USING LINUX I

EDWARD O'CALLAGHAN

Contents

1. Prelude

Typically a desktop computer consists of three main components;

(1) The physical hardware.
(2) The supervisor operating system or *kernel*.
(3) The client applications, (i.e., web browsers, text editors,..).

There are various choices in operating system supervisors around, such as, to name a few;

- Microsoft Windows NT (found in Windows).
- Darwin (found in Apple OSX).
- BSD.
- Linux (typically GNU/Linux).

Typically we do not interact with the supervisor, or kernel, directly but though client applications. Hence, the operating system is typically misconstrued as being the installed client applications when this is in fact not the case. In this course we shall study the GNU/Linux operating system and consider some usual client applications and their configuration.

Various GNU/Linux *distributions* exist, each of which tailored to a particular audience. The **ArchLinux** distribution is favoured for its neutrality of not imposing any particular pre-configuration on the user. In ArchLinux you are required to configure everything yourself how you wish. In this way, ArchLinux is somewhat true to the spirit of GNU/Linux's simplicity and individuality is better in regards to program design. The ArchLinux distribution simply provides a few various scripts to get it installed and a core bootable system and package manager to allow you to install your preferred software without imposing any choice on you.

Example 1.1 (Various common GNU/Linux distributions)**.** ArchLinux is not the only distribution of course. Some common alternatives are listed;

- Red Hat Fedora Linux,
- Novel SuSE Linux,
- Slackware Linux,
- Debian Linux,
- Ubuntu Linux (made from Debian).

2. Background

The Linux kernel is the piece of software that mediates the interaction of the needs of client applications to the underling physical hardware. A typical scenario is that of a text editor which asks the operating system to print some displayed text, the operating system then works out how to talk to the printer hardware correctly to get this done. Another such example is reading a file off a USB storage device in a file manager. In this case the file manager asks the operating system what files are stored on the media and the operating system returns a list of names and locations.

In this course we study the GNU/Linux operating system. This is the Linux kernel with the GNU *userland* built on top. The GNU userland provides various simple client applications to get common tasks done such as moving, copying and renaming files. The userland also provides system *libraries* that provide common bits of code that these userland applications share in their functionality. The central system library on GNU/Linux is called **glibc**. This library is probably the most important since it is the glue that fuses the Linux kernel to userland programs for the fundamental operations such as file manipulation. Any other client program you may be running is also part of the userland however need not be part of the GNU core system tools and libraries.

GNU itself stands for Gnu Not Unix. GNU is arguably the ordinal ideology behind the *open source* movement. Although other ideologies now exist, such as the BSD's, the **GNU/GPL**, or General Public License, has dominated the scene with its far left ideals. The GPL essentially asserts that any changes to the code of the system must be published with the original code and a copy of the GPL license with corresponding references to who wrote the original works. The central idea is to *force* the workings of the system to be *always open* and free to use! In this way, the system is owned by no one and made by everyone who is feels the need to contribute. Companies may contribute to GNU/Linux and may own the rights to the original work. However, they do not own the right to demand anything from you or force upon you questionable practices typically found in the world of proprietary software.

Open source empowers you with the control and ownership of your own system. Proprietary technologies are typically licensed in a way that gives you the rights to *use* the software *subject* to be removed at *any* time they see fit and not *own* the software you may of just payed for! In actual fact, if you do not use the software as they specify in their license agreement you are typically legally liable of braking sometimes multiple laws if you ever got caught.

3. The Console

The *console* is essentially the way in which we interact with a computer. The console is just a fancy name for the mouse, keyboard and screen set. Before graphical enabled console was introduced consoles were presented as a terminal. Today our consoles are graphical and so we have **terminal emulators**.

