# Lecture 0: Intro

Timetable:

* Tuesdays – C
* Thursdays – Linux

Assessment:

* Formative Class Test (Optional)
  + Results in following exam
* Practical Exam (no resit, March) (25%)
* Labs (25%)
  + C and Linux components, both required
* Final Written Exam (50%)

# Linux Lecture 1: Intro

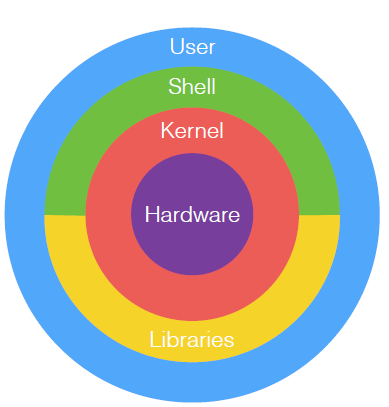
Brief History

* Centralised computing (only available to rent from big companies like IBM)
* Kenneth Lane Thomson (1943-)
  + Received his Bachelor’s and Master’s degrees in E&EE from the University of California at Berkley
  + In 1966, hired by Bell Labs (AT&T) to work on MULTICS
    - MULTICS (Multiplexed Information and Computing Service) was a mainframe time sharing OS
  + Worked on developing the OS until Bell LABS withdrew from the consortium in 1969
  + Space Travel
    - While working on MULTICS, he created a game called Space Travel
    - It allowed the player to travel around a 2D solar system
    - Originally intended for MULTICS, Ken rewrote it to run on an obsolete PDP-7 located in the lab
    - Ken wanted a better system to run his game on, in 1969, he took 1 month to write:
      * kernel
      * shell
      * editor
      * assembler
  + Birth of UNIX
    - Bell Labs left the MULTICS consortium in 1969
    - Bell Labs purchased the new PDP-11, and development continued on what would become UNIX
    - 1970: Ken created programming language ‘B’ (precursor to C)
    - 1972: rewrote UNIX kernel in C with Dennis Ritchie
    - 1973: UNIX presented to the world
  + UNIX?
    - Multi-user, multi-tasking OS (at present, a family of OSs)
    - Any system has key features:
      * use of plain text for data storage
      * hierarchical file system
      * devices and processes represented as files
      * use of a large number of software tools, small programs that can be composed as opposed to using a single monolithic program
    - 2015: certification from OpenGroup needed to be called UNIX
* Linus Benedict Torvalds (1969-)

Why Linux?

* **Free**
* Source code is freely available and can be modified by anyone
* Multi-user, multi-tasking OS based on tested principles of UNIX
* Ubiquitous in scientific computing (all top 500 supercomputers run Linux)
* Scientific code bases for Cosmology, Particle Physics, Nuclear, Atmospheric, Fluid Dynamic all run on Linux
* Ability to port to many platforms and architectures
  + From embedded devices like Raspberry Pi to companies like Google and Facebook
  + Phones (Android), routers (dd-wrt), TV, cars (BMW), Munich, Facebook/Google/DropBox
  + Internet services

Parts of an OS:

* Linux Architecture
  + 
  + **Hardware**
    - CPU, GHz; Numeric / Logic / Shift / Vector Operations
    - Memory; GB; running software and data storage
    - Network; Gbit; data transmission, TCP/IP packet decoding
    - Storage; TB; data storage preserved without power requirements
    - Graphics; MOps; VGA display, 2D/3D acceleration, floating point operations
  + **Kernel** – core part of Linux, includes all that drivers need to interact with the underlying hardware
    - Looks after most interactions with Hardware and all of the processes running on the system
    - Process scheduler
    - Inter-Process Communication
    - Memory Management
    - Virtual Filesystem
    - Networking
    - Device Mapper
    - Sound Subsytem
  + **Shell** – allows interactive access to systems resources through a set of built-in commands or by running commands found on the filesystem
    - In Linux, handles interaction between users and system
    - Can be via text interface (command line) or through a graphical interface (e.g., gnome-shell)
    - Allows other process/apps to be started
  + Access to system resources can be via system **libraries** that expose access to the underlying kernel (c.f. glibc)
    - Running programs also need to interact with system resources
    - Wide range
    - Used in this course: standard C libraries
  + **User** applications will usually be started by the shell or by the init process at system start-up
    - Final level of OS
    - Ex: web browsers, office suites, editors, etc
* Boot process
  + System start
    - Power On Self-Test (POST)
      * Verify BIOS code
      * Verify CPU and Registers
      * Find and Verify Memory
      * Initialise the BIOS
      * Discover and initialise Hardware
  + 1st stage boot loader
    - Master Boot Record
      * 512 bytes located on first sector
      * small program that loads 2nd stage
  + 2nd stage boot loader
    - loads kernel
      * loads required components into memory
      * enters protected mode
      * transfers control to the kernel
  + kernel
    - Starts and brings up system services. Once devices have been initialised and important subsystems started, starts init services
  + init

