Operating systems fundamentals - B02

David Kendall

Northumbria University

Introduction – Getting started with Linux

- How the operating system starts running
- Terminal, console, shell
- The command line - why bother?
- The file system - managing files
- I/O redirection
- Advanced use of the terminal - tmux

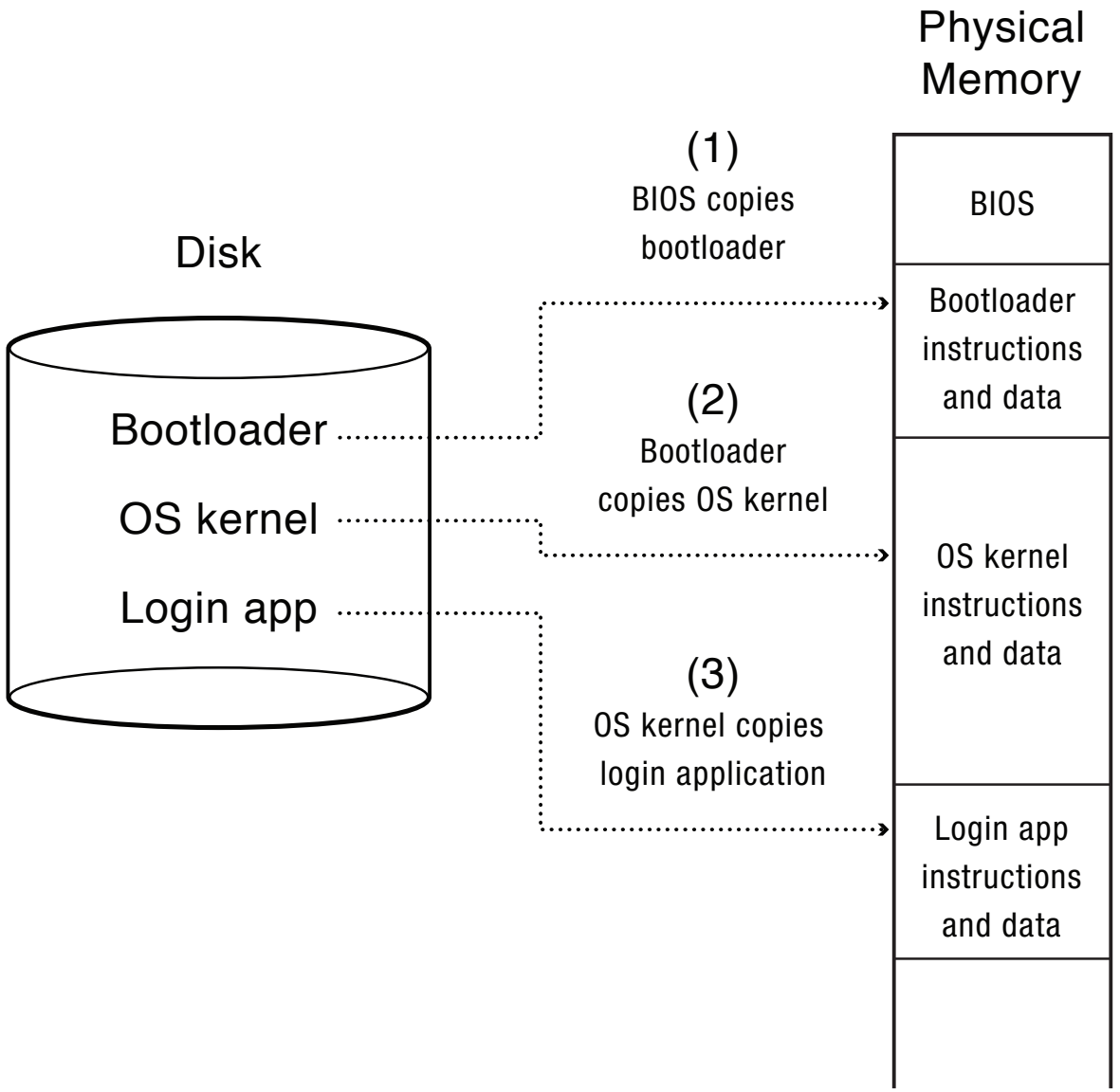
How the operating system starts running

- Turning on the computer (or pressing the reset button) forces the CPU to begin executing instructions from a fixed location in the computer memory
- The contents of memory starting at this location need to be *non-volatile*, i.e. to survive a power down/power on – usually use *ROM* (Read Only Memory).
- The ROM that stores the program used to start the operating system is called the *boot* ROM. On most PCs, the boot program is called the *BIOS* (Basic Input/Output System)
- Don't store the complete OS in the boot ROM
  - ROM is slow and expensive compared to RAM
  - ROM is hard to update
  - OS needs frequent updates

