CURS 1 LINUX

1.1 Linux is Everywhere

Hello and welcome to **NDG Linux Essentials**!

Linux is everywhere; Linux jobs are everywhere. There is a demand for Linux skills in just about every industry and job category on the planet, and not enough Linux talent to meet this growing demand. It’s also fun and rewarding to work with something that’s so much a part of our modern lives yet which so few people understand.

If someone says they have Linux experience, it might refer to configuring systems, running web servers, or any number of other services and programs that operate on top of Linux. Over time, Linux administration has evolved to encompass just about every task that a modern business, educational institution or government organization might use in their daily operations.

The journey of learning you are beginning today has no ending point. It can take you in a myriad of different directions, from cybersecurity to application and game development, system administration, networking, big data, and artificial intelligence; all of these fields are rooted in Linux.

Every time you execute a search on the internet, watch a video on your phone or order something online, that’s likely Linux at work. It epitomizes a whole that is greater than the sum of its parts, a vast undertaking, done voluntarily, by some of the smartest people on the planet.

While your journey will be ongoing, be comforted that you are learning a set of technologies, commands, and methods that have stood the test of time. Linux utilizes and expands upon many of the commands and ways of accomplishing computing that UNIX began, with a rate of change per year that’s very manageable. Now, some 30+ years on, many of those learned commands are still active and used every day by sysadmins, devops, and architects. Linux is a revolution of evolution, allowing you to learn the majority once and keep up with the small percentage of changes in a continual learning process.

## Linux is a Kernel

The definition of the word Linux depends on the context in which it is used. Linux means the kernel of the system, which is the central controller of everything that happens on the computer.

When most people refer to Linux, they are really referring to a combination of software called **GNU/Linux**, which defines the operating system. **GNU** is the free software that provides open source equivalents of many common UNIX commands. The **Linux** part of this combination is the Linux kernel, which is the core of the operating system. The kernel is loaded at boot time and stays running to manage every aspect of the functioning system.

The story of Linux begins with **UNIX**, an operating system developed at AT&T Bell Labs in the 1970s. UNIX is written in the **C** language making it uniquely portable amongst competing operating systems, which were typically closely tied to the hardware for which they were written. It quickly gained popularity in research and academic settings, as well as amongst programmers who were attracted to its modularity. Over time it was modified and forked (that is, people modified it, and those modifications served as the basis for other systems) such that at present there are many different variants of UNIX. However, UNIX is now both a trademark and a specification, owned by an industry consortium called the **Open Group**. Only software that has been certified by the Open Group may call itself UNIX.

Linux started in 1991 as a hobby project of **Linus Torvalds**, a Finnish-born computer scientist studying at the University of Helsinki. Frustrated by the licensing of MINIX, a UNIX-like operating system designed for educational use, and its creator’s desire not to make it a full operating system, Linus decided to create his own OS kernel.

From this humble beginning, Linux has grown to be the dominant operating system on the Internet, and arguably the most important computer program of any kind. Despite adopting all the requirements of the UNIX specification, Linux has not been certified, so Linux really isn’t UNIX! It’s just… UNIX-like.

Prior to and alongside this development was the **GNU Project**, created by **Richard Stallman** in 1983. While GNU initially focused on building their own operating system, they ultimately were far more effective at building tools that go along with a UNIX-like operating system, such as the editors, compilers and user interfaces that make a kernel usable. Since the source was all freely available, Linux programmers were able to incorporate the GNU tools to provide a complete operating system. As such, many of the tools and utilities that are part of the Linux system evolved from these early GNU tools.

**Consider This**

Linus originally named the project Freax, however, an administrator of the server where the development files were uploaded renamed it Linux, a portmanteau of Linus’ name and UNIX. The name stuck.

GNU is a recursive acronym for “GNU’s Not Unix,” and it’s pronounced just like the African horned antelope that is its namesake.

## Linux is Open Source

Historically, most software has been issued under a closed-source license, meaning that you get the right to use the machine code, but cannot see the source code. Often the license explicitly says that you may not attempt to reverse engineer the machine code back to source code to figure out what it does!

The development of Linux closely parallels the rise of open source software. Open source takes a source-centric view of software. The open source philosophy is that you have a right to obtain the software source code and to modify it for your own use.

Linux adopted this philosophy to great success. Linus made the source programming code (the instructions a computer uses to operate) freely available, allowing others to join in and shape this fledgling operating system. It was not the first system to be developed by a volunteer group, but since it was built from scratch, early adopters could influence the project’s direction. People took the source, made changes, and shared them back with the rest of the group, greatly accelerating the pace of development, and ensuring mistakes from other operating systems were not repeated.

**Consider This**

The source code may be written in any of hundreds of different languages. Linux happens to be written in **C**, a versatile and relatively easy language to learn, which shares history with the original UNIX. This decision, made long before it’s utility was proven, turned out to be crucial in its nearly universal adoption as the primary operating system for internet servers.

## 1.4 Linux Has Distributions

People that say their computer runs Linux usually refer to the kernel, tools, and suite of applications that come bundled together in what is referred to as a distribution.

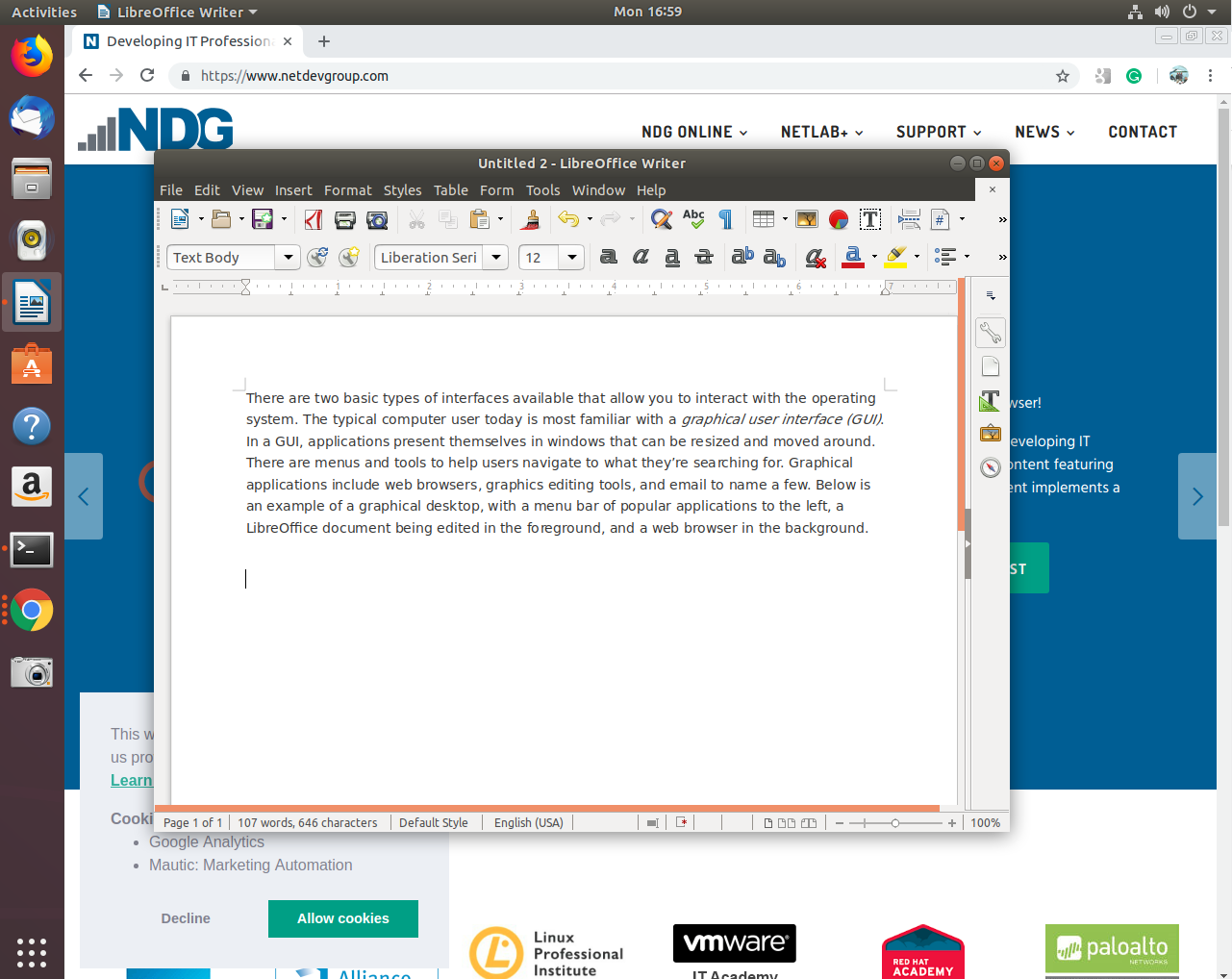
Take Linux and the GNU tools, add some user-facing applications like a web browser and an email client, and you have a full Linux system. Individuals and even companies started bundling all this software into distributions almost as soon as Linux became usable. The distribution includes tools that take care of setting up the storage, installing the kernel, and installing the rest of the software. The full-featured distributions also include tools to manage the system and a package manager to help you add and remove software after the installation is complete.

Like UNIX, there are distributions suited to every imaginable purpose. There are distributions that focus on running servers, desktops, or even industry-specific tools such as electronics design or statistical computing. The major players in the market can be traced back to either Red Hat, Debian or Slackware. The most visible difference between Red Hat and Debian derivatives is the package manager though there are other differences in everything from file locations to political philosophies.

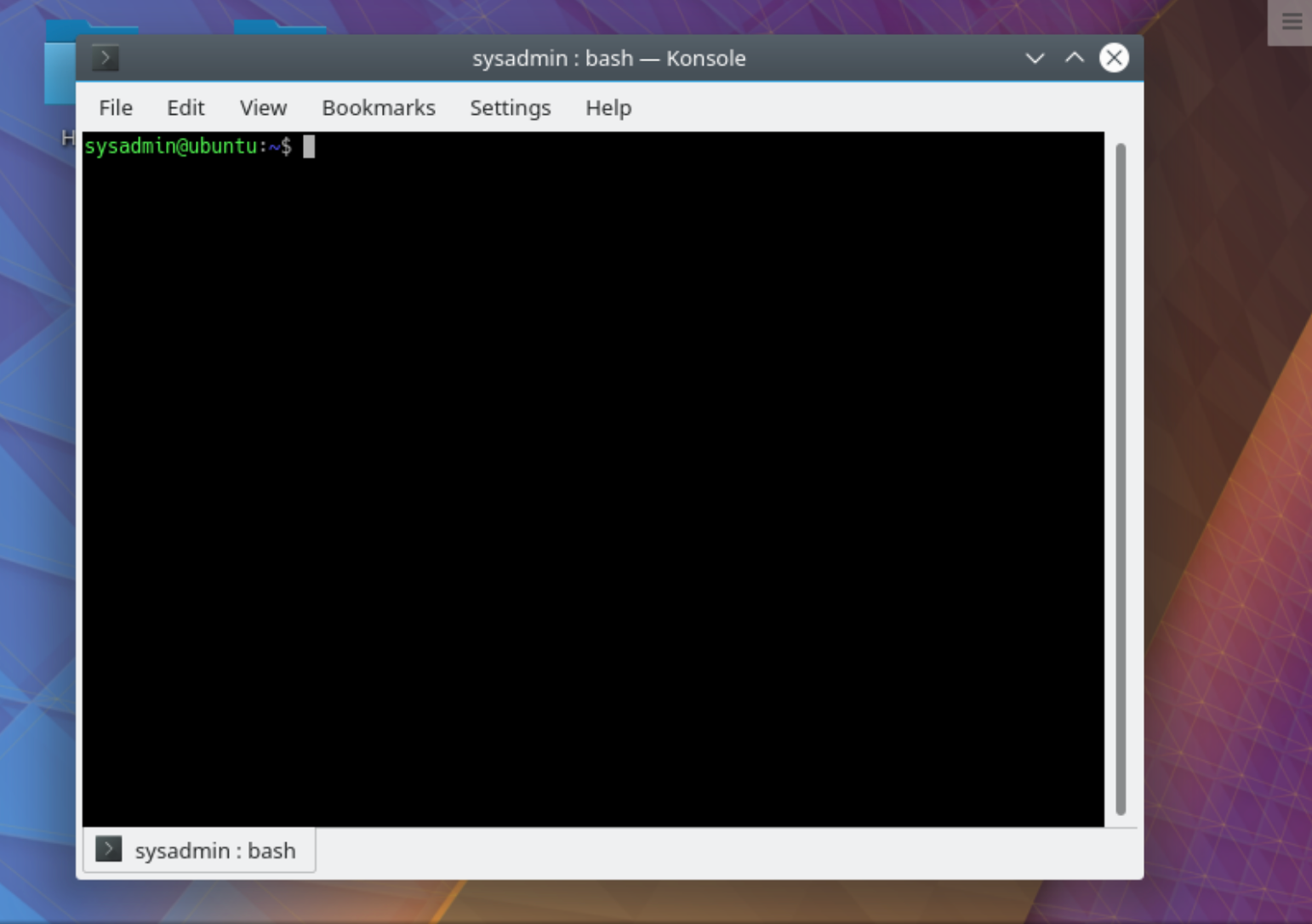
Linux Embraces the CLI

There are two basic types of interfaces available that allow you to interact with the operating system. The typical computer user today is most familiar with a *graphical user interface (GUI)*. In a GUI, applications present themselves in windows that can be resized and moved around. There are menus and tools to help users navigate. Graphical applications include web browsers, graphics editing tools and email, to name a few.

Below is an example of a graphical desktop, with a menu bar of popular applications to the left, a LibreOffice document being edited in the foreground and a web browser in the background.



The second type of interface is the *command line interface (CLI)*, a text-based interface to the computer. The CLI relies primarily on keyboard input. Everything the user wants the computer to do is relayed by typing commands rather than clicking on icons. It can be said that when a user clicks on an icon, the computer is telling the user what to do, but, when the user types a command, they are telling the computer what to do.



Typically operating systems offer both GUI and CLI interfaces. However, most consumer operating systems (Windows, macOS) are designed to shield the user from the complexity of the CLI. The Linux community is different in that it positively celebrates the CLI for its power, speed and ability to accomplish a vast array of tasks with a single command line instruction. The virtual machines used for the chapters and labs in this course provide a CLI for you to practice on without fear of damaging anything.

When a user first encounters the CLI, they can find it challenging because it requires memorizing a dizzying amount of commands and their options. However, once a user has learned the structure of how commands are used, where the necessary files and directories are located and how to navigate the hierarchy of a filesystem, they can be immensely productive. This capability provides more precise control, greater speed and the ability to easily automate tasks through scripting.

Furthermore, by learning the CLI, a user can easily be productive almost instantly on ANY distribution of Linux, reducing the amount of time needed to familiarize themselves with a system because of variations in a GUI.