In Microsoft Windows the emphasis is on the windowing graphical side of the console whereas GNU/Linux emphasises *the terminal* as the preferred choice of control. This is not to say that GNU/Linux is not capable of a comprehensive array of graphical capabilities, in fact the case is quite the opposite. However, the terminal provides greater flexibility and more precise control of the system console and most importantly, efficiently, the primary reason for desktop computing in many respects. Some drawbacks exist as with any trade off, with increased functionality there is increased complexity. The GNU/Linux paradigm of simplicity leads to vast flexibility and so inherently can be a little tough to understand at first sight coming from a graphical only background.

To motivate consider the summary that, if there is no button to click then you can't do it, whereas in the terminal when there is a will there is a way! Thus, some old ideas should be carefully considering before throwing them away so readily. Perhaps for some wizy-bang pretty looking graphical interface some company is trying to sell you but really does not actually add to your productivity in reality.

Another main task of the terminal emulator is to authenticate the user upon login, although this is typically done by the *virtual terminal*. The virtual terminal is the terminal emulator that is run by the kernel itself.

Various terminal emulators exist however we consider *urxvt* as the best blend of simplicity and stability while maintaining various useful features. The *command interpreter* we shall use runs inside the terminal emulator and allows the machine to interpret user command input.

Various command interpreters exist (i.e., zsh, ksh93, bash,...), again giving you the freedom of choice. However **bash** or Born Again SHell is the typically most commonly found command interpreter, or just *shell*. Since the shell is an interpreter each shell has its own little language and so you can *script* how you wish the console to play. That is, you write **shell scripts** to automate common tasks in our shell that you would normally do at your console. For example, rename a directory of pictures according to some naming scheme such as by date-location. For such a task, as a graphical user, you would typically download some random special program in the hope that it would do something similar to what you wish or you may spend hours manually renaming each file at a time. This is where the power of interacting with the console in a command fashion comes into its own. Here, we need only specify the task in terms of a chain of a few short commands in some file, which we call a script, and then run this file in the shell interpreter. The shell interpreter will interpret the script as if you were siting in front of the console doing the task and automate this task exactly how you specified. The detail of interacting in this way with the console will take some time to get use to and only comes with practice however give yourself time and take it easy.

4. ViM Text Editor

Try running the *vimtutor* command.

5. Files

First consider the following proposition and remember this as it is essentially the whole design premise of the Linux operating system!

Proposition 5.1. *Everything in Linux is consider a file of two main kinds:*

- *Block, or*
- *Character Stream.*

A ***Block*** *is a contiguous chunk of binary data whereas a* ***Stream*** *is a flow of binary data bit by bit where a bit is one or zero.*

Now we may consider some concrete examples in order to fix our ideas of what this actually means in practice.

Example 5.2 (Block Device)**.** A block device or block file comes up in various places such as;

(1) A hard disk.
(2) A USB pen (data is buffered in VFS since the USB port is serial).
(3) The screen "framebuffer".
(4) A file on the disk.
(5) A allocated chunk of memory can be consider a block of memory.

Example 5.3 (Character Device)**.** A character device is a very typical type of file used to stream binary content to some kind of media such as;

(1) A printer.
(2) A serial port, (i.e., RS232, i2c, USB).
(3) Audio jack out.
(4) A Linux pipe stream.

The Linux kernel deals with files by a layer called **VFS** or *Virtual File System*. The detail of this is not important to understand how things work in general.

Three main special stream device files are used to interact with the system:

Definition 5.4 (Standard Input/Output (IO))**.** The *Standard IO* streams are how programs talk with the screen and other programs, in this case programs interact in a general way with the following standard IO streams:

1 Standard Input or /dev/stdin
2 Standard Out or /dev/stdout
3 Standard Error or /dev/stderr

Consider some web browser application that would like to print a page then save it. The browser need not known anything about the type of printer or type of media it is storing files to. Only that it needs to *write* something to a printer and *write* something to a file. This is exactly how applications interact with VFS in the Linux kernel.

A application simply writes to a printer file or writes to a disk file and VFS **abstracts** the writing operation to whatever subsystem takes care of the detail of the write.

In Linux printer files are represented though a userland program called CUPS or *Common Unix Printer System* which presents *printer files* that are software block devices, called a *spool*, that buffer the page and streams it to the physical device.