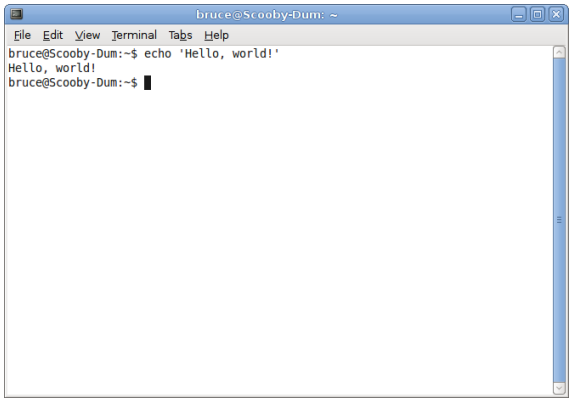
How the operating system starts running

- So load the operating system in stages:
  - The boot ROM contains a small program that is able to read a fixed-size block of bytes from a fixed position on the disk (*the bootloader*)
    - Note the boot program doesn't need to know about the file system – it just has to be able to read a block of raw bytes from a known location
    - The bootloader may have a *digital signature* to ensure that it hasn't been tampered with
  - Once the BIOS has loaded the bootloader into memory, it jumps to its first instruction and starts executing code from there
  - The job of the bootloader is to load the OS kernel into memory
    - The OS kernel is usually stored in the file system on disk, so the bootloader needs to know how to find a file in the file system and read it.
  - Once the kernel has been loaded, it can initialise its data structures and then start the first process (called *init* in Linux), which can start a login process so that the user can login and start work.

How the operating system starts running



Terminal, Console, Shell



- DEC VT220 terminal, popular in the 1980s
- A physical terminal used to communicate with a mini or mainframe computer over a serial communication link
- A console was the keyboard/monitor directly connected to the console port of the computer
- Gnome terminal – a *virtual terminal* that *emulates* the behaviour of a physical terminal – also KDE and xterm
- The *shell* is a program that receives command input from the terminal and makes calls to the computer operating system to execute the commands. Shells include bash, zsh, csh.

Command line – why bother?

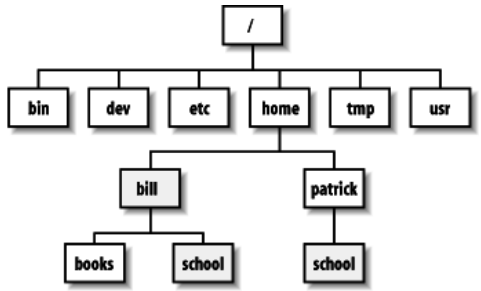
Command line advantages

- Available everywhere – many servers and small embedded systems don't have a GUI
- Uses less resources
- More efficient once you've learnt the commands – faster than clicking and scrolling, then clicking and scrolling, then clicking and scrolling some more just to do something simple
- Can easily compose commands to do complex tasks
- Can automate commands, go away and leave them running

GUI's are also good

- Browse the web using GUI
- Read email using GUI (although there are good terminal mail readers)
- Read pdf documents
- View and edit photos
- Use graphical design tools, stream video, ...

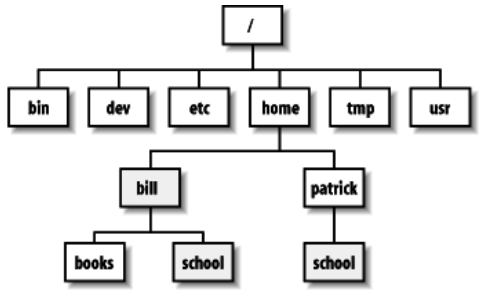
File system tree



http://etutorials.org/shared/images/tutorials/tutorial_95/rh4_0401.gif

- When we first login, we are positioned in the file system directory tree at our *home directory*, e.g. `/home/bill`
- Wherever we are in the tree is called the *current working directory*
- Every directory has a *parent directory* (except the top-level directory, `/`, called the *root directory*). The parent of a directory is the one directly above it in the tree, e.g. `/home` is the parent of `/home/bill`
- The parent of the current working directory can be referred to using the symbol, `..` (the current working directory is referred to using `.`)