#### [**Chapter 2 - Operating Systems**](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/#c1)

**Android**

A Linux distribution that provides a platform for mobile users but lacks the traditional GNU/Linux packages to make it compatible with desktop Linux distributions.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**CentOS**

A Linux distribution that is compatible with Red Hat Enterprise Linux but does not offer the paid support that Red Hat does.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**Debian**

An operating system that uses the Linux kernel. It that promotes the use of open source software and adherence to standards.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**Linux Mint**

A Linux distribution that is a derivative of Ubuntu and still relies upon the Ubuntu repositories.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**Raspberry Pi**

A hardware platform used in training for programmers and hardware designers at all levels. Its low cost and ease of use have made it popular with educators.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**Raspbian**

A specialized Linux distribution optimized to run on Raspberry Pi hardware.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**Red Hat**

A Linux distribution that introduced Red Hat Package Manager (RPM). The developer formed a company by the same name which specializes in open source software.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**SUSE**

One of the first comprehensive Linux distributions. It is derived from Slackware which offers many similarities with Red Hat Enterprise Linux.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**Scientific Linux**

A specific use distribution based on Red Hat. It was designed to enable scientific computing.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**Ubuntu**

The most popular Debian derived distribution. It has several different variants for desktop, server, and various specialized applications as well as an LTS version.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**beta**

A software release that has many new features that haven’t been tested.  
[Section 2.1.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.1.1)

**command line interface (CLI)**

A text based interface in which the user enters commands. Feedback, output and programs are presented in text format only.  
[Section 2.1.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.1.1)

**desktop configuration**

Desktop are preferred if the user interacts with the system directly. Desktop system primarily run a GUI for the ease of use of its user.2.1.1  
[Section 2.1.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.1.1)

**graphical user interface (GUI)**

A visual user interface that allows the user to interact with the system using windows, icons, a cursor, etc.  
[Section 2.1.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.1.1)

**long-term support (LTS)**

Associated with the life cycle of distributions, this feature states that software is supported for 5 years or more.  
[Section 2.4](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4)

**maintenance cycles**

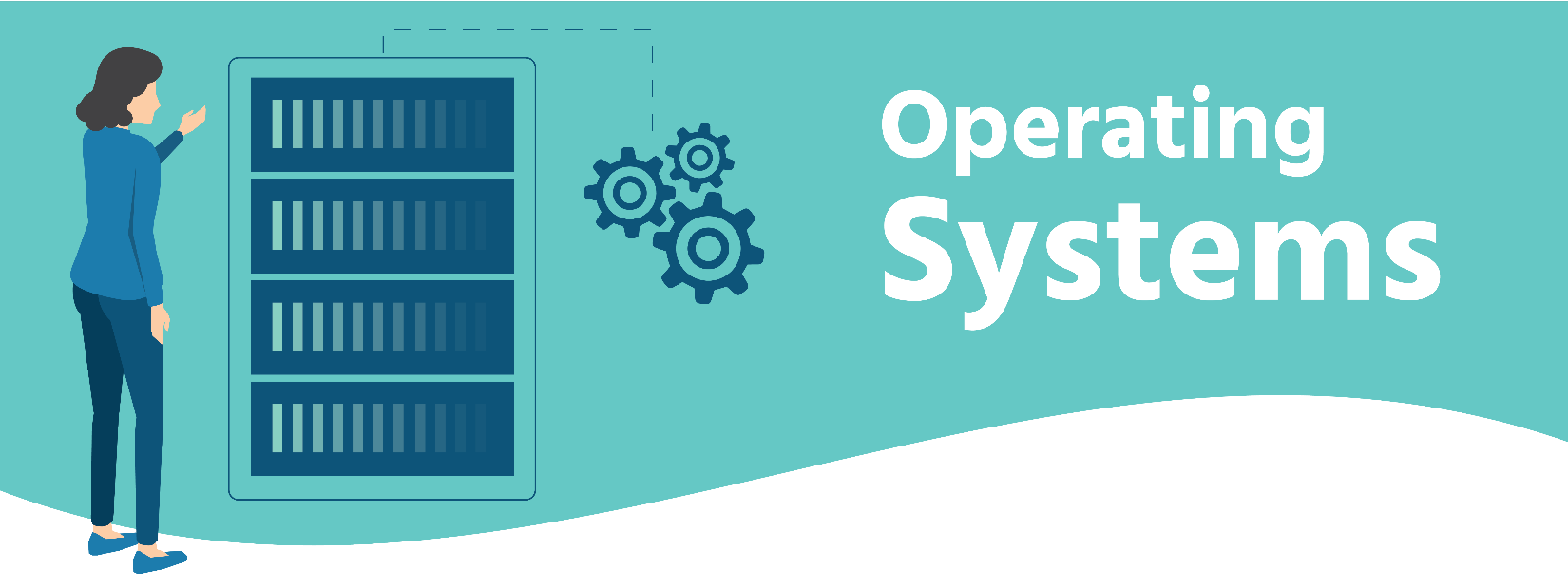
The period of time vendors support older versions of software before not offering any updates.  
[Section 2.1.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.1.1)

**openSUSE**

A Linux distribution that is a completely open, free version of SUSE Linux Enterprise with multiple desktop packages similar to CentOS and Linux Mint.  
[Section 2.4.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.4.1)

**stable**

A software release whose updates have been tested in the field.  
[Section 2.1.1](https://content.netdevgroup.com/contents/linux-essentials/MX53hsCaJo/2.1.1)

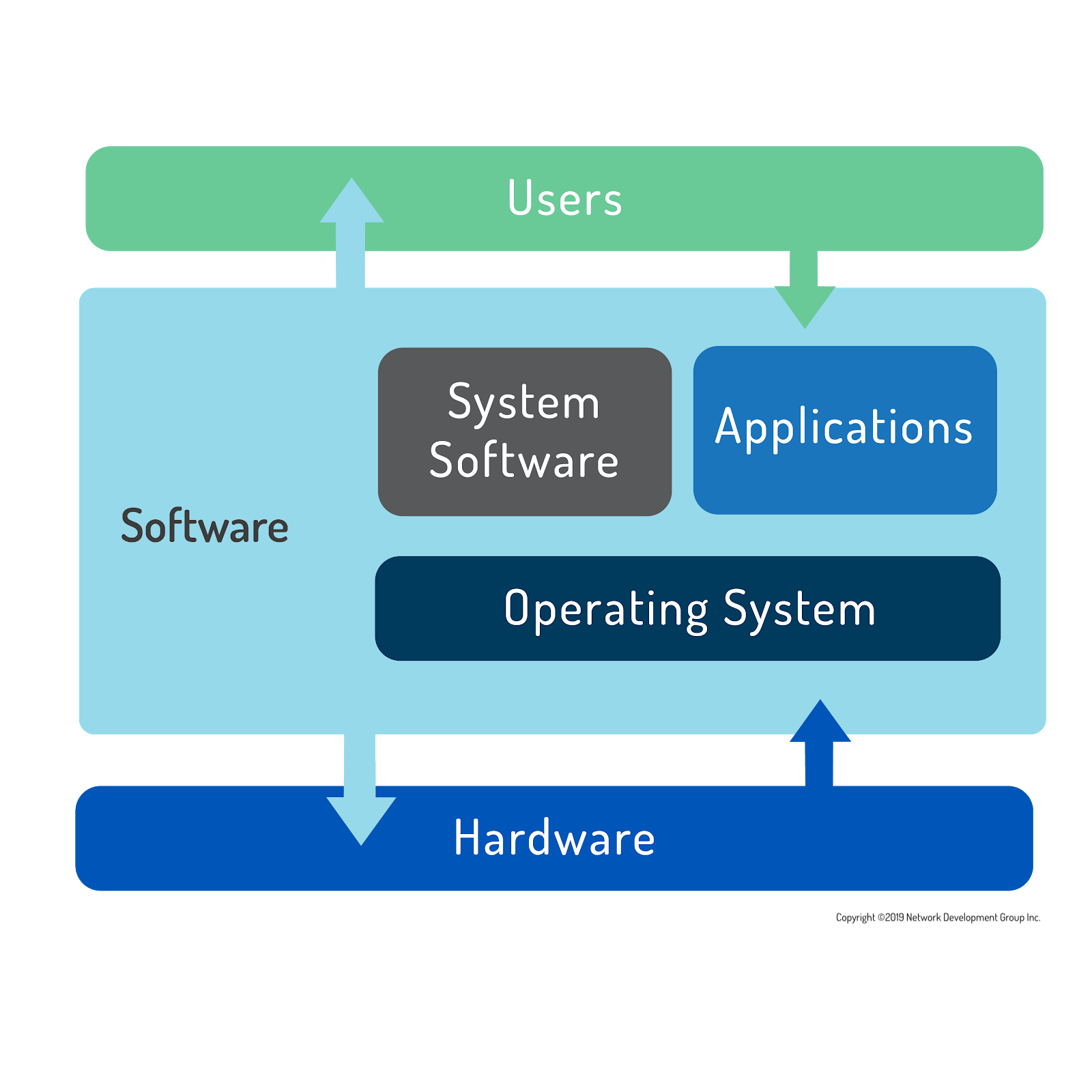


2.1 Operating Systems

An *operating system* is software that runs on a computing device and manages the hardware and software components that make up a functional computing system.

Modern operating systems don’t just manage hardware and software resources, they schedule programs to run in a multi-tasking manner (sharing the processor so that multiple tasks can occur apparently simultaneously), provide standard services that allow users and programs to request something happen (for example a print job) from the operating system, and provided it’s properly requested, the operating system will accept the request and perform the function needed.

Desktop and server operating systems are by nature more complex than an operating system that runs on a single-purpose device such as a firewall, or a mobile phone. From a simple set-top box that provides a menu interface for a cable provider, to supercomputers and massive, parallel computing clusters, the generic term *operating system* is used to describe whatever software is booted and run on that device.



Computer users today have a choice mainly between three major operating systems: **Microsoft Windows**, **Apple macOS**, and **Linux**.

Of the three major operating systems listed only Microsoft Windows is unique in its underlying code. Apple’s macOS is a fully-qualified UNIX distribution based on BSD Unix (an operating system distributed until 1995), complemented by a large amount of proprietary code. It runs on hardware specifically optimized to work with Apple software. Linux can be any one of hundreds of distribution packages designed or optimized for whatever task is required. Only Microsoft Windows is based on a proprietary code base that isn’t either UNIX- or Linux-based.

A user can easily interact with any of these systems by pointing and clicking their way through everyday productivity tasks that all behave similarly regardless of the underlying operating system. Except for Windows, which is mostly administered via the GUI, most system administration tasks are performed using typed commands in a terminal. An administrator that is familiar with UNIX can typically perform tasks on a Linux system and vice versa. Many Linux command line functions also have Microsoft equivalents that administrators use to do their work efficiently.

## Decision Points

### **Role**

The first decision when specifying any computer system is the machine’s role. Will you be sitting at the console running productivity applications or web browsing? If so, a familiar desktop is best. Will the machine be accessed remotely by many users or provide services to remote users? Then it’s a server.

Servers typically sit in a rack and share a keyboard and monitor with many other computers, since console access is generally only used for configuration and troubleshooting. Servers generally run as a CLI, which frees up resources for the real purpose of the computer: serving information to clients (any user or system that accesses resources remotely). Desktop systems primarily run a GUI for the ease of use of their users.

### **Function**

Next, determine the functions of the machine. Is there specific software it needs to run, or specific functions it needs to perform? Will there be hundreds, even thousands, of these machines running at the same time? What is the skill-set of the team managing the computer and software?

### **Life Cycle**

The service lifetime and risk tolerance of the server also needs to be determined. Operating systems and software upgrades come on a periodic basis, called a release cycle. Vendors only support older versions of software for a certain period of time before not offering any updates; this is called a maintenance cycle or life cycle.

In an enterprise server environment, maintenance and release cycles are critical considerations because it is time-consuming and expensive to do major upgrades. Instead, the server hardware itself is often replaced because increased performance is worth the extra expense, and the resources involved are often many times more costly than the hardware.

**Consider This**

There is a fair amount of work involved in upgrading a server due to specialized configurations, application software patching and user testing, so a proactive organization will seek to maximize their return on investment in both human and monetary capital.

Modern data centers are addressing this challenge through virtualization. In a virtual environment, one physical machine can host dozens, or even hundreds of virtual machines, decreasing space and power requirements, as well as providing for automation of many of the tasks previously done manually by systems administrators. Scripting programs allow virtual machines to be created, configured, deployed and removed from a network without the need for human intervention. Of course, a human still needs to write the script and monitor these systems, at least for now.

The need for physical hardware upgrades has also been decreased immensely with the advent of cloud services providers like **Amazon Web Services**, **Rackspace**, and **Microsoft Azure**. Similar advances have helped desktop administrators manage upgrades in an automated fashion and with little to no user interruption.

### **‌⁠​​⁠​Stability**

Individual software releases can be characterized as beta or stable depending on where they are in the release cycle. When a software release has many new features that haven’t been tested, it’s typically referred to as beta. After being tested in the field, its designation changes to stable.

Users who need the latest features can decide to use beta software. This is often done in the development phase of a new deployment and provides the ability to request features not available on the stable release.

Production servers typically use stable software unless needed features are not available, and the risk of running code that has not been thoroughly tested is outweighed by the utility provided.

Software in the open source realm is often released for peer review very early on in its development process, and can very quickly be put into testing and even production environments, providing extremely useful feedback and code submissions to fix issues found or features needed.

Conversely, proprietary software will often be kept secret for most of its development, only reaching a public beta stage when it’s almost ready for release.

### **Compatibility**

Another loosely-related concept is backward compatibility which refers to the ability of later operating systems to be compatible with software made for earlier versions. This is usually a concern when it is necessary to upgrade an operating system, but an application software upgrade is not possible due to cost or lack of availability.

The norm for open source software development is to ensure backward compatibility first and break things only as a last resort. The common practice of maintaining and versioning libraries of functions helps this greatly. Typically, a library that is used by one or more programs is versioned as a new release when significant changes have occurred but also keeps all the functions (and compatibility) of earlier versions that may be hard-coded or referred to by existing software.

### **Cost**

Cost is always a factor when specifying new systems. Microsoft has annual licensing fees that apply to users, servers and other software, as do many other software companies. Ultimately, the choice of operating system will be affected by available hardware, staff resources and skill, cost of purchase, maintenance, and projected future requirements.

Virtualization and outsourced support services offer the modern IT organization the promise of having to pay for only what it uses rather than building in excess capacity. This not only controls costs but offers opportunities for people both inside and outside the organization to provide expertise and value.

### **Interface**

The first electronic computer systems were controlled by means of switches and plugboards similar to those used by telephone operators at the time. Then came punch cards and finally a text-based terminal system similar to the Linux command line interface (CLI) in use today. The graphical user interface (GUI), with a mouse and buttons to click, was pioneered at Xerox PARC (Palo Alto Research Center) in the early 1970s and popularized by Apple Computer in the 1980s.

Today, operating systems offer both GUI and CLI interfaces, however, most consumer operating systems (Windows, macOS) are designed to shield the user from the ins and outs of the CLI.

## 2.2 Microsoft Windows

**Microsoft** offers different versions of its operating system according to the machine’s role: desktop or server? The desktop version of Windows has undergone various naming schemes with the current version (as of this writing) being simply **Windows 11**. While new versions of most Linux distributions come out twice a year, around March and September, new versions of Windows tend to be released only every few years. In all, there have been 16 versions of Windows since 1985. Backward compatibility is a priority for Microsoft, even going so far as to bundle virtual machine technology so that users can run older software.

**Windows Server** currently (as of this writing) is at version **2019** to denote the release date. The server can run a GUI but recently Microsoft, largely as a competitive response to Linux, has made incredible strides in its command line scripting capabilities through PowerShell and Windows Subsystem for Linux (WSL). There is also an optional Desktop Experience package which mimics a standard productivity machine. Microsoft also actively encourages enterprise customers to incorporate its Azure cloud service.

## 2.3 Apple macOS

**Apple** makes the **macOS** operating system, which is partially based on software from the FreeBSD project and has undergone UNIX certification. macOS is well known for being “easy to use”, and as such has continued to be favored by users with limited access to IT resources like schools and small businesses. It is also very popular with programmers due to its robust UNIX underpinnings.

On the server side, **macOS Server** is primarily aimed at smaller organizations. This low-cost addition to macOS desktop allows users to collaborate, and administrators to control access to shared resources. It also provides integration with iOS devices like the iPhone and iPad.

Some large corporate IT departments allow users to choose macOS since users often require less support than standard Microsoft productivity deployments. The continued popularity of macOS has ensured healthy support from software vendors. macOS is also quite popular in the creative industries such as graphics and video production. For many of these users, application choice drives the operating system decision. Apple hardware, being integrated so closely with the operating system, and their insistence on adherence to standards in application programming gives these creative professionals a stable platform to perform many computing-intense functions with fewer concerns about compatibility.

## 2.4 Linux

Linux users typically obtain an operating system by downloading a distribution. A Linux distribution is a bundle of software, typically comprised of the Linux kernel, utilities, management tools, and even some application software in a package which also includes the means to update core software and install additional applications.

The distribution takes care of setting up the storage, building the kernel and installing hardware drivers, as well as installing applications and utilities to make a fully functional computer system. The organizations that create distributions also include tools to manage the system, a package manager to add and remove software, as well as update programs to provide security and functionality patches.

The number of Linux distributions available numbers in the hundreds, so the choice can seem daunting at first. However, the decision points are mostly the same as those highlighted for choosing an operating system.

### **Role**

With Linux, there are multiple options to choose from depending on organizational needs. The variety of distributions and accompanying software allows the operating system to be significantly more flexible and customizable. Distributions are available for a much wider variety of systems, from commercial offerings for the traditional server or desktop roles, to specialized distributions designed to turn an old computer into a network firewall; from distributions created to power a supercomputer, to those that enable embedded systems. These might focus on running application or web servers, productivity desktops, point-of-sale systems, or even tools dedicated to electronics design or statistical computing.

### **Function**

Governments and large enterprises may also limit their choices to distributions that offer commercial support because paying for another tier of support may be better than risking extensive outages. For the most part, concerns over security have been addressed through the large open source community, which monitors kernel changes for vulnerabilities and provides bug reporting and fixes at a much larger scale than closed source vendors can achieve.

Support for necessary applications may vary and is, therefore, an additional consideration. Often application vendors choose a subset of distributions to support. Different distributions have different versions of key libraries, and it is difficult for a company to support all these different versions. However, some applications like Firefox and LibreOffice are widely supported and available for all major distributions.

### **Life Cycle**

Most distributions have both major and minor update cycles to introduce new features and fix existing bugs. Additionally, there are development packages where users can contribute code and submit patches for possible inclusion into new releases.

Linux distributions can be broadly classed in two main categories: enthusiast and enterprise. An enthusiast distribution such as openSUSE’s Tumbleweed has a fast update cycle, is not supported for enterprise and may not contain (or drop) features or software in the next version that are in the current one. Red Hat’s Fedora project uses a similar method of development and release cycle, as does Ubuntu Desktop.

Enterprise distributions are almost the exact opposite, in that they take care to be stable and consistent, and offer enterprise-grade support for extended periods, anywhere from 5-13 years in the case of SUSE. Enterprise distributions are fewer by far, being offered mainly by Red Hat, Canonical and SUSE.

Application software may be written such that it only supports a specific release of a distribution, requiring users to remain on an older, less secure operating system than they might like. Therefore, some Linux releases are considered to have long-term support (LTS) of 5 years or more while others are only supported for two years or less.

### **Stability**

Some distributions offer stable, testing, and unstable releases. When choosing an unstable release for required features, consideration must be given to the fact that those features may change at any point during the development cycle. When features have been integrated into the system for a long time, with most of the bugs and issues addressed, the software moves through testing into the stable release.