In the situation where we are writing a file to a disk, VFS now presents a software block device that *represents* the physical disk. In particular, VFS takes the content of the software block device buffer, finds the correct filesystem software that the disk is formated with and uses this to correctly write the data to the physical media in the respective filesystem formating. Some example disk filesystems are; XFS, JFS, NTFS, FAT32, etc..

Remark. Notice that VFS represents physical storage media as software buffers in memory. This is why we must *umount* a disk such as a USB storage pen **before** physically removing it from the system. Since, when we write to the software representation we are *not* writing to the actual disk and so we *must* ensure that everything in the software buffer has been flushed or *sync*'ed to the physical disk! Failure to do this may result in loss of data!

Similarly to the printer situation, in the case of audio mixing Linux presents the sound card by an equivalent software representation. This particular representation is achieved by *ALSA* or the Advanced Linux Sound Architecture. That is, when playing a song we write the song file to ALSA's representation of the sound card (i.e., a software buffer block device) and ALSA takes care of reading this block of data out of memory bit by bit and streaming that to the physical sound card.

A somewhat special file exists that you may of considered by now, a *directory.* A directory is simply a file that contains the location of all the files it has assigned as children. This design choice naturally leads to the representation of all files on the system as a tree-like structure. Note that the structure is not a pure tree, since some files are software links to other files or essentially a directory with one and only one entry. Hence, the structure in actual fact is more like a graph on a real system. In any case this should provide us with the understanding of why *mount* points are required.

A *mount point* is simply a directory in which we mount the software file representation from VFS of a physical disk. The directory we mount to simply has the pointer to location of the memory where VFS looks for data to read and write to a physical disk. When we call the command **mount** we are writing this address into the directory used as the mount point. When we call the command **umount** we are forcing the data in VFS's buffer to be flush to the physical disk and then removing the address entry from the original mount point directory.

5.1. **File-system Heredity.** ..

5.2. **File Manipulation.**

Definition 5.5 (Moving files)**.** To move a file we use the command **mv**.

Definition 5.6 (Copying files)**.** To copy a file we use the command **cp**.

Definition 5.7 (Renaming files)**.** To rename a file we use the command **rename**.

Definition 5.8 (Removing files)**.** To remove a file we use the command **rm**.

6. Manual Pages

The *man pages*, or Manual Pages, are the first source for documentation to shell based activities. Typically every shell command has a corresponding man page. If you are unsure of the detail as to a particular command you should consult its man page.

Problem 6.1. *Type in the shell,* man man*, to bring up the manual page of the manual pages. Another example type,* man ls*, to bring up the manual page for the* ls *command. To quit the manual page, simply press* q.

7. Package Manager

The package manager allows the system to track the installed software and check it for consistency and updates. Another important point of the package manager is to allow the user to install new software easily.

Example 7.1 (Common package managers)**.** GNU/Linux distributions typically come with their own home grown style of package manager. Here is a short list of common package managers;

- RPM or Red-Hat Package Manager, found in Fedora.
- Pacman, found in ArchLinux.
- APT, found in Debian and Ubuntu.

8. Installation

8.1. **Disk Partitions.**

Disk partitions are essentially the way in which we may divide up the disk. For example, we may wish for our personal files in our *home* directory to be separate from the rest of the system files. By partitioning the disk we may *format* each partition separately, typically with different filesystems. The advantage in the case of the *home* directory having a different filesystem is that of performance and increased safety. For example, if, for whatever reason, our system was unbootable and irreparable (broken) our files are safe from doing a clean install without the worry of overwriting them. Although it is **always** a good idea to regularly backup important personal data and configuration files.

8.2. **File-systems.**

The filesystem is the software to which the Linux kernel uses when reading and writing data to some storage media such as a disk. The filesystem is essentially a algorithmic software description as to how to organise data on the storage media in an optimal and consistent way. Hence, we typically say that a disk has been *formated* with a particular filesystem. For example, "the USB pen has been formated with FAT32".