File system tree



http://etutorials.org/shared/images/tutorials/tutorial_95/rh4_0401.gif

pwd where am I? (**p**rint **w**orking **d**irectory)

cd go somewhere else (**c**hange **d**irectory)

ls what's in here? (**ls**t contents of directory)

- A *path name* is a description of the location of a directory in the file system tree
- We can use *relative* path names, which start at the current working directory, e.g. `ls books`
- or *absolute* path names, which start at the root of the file system, e.g. `ls /home/bill/books`

Useful commands for working with files

ls lists contents of directory
less display contents of text file
file indicate file type
cp copy files and directories
mv move/rename files and directories
rm remove files and directories

- Command format: `command` *-option arguments*
- e.g. `ls -l /home/bill`

man look up the manual entry for a command

- e.g. `man ls`

Wildcards

Using *wildcard* characters can make the use of the file commands even more powerful

Wildcard	Matches
`*`	Any characters
`?`	Any single character
`[` *characters* `]`	Any character that is in the set *characters*
`[ !` *characters* `]`	Any character that is *not* in the set *characters*

Pattern	Matches
`*`	All files
`s*`	All files beginning with `s`
`f*.txt`	All files beginning with `f` followed by any characters and ending in `.txt`
`Log??`	All files beginning with `Log` followed by exactly two characters
`[xyz]*`	All files beginning with `x`, `y`, or `z`

I/O Redirection – Output

- Most the commands that we've used generate some *output*
- Sometimes the output is the data that we were after, sometimes it's an error message or status information
- All output is sent to a file: good data is usually sent to the *standard output* file, error messages etc. are sent to the *standard error* file
- Usually the standard output file and the standard error file are both mapped to our display, so we see the output appearing on the screen
- We can choose to *redirect* the standard output to a different file, e.g.

```
ls -l /home/bill/books > ls_output.txt
```

- We can also redirect standard error, e.g.

```
ls -l /bill/home/books 2> ls_errors.txt
```

- We can redirect standard output and standard error to the *same* file, e.g.

```
ls -l /home/bill/books > ls_output.txt 2>&1
```

I/O Redirection – Input

- Just as the standard output and standard error files are usually mapped to the display, the *standard input* file is usually mapped to the keyboard
- `cat` is a program that normally reads data from the standard input (keyboard) and sends the data to the standard output (display), e.g.

```
$ cat
Much have I travelled in the realms of gold
Much have I travelled in the realms of gold
```

- We can redirect the output as usual, e.g.

```
$ cat > gold.txt
Much have I travelled in the realms of gold
```

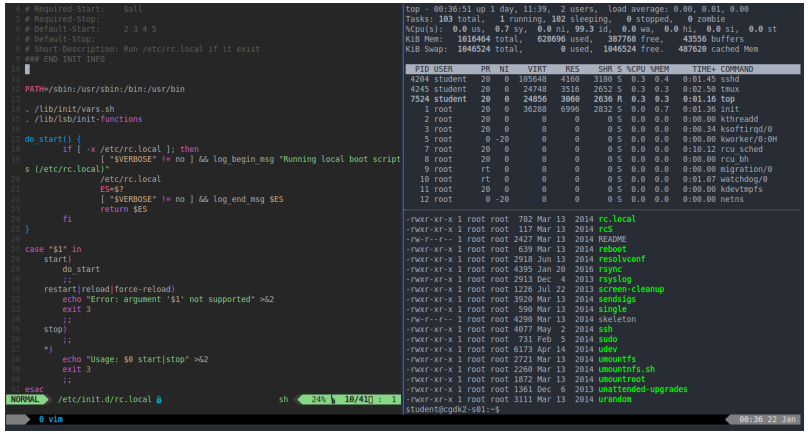
- We can also redirect the *input*, e.g.

```
$ cat < gold.txt > keats.txt
```

Next steps . . .

- We've just scratched the surface of the capabilities of the command line
- You can find out more by reading chapters 6 and 7 in the The Linux Command Line
- We'll also be covering additional commands in the lab session this week and in later sessions of the module as appropriate
- The key steps to making progress are to read more and to play around with a Linux system as much as possible

Advanced use of the terminal: tmux [Optional]



- **tmux** – **t**erminal **mu**ltiple**x**er
- create sessions, attach and detach from a session, leaving it running exactly as it was
- create multiple windows
- create multiple panes within a window
- Makes it more convenient to use a terminal to manage a remote server
- There's a nice tmux blog post and a tmux home page.