Other releases depend on beta distributions. For instance, the Fedora distribution releases beta or pre-releases of its software ahead of the full release to minimize bugs. Fedora is often considered the community-oriented beta release of RedHat. Features are added and changed in the Fedora release before finding their way into the enterprise-ready RedHat distribution.

openSUSE and its enterprise counterpart SLES (SUSE Linux Enterprise Server) are similar, in that the community edition is used as a testing ground for the features and functions that will eventually be migrated into the enterprise version. Previously somewhat dissimilar, later versions of the openSUSE and SLES distribution codebases are nearly identical, allowing for easier migration of features and code from one to the other.

**Consider This**

The Debian distribution warns users about the pitfalls of using the “sid” (unstable) release with the following warning:

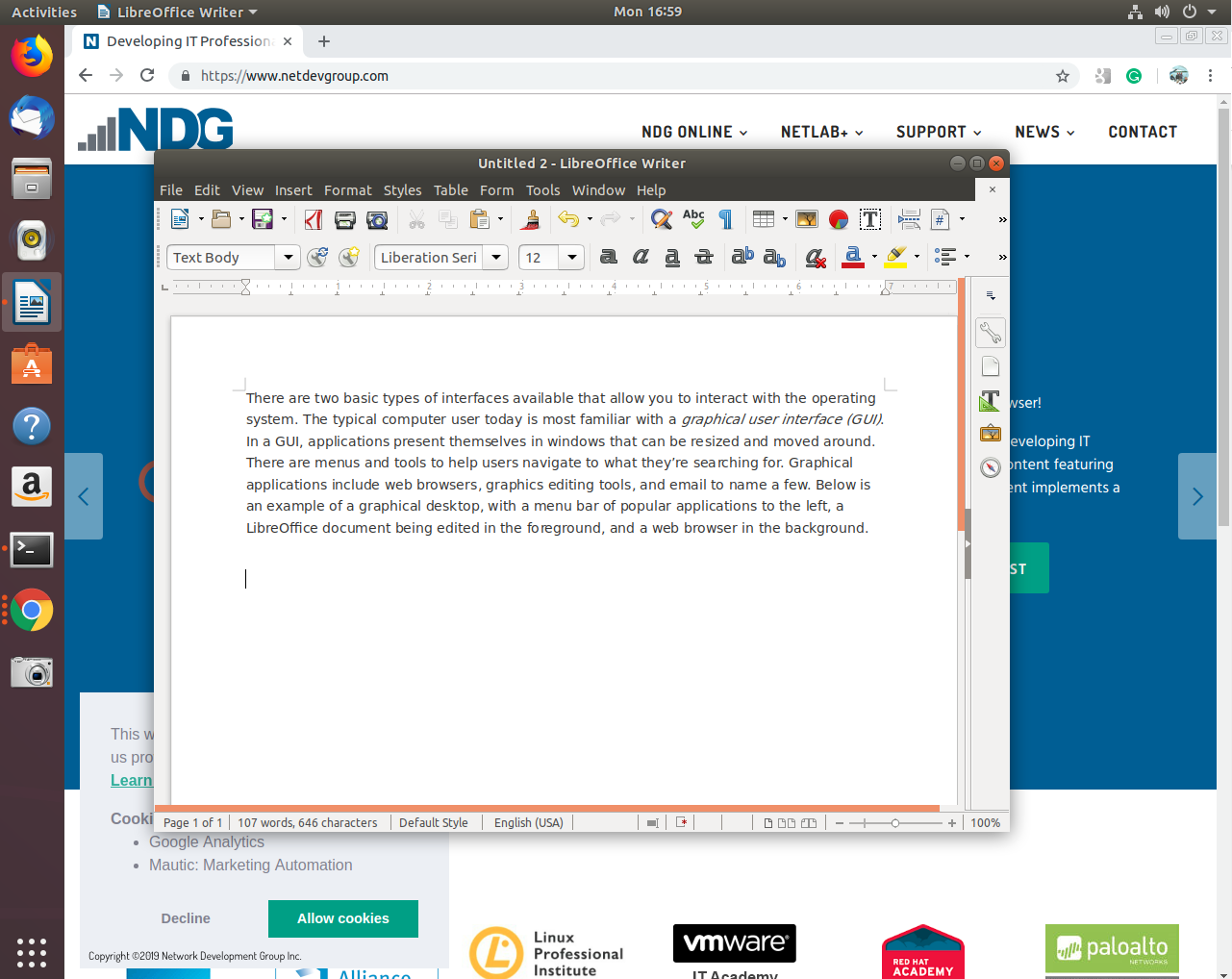
‘"sid" is subject to massive changes and in-place library updates. This can result in a very "unstable" system which contains packages that cannot be installed due to missing libraries, dependencies that cannot be fulfilled, etc. Use it at your own risk!’

### **Cost**

Your chosen Linux distribution itself might be zero cost, but paying for support may be worthwhile depending on organizational needs and capabilities.

### **Interface**

Like most operating systems, Linux can be used in one of two ways: graphical (GUI) and non-graphical (CLI). Below is an example of a graphical desktop, with a menu bar of popular applications to the left, a LibreOffice document being edited in the foreground, and a web browser in the background.



In graphical mode, users can have several different windows with terminal applications (shells) open, which is very helpful when performing tasks on multiple remote computers. Administrators and users can log-in with their username and password through a graphical interface.

The second type of interface, the CLI, is a text-based interface to the computer, where the user types in a command and the computer then executes it. The CLI environment is provided by an application on the computer known as a terminal. ‌⁠​⁠​ The terminal accepts what the user types and passes to a shell. The shell interprets what the user has typed into instructions that can be executed by the operating system. If output is produced by the command, then this text is displayed in the terminal. If problems with the command are encountered, then an error message is displayed.

The CLI starts with a text-based login as shown below. In a successful login, after being prompted for a username and password, you are taken to a CLI shell customized for the particular user.

ubuntu 18.04 ubuntu tty2

ubuntu login:

In CLI mode there are no windows to move around. Text editors, web browsers, and email clients are all presented in text format only. This is how UNIX operated before graphical environments were the norm. Most servers run in this mode too, since people don’t log into them directly, making a graphical interface a waste of resources. Here is an example of a CLI screen after logging in:

ubuntu 18.04 ubuntu tty2

ubuntu login: sue

Password:

The programs included with the Ubuntu system are free software;

the exact distribution terms for each program are described in the

individual files in /usr/share/doc/\*/copyright.

‌⁠​​⁠​

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by

applicable law.

Welcome to Ubuntu 18.04 LTS (GNU/Linux 4.4.0-72-generic x86\_64)

\* Documentation: https://help.ubuntu.com/

212 packages can be updated.

91 updates are security updates.

sue@ubuntu:~$ w

17:27:22 up 14 min, 2 users, load average: 1.73, 1.83, 1.69

USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT

sue tty2 20:08 14.35 0.05s 0.00s w

The original login prompt is at the top, with newer text added below. During login there may be some text displayed called the message of the day (MOTD). This is an opportunity for the systems administrator to pass information to users, or just make a silly joke. Following the MOTD is the command prompt, in the example above, the user has entered the w command which shows who is logged in. As new commands are entered and processed, the window scrolls up and older text is lost across the top. The terminal itself is responsible for keeping any history, such as to allow the user to scroll up and see previously entered commands. As far as Linux is concerned, what is on the screen is all that there is. There’s nothing to move around.

## 2.4.1 Linux Distributions

### **Red Hat**

**Red Hat** started as a simple distribution that introduced the Red Hat Package Manager (RPM). The developer eventually formed a company around it, which tried to commercialize a Linux desktop for business. Over time, Red Hat started to focus more on the server applications, such as web- and file-serving and released **Red Hat Enterprise Linux (RHEL)**, which was a paid service on a long release cycle. The release cycle dictates how often software is upgraded. A business may value stability and want long release cycles, while a hobbyist or a startup may want the latest software and opt for a shorter release cycle. To satisfy the latter group, Red Hat sponsors the **Fedora Project** which makes a personal desktop comprising the latest software but is still built on the same foundations as the enterprise version.

Because everything in Red Hat Enterprise Linux is open source, a project called **CentOS** came to be. It recompiled all the RHEL packages (converting their source code from the programming language they were written into language usable by the system) and gave them away for free. CentOS and others like it (such as Scientific Linux) are largely compatible with RHEL and integrate some newer software, but do not offer the paid support that Red Hat does.

**Scientific Linux** is an example of a specific-use distribution based on Red Hat. The project is a Fermilab-sponsored distribution designed to enable scientific computing. Among its many applications, Scientific Linux is used with particle accelerators including the Large Hadron Collider at CERN.

### **SUSE**

**SUSE**, originally derived from **Slackware**, was one of the first comprehensive Linux distributions, it has many similarities to Red Hat Enterprise Linux. The original company was purchased by Novell in 2003, which was then purchased by the Attachmate Group in 2011. The Attachmate group then merged with Micro Focus International in 2014, and in 2018 SUSE announced plans to go forward as an independent business. Through all of the mergers and acquisitions, SUSE has managed to continue and grow.

While SUSE Linux Enterprise contains proprietary code and is sold as a server product, **openSUSE** is a completely open, free version with multiple desktop packages similar to CentOS and Linux Mint.

### **Debian**

**Debian** is more of a community effort, and as such, also promotes the use of open source software and adherence to standards. Debian came up with its own package management system based on the .deb file format. While Red Hat leaves non-Intel and AMD platform support to derivative projects, Debian supports many of these platforms directly.

**Ubuntu** is the most popular Debian-derived distribution. It is the creation of **Canonical**, a company that was made to further the growth of Ubuntu and makes money by providing support. Ubuntu has several different variants for desktop, server and various specialized applications. They also offer an LTS version that is kept up-to-date for 3 years on desktops and 5 years on servers, which gives developers and the companies they work for confidence to build solutions based on a stable distribution.

**‌⁠⁠Linux Mint** was started as a fork of Ubuntu Linux, while still relying upon the Ubuntu repositories. There are various versions, all free of cost, but some include proprietary codecs, which cannot be distributed without license restrictions in certain countries.

### **Android**

Linux is a kernel, and many of the commands covered in this course are actually part of the GNU package. That is why some people insist on using the term ***GNU/Linux*** instead of Linux alone.

**Android**, sponsored by Google, is the world’s most popular Linux distribution. It is fundamentally different from its counterparts. Android uses the **Dalvik** virtual machine with Linux, providing a robust platform for mobile devices such as phones and tablets. However, lacking the traditional packages that are often distributed with Linux (such as GNU and Xorg), Android is generally incompatible with desktop Linux distributions.

This incompatibility means that a Red Hat or Ubuntu user cannot download software from the Google Play store. Likewise, a terminal emulator in Android lacks many of the commands of its Linux counterparts. It is possible, however, to use BusyBox with Android to enable most commands to work.

### **Other**

**Raspbian** is a specialized Linux distribution optimized to run on **Raspberry Pi** hardware. This combination has seen significant use in training for programmers and hardware designers at all levels. Its low cost and ease of use have made it a favorite of educators worldwide, and many add-on devices are available to extend its capabilities into the physical world. There is a multitude of labs and projects available that teach everything from environmental monitoring to circuit design, machine learning, and robotics.

**Linux From Scratch (LFS)** is more of a learning tool than a working distribution. This project consists of an online book, and source code, with “step-by-step instructions” for building a custom Linux distribution from the source code up. This “distribution” embodies the true spirit of Linux whereby users can modify any aspect of the operating system and learn how all the pieces work together. It’s also a good starting point for anyone who needs specialized functionality or an ultra-compact build for an embedded system project.

We have discussed the distributions explicitly mentioned in the Linux Essentials objectives. Be aware that there are hundreds, if not thousands more that are available. While there are many different distributions of Linux, many of the programs and commands remain the same or are very similar.



## 2.4.2 Embedded Systems

Linux started out as something that would only run on a computer like Linus Torvald's: an Intel 386 PC with a specific hard drive controller, but since anyone could add to or change Linux, people started building support for other hardware. Eventually, Linux started supporting other chips with an emphasis on small size and low power consumption.

Because of this flexibility, a significant number of device makers have used Linux as the operating system for their hardware products. Today we call these embedded systems because they are designed to do a specific task on hardware optimized for only that purpose. These systems encompass a tremendous diversity of devices that are used today, from cell phones to smart TVs and appliances, to remote monitoring systems for pipelines and factories.

As Linux evolved, specialized processor chips were developed for consumer and industrial devices to take advantage of its capabilities. Support for Linux has become so ubiquitous that it is possible to prototype and bring to market new devices using off-the-shelf components. The rise of cheap, small, adaptable single-board computers like the Raspberry Pi has given experimenters and entrepreneurs everywhere tools to quickly build custom solutions, powered by Linux, that would have taken months of work by specialized teams just a few years ago.

While consumers are familiar with embedded Linux entertainment devices like digital video recorders (DVRs) and “smart TVs,” the real impact of embedded Linux is just starting to be realized. The internet of things (IoT) is just ramping up with cheap, ubiquitous devices being deployed on everything from oil wells to solar generating farms. These networks of smart sensors and controllers enable engineers to adjust critical processes in real time while monitoring and reporting back to central control stations. As more processes are being monitored and more data is being integrated with machine learning and artificial intelligence (AI) we can anticipate gains in efficiency, safety and productivity only dreamed of by past generations.

#### [**Chapter 3 - Working in Linux**](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/#c1)

**Apache HTTPD**

Apache HTTPD is a server application program, or daemon, that manages web page requests on an Apache web server.  
[Section 3.2.2.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.2.1)

**C**

A compiled computer programming language. C is the language in which Linux is written.  
[Section 3.5](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.5)

**Firefox**

An open source, cross-platform web browser developed by the Mozilla Foundation.  
[Section 3.2.3.4](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.3.4)

**GIMP**

An open source application which handles 2D image manipulation.  
[Section 3.2.3.2](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.3.2)

**Java**

A compiled, object-oriented programming language owned by Oracle.  
[Section 3.5](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.5)

**JavaScript**

A cross-platform scripting language for adding interactive elements to web pages, that is in wide use across the internet.  
[Section 3.5](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.5)

**LibreOffice**

An open source office suite forked from Open Office. It includes tools that strive for compatibility with Microsoft Office in both features and file formats.  
[Section 3.2.3.3](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.3.3)

**MariaDB**

An open source database application forked from MySQL. It records data written to it by dynamic web applications.  
[Section 3.2.2.3](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.2.3)

**MySQL**

A relational database management system used for web development.  
[Section 3.2.2.3](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.2.3)

**NFS**

The native file sharing protocol for Unix and Linux. Short for Network File System.  
[Section 3.2.2.5](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.2.5)

**NGINX**

An open source web server based out of Russia focused on the use of more modern UNIX kernels.  
[Section 3.2.2.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.2.1)

**Nextcloud**

An open source private cloud server software forked from ownCloud. It is provided under a GNU AGPLv3 that can be deployed and administered internally by an organization.  
[Section 3.2.2.2](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.2.2)

**OpenOffice.org**

An open source office suite which has been discontinued.  
[Section 3.2.3.3](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.3.3)

**PHP**

A scripting language designed to create dynamic web pages.  
[Section 3.5](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.5)

**Perl**

An interpreted programming language popular with system adminsitrators. It was orginally developed to perform text manipulation.  
[Section 3.5](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.5)

**Python**

A scripting language which simplifies complex tasks, has excellent statistical processing, and it popular in academia.  
[Section 3.5](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.5)

**Samba**

A file sharing application ideal for use with Windows systems.  
[Section 3.2.2.5](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.2.5)

**Thunderbird**

A full-featured, open source desktop email client developed by the Mozilla Foundation.  
[Section 3.2.3.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.3.1)

**apt-get**

A front-end program for Debian package management.  
[Section 3.4.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.4.1)

**console**

Traditionally used to refer to a physical terminal.  
[Section 3.1.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.1.1)

**dpkg**

The package management system used on Debian derived Linux distributions.  
[Section 3.4.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.4.1)

**ownCloud**

An open source file hosting service. The project was launched in 2010 by Frank Karlitschek to provide software to store, sync and share data from private cloud servers.  
[Section 3.2.2.2](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.2.2)

**password issues**

[Section 3.6.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.6.1)

**privacy issues and tools**

[Section 3.6.2](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.6.2)

**rpm**

The package management system used on Red Hat derived Linux distributions. According to the Linux Standards Base, the standard package management system is RPM.  
[Section 3.4.2](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.4.2)

**shell**

The user interface of a Linux system. It interacts with the user by accepting commands, passes them to the kernel for execution, and displays any output to the terminal.  
[Section 3.3.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.3.1)

**terminal**

The environment or device in which the user interacts with the software. It can also refer to a graphical program which emulates a console.  
[Section 3.1.1](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.1.1)

**use of common open source applications in presentations and projects**

[Section 3.2.3.3](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.2.3.3)

**using a browser, privacy concerns, configuration options, searching the web and saving content**

[Section 3.6.2](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.6.2)

**yum**

A front-end tool for RPM package management.  
[Section 3.4.2](https://content.netdevgroup.com/contents/linux-essentials/GV3sPWsQ7r/3.4.2)

## 3.1 Navigating the Linux Desktop

To be a Linux systems administrator, it is necessary to be comfortable with Linux as a desktop operating system and have proficiency with basic Information and Communication Technology (ICT) skills. Using Linux for productivity tasks, rather than depending on Windows or Macintosh systems, accelerates learning by working with Linux tools on a daily basis. Systems administrators do far more than manage servers; they are often called upon to assist users with configuration issues, recommend new software, and update documentation among other tasks.