Example 8.1 (Common Filesystems)**.** Here is a short list of commonly found Linux filesystems:

- XFS,
- JFS,
- EXTended 2,3 and 4,
- BTFS,
- ZFS,
- procfs,
- tempfs,
- swapfs.

Some common filesystems found both in Linux and other places such as Microsoft Windows and Apple OSX are given:

- FAT32,
- NTFS,
- HFS+.

The EXTended filesystem, or *ext*, is the usual default Linux filesystem software. However, ethier JFS or XFS is recommended for its greater maturity and hence stability and reliability. Something of a critical concern considering the filesystem takes care of your data! USB storage devices are usually formated with FAT32 since almost every system can read and write this very basic filesystem formatting and hence is very compatible, however it is very minimal.

In Windows, NTFS is the default filesystem and you have little to no choice about that. NTFS is proprietary software and so if something goes wrong you really have no possible way to find out how NTFS went wrong since we have no access to its code. Apple's OSX uses HFS+ which has its own set of problems and is becoming somewhat long in the tooth. The most critical point about proprietary filesystem software, like NTFS, is that your data is trapped in this format and only systems that have access to the code that makes NTFS work may read and write to these filesystems. In this way, proprietary software traps you and your rights to your data! Worse still is that, if you recall, propriary software licensing is typically written so that you buy the rights to *use* the software not *own* it. Hence, by storing your data on filesystems like NTFS you only have the rights to *use* your data even though you may own your data you may not always have the right to *access* it!

9. Networking

9.1. Basics. Basic networking starts with setting up what the machines *hostname* should be. This is some arbitrary name we give to the physical machine console on the network. To specify the name of the machine, simply make something up and do the follow:

```
$ echo ”starkid” > /etc/hostname
```

You will also need to append this name in /etc/hosts as an alias to *localhost*. The *localhost* is the default name that simply means the local machine.

A *fully qualified username* is then of the form:

username@hostname

For example, joebobs@starkid.

9.2. **Dynamic IP.**

9.3. **Static IP.**

9.4. **Wired.**

9.5. **Wireless.**

10. Environments

Two main console environment exist, the textural and the graphical. In the textural environment we make use of the shell interpreter to communicate our commands to the system console. Whereas in the graphical environment we make use of abstract graphical constructs to communicate our ideas on what we wish the system console to do for us.

Both environment have various runtime variables that can be tweaked as needed. For example, in the graphical environment we may wish to tweak the wallpaper environment variable. Whereas in the shell we may wish to set things like the system search path. Various path variables exist such as where to look for executable commands, set by **PATH** or where to find system libraries, set by **LIBS**. The details vary and you will need to consult your environment documentation for the details. That is, your shell man page or your window manager documentations.

10.1. **Shell Interpreter.** The shell interpreter is the textural environment that runs inside the terminal emulator. It is though the shell we command the system console to do certain tasks. The shell can run more than one command at a time, called jobs. This is called job control.

Problem 10.1 (Job Control)**.** *Try running the* firefox *command to start the Mozilla Firefox web browser and try some various bits of job control.*

10.2. **X Server.** The *X Server*, much like the terminal emulator, provides graphical windowing functionality as the fundamental object of control. This is in contrast with the terminal emulator that instead uses shells that use individual commands as the fundamental objects of control. Notice the distinction! The X server windowing system *environment* is where graphical client applications can ask the X server to draw them a window from which you can control the console though the mouse. Conversely, the shell asks the terminal emulator to output results from commands and takes input from the console though the terminal emulator.

In summary:

- A terminal emulator *abstracts* the console away from commands in a shell.

- The X server *abstracts* the drawing of graphical objects on the console away from the client applications.

