the past day. Only include regular files in this list.
   b. Create a tarball containing all files under */home/* changed modified in the past day. Why is including directories in this list not sensible?
   c. Create a tarball containing all files on the system that have changed in the past day, and which are in directories you deem worthy of being backed up.
   d. Set up a cron job to make a daily incremental backup of the system.
      - It should run at 18:00 every day.
      - The files created should be stored under */var/tmp/backup/*.
      - Each day's backup should be in a file named with that day's date, such as */var/tmp/backup/2003/04-07.tgz*