Most Linux distributions allow users to download a “desktop” installation package that can be loaded onto a USB key. This is one of the first things aspiring system administrators should do; download a major distribution and load it onto an old PC. This process is fairly straightforward, and tutorials are available online. The Linux desktop should be familiar to anyone who has used a PC or Macintosh with icons to select different programs and a “settings” application to configure things like user accounts, WiFi networks, and input devices. After familiarizing oneself with the Linux Graphical User Interface (GUI), or desktop, the next step is learning how to perform tasks from the command line.

3.1 Getting to the Command Line

The *command line interface (CLI)* is a simple text input system for entering anything from single-word commands to complicated scripts. Most operating systems have a CLI that provides a direct way of accessing and controlling the computer.

On systems that boot to a GUI, there are two common ways of accessing the command line—a GUI-based terminal, and a virtual terminal:

* A GUI terminal is a program within the GUI environment that emulates a terminal window. GUI terminals can be accessed through the menu system. For example, on a CentOS machine, you could click on **Applications** on the menu bar, then **System Tools** > and, finally, **Terminal**. If you have a search tools, you can search for **terminal**, as shown here.



* A virtual terminal can be run at the same time as a GUI but requires the user to log in via the virtual terminal before they can execute commands (as they would before accessing the GUI interface).

Each Linux desktop distribution is slightly different, but the application **terminal** or **x-term** will open a terminal window from the GUI. While there are subtle differences between the terms *console* and *terminal* window sessions, they are all the same from an administrators standpoint and require the same knowledge of commands to use.

Ordinary command line tasks are starting programs, parsing scripts, and editing text files used for system or application configuration. Most servers boot directly to a terminal, as a GUI can be resource intensive and is generally not needed to perform server-based operations.‌⁠​​

## 3.2 Applications

The kernel of the operating system is like an air traffic controller at an airport, and the applications are the airplanes under its control. The kernel decides which program gets which blocks of memory, it starts and kills applications, and it handles displaying text or graphics on a monitor.

Applications make requests to the kernel and in return receive resources, such as memory, CPU, and disk space. If two applications request the same resource, the kernel decides which one gets it, and in some cases, kills off another application to save the rest of the system and prevent a crash.

The kernel also abstracts some complicated details away from the application. For example, the application doesn’t know if a block of disk storage is on a solid-state drive, a spinning metal hard disk, or even a network file share. Applications need only follow the kernel’s Application Programming Interface (API) and therefore don’t have to worry about the implementation details. Each application behaves as if it has a large block of memory on the system; the kernel maintains this illusion by remapping smaller blocks of memory, sharing blocks of memory with other applications, or even swapping out untouched blocks to disk.

The kernel also handles the switching of applications, a process known as multitasking. A computer system has a small number of central processing units (CPUs) and a finite amount of memory. The kernel takes care of unloading one task and loading a new one if there is more demand than resources available. When one task has run for a specified amount of time, the CPU pauses it so that another may run. If the computer is doing several tasks at once, the kernel is deciding when to switch focus between tasks. With the tasks rapidly switching, it appears that the computer is doing many things at once.

When we, as users, think of applications, we tend to think of word processors, web browsers, and email clients, however, there are a large variety of application types. The kernel doesn’t differentiate between a user-facing application, a network service that talks to a remote computer, or an internal task. From this, we get an abstraction called a process. A process is just one task that is loaded and tracked by the kernel. An application may even need multiple processes to function, so the kernel takes care of running the processes, starting and stopping them as requested, and handing out system resources.

3.2.1 Major Applications

The Linux kernel can run a wide variety of software across many hardware platforms. A computer can act as a *server*, which means it primarily handles data on others’ behalf, or as a *desktop*, which means a user interacts with it directly. The machine can run software or be used as a development machine in the process of creating software. A machine can even adopt multiple roles as Linux makes no distinction; it’s merely a matter of configuring which applications run.

One resulting advantage is that Linux can simulate almost all aspects of a production environment, from development to testing, to verification on scaled-down hardware, which saves costs and time. A Linux administrator could run the same server applications on a desktop or inexpensive virtual server that are run by large internet service providers. Of course, a desktop would not be able to handle the same volume as a major provider would, but almost any configuration can be simulated without needing powerful hardware or server licensing.

Linux software generally falls into one of three categories:

* **Server Applications**

Software that has no direct interaction with the monitor and keyboard of the machine it runs on. Its purpose is to serve information to other computers, called *clients*. Sometimes server applications may not talk to other computers but only sit there and crunch data.

* **Desktop Applications**

Web browsers, text editors, music players, or other applications with which users interact directly. In many cases, such as a web browser, the application is talking to a server on the other end and interpreting the data. This is the “client” side of a client/server application.

* **Tools**

A loose category of software that exists to make it easier to manage computer systems. Tools can help configure displays, provide a Linux shell that users type commands into, or even more sophisticated tools, called compilers, that convert source code to application programs that the computer can execute.

The availability of applications varies depending on the distribution. Often application vendors choose a subset of distributions to support. Different distributions have different versions of key libraries, and it is difficult for a company to support all these different versions. Some applications, however, like Firefox and LibreOffice are widely supported and available for all major distributions.

The Linux community has come up with lots of creative solutions for both desktop and server applications. These applications, many of which make up the backbone of the Internet, are critical to understanding, and utilizing the power of Linux.

Most computing tasks can be accomplished by any number of applications in Linux. There are many web browsers, web servers, database servers, and text editors from which to choose. Evaluating application software is an important skill to be learned by the aspiring Linux administrator. Determining requirements for performance, stability, and cost are just some of the considerations needed for a comprehensive analysis.

## 3.2.2 Server Applications

Linux excels at running server applications because of its reliability and efficiency. The ability to optimize server operating systems with just needed components allows administrators to do more with less, a feature loved by startups and large enterprises alike.

## 3.2.2.1 Web Servers

One of the early uses of Linux was for web servers. A web server hosts content for web pages, which are viewed by a web browser using the **HyperText Transfer Protocol (HTTP)** or its encrypted flavor, **HTTPS**. The web page itself can either be static or dynamic. When the web browser requests a static page, the web server sends the file as it appears on disk. In the case of a dynamic site, the request is sent by the web server to an application, which generates the content.

**WordPress** is one popular example. Users can develop content through their browser in the WordPress application, and the software turns it into a fully functional dynamic website.

**Apache** is the dominant web server in use today. Apache was originally a standalone project, but the group has since formed the **Apache Software Foundation** and maintains over a hundred open source software projects. **Apache HTTPD** is the daemon, or server application program, that “serves” web page requests.

Another web server is **NGINX**, which is based out of Russia. It focuses on performance by making use of more modern UNIX kernels and only does a subset of what Apache can do. Over 65% of websites are powered by either NGINX or Apache.

## 3.2.2.2 Private Cloud Servers

As individuals, organizations, and companies start to move their data to the cloud, there is a growing demand for private cloud server software that can be deployed and administered internally.

The **ownCloud** project was launched in 2010 by Frank Karlitschek to provide software to store, sync and share data from private cloud servers. It is available in a standard open source GNU AGPLv3 license and an enterprise version that carries a commercial license.

The **Nextcloud** project was forked from ownCloud in 2016 by Karlitschek and has been growing steadily since then. It is provided under a GNU AGPLv3 and aims for “an open, transparent development process.”

Both projects focus on providing private cloud software that meets the needs of both large and small organizations that require security, privacy, and regulatory compliance. While several other projects aim to serve the same users, these two are by far the largest in terms of both deployment and project members.

## 3.2.2.3 Database Servers

Database server applications form the backbone of most online services. Dynamic web applications pull data from and write data to these applications. For example, a web program for tracking online students might consist of a front-end server that presents a web form. When data is entered into the form, it is written to a database application such as **MariaDB**. When instructors need to access student information, the web application queries the database and returns the results through the web application.

MariaDB is a community-developed fork of the **MySQL** relational database management system. It is just one of many database servers used for web development as different requirements dictate the best application for the required tasks.

A database stores information and also allows for easy retrieval and querying. Some other popular databases are **Firebird** and **PostgreSQL**. You might enter raw sales figures into the database and then use a language called **Structured Query Language (SQL)** to aggregate sales by product and date to produce a report.

3.2.2.4 Email Servers

Email has always been a widespread use for Linux servers. When discussing email servers, it is always helpful to look at the 3 different tasks required to get email between people:

* **Mail Transfer Agent (MTA)**

The most well known MTA (software that is used to transfer electronic messages to other systems) is **Sendmail**. **Postfix** is another popular one and aims to be simpler and more secure than Sendmail.

* **Mail Delivery Agent (MDA)**

Also called the **Local Delivery Agent**, it takes care of storing the email in the user’s mailbox. Usually invoked from the final MTA in the chain.

* **POP/IMAP Server**

The **Post Office Protocol (POP)** and **Internet Message Access Protocol (IMAP)** are two communication protocols that let an email client running on your computer talk to a remote server to pick up the email.

**Dovecot** is a popular POP/IMAP server owing to its ease of use and low maintenance. **Cyrus IMAP** is another option. Some POP/IMAP servers implement their own mail database format for performance and include the MDA if the custom database is desired. People using standard file formats (such as all the emails in one text file) can choose any MDA.

There are several significant differences between the closed source and open source software worlds, one being that of inclusion of other projects as components to a project or package.In the closed source world, **Microsoft Exchange** is shipped primarily as a software package/suite that includes all the necessary or approved components, all from Microsoft, so there are few if any options to make individual selections. In the open source world, many options can be modularly included or swapped out for package components, and indeed some software packages or suites are just a well-packaged set of otherwise individual components all harmoniously working together.

## 3.2.2.5 File Sharing

For Windows-centric file sharing, **Samba** is the clear winner. Samba allows a Linux machine to look and behave like a Windows machine so that it can share files and participate in a Windows domain. Samba implements the server components, such as making files available for sharing and certain Windows server roles, and also the client end so that a Linux machine may consume a Windows file share.

The **Netatalk** project lets a Linux machine perform as an Apple Macintosh file server. The native file sharing protocol for UNIX/Linux is called the **Network File System (NFS)**. NFS is usually part of the kernel which means that a remote file system can be mounted (made accessible) just like a regular disk, making file access transparent to other applications.

As a computer network becomes more substantial, the need for a directory increases. One of the oldest network directory systems is the **Domain Name System (DNS)**. It is used to convert a name like https://www.icann.org/ to an IP address like 192.0.43.7, which is a unique identifier of a computer on the Internet. DNS also holds global information like the address of the MTA for a given domain name. An organization may want to run their own DNS server to host their public-facing names, and also to serve as an internal directory of services. The **Internet Software Consortium** maintains the most popular DNS server, simply called bind after the name of the process that runs the service.

The DNS is focused mainly on computer names and IP addresses and is not easily searchable. Other directories have sprung up to store information such as user accounts and security roles. The **Lightweight Directory Access Protocol (LDAP)** is one common directory system which also powers Microsoft’s Active Directory. In LDAP, an object is stored in a tree, and the position of that object on the tree can be used to derive information about the object and what it stores. For example, a Linux administrator may be stored in a branch of the tree called “IT Department,” which is under a branch called “Operations.” Thus one can find all the technical staff by searching under the “IT Department” branch. **OpenLDAP** is the dominant program used in Linux infrastructure.

One final piece of network infrastructure to discuss here is called the **Dynamic Host Configuration Protocol (DHCP)**. When a computer boots up, it needs an IP address for the local network so it can be uniquely identified. DHCP’s job is to listen for requests and to assign a free address from the DHCP pool. The Internet Systems Consortium (known until January 2004 as the Internet Software Consortium) also maintains the **ISC DHCP** server, which is the most common open source DHCP server.

## 3.2.3 Desktop Applications

The Linux ecosystem has a wide variety of desktop applications. There are games, productivity applications, creative tools, web browsers and more.

## 3.2.3.1 Email

The Mozilla Foundation came out with **Thunderbird**, a full-featured desktop email client. Thunderbird connects to a POP or IMAP server, displays email locally, and sends email through an external SMTP server.

Other notable email clients are **Evolution** and **KMail** which are the GNOME and KDE projects' email clients. Standardization through POP and IMAP and local email formats means that it’s easy to switch between email clients without losing data.

## 3.2.3.2 Creative

For the creative types, there is **Blender**, **GIMP (GNU Image Manipulation Program)**, and **Audacity** which handle 3D movie creation, 2D image manipulation, and audio editing respectively. They have had various degrees of success in professional markets. Blender is used for everything from independent films to Hollywood movies, for example. GIMP supports high-quality photo manipulation, original artwork creation, graphic design elements, and is extensible through scripting in multiple languages. Audacity is a free and open source audio editing tool that is available on multiple operating systems.

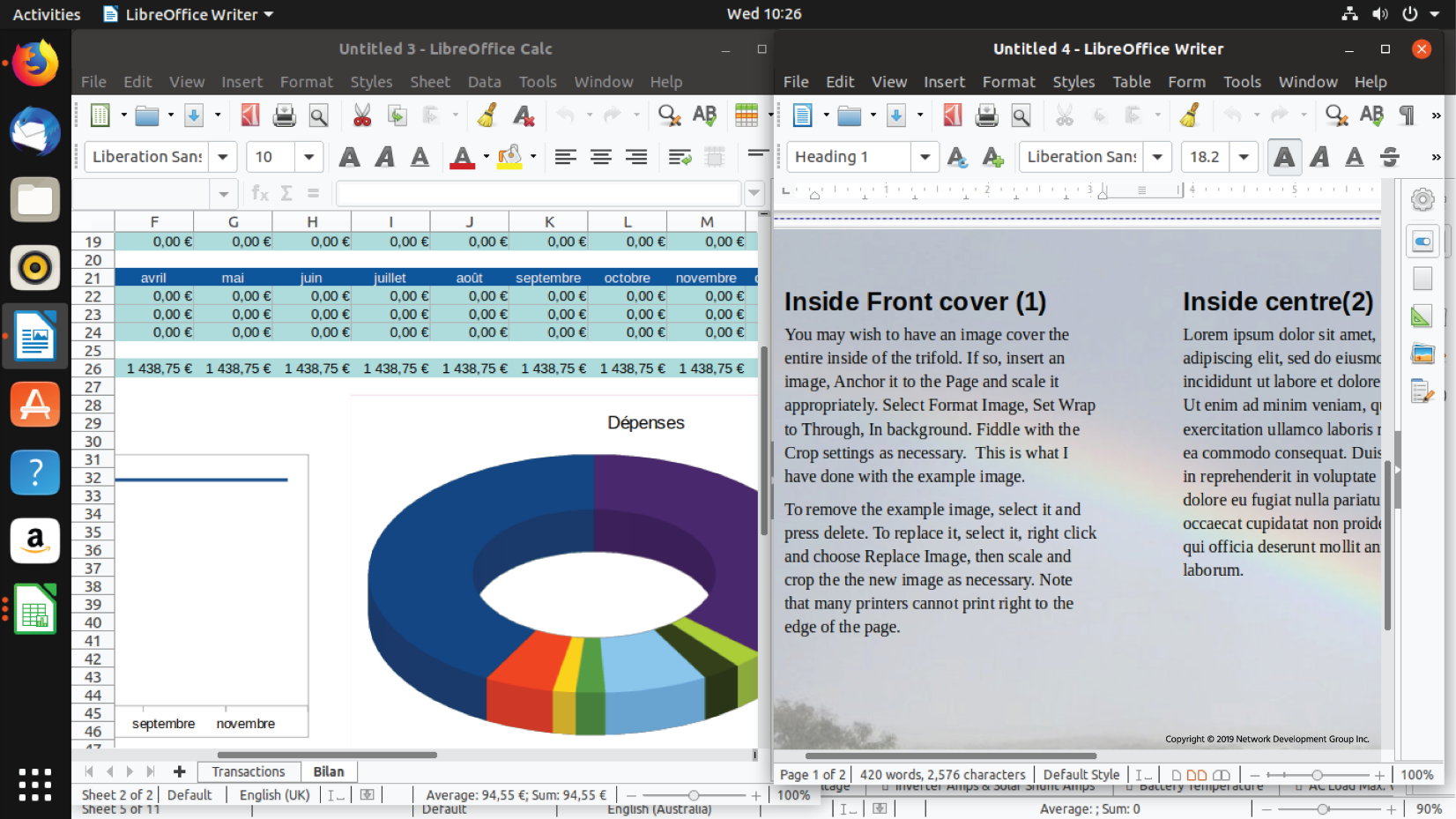
3.2.3.3 Productivity

Use of common open source applications in presentations and projects is one way to strengthen Linux skills. The basic productivity applications, such as a word processor, spreadsheet, and presentation package are valuable assets. Collectively they’re known as an *office suite*, primarily due to Microsoft Office, the dominant player in the market.

**LibreOffice** is a fork of the **OpenOffice** (sometimes called **OpenOffice.org**) application suite. Both offer a full office suite, including tools that strive for compatibility with Microsoft Office in both features and file formats.

Shown below is the spreadsheet and the document editor of LibreOffice. Note how the spreadsheet, **LibreOffice Calc**, is not limited to rows and columns of numbers. The numbers can be the source of a graph, and formulas can be written to calculate values based on information, such as pulling together interest rates and loan amounts to help compare different borrowing options.