Command Line:

* Large variety: Sh, BASh, CSh, TCSh, ZSh, …
* C-style shells: C-like syntax for scripting
* Bash-style shells: slightly more consistent, more widely adopted in research community
* Using:
  + Form:
    - prompt# <command> <flags> <arguments>
    - prompt# du -h –max-depth=1
  + Can be chained together by piping
  + Useful:
    - history
      * gives all commands you’ve entered
    - !! – runs last command
    - !<number> - runs the command at <number>
    - [up/down arrow] – cycles through previous commands
    - [tab] – auto completes names
    - man – gives manual page for command
      * [spacebar] – next page
      * [q] – quit
    - [CTRL+R] – search through previous commands
    - clear – empties screen

# Lecture 2: Filesystem

## Filesystem

On Linux, everything is either a file or a process

Hierarchical – a tree-like structure that begins at the ‘root’ nodes

* Each node in the tree can be a file or directory (or link, socket, …)
* Directory – special file that contains a list of other files (and directories)
* Each “file” has a name (case sensitive)
* Each has a unique identifier (**inode**)

File types:

* Regular (-) – standard text files
* Directories (d) – Lists of files
* Links (l) – allows another file to appear in multiple locations
* Special File (c) – IO sources
* Sockets (s) – Inter-process networking
* Named Pipes (p) – Allows process to communicate with another

File Paths

* Absolute
  + The location of each file is given by its path
  + Starts from root node (/)
  + Each time a directory is descended, add additional ‘/’
* Relative
  + Reference files at other parts of the file system

Hidden Files

* Don’t appear in a directory
* All begin with ‘.’ character
* Often used for configuration files
* . – current directory
* .. – parent directory

Standard locations

* /etc
  + Holds a lot of configuration files
* /dev
  + holds links to the various pieces of hardware being managed by the kernel
* /proc
* /var
  + variable data (logging files, mail, printer spools)
* /usr
* /home
* /bin
* /sbin

## Navigating

Look around:

* pwd – current position in filesystem
* tree – prints out complete tree of current location
* ls – lists contents of current directory
  + ls -a – lists hidden files, too (including . and ..)
  + ls -l – lists in long format (with permissions)

File Globbing

* Searching files with wildcards
  + \* - any symbol (and amount)
  + ? – single character
  + [] – single characters of a choice in the brackets
  + ^ - negation of a character

Moving Around

* cd – change directory

Finding Files

* find
* locate (faster, but less reliable if file is new)
* whereis – returns the location of programs and manages
* which

## Users & Permissions

General:

* Multi-user OS
* Each User has a unique ID (uid) and is associated with a set of groups. Each group also has a unique id (gid)
* What a user can do is restricted by the rights associated with each user or group
* whoami – shows current user
* id – current id and group info
* Root/super user

Permissions

* All restrictions are handled at file level
* Each file/dir has the notion of ownership by a user/group
* Three types
  + User
  + Group
  + Other
* Permission types
  + r – read access
  + w – write access
  + x – execute
* Changing:
  + chmod [who][op][what] filename
  + Who:
    - u – user
    - g – group
    - o – other
    - a – all
  + Op:
    - + add
    - - remove
  + What: rwx

## Working with Files

Common Tasks

* Create, move, copy, delete, look at contents, look at beginning

Creation

* touch – create a file
* mkdir – make a directory
* {f1,f2,f3} makes all

Moving

* mv <src> <dest>
* Globbing possible

Copying

* cp <src> <destination>

Deleting

* rm <filename>
  + -r has to be used when deleting directories with content

Looking at file

* file <filename>
* cat <filename>
* less <filename>
* Bits:
  + head <filename>
  + tail <filename>
  + Ability to follow changes

