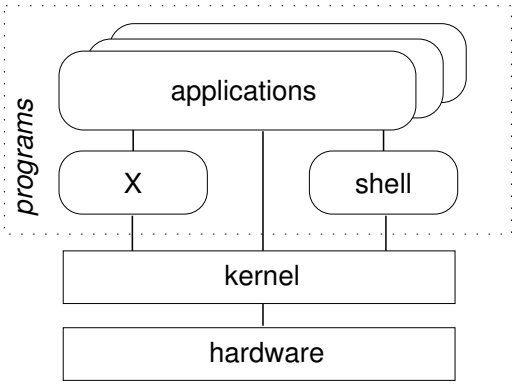
Module 1

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.

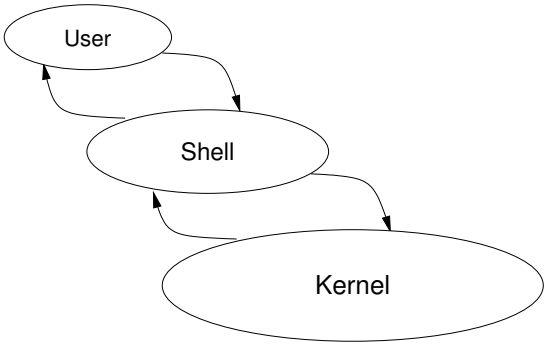
   e. Finally, use the `rm` command to delete the *poetry* directory and the poems in it.

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:

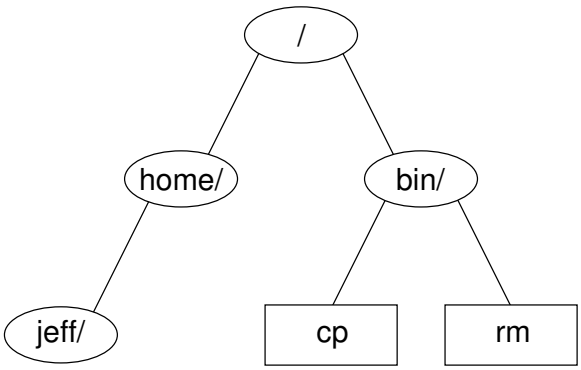
```
$ chmod a+rx my_ls
```

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
   ```

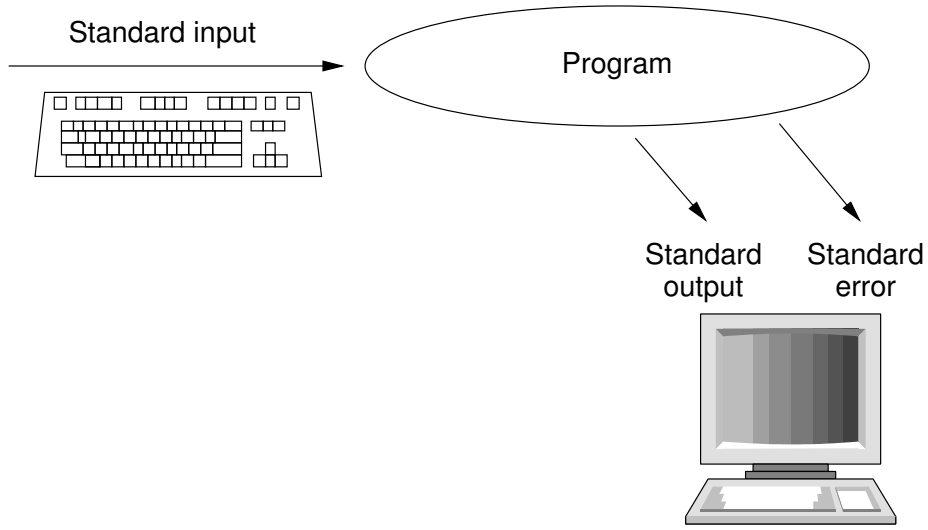
   c. Wait for a minute, and then try the previous two steps again, and see what changes. What happens when we don't specify a time to `touch`?
   d. Try setting the timestamp on the file to a value in the future.
   e. When you're finished with it, delete the file.

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`

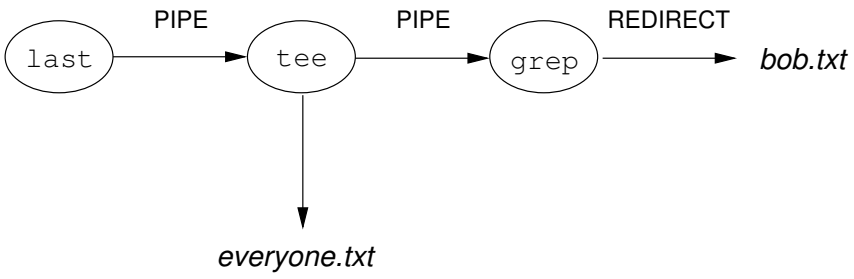
- `xargs` reads pieces of text and runs another program with them as its arguments
  - Usually its input is a list of filenames to give to a file processing program
- Syntax: `xargs` *`command`* `[`*`initial args`*`]`
- Use `-l` *n* to use *n* items each time the command is run
  - The default is 1
-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