Using **LibreOffice Writer**, a document can contain text, graphics, data tables, and much more. You can link documents and spreadsheets together, for example, so that you can summarize data in a written form and know that any changes to the spreadsheet will be reflected in the document.



LibreOffice can also work with other file formats, such as Microsoft Office or **Adobe Portable Document Format (PDF)** files. Additionally, through the use of extensions, LibreOffice can be made to integrate with Wiki software to give you a powerful intranet solution.

## 3.2.3.4 Web Browsers

Linux is a first class citizen for the **Mozilla Firefox** and **Google Chrome** browsers. Both are open source web browsers that are fast, feature-rich, and have excellent support for web developers. These packages are an excellent example of how competition helps to drive open source development – improvements made to one browser spur the development of the other browser. As a result, the Internet has two excellent browsers that push the limits of what can be done on the web, and work across a variety of platforms. Using a browser, while second nature for many, can lead to privacy concerns. By understanding and modifying the configuration options, one can limit the amount of information they share while searching the web and saving content.

## 3.3 Console Tools

Historically, the development of UNIX shows considerable overlap between the skills of software development and systems administration. The tools for managing systems have features of computer languages such as loops (which allow commands to be carried out repeatedly), and some computer programming languages are used extensively in automating systems administration tasks. Thus, one should consider these skills complementary, and at least a basic familiarity with programming is required for competent systems administrators.

## 3.3.1 Shells

At the basic level, users interact with a Linux system through a shell whether connecting to the system remotely or from an attached keyboard. The shell’s job is to accept commands, like file manipulations and starting applications, and to pass those to the Linux kernel for execution. ‌⁠​​⁠​  The Linux shell provides a rich language for iterating over files and customizing the environment, all without leaving the shell. For example, it is possible to write a single command line that finds files with contents matching a specific pattern, extracts useful information from the file, then copies the new information to a new file.

Linux offers a variety of shells to choose from, mostly differing in how and what can be customized, and the syntax of the built-in scripting language. The two main families are the **Bourne shell** and the **C shell**. The Bourne shell was named after its creator Stephen Bourne of Bell Labs. The C shell was so named because its syntax borrows heavily from the C language. As both these shells were invented in the 1970s, there are more modern versions, the **Bourne Again Shell (Bash)** and the **tcsh** (pronounced as tee-cee-shell). Bash is the default shell on most systems, though tcsh is also typically available.

Programmers have taken favorite features from Bash and tcsh and made other shells, such as the **Korn shell (ksh)** and the **Z shell (zsh)**. The choice of shells is mostly a personal one; users who are comfortable with Bash can operate effectively on most Linux systems. Other shells may offer features that increase productivity in specific use cases.

## 3.3.2 Text Editors

Most Linux systems provide a choice of text editors which are commonly used at the console to edit configuration files. The two main applications are **Vi** (or the more modern **Vim**) and **Emacs**. Both are remarkably powerful tools to edit text files; they differ in the format of the commands and how plugins are written for them. Plugins can be anything from syntax highlighting of software projects to integrated calendars.

Both Vi and Emacs are complex and have a steep learning curve, which is not helpful for simple editing of a small text file. Therefore, **Pico** and **Nano** are available on most systems and provide very basic text editing.

**Consider This**

The Nano editor was developed as a completely open source editor that is loosely based on Pico, as the license for Pico is not an open source license and forbids making changes and distributing it.

While Nano is simple and easy to use, it doesn’t offer the extensive suite of more advanced editing and key binding features that an editor like Vi does. Administrators should strive to gain some basic familiarity with Vi, though, because it is available on almost every Linux system in existence. When restoring a broken Linux system by running in the distribution’s recovery mode, Vi can be a critical tool, and the best time to learn Vim or any editor is before you desperately need it to fix a broken system.

## 3.4 Package Management

Every Linux system needs to add, remove, and update software. In the past this meant downloading the source code, setting it up, compiling it, and copying files onto each system that required updating. Thankfully, modern distributions use packages, which are compressed files that bundle up an application and its dependencies (or required files), greatly simplifying the installation by making the right directories, copying the proper files into them, and creating such needed items as symbolic links.

A package manager takes care of keeping track of which files belong to which package and even downloading updates from repositories, typically a remote server sharing out the appropriate updates for a distribution. In Linux, there are many different software package management systems, but the two most popular are those from Debian and Red Hat.

## 3.4.1 Debian Package Management

The Debian distribution, and its derivatives such as Ubuntu and Mint, use the Debian package management system. At the heart of Debian package management are software packages that are distributed as files ending in the .deb extension.

The lowest-level tool for managing these files is the dpkg command. This command can be tricky for novice Linux users, so the **Advanced Package Tool**, apt-get (a front-end program to the dpkg tool), makes management of packages easier. Additional command line tools which serve as front-ends to dpkg include aptitude and GUI front-ends like **Synaptic** and **Software Center**.

## 3.4.2 RPM Package Management

The **Linux Standards Base**, which is a **Linux Foundation** project, is designed to specify (through a consensus) a set of standards that increase the compatibility between conforming Linux systems. According to the Linux Standards Base, the standard package management system is RPM.

RPM makes use of an .rpm file for each software package. This system is what distributions derived from Red Hat, including Centos and Fedora, use to manage software. Several other distributions that are not Red Hat derived, such as SUSE, OpenSUSE, and Arch, also use RPM.

‌⁠​​⁠​ Like the Debian system, RPM Package Management systems track dependencies between packages. Tracking dependencies ensures that when a package is installed, the system also installs any packages needed by that package to function correctly. Dependencies also ensure that software updates and removals are performed properly.

The back-end tool most commonly used for RPM Package Management is the rpm command. While the rpm command can install, update, query and remove packages, the command line front-end tools such as yum and up2date automate the process of resolving dependency issues.

**Note**

A back-end program or application either interacts directly with a front-end program or is "called" by an intermediate program. Back end programs would not interact directly with the user. Basically, there are programs that interact with people (front-end) and programs that interact with other programs (back-end).

There are also GUI-based front-end tools such as **Yumex** and **Gnome PackageKit** that also make RPM package management easier.

Some RPM-based distributions have implemented the **ZYpp** (or **libzypp**) package management style, mostly openSUSE and SUSE Linux Enterprise, but mobile distributions MeeGo, Tizen and Sailfish as well.

The zypper command is the basis of the ZYpp method, and it features short and long English commands to perform functions, such as zypper in packagename which installs a package including any needed dependencies.

Most of the commands associated with package management require root privileges. The rule of thumb is that if a command affects the state of a package, administrative access is required. In other words, a regular user can perform a query or a search, but to add, update or remove a package requires the command to be executed as the root user.

## 3.5 Development Languages

It should come as no surprise that as software built on contributions from programmers, Linux has excellent support for software development. The shells are built to be programmable, and there are powerful editors included on every system. There are also many development tools available, and many modern programming languages treat Linux as a first-class citizen.

Computer programming languages provide a way for a programmer to enter instructions in a more human readable format, and for those instructions to eventually become translated into something the computer understands. Languages fall into one of two camps: interpreted or compiled. An interpreted language translates the written code into computer code as the program runs, and a compiled language is translated all at once.

Linux itself was written in a compiled language called **C**. The main benefit of C is that the language itself maps closely to the generated machine code so that a skilled programmer can write code that is small and efficient. When computer memory was measured in kilobytes, this was very important. Even with large memory sizes today, C is still helpful for writing code that must run fast, such as an operating system.

C has been extended over the years. There is **C++**, which adds object support to C (a different style of programming), and **Objective C** that took another direction and is in heavy use in Apple products.

The **Java** language puts a different spin on the compiled approach. Instead of compiling to machine code, Java first imagines a hypothetical CPU called the **Java Virtual Machine (JVM)** and then compiles all the code to that. Each host computer then runs JVM software to translate the JVM instructions (called bytecode) into native instructions.

The additional translation with Java might make you think it would be slow. However, the JVM is relatively simple so it can be implemented quickly and reliably on anything from a powerful computer to a low power device that connects to a television. A compiled Java file can also be run on any computer implementing the JVM!

Another benefit of compiling to an intermediate target is that the JVM can provide services to the application that usually wouldn’t be available on a CPU. Allocating memory to a program is a complex problem, but it’s built into the JVM. As a result, JVM makers can focus their improvements on the JVM as a whole, so any progress they make is instantly available to applications.

Interpreted languages, on the other hand, are translated to machine code as they execute. The extra computer power spent doing this can often be recouped by the increased productivity the programmer gains by not having to stop working to compile. Interpreted languages also tend to offer more features than compiled languages, meaning that often less code is needed. The language interpreter itself is usually written in another language such as C, and sometimes even Java! This means that an interpreted language is being run on the JVM, which is translated at runtime into actual machine code.

**JavaScript** is a high-level interpreted programming language that is one of the core technologies on the world wide web. It is similar to but fundamentally different from Java, which is a completely object-oriented programming language owned by Oracle. JavaScript is a cross-platform scripting language for adding interactive elements to web pages, that is in wide use across the internet. By using JavaScript libraries, web programmers can add everything from simple animations to complex server-side applications for internet users. JavaScript is continuously evolving to meet the functionality and security needs of internet users and is capable of being released under a GNU GPL License.

**Consider This**

The term object-oriented refers to programing that abstracts complex actions and processes so that the end user only deals with basic tasks. To visualize this concept, think of a machine that performs a complex set of tasks by simply pushing a button.

**Perl** is an interpreted language. Perl was originally developed to perform text manipulation. Over the years, it gained favor with systems administrators and continues to be improved and used in everything from automation to building web applications.

**PHP** is a language that was initially built to create dynamic web pages. A PHP file is read by a web server such as Apache. Special tags in the file indicate that parts of the code should be interpreted as instructions. The web server pulls all the different parts of the file together and sends it to the web browser. PHP’s main advantages are that it is easy to learn and available on almost any system. Because of this, many popular projects are built on PHP. Notable examples include WordPress (for blogging), cacti (for monitoring), and even parts of Facebook.

**Ruby** is another language that was influenced by Perl and Shell, along with many other languages. It makes complex programming tasks relatively easy, and with the inclusion of the Ruby on Rails framework, is a popular choice for building complex web applications. Ruby is also the language that powers many of the leading automation tools like **Chef** and **Puppet**, which make managing a large number of Linux systems much simpler.

**Python** is another scripting language that is in general use. Much like Ruby it makes complex tasks easier and has a framework called **Django** that makes building web applications very easy. Python has excellent statistical processing abilities and is a favorite in academia.

A computer programming language is just a tool that makes it easier to tell the computer what you want it to do. A library bundles common tasks into a distinct package that can be used by the developer. **ImageMagick** is one such library that lets programmers manipulate images in code. ImageMagick also ships with some command line tools that enable programmers to process images from a shell and take advantage of the scripting capabilities there.

**OpenSSL** is a cryptographic library that is used in everything from web servers to the command line. It provides a standard interface for adding cryptography into a Perl script, for example.

At a much lower level is the **C library**. The C library provides a basic set of functions for reading and writing to files and displays, and is used by applications and other languages alike.

3.6 Security

Administrators and computer users are increasingly aware of privacy concerns in both their personal and professional lives. High-profile data breaches have been in the news all too often recently, and the cost of these break-ins can reach into the millions of dollars for the institutions that fall victim to hackers and ransomware attacks. Many times the cause of these breaches is simply human error such as opening a suspicious email or entering passwords into a phony login page.

*Cookies* are the primary mechanism that websites use to track you. Sometimes this tracking is good, such as to keep track of what is in your shopping cart or to keep you logged in when you return to the site.

As you browse the web, a web server can send back the cookie, which is a small piece of text, along with the web page. Your browser stores this information and sends it back with every request to the same site. Cookies are normally only sent back to the site they originated from, so a cookie from example.com wouldn’t be sent to example.org.

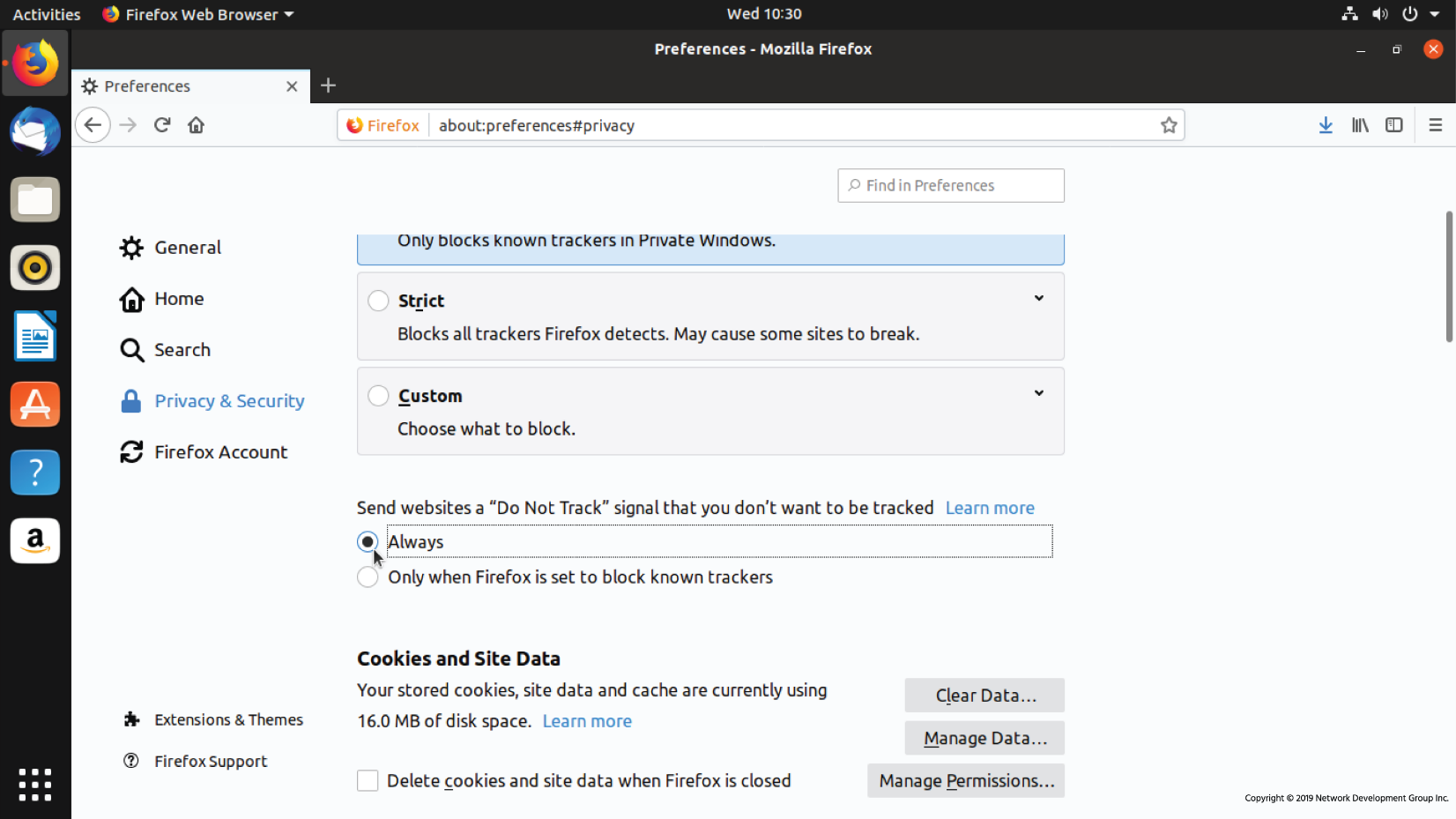
However, many sites have embedded scripts that come from third parties, such as a banner advertisement or Google analytics pixel. If both example.com and example.org have a tracking pixel, such as one from an advertiser, then that same cookie will be sent when browsing both sites. The advertiser then knows that you have visited both example.com and example.org.

With a broad enough reach, such as placement on social network sites with “Like” buttons and such, a website can gain an understanding of which websites you frequent and figure out your interests and demographics.

There are various strategies for dealing with this. One is to ignore it. The other is to limit the tracking pixels you accept, either by blocking them entirely or clearing them out periodically.

Browsers typically offer cookie-related settings; users can opt to have the browser tell the site not to track. This voluntary tag is sent in the request, and some sites will honor it. The browser can also be set never to remember third-party cookies and remove regular cookies (such as from the site you are browsing) after being closed.

Tweaking privacy settings can make you more anonymous on the Internet, but it can also cause problems with some sites that depend on third-party cookies. If this happens, you might have to explicitly permit some cookies to be saved.



Browsers also offer a *private* or *incognito* mode where cookies and tracking pixels are deleted upon exiting the window. This mode can be helpful if you would like to search for something without letting other websites know what you are looking for.

## 3.6.1 Password Issues

Good password management is essential to security in any computing environment. The Linux systems administrator is often the person responsible for setting and enforcing password policies for users at all levels. The most privileged user on any Linux system is root; this account is the primary administrator and is created when the operating system is installed. Often administrators will disable root access as the first line of defense against intrusion since computer hackers will try to gain root access in order to take control of the system.

There are many levels of access and various means of password management on a Linux system. When users are created, they are given different login permissions depending on what groups they are assigned to. For example, administrators can create and manage users while regular users cannot. Services that run on systems such as databases can also have login permissions with their own passwords and privileges. Additionally, there are specific passwords for accessing systems remotely through SSH, FTP, or other management programs.