Searching

* grep <expression> <filename>

Other useful commands:

* wc

# Lecture 3: Processes

## Variables and the Shell

Variables

* The “shell” stores parameters in variables (e.g., bash)
* To assign a value to a bash variable on the command line,
  + export SOME\_VAR=”some value” (no spaces when assignment)
* Variables store configuration info
* Most environment variable names are in uppercase
* **env** or **export** – all variables set in a running shell
* To access a specific variable, **echo**
  + Writes to the screen
  + echo “hello”
    - puts hello
  + echo $HELLO
    - puts the content of variable
    - alt (better): echo ${HELLO}
      * able to embed in strings
* Important:
  + $PATH – the places to look for a particular command
  + $HOME – user’s home dir (alt - ~)
  + $PWD – current working dir (alt – pwd)
  + $USER – username (alt – whoami)
  + $LD\_LIBARY\_PATH – paths to search for library files when building C code

Aliases

* Typed commands made shorter
* ex: alias ll=’ls -laF’
  + Then ll will run that

Persistent Variables

* All variables in bash are transient
* Variables set in 1 shell will not persists in another
* Once a shell is opened, bash reads a number of config files to set its state
* Variables can be made persistent by adding them to one of these config files
* Bash has 2 config files in home dir:
  + .bashrc
  + .bash\_profile
* Once edited, can be loaded with **source .bashrc**

## Processes

Process

* Each program that runs on a Linux system runs inside a process
* Each process is self-contained, even if it’s for the same program
* The Linux Kernel schedules CPU time for each process to run
* Processes are created as children of other processes by “**forking**”
* **pstree**
  + shows all processes running on a system, hierarchical
* Process:
  + Contains:
    - a copy of the program
    - a section of Virtual Memory assigned to it (containing Stack and Heap)
    - a link to any open files (file descriptors)
    - a snapshot of the processor state (including register contents)
    - attributes (owner, permissions)
  + Each process is isolated from every other process in the system
* Forking
  + Creation of processes
  + Process creates a copy of itself and then starts the program it desires to run
    - Parent-child
  + Initial process – ‘init’

## Working with Processes

Seeing processes:

* **ps**
  + no arguments:
    - those running in shell
  + shows process id (pid), how long the command has been running, and the command that is run
  + **ps -e** (or **ps -ax**) will show all processes running on the system
  + other filters:
    - **ps -elf**
    - **ps -auxwf**
* **top**
  + sorts by %CPU
    - changed by typing **F** and selecting sort field
  + quit by **q**

Foreground and Background

* Foreground jobs – processes that have interactive access to the command line
* Background processes don’t have access to the interactive command line but still run
* **&** (after command) – starts in background
  + Allows to continue working in terminal for other things
  + **jobs** – check what processes are
* Job Control
  + CTRL+C - kill a job in fg
  + CTRL+Z – suspend a job in fg
    - retains use but doesn’t run/use CPU time
  + **fg** <job id> - brings job to fg
  + **bg** <job id> - puts suspended job into bg
* Killing jobs
  + **kill** <pid> - removes a process from the system
  + Find pid:
    - **ps**
    - **pidof** <name>
  + **kill -9** <pid> - kill process and all its children

## IO Streams

Standard Streams

* Each process has 3 standard input/output streams
* These come from the C standard and are built-in to every executable that has a “main” function
* **stdin** – standard source of input to the program
* **stdout** – standard output of the program, e.g., writing to the screen/shell
* **stderr** – standard error of the program, which allows errors to be handled separately from output
* Bash:
  + stdin – keyboard
  + stderr – screen
  + stdout – screen

## Pipes and Redirections

Redirection

* Standard streams can be redirected to places
* Ex:
  + redirect stdin to put a list of typed commands into a program
  + redirect stdout to save output of a program to a file for later processing
  + redirect stderr to save error messages to look at/fix later
* Commands:
  + **<** - redirect stdin from a file/another stream
  + **1>** or **>** - redirect stdout to a file/another stream
    - overwrite or create
  + **2>** - redirect stderr to file/another stream
  + **>&1** – redirect to stdout (**2>&1** to redirect stderr to stdout)
  + **>&2** – redirect to stderr
  + **>>** - append (not overwrite)

Pipes

* Takes the output of one command and uses it as input for another
* **ps -elf | grep n\_tty**
* | means ‘take stdout of 1st command and attach it to 2nd’