10.2.1. *Toolkits.* Applications typically want to draw more complex things such as triangles and squares. However the X server does not attempt to draw sophisticated graphical objects, keeping true to the GNU/Linux philosophy. In this way the X server provides the flexibility to draw whatever kind of object you would like in a graphical way on the screen. However, applications typically would like to draw the same kind of thing over and over and that various applications on a graphical desktop would ideally like to look consistent from a usability prospective. Hence, on a typical GNU/Linux installation one normally installs a *toolkit* to go with the X server and the preferred range of applications that make use of the said toolkit. Toolkits provide another abstraction on top of the X server to draw graphical applications to look in a consistent and feature full way.

Example 10.2 (Some common X toolkits)**.** Some example toolkits that are typically found;

- Gtk2/3,
- Qt,
- Clutter.

The X server is not the only place graphical toolkits are found. In fact, in the shell one can draw *ASCII* art type graphical interfaces though a common toolkit known as **ncurses**. Ncurses once again abstracts away the terminal emulators ability to draw ASCII text to the screen and provides a rang of *API*'s, or Application Programmer Interface's, that allow programmers to draw graphical interfaces in the form of text on the console though a terminal emulator.

10.2.2. *Window Manager.* The window manager is essentially the *job control* of the graphically world on a GNU/Linux system. The window manager is the main program the X server runs that controls how applications behave with one another and how they are presented to the user in terms of method of control. Typically the window manager is what the user perceives to be *the desktop*! So the window manager is a important choice as to how you prefer to work and prefer your desktop to act and look. The window managers environment is analogous to the shell language that comes with the shell of choice. Hence, desktop environment settings such as setting your wallpaper is equivalent to setting, say for example, the *PATH* environment variable of your shell.

Example 10.3 (Some common window managers)**.** Some example window managers that are popular;

- XMonad,
- KDE,
- Gnome.

Typically two main paradigms exist for window managers, **tilling** and *stacking*. The most popular of which is typically the stacking paradigm in which program windows are layer on top of each other. Both Gnome and KDE are example window managers that are of the stacking sort, although can be manually configured otherwise. However the very minimalistic XMonad window manager is extremely small, uses almost no memory at all (i.e., less than 1MB) and is of the tiling sort. The main advantage of tiling window managers comes in when you are running more than a few programs and spend more time switching between programs than actually doing your intended work inside the programs. Another advantage of the usually minimal nature of tiling window managers is the desktop is not cluttered with random applications popping out everywhere and arbitrarily large unnecessary boarders around menu parts taking up valuable screen real estate.

10.2.3. *Login Manager.* To start the X server one can run *startx* as the intended user and configure which window manager the X server should start in *.xinitrc* in their home directory. However, on a desktop workstation these days you typically wish for the X server to start immediately and be presented a graphical method of login. This is the intended task of a login manager, at boot the system start the graphical login manager as a system service and the login manager starts the X server in turn. Various login managers exist.

Example 10.4 (Some common login managers)**.** Some example login managers, some of which come with the window manager for a more transparent experience.

- KDM - comes with the KDE window manager,
- GDM - comes with the Gnome window manager,
- SLiM - a light weight login manager that is independent of any particular window manger.

A good match with the XMonad window manager is the SLiM login manager. Since SLiM is very minimal and does not reply on the toolkit libraries Qt or Gtk3 from KDE or GDM respectively.

11. System Services

System services are the client applications that work in the background when the system boots. These services provide various house keeping and user services which are called demons. Some of these demons you interact with directly such as SLiM and others you most likely don't even think about such as NTP.

Example 11.1 (Various common services)**.** Some example services are as follows;

- NTP - Keeps the system clock in sync with the time from the internet.
- SLiM - Provides a graphical login authentication service for the window manager though the X server.
- Autofs - Checks for things like USB storage devices and automatically mounts them for you.

- WPA_Supplicant - Configures wireless connections by automatically authenticating with the access point and connecting for you.
- DHCPD - Automatically sets up your IP address and DNS settings for your network connection.

Various GNU/Linux distributions come with their own system service manager. However, a typically common one is called *systemd*. SystemD handles when services should start/stop and reports on their status should you wish to inquire about it. Consult the man page for *systemd* for details.