Managing all these accounts, and their accompanying passwords is a complicated and necessary part of the systems administrator role. Passwords need to be complex enough not to be easily guessed by hackers, yet easy to remember for users. Increasingly users and administrators are turning to password manager programs to store login credentials in encrypted form. Another trend is two-factor authentication (2FA), a technique where a password is supplemented by a second “factor,” often a passcode sent to the user's phone or other devices. Keeping up with current security trends, while ensuring authorized users' ease of access, is an ongoing challenge that must be met.

3.6.2 Protecting Yourself

As you browse the web, you leave a digital footprint. Much of this information goes ignored; some of it is gathered to collect statistics for advertising, and some can be used for malicious purposes.

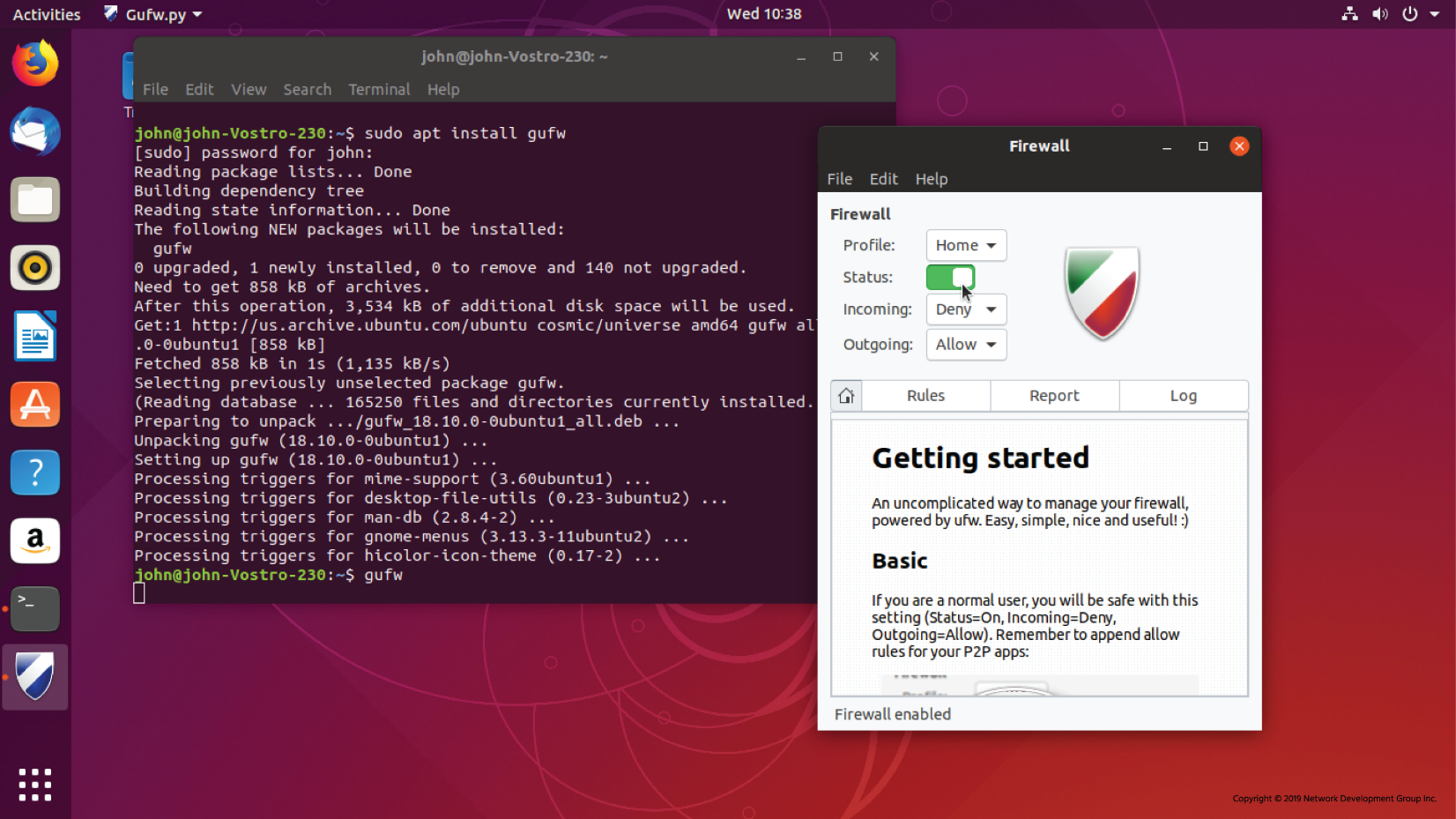
The easiest thing you can do is to use a good, unique password everywhere you go, especially on your local machine. A good password is at least 10 characters long and contains a mixture of numbers, letters (both upper and lower case) and special symbols. Use a *password manager* like **KeePassX** to generate passwords, and then you only need to have a login password to your machine and a password to open up your KeePassX file.

Also, limit the information you give to sites to only what is needed. While giving your mother’s maiden name and birthdate might help unlock your social network login if you lose your password, the same information can be used to impersonate you to your bank.

After that, make a point of checking for updates periodically. The system can be configured to check for updates on a regular basis. If there are security-related updates, you may be prompted immediately to install them.



Finally, you should protect your computer from accepting incoming connections. A *firewall* is a device that filters network traffic, and Linux has one built-in. If you are using Ubuntu, then the **Gufw** is a graphical interface to **Ubuntu’s Uncomplicated Firewall (UFW)**.



Under the hood, you are using **iptables**, which is the built-in firewall system. Instead of entering complicated iptables commands you use a GUI. While this GUI lets you build an effective policy for a desktop, it barely scratches the surface of what iptables can do.

## 3.6.3 Privacy Tools

The use of modern privacy tools, both at the server and user level, can help prevent system intrusions and unauthorized access to data.

The good news is that Linux is by default one of the most secure operating systems ever created. Many of the exploits that plague other operating systems simply won’t work on Linux due to the underlying architecture. However, there are still many known weaknesses that hackers can take advantage of so the proactive systems administrator is wise to deploy privacy tools that protect their users as well as the systems they use.

Encryption is probably the best-known and most widely-deployed privacy tool in use today. Administrators deploy encryption with authentication keys on almost every system that communicates with the outside world. One well-known example is the **HyperText Transfer Protocol Secure (HTTPS)** standard used on web servers to ensure that data transmitted between users and online resources cannot be intercepted as it travels on the open Internet.

Virtual private networks (VPN) have been in use by companies to connect their remote servers and employees for many years. Now they are gaining popularity amongst ordinary users looking to protect their privacy online. They work by creating an encrypted channel of communication between two systems, so the data transmitted between them is scrambled by an algorithm only the systems know.

The **Tor** project has long been involved in creating privacy tools like it’s **Tor Browser** that works by relaying internet requests through a network of servers that prevents websites and others from learning the identity of the person making the request.

These tools are constantly evolving and choosing which ones are appropriate for the users and systems involved is an essential part of the systems administrator's role.

## 3.7 The Cloud

No doubt you’ve heard of the cloud. Whether you’re using Google Docs for your homework or storing music and photos on iCloud, you probably have at least some of your digital content hosted on a cloud server somewhere.

Cloud computing has revolutionized the way we access technology. As Internet connectivity and speeds have increased, it’s become easier to move computing resources to remote locations where content can be accessed, manipulated and shared around the globe. Organizations are increasingly looking at the cloud as essential to their businesses and operations. The migration of an organization's IT applications and processes to cloud services, known as cloud adoption, is rapidly becoming a strategic business decision for many. With cloud adoption rising significantly all over the globe, cloud computing is not the catchphrase that it once was. Cloud computing is seen as one of the major disruptive technologies of the coming decade which will significantly transform businesses, economies, and lives globally.

Physically, a cloud can be described as computing resources from one or many off-site data centers which can be accessed over the internet. The cloud builds on the benefits of a data center and provides computing solutions to organizations who need to store and process data, and it allows them to delegate management of IT infrastructure to a third-party. The data and resources that organizations store in the cloud can include data, servers, storage, application hosting, analytics and a myriad of other services.

A cloud deployment model provides a basis for how cloud infrastructure is built, managed, and accessed. There are four primary cloud deployment models:

* **Public Cloud**: A public cloud is a cloud infrastructure deployed by a provider to offer cloud services to the general public and organizations over the Internet. In the public cloud model, there may be multiple tenants (consumers) who share common cloud resources. More than likely, many of us have accessed public cloud resources at some point through providers such as Amazon, Google, and other popular public cloud providers.
* **Private Cloud**: A private cloud is a cloud infrastructure that is set up for the sole use of a particular organization. When compared to a public cloud, a private cloud offers organizations a greater degree of privacy, and control over the cloud infrastructure, applications, and data. It can be hosted either on servers managed by the company that is using it or through a managed private cloud provider such as Rackspace or IBM.
* **Community Cloud**: A community cloud is a cloud infrastructure that is set up for the sole use by a group of organizations with common goals or requirements. The organizations participating in the community typically share the cost of the community cloud service. This option may be more expensive than the public cloud; however, it may offer a higher level of control and protection against external threats than a public cloud.
* **Hybrid Cloud**: A hybrid cloud is composed of two or more individual clouds, each of which can be a private, community, or public cloud. A hybrid cloud may change over time as component clouds join and leave. The use of such technology enables data and application portability. It also allows companies to leverage outside resources while retaining control of sensitive resources.

## 3.7.1 Linux in the Cloud

Linux plays a pivotal role in cloud computing. It powers 90% of the public cloud workload, most virtual servers are based on some version of the Linux kernel, and Linux is often used to host the applications behind cloud computing services. So what makes Linux uniquely suited to enabling cloud computing?

**Flexibility**

Cloud computing provides the capability to provision IT resources quickly and at any time. This agility enables rapid development and experimentation that, in turn, facilitates innovation which is essential for research and development, the discovery of new markets and revenue opportunities, creating new customer segments, and the development of new products.

As a result, cloud computing must compensate for the fact that each organization has a unique, evolving set of resource requirements.

Linux stands out here because it is highly adaptable. For starters, Linux is modular by design, and at the center of an enormous ecosystem of open source applications providing endless configuration options to suit various systems and use cases. On top of that, Linux scales efficiently, allowing it to run anything from a tiny remote sensor to an entire server farm.

**Accessibility**

In a traditional environment, IT resources are accessed from dedicated devices, such as a desktop or a laptop. In cloud computing, applications and data reside centrally and are accessed from anywhere over a network from any device, such as desktop, mobile, or thin client, and there is a version of Linux for every single one of these devices.

**Cost-Effective**

Cloud computing is attractive as it has the potential for consumers to reduce their IT costs. In cloud computing, consumers can unilaterally and automatically scale IT resources to meet workload demand, thereby eliminating overhead from underutilized resources. Additionally, the expenses associated with IT configuration, management, floor space, power, and cooling are reduced.

Cloud providers absorb these infrastructure costs but must remain a low-cost alternative. Choosing Linux is one of the most cost-effective solutions providers can deploy. Linux is one of the most power efficient operating systems, and the Linux kernel is completely free, as are many associated applications, utilities, and additional software components.

Enterprise and government organizations can opt to pay for commercially-supported distributions, which are still more cost-effective when compared to licensed competitors. Non-commercial distributions that support cloud computing also are a viable option for many organizations.

Not only can vendors pass these savings onto the customers, offering Linux-based solutions can be cheaper for the client to implement. Setting up Linux on their own systems eliminates expensive user licensing fees potentially associated with competing operating systems.

**Manageability**

While Linux began as a niche operating system, its widespread presence in the IT industry has made Linux use and administration a necessary skill for IT professionals. It is becoming increasingly easy for cloud vendors and consumers to acquire the necessary talent, or reallocate existing team members.

The nature of Linux, built on the C programming language, also lends itself to automated management tools. A significant portion of Linux servers operating in the cloud are created and managed by automated management programs rather than human operators. This process frees up administrators to monitor computing operations rather than manually configuring and updating systems.

**Security**

When using a cloud solution, especially a public cloud, an organization may have concerns related to privacy, external threats, and lack of control over the IT resources and data.

Linux can help offset these issues because it is one of the most secure and reliable operating systems available. Linux is open source, meaning its source code is available for anyone to obtain, review, and modify. This also means the code can be inspected for vulnerabilities and compatibility issues, resulting in an extensive community effort to rectify these issues and uphold the robust reputation of Linux.

**Virtualization**

Virtualization is one of the most significant advancements that has contributed to the enablement cloud of computing.

Linux is a multi-user operating system, which means that many different users can work on the same system simultaneously and for the most part can’t do things to harm other users. However, this does have limitations – users can hog disk space or take up too much memory or CPU resources and make the system slow for everyone. Sharing the system in multi-user mode also requires that everyone run as unprivileged users, so letting each user run their own web server, for example, is challenging.

Virtualization is the process where one physical computer, called the host, runs multiple copies of an operating system, each copy called a guest. These guest images can be pre-configured for specific functions to allow rapid deployment, often automatically, when needed. The host system runs software called a hypervisor that switches resources between the various guests just like the Linux kernel does for individual processes. With bare metal hypervisors, the hypervisor runs directly on computer hardware rather than on top of an OS freeing up more resources for guest images.

Virtualization works because servers spend most of their time idling and don’t need physical resources such as a monitor and keyboard. With software from companies like **VMWare** and **Openbox**, you can now take a powerful CPU and by using it to run multiple virtual machines administrators can optimize usage of physical resources and dramatically reduce costs over the previous one-machine, one-OS data center model. The main limitation is usually memory, however, with advances in hypervisor technology and CPUs, it is possible to put more virtual machines on one host than ever.

In a virtualized environment one host can run dozens of guest operating systems, and with support from the CPU itself, the guests don’t even know they are running on a virtual machine. Each guest gets its own virtual resources and communicates with the network on its own. It is not even necessary to run the same operating system on all the guests, which further reduces the number of physical servers needed.

Virtualization offers a way for an enterprise to lower power usage and reduce data center space over an equivalent fleet of physical servers. Guests are now just software configurations, so it is easy to spin up a new machine for testing and destroy it when its usefulness has passed.

Since it is possible to run multiple instances of an operating system on one physical machine and connect to it over the network, the location of the machine doesn’t matter. Cloud computing takes this approach and allows administrators to have virtual machines in a remote data center owned by another company, and only pay for the resources used. Cloud computing vendors can take advantage of scales of economy to offer computing resources at far lower prices than operating an on-site data center.

**Containers and Bare Metal Deployments**

With the rise of containerization technologies like **Docker** and **Kubernetes** application software is now being written that runs in a serverless environment. Essentially, programmers are creating software that does one single function of a system (like database processing or storage) that runs in a container. These containers are organized in pods that run within a node and can talk with each other, and the outside world if needed. Nodes, in turn, are organized and controlled by a master node that provides services to each component within the structure. Building applications in this way decouples each of the components from the others, and from the overhead of running an OS. Since each piece of the puzzle can be automatically destroyed and recreated by the master node they no longer need to be as robust as software that runs on top of an OS. Although these new programming architectures are in many ways bypassing the need for a traditional OS the underlying technology that makes them work is still Linux. So, working in Linux will increasingly be working within a development team that draws on the disciplines of programming, database design, networking, and systems administration to create the systems of the future.

#### [**Chapter 4 - Open Source Software and Licensing**](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/#c1)

**BSD**

An open source license which states that you may redistribute the source software and binaries as long as copyright notices are maintained and there is no implication that the original creator endorses your version. Short for Berkley Software Distribution.  
[Section 4.2.2](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.2)

**Creative Commons**

An organization created to address the intentions behind FOSS licenses for non-software entities.  
[Section 4.2.3](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.3)

**FLOSS**

Free/Libre/Open Source Software (FLOSS) is a collective term for the open source community which uses the term libre to define the difference between free from restrictions on software usage (libre) and free from cost (free).  
[Section 4.2.2](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.2)

**FOSS**

An open source body called Free Open Source Software (FOSS) which consists of both Free Software and Open Source as a collective in order to downplay the differences between the two organizations.  
[Section 4.2.2](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.2)

**Free Software**

The freedom to share, study and modify the underlying source code of the software.  
[Section 4.2.1](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.1)

**GPL**

Licenses developed by FSF that are meant to be included in the source code to ensure that all future variants and modifications of the original program continue to have the same freedom of use as the original. Short for GNU General Public Licenses.  
[Section 4.2.1](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.1)

**Open Source Software**

Software that meets the OSI Open Source software guidelines where no restrictions are placed on the use of the software regardless of the intended use.  
[Section 4.2.2](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.2)

**copyleft**

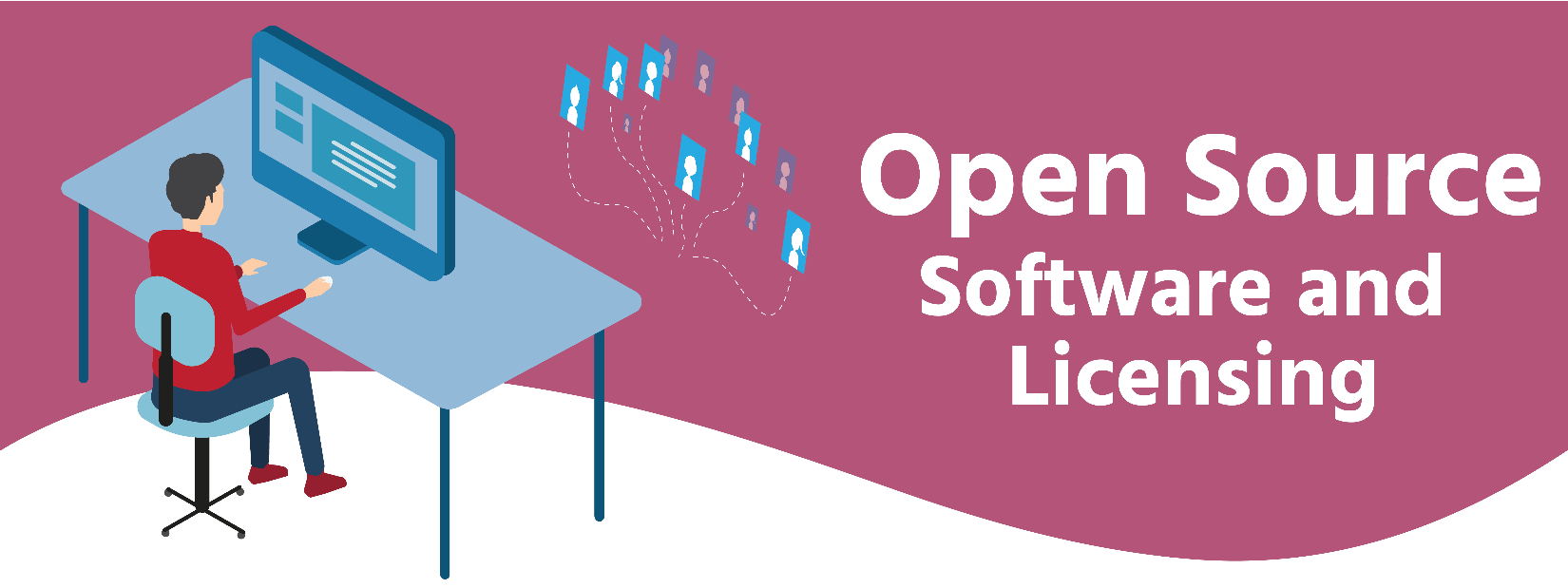
A philosophy held by the Free Software Foundation (FSF) that someone who modifies free software should be required to share any changes they have made.  
[Section 4.2.1](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.1)

**open source business models**

[Section 4.3](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.3)

**permissive**

Open source licenses that do not contain copyleft provisions and therefore are more permissive in how they allow software to be redistributed.  
[Section 4.2.2](https://content.netdevgroup.com/contents/linux-essentials/aGrhXa6xoz/4.2.2)



4.1 Introduction

Software projects take the form of *source code*, which is a human-readable set of computer instructions. Since source code is not understood directly by the computer, it must be compiled into machine instructions by a compiler. The compiler is a special program that gathers all of the source code files and generates instructions that can be run on the computer, such as by the Linux kernel.

Historically, commercial software has been sold under a *closed source license*, meaning that users have the right to use the machine code, also known as the binary or executable, but cannot see the source code. Often the license explicitly states that users may not attempt to reverse engineer the machine code back to source code to figure out what it does.

**Consider This**

Source code compiled into binary programs is one method of creating programs and running computing instructions. Another is the many types of interpreted languages, such as PERL, Python and even BASH scripting, where the code is not compiled, but fed to an interpreting program, typically a binary executable that understands and implements the instructions contained in the source code or scripts.

The development of Linux closely parallels the rise of *open source software*. Early on there was shareware, freely available programs where users did not necessarily have access to the source code. There were a lot of good things about this, but it was also problematic because malicious programs could be disguised as innocent-looking games, screensavers, and utilities.

Open source takes a source-centric view of software. The open source philosophy is that users have the right to obtain the software source code, and to expand and modify programs for their own use. This also meant the code could be inspected for backdoors, viruses, and spyware. By creating a community of developers and users, accountability for bugs, security vulnerabilities, and compatibility issues became a shared responsibility. This new, global community of computer enthusiasts was empowered by the growing availability of faster internet services and the world wide web.

There are many different variants of open source, but all agree that users should have access to the source code. Where they differ is in how one can, or must, redistribute changes.

Linux has adopted this philosophy to great success. Since Linux was written in the C programming language, and it mirrored the design and functionality of already established UNIX systems, it naturally became a forum where people could develop and share new ideas. Freed from the constraints of proprietary hardware and software platforms, large numbers of very skilled programmers have been able to contribute to the various distributions, making for software that is often more robust, stable, adaptable, and, frankly, better than the proprietary, closed source offerings which dominated the previous decades.

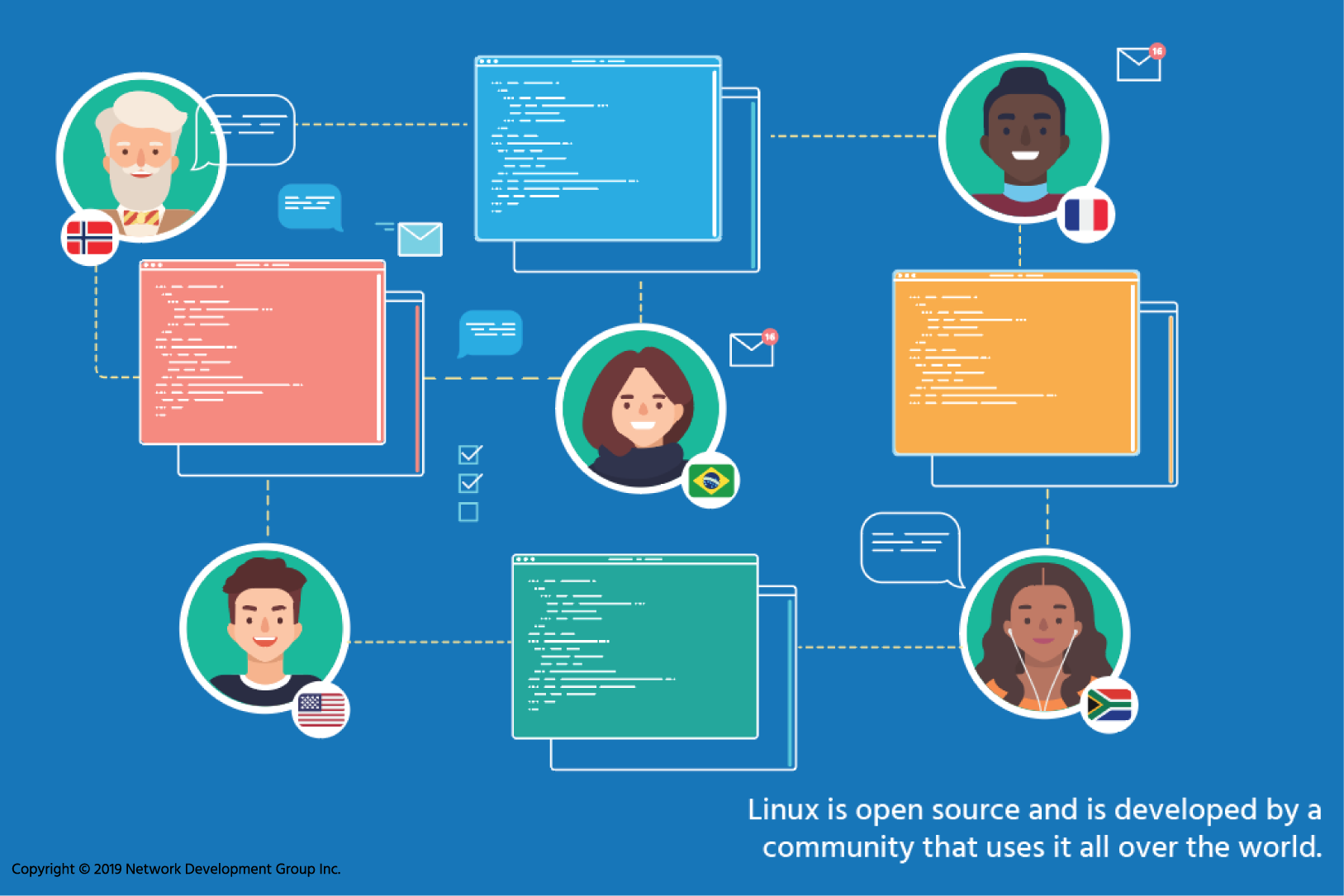
Large organizations were understandably suspicious about using software built in this new way, but over time they realized their best programmers were working on Linux-based open source projects in their spare time. Soon, Linux servers and open source programs began to outperform the expensive, proprietary systems already in place. When it came time to upgrade outdated hardware the same programmers, engineers, and system administrators who had started working on Linux as a hobby were able to convince their bosses to give Linux a try. The rest is, as they say, history.

Before the development of Linux, many corporate and scientific applications ran on proprietary UNIX systems. Companies, universities, and governments that run large server farms liked the stability and relative ease of application development these platforms offered.

**UNIX** was initially created in 1969. By its fourth edition, in 1973, it had been rewritten in the C programming language that is still prominent today. In 1984 the University of California Berkeley released 4.2BSD which introduced TCP/IP, the networking specification that underpins the Internet. By the early 1990’s, when Linux development started, different companies developing UNIX operating systems realized their systems needed to be compatible, and they started working on the X/Open specification that is still used today.

Over the years, computer scientists and the organizations that employ them have realized the benefit of systems that provide familiar tools and consistent ways of accomplishing specific tasks. The standardization of application programming interfaces (APIs) allows programs written for one specific UNIX or Linux operating system to be ported (converted) relatively easy to run on another. So, while proprietary UNIX systems are still in use throughout the world in environments where "certified" solutions are preferred, the interoperability of these systems alongside Linux computers is valued by industry, academia, and governments that use them.

The importance of standards organizations cannot be overstated. Groups like the **IEEE (Institute of Electrical and Electronics Engineers)** and **POSIX (Portable Operating System Interface)**, allow professionals from different companies and institutions to collaborate on specifications that make it possible for different operating systems and programs to work together. It doesn’t matter if a program is closed or open source, simple or complex, if it is written to these standards others will be able to use and modify it in the future. Every innovation in computing is built on the work of others who came before. Open source software is a collaboration of different people with different needs and backgrounds all working together to make something better than any one of them could have made individually. Standards are what makes this possible, and the many organizations that create, maintain and promote them are integral to the industry.



4.2 Open Source Licensing

When talking about buying software, there are three distinct components:

* **Ownership** – Who owns the intellectual property behind the software?
* **Money Transfer** – How does money change hands, if at all?
* **Licensing** – What do you get? What can you do with the software? Can you use it on only one computer? Can you give it to someone else?

In most cases, the ownership of the software remains with the person or company that created it. Users are only granted a license to use the software; this is a matter of copyright law. The money transfer depends on the business model of the creator. It’s the licensing that differentiates open source software from closed source software.

Two contrasting examples will get things started.

With Microsoft Windows, the Microsoft Corporation owns the intellectual property. The license itself, the **End User License Agreement (EULA)**, is a custom legal document that you must click through, indicating your acceptance, in order to install the software. Microsoft keeps the source code and distributes only binary copies through authorized channels. For most consumer products you are allowed to install the software on one computer and are not allowed to make copies of the disk other than for a backup. You are not allowed to reverse engineer the software. You pay for one copy of the software, which gets you minor updates but not major upgrades.

Linux is owned by Linus Torvalds. He has placed the code under a license called **GNU General Public License version 2 (GPLv2)**. This license, among other things, says that the source code must be made available to anyone who asks and that anyone is allowed to make changes. One caveat to this is that if someone makes changes and distributes them, they must put the changes under the same license so that others can benefit. GPLv2 also says that no one is allowed to charge for distributing the source code other than the actual costs of doing so (such as copying it to removable media).

In general, when someone creates something, they also get the right to decide how it is used and distributed. **Free and Open Source Software (FOSS)** refers to software where this right has been given up; anyone is allowed to view the source code and redistribute it. Linus Torvalds has done that with Linux – even though he created Linux he can’t forbid someone from using it on their computer because he has given up that right through the GPLv2 license.

Software licensing is a political issue, therefore it should come as no surprise that there are many different opinions. Organizations have come up with their own license that embodies their particular views, so it is easier to choose an existing license than come up with your own. For example, universities like the Massachusetts Institute of Technology (MIT) and University of California have come up with licenses, as have projects like the Apache Foundation. Also, groups like the Free Software Foundation have created their own licenses to further their agenda.

## 4.2.1 The Free Software Foundation

Two groups can be considered the most influential forces in the world of open source: the Free Software Foundation and the Open Source Initiative.

Only a few years after the development of the GNU project, Richard Stallman founded the **Free Software Foundation (FSF)** in 1985 with the goal of promoting free software. In this context, the word "free" does not refer to the price, but to the freedom to share, study, and modify the underlying source code. According to their website, the FSF believes that users should have "control over the technology we use in our homes, schools, and businesses".

FSF also advocates that software licenses should enforce the openness of modifications. It is their view that if someone modifies free software that they should be required to share any changes they have made when they share it again. This specific philosophy is called copyleft. According to FSF, "copyleft is a general method for making a program (or other work) free (in the sense of freedom, not "zero price"), and requiring all modified and extended versions of the program to be free as well".

The FSF also advocates against software patents and acts as a watchdog for standards organizations, speaking out when a proposed standard might violate the free software principles by including items like Digital Rights Management (DRM) which restrict what can be done with compliant programs.

The FSF have developed their own set of licenses which are free for anyone to use based on the original **GNU General Public License (GPL)**. FSF currently maintains GNU General Public License version 2 (GPLv2) and version 3 (GPLv3), as well as the GNU Lesser General Public Licenses version 2 (LGPLv2) and version 3 (LGPLv3). These licenses are meant to be included in the actual source code to ensure that all future variants and modifications of the original program continue to have the same freedom of use as the original. The GPL license and its variants are powerful legal tools to advance the cause of free software worldwide. What started off in 1983 as one man’s desire to share and improve software by letting others change it has ended up changing the world.

**Consider This**

The changes between GPLv2 and GPLv3 largely focused on using free software on a closed hardware device which has been coined Tivoization. TiVo is a company that builds a television digital video recorder on their own hardware and used Linux as the base for their software. While TiVo released the source code to their version of Linux as required under GPLv2, the hardware would not run any modified binaries. In the eyes of the FSF, this went against the spirit of the GPLv2, so they added a specific clause to version 3 of the license. Linus Torvalds agrees with TiVo on this matter and has chosen to stay with GPLv2.

## 4.2.2 The ‌⁠​​⁠​ Open Source Initiative

The **Open Source Initiative (OSI)** was founded in 1998 by Bruce Perens and Eric Raymond. They believed that the Free Software Foundation was too politically charged and that less extreme licenses were necessary, particularly around the copyleft aspects of FSF licenses. OSI believes that not only should the source be freely available, but also that no restrictions should be placed on the use of the software, no matter what the intended use. Unlike the FSF, the OSI does not have its own set of licenses. Instead, the OSI has a set of principles and adds licenses to that list if they meet those principles, called open source licenses. Software that conforms to an Open Source license is, therefore, open source software.

One type of Open Source license is the **BSD (Berkeley Software Distribution)** and its derivatives, which are much simpler than GPL. There are currently two actual "BSD" licenses approved by OSI, a 2-Clause and a 3-Clause. These licenses state that you may redistribute the source and binaries as long as you maintain copyright notices and don’t imply that the original creator endorses your version. In other words "do what you want with this software, just don’t say you wrote it." According to FSF, the original BSD license had a serious flaw in that it required developers to add a clause acknowledging the University of California, Berkeley in every advertisement for software licensed this way. As others copied this simple license, they included acknowledgment for their own institutions which led to over 75 such acknowledgments in some cases.

FSF licenses, such as GPLv2, are also open source licenses. However, many open source licenses such as BSD and MIT do not contain the copyleft provisions and are thus not acceptable to the FSF. These licenses are called permissive free software licenses because they are permissive in how you can redistribute the software. You can take BSD licensed software and include it in a closed software product as long as you give proper attribution.

Rather than dwell over the finer points of Open Source and Free Software, the community has started referring to them collectively as **Free and Open Source Software (FOSS)**. The English word "free" can mean "free as in lunch" (as in no cost) or "free as in speech" (as in no restrictions). This ambiguity led to the inclusion of the word "libre" to refer to the latter definition. Thus, we end up with **Free/Libre/Open Source Software (FLOSS)**.

4.2.3 Creative Commons

FOSS licenses are mostly related to software. People have placed works such as drawings and plans under FOSS licenses, but this was not the intent.

When software has been placed in the public domain, the author has relinquished all rights, including the copyright on the work. In some countries, this is the default when the work is done by a government agency. In some countries, copyrighted work becomes public domain after the author has died and a lengthy waiting period has elapsed.

The **Creative Commons (CC)** organization has created the Creative Commons Licenses which try to address the intentions behind FOSS licenses for non-software entities. CC licenses can also be used to restrict commercial use if that is the desire of the copyright holder. The CC licenses are made up of the following set of conditions the creator can apply to their work:

* **Attribution (BY)** – All CC licenses require that the creator must be given credit, without implying that the creator endorses the use.
* **ShareAlike (SA)** – This allows others to copy, distribute, perform, and modify the work, provided they do so under the same terms.
* **NonCommercial (NC)** – This allows others to distribute, display, perform, and modify the work for any purpose other than commercially.
* **NoDerivatives (ND)** – This allows others to distribute, display, and perform only original copies of the work. They must obtain the creator’s permission to modify it.

These conditions are then combined to create the six main licenses offered by Creative Commons:

* **Attribution (CC BY)** – Much like the BSD license, you can use CC BY content for any use but must credit the copyright holder.
* **Attribution ShareAlike (CC BY-SA)** – A copyleft version of the Attribution license. Derived works must be shared under the same license, much like in the Free Software ideals.
* **Attribution NoDerivs (CC BY-ND)** – You may redistribute the content under the same conditions as CC-BY but may not change it.
* **Attribution-NonCommercial (CC BY-NC)** – Just like CC BY, but you may not use it for commercial purposes.
* **Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)** – Builds on the CC BY-NC license but requires that your changes be shared under the same license.
* **Attribution-NonCommercial-NoDerivs (CC BY-NC-ND)** – You are sharing the content to be used for non-commercial purposes, but people may not change the content.
* **No Rights Reserved (CC0)** – This is the Creative Commons version of public domain.

## 4.3 Open Source Business Models

If all this software is free, how can anyone make money off of it?

First, you must understand there isn’t anything in the GPL that prohibits selling software. In fact, the right to sell software is part of the GPL license. Again, recall that the word free refers to freedom, not price. Companies that add value to these free programs are encouraged to make as much money as they can, and put those profits back into developing more and better software.

One of the simplest ways to make money is to sell support or warranty around the software. Companies like Canonical, the developer of Ubuntu, and Redhat have grown into huge enterprises by creating Linux distributions and tools that enable commercial users to manage their enterprises and offer products and services to their customers.

Many other open source projects have also expanded into substantial businesses. In the 1990s, Gerald Combs was working at an Internet service provider and started writing his own network analysis tool because similar tools at the time were costly. Over 600 people have now contributed to the project, called Wireshark. It is now often considered better than commercial offerings and led to a company being formed to sell products and support. Like many others, this company was purchased by a larger enterprise that supports its continued development.

Companies like Tivo have packaged hardware or add extra closed source software to sell alongside the free software. Appliances and embedded systems that use Linux are a multi-billion dollar business and encompass everything from home DVRs to security cameras and wearable fitness devices. Many consumer firewalls and entertainment devices follow this model.

Today, both large and small employers have individuals and sometimes whole groups devoted to working on open source projects. Technology companies compete for the opportunity to influence projects that will shape the future of their industries. Other companies dedicate resources towards projects they need for internal use. As more business is done on cloud resources, the opportunity for open source programmers continues to expand.

#### [**Chapter 5 - Command Line Skills**](https://content.netdevgroup.com/contents/linux-essentials/VDSPhwELN6/#c1)

**Bash**

The most commonly used shell for Linux distributions.  
[Section 5.2](https://content.netdevgroup.com/contents/linux-essentials/VDSPhwELN6/5.2)

**PATH environment variable**

Variable containing a list that defines which directories the shell looks in for commands.  
[Section 5.4.3](https://content.netdevgroup.com/contents/linux-essentials/VDSPhwELN6/5.4.3)

**echo**

Command that displays output in the terminal.  
[Section 5.4.1](https://content.netdevgroup.com/contents/linux-essentials/VDSPhwELN6/5.4.1)

**export**

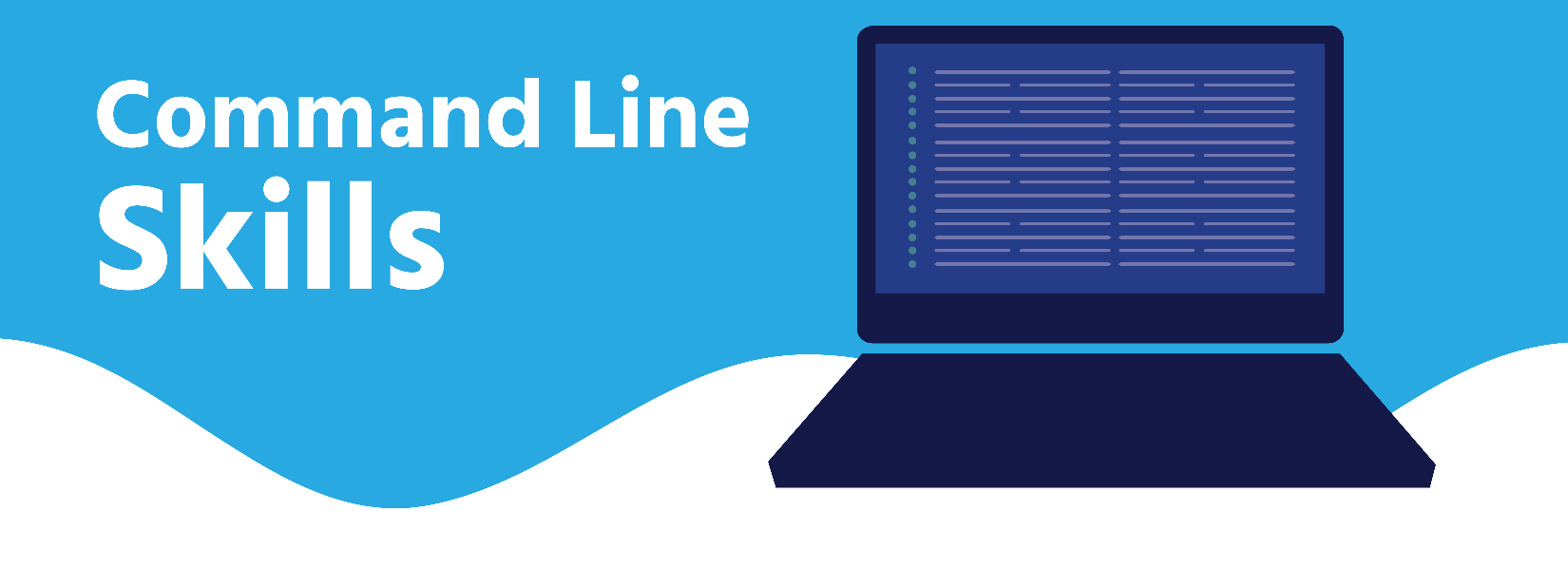
Command that turns a local variable into an environment variable.  
[Section 5.4.2](https://content.netdevgroup.com/contents/linux-essentials/VDSPhwELN6/5.4.2)

**history**

Command that outputs a list of previously executed commands.  
[Section 5.3.3](https://content.netdevgroup.com/contents/linux-essentials/VDSPhwELN6/5.3.3)

**type**

Command that determines information about command type.  
[Section 5.5](https://content.netdevgroup.com/contents/linux-essentials/VDSPhwELN6/5.5)

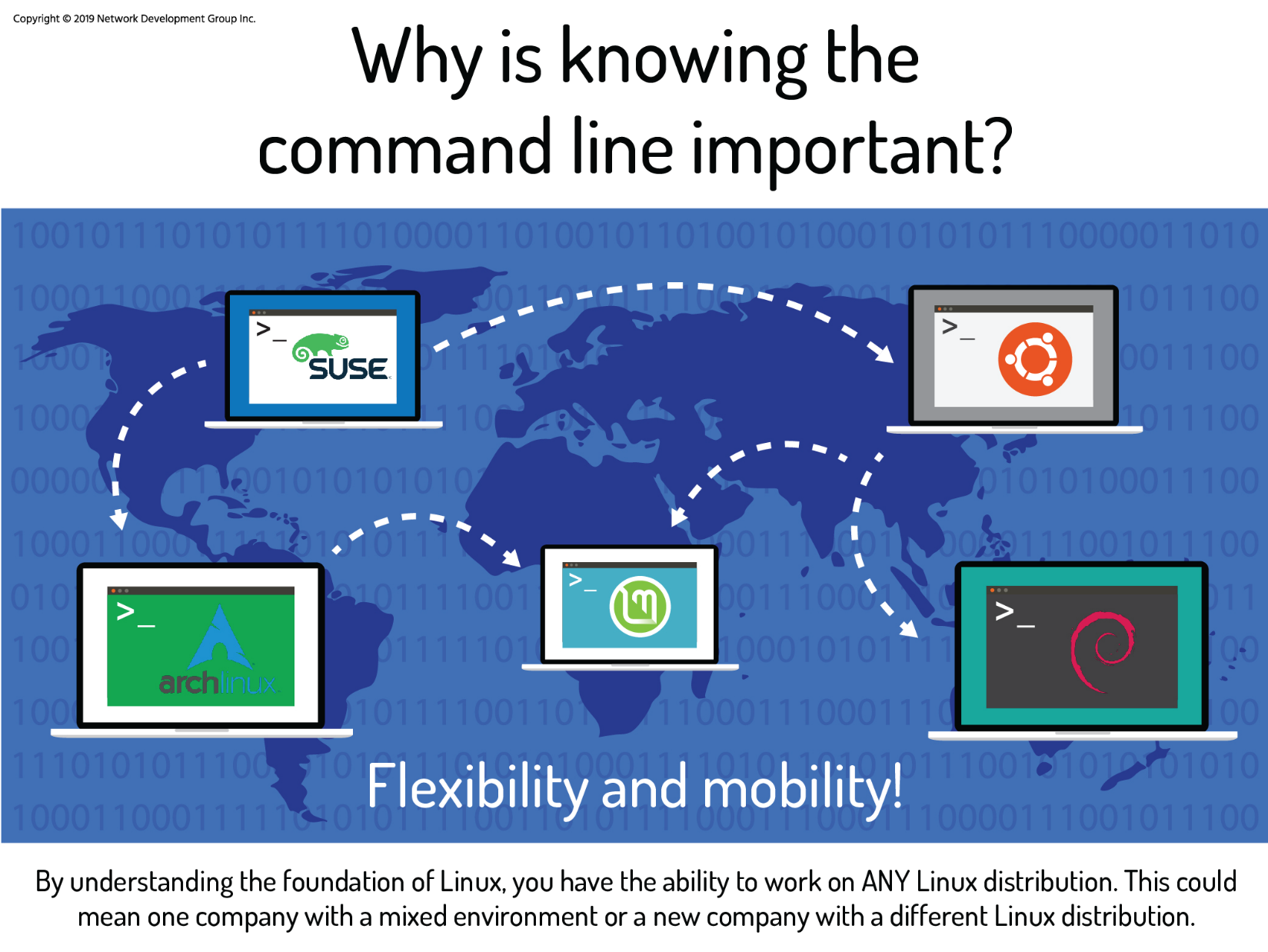


5.1 Introduction

Most consumer operating systems are designed to shield the user from the ins and outs of the CLI. The Linux community is different in that it positively celebrates the CLI for its power, speed and ability to accomplish a vast array of tasks with a single command line instruction.

When a user first encounters the CLI, they can find it challenging because it requires memorizing a dizzying amount of commands and their options. However, once a user has learned the structure of how commands are used, where the necessary files and directories are located and how to navigate the hierarchy of a file system, they can be immensely productive. This capability provides more precise control, greater speed and the ability to automate tasks more easily through scripting.

Furthermore, by learning the CLI, a user can easily be productive almost instantly on ANY flavor or distribution of Linux, reducing the amount of time needed to familiarize themselves with a system because of variations in a GUI.



5.2 Shell

Once a user has entered a command the terminal then accepts what the user has typed and passes it to a *shell*. The shell is the command line interpreter that translates commands entered by a user into actions to be performed by the operating system. If output is produced by the command, then text is displayed in the terminal. If problems with the command are encountered, an error message is displayed.

The Linux environment allows the use of many different shells, some of which have been around for many years. The most commonly-used shell for Linux distributions is called the **Bash** shell. Bash provides many advanced features, such as command history and inline editing, which allows a user to easily re-execute previously executed commands or a variation of them via simple editing.

The Bash shell also has other popular features, a few of which are listed below:

* **Scripting:** The ability to place commands in a file and then interpret (effectively use Bash to execute the contents of) the file, resulting in all of the commands being executed. This feature also has some programming features, such as conditional statements and the ability to create functions (AKA subroutines).
* **Aliases:** The ability to create short nicknames for longer commands.
* **Variables:** Used to store information for the Bash shell and for the user. These variables can be used to modify how commands and features work as well as provide vital system information.

Bash has an extensive feature list; this is only a sampling of its capabilities.

When a terminal application is run, and a shell appears, displaying an important part of the interface—the *prompt*. Not only is the prompt there to indicate that commands can be run, but it also conveys useful information to the user. The prompt is fully configurable and can be as sparse or as full-featured as is practical and useful.

The structure of the prompt may vary between distributions, but typically contains information about the user and the system. Below is a common prompt structure:

**sysadmin@localhost:~$**

The prompt shown contains the following information:

* **User Name:**

**sysadmin@localhost:~$**

* **System Name:**

**sysadmin@localhost:~$**

* **Current Directory:**

**sysadmin@localhost:~$**

The ~ symbol is used as shorthand for the user's home directory. Typically the home directory for the user is under the /home directory and named after the user account name; for example, /home/sysadmin.

5.3 Commands

What is a command? The simplest answer is that a *command* is a software program that, when executed on the CLI, performs an action on the computer.

To execute a command, the first step is to type the name of the command. Click in the terminal on the right. Type ls and hit **Enter**. The result should resemble the example below:

**sysadmin@localhost:~$** ls

Desktop Documents Downloads Music Pictures Public Templates Videos

**Note:** By itself, the ls command lists files and directories contained in the current working directory. At this point, you shouldn't worry too much about the output of the command, instead, focus on understanding how to format and execute commands.

*The ls command will be covered in complete detail later in the course.*

Many commands can be used by themselves with no further input. Some commands require additional input to run correctly. This additional input comes in two forms: *options* and *arguments*.

The typical format for a command is as follows:

command [options] [arguments]

Options are used to modify the core behavior of a command while arguments are used to provide additional information (such as a filename or a username). Each option and argument is normally separated by a space, although options can often be combined.

Keep in mind that Linux is case-sensitive. Commands, options, arguments, variables, and file names must be entered exactly as shown.

## 5.3.1 Arguments

command [options] **[arguments]**

An argument can be used to specify something for the command to act upon. If the ls command is given the name of a directory as an argument, it lists the contents of that directory. In the following example, the /etc/ppp directory is used as an argument; the resulting output is a list of files contained in that directory:

**sysadmin@localhost:~$** ls /etc/ppp

ip-down.d ip-up.d

The ls command also accepts multiple arguments. To list the contents of both the /etc/ppp and /etc/ssh directories, pass them both as arguments:

**sysadmin@localhost:~$** ls /etc/ppp /etc/ssh

/etc/ppp:

ip-down.d ip-up.d

/etc/ssh:

moduli ssh\_host\_dsa\_key.pub ssh\_host\_rsa\_key sshd\_configssh\_config

ssh\_host\_ecdsa\_key ssh\_host\_rsa\_key.pub

ssh\_host\_dsa\_key ssh\_host\_ecdsa\_key.pub ssh\_import\_id

5.3.2 Options

command **[options]** [arguments]

Options can be used with commands to expand or modify the way a command behaves. For example, using the -l option of the ls command results in a *long listing*, providing additional information about the files that are listed, such as the permissions, the size of the file and other information:

**sysadmin@localhost:~$** ls -l

total 0

drwxr-xr-x 1 sysadmin sysadmin 0 Jan 29 20:13 Desktop

drwxr-xr-x 1 sysadmin sysadmin 0 Jan 29 20:13 Documents

drwxr-xr-x 1 sysadmin sysadmin 0 Jan 29 20:13 Downloads

drwxr-xr-x 1 sysadmin sysadmin 0 Jan 29 20:13 Music

drwxr-xr-x 1 sysadmin sysadmin 0 Jan 29 20:13 Pictures

drwxr-xr-x 1 sysadmin sysadmin 0 Jan 29 20:13 Public

drwxr-xr-x 1 sysadmin sysadmin 0 Jan 29 20:13 Templates

drwxr-xr-x 1 sysadmin sysadmin 0 Jan 29 20:13 Videos

Note that, in the command above, -l is a lowercase letter "L". An easy way to remember this is -l is a mnemonic (easy to memorize programming code) for ***l****ong listing*).

Often the character is chosen to be mnemonic for its purpose, like choosing the letter *l* for *long* or *r* for *reverse*. By default, the ls command prints the results in alphabetical order, and so by adding the -r option, it prints the results in reverse alphabetical order.

**sysadmin@localhost:~$** ls -r

Videos Templates Public Pictures Music Downloads Documents Desktop

In most cases, options can be used in conjunction with other options. They can be given as separate options, as in -l -r, or combined, as in -lr. The combination of these two options would result in a long listing output in reverse alphabetical order:

**sysadmin@localhost:~$** ls -lr

total 32

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 20:13 Videos

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 20:13 Templates

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 20:13 Public

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 20:13 Pictures

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 20:13 Music

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 20:13 Downloads

drwxr-xr-x 4 sysadmin sysadmin 4096 Oct 31 20:13 Documents

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 20:13 Desktop

The order of the combined options isn't important. The output of all of these examples would be the same:

ls -l -r

ls -rl

ls -lr

By default the -l option of the ls command displays files sizes in bytes:

**sysadmin@localhost:~$** ls -l /usr/bin/perl

-rwxr-xr-x 2 root root 10376 Feb 4 2018 /usr/bin/perl

If the -h option is added the file sizes will be displayed in *human-readable* format:

**sysadmin@localhost:~$** ls -lh /usr