# Lecture 4: Bash Scripting

Shell Scripts

* Scripting allows the automation of easy, repetitive tasks
* Ex: checking webserver logs, backing up data to remote disk, creating dir structures, checking the working of an automates experiment
* Shell scripts are deceptively simple, but can be powerful when combined with other tools
* ex scripting languages: perl, python, Ruby, BASH

Scripts:

* Text file with a series of commands
* \*.sh
* #! – interpretive directive. Instructs the shell to run the command specified with the current file
  + “shebang”, “hash-bang”, “hash-pling”
  + All bash scripts start with this line
* Comments are essential
* Possible commands:
  + exit
    - returns numeric value
    - 0 – success
    - any other number – failure

Running:

* **source simple.sh**
  + runs commands in current shell
  + can be dangerous as it can overwrite variables in current session
  + will exit current session if there’s “**exit 0**” at the end
* **chmod +x simple.sh; ./simple.sh**
  + forks a new process

Variables

* Possible to assign output of a command to a variable:
  + **MYSUER=&(whoami)**
  + **MYUSER=`whoami`**
* 2 types of string
  + double (“”)
    - Variables are replaced with their values
  + single (‘’)
    - Variable substitution does not take place

Conditional Branching

* Types:
  + if x then y elif a then z
  + if x then y
  + if x then y else z
* Conditionals
  + Format: [ flag file/string ]
* ???

Ranges

* {start..stop} notation
  + ex:
    - {1..3} = 1 2 3
    - {01..03} – 01 02 03
    - {a..d} – a b c d
* **seq**

Lists

* Can specify list with brace expansion: {a,b,c,d}
* String with spaces (double or single quotes)

Arrays

* Can be accessed via indices
* ex: **MYARRAY=(a b c d)**
* Indices don’t need to be consecutive
  + Arrays in bash are more like hash maps

Loops

* For loop types:
  + for (( exp1;exp2;exp3 ))  
    do  
     command  
    done
    - C-style: init, condition, increment
    - rare
  + for i in {1..5}  
    do  
     command  
    done
* While loops

Special Variables

* $$ - process ID of the running Bash script
* $0 – name of the Bash script
* $1, $2, … - first, second, … argument passed to the script
* $# - number of arguments passed to the Bash script
* $? – return code of the previous command

# Lecture 5: Bash+, Compilation

Input:

* User
  + **read**
    - takes a variable to store or stores to $REPLY by default
    - **-p** – specifies prompt
    - **-s** – specifies that input is not printed to screen
    - **-n** – specified number of characters to read
* File
  + also **read**
    - **cat $1 | while read LINE**

Functions

* Declaring
  + label and block
  + Two ways:
    - **function** <name> {}
    - <**name**>(){}
* Way of reusing code without much repetition
* Arguments
  + $ variables in functions refer to the parameters to the function, not whole script
* Global & Local Variables
  + **All variables are global within the running script by default**
  + **local** keyword makes a variable local

Good practice

* “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”
  + Brian Kernighan
* **ALWAYS use comments**
  + Include at least a single line comment telling what the script does
* Use white space to separate ideas
  + easier to understand each section of code
* Use quoted variables (“$MYVAR”)
  + this will get around problems with Bash “word splitting”
* Always use exit statements and test sub-commands’ return codes
  + Bash tends to fail quietly

Debugging

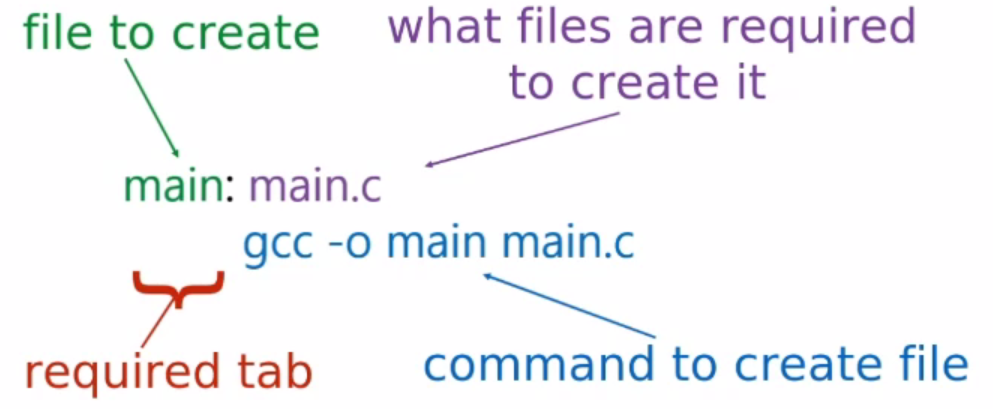
* **set -x** – simple trace of execution
* **set -n** – stops any commands from running
  + good for looking at syntax errors
* **set -v** – prints shell input line as interpreter runs

Compilation

* Bash does not need compiling, C does
* Compilation – taking human-readable source code and translating it into a set of instructions for a machine and OS
* A compiled program runs independently of other programs (unlike a shell script). Does not require a shell to interpret
* Components of a C programs
  + Header file: definitions & Prototypes (\*.h)
  + Source File: Functions, etc (\*.c)
  + Library Header: Library definitions and Prototypes (\*.h)
  + Library File: Pre-compiled Library functions (\*.a, \*.so)
  + Executable – created from several source and header files (created by dev and libraries)
* Stages:
  + Pre-processor
    - takes source and header files and combines the two, while expanding all #defines, etc
  + Compilation
    - takes pre-processed C files and converts them to machine code, labels each function with name (symbol) and creates an object file
  + Linking
    - Takes all object files and mathes the symbols to actual machine code fragments, places all fragments in the executable and replaces the names with addresses
* GNU Compiler Collection
  + **gcc**
    - -l libraries (library name)
    - -L path (path to search for library code)
    - -I path (capital i)
    - -o name
    - source.c
  + **nm**
    - look at contents of object file
    - U – unknown symbol
    - T – symbol exists (in text/code segment)
* Libraries (Dynamic and Static)
  + External libraries need to be specified with **-l**
    - Searches LD\_LIBRARY\_PATH env variable or path (**-L**)
  + **ldd**
    - Sees what libraries a program will load dynamically
    - **static** – forces all library functions to be included at compile time

# Lecture 6: Makefile, Git

Makefile

* Way of automating build process without losing flexibility of only building object files
* Makefile – set of rules that explain how a piece of code is built
* **make** – runs through each rule required to build a target, ensuring they are met
  + Will look for a file called makefile/Makefile
* Rules
  + Target, dependencies, command needed to produce the target from its dependencies
  + 
* Variables
  + Similar to Bash
    - Refer: $(MYVAR)
* Targets and Phony
  + .PHONY
  + Targets that always run
* Automatic Variables
  + % - analogous to \* in a shell script. Matches anything but NULL, and remembers what it matched
  + $@ - name of target
  + $< - name of 1st prerequisite
  + $^ - list of prerequisites for target
  + $? – list all prerequisites newer than target
* [www.gnu.org/software/](http://www.gnu.org/software/) … /make/
* Make+

Revision Control

* \* - method for managing the changes to a file and maintaining a history of the changes that have taken place
* Basic terms:
  + Repository – location of stored files and file changes
  + Checkout – get a complete version of the code from the repository
  + Commit – send an updated version of the code to the repository (along with some message/log)
* Branching:
  + Branch – create a complete copy of the code with its own revision history
  + Merge – take all occurred changes within a branch and merge them (including history)
* Centralised vs Distributed
  + Centralised
    - All history, tags, branches are kept on the server
    - Only 1 canonical source for all code
    - Simple access controls, and user management
  + Distributed
    - Each client has a complete copy of the history
    - Easy for offline work, simple syncing mechanisms
    - No single canonical source for all changes
* Git
  + Getting Started
    - Created in 2 ways:
      * Create a new, empty repository
        + **git init <directory>**
      * Clone an existing one from a remote repo
        + **git clone <url>**
  + Adding & Committing
    - Add files to repo – **git add <filename>**
    - To save a change to local repo – **git commit -m “desc”**
    - **git commit -a -m “Commit and Add”**
  + Status
    - **git status**
    - Tells about files present, but not saved, committed
    - **-s** – shorter version
  + Log
    - **git log**
    - History of changes
      * full hash of commit (ID)
    - **--oneline** – each change as one line
  + Branching
    - **git branch <branch>**
      * **--list** – shows all branches
    - **git checkout <branch>** - switch to the branch
    - **git merge <branch>** - merges branch to master

# Lecture 8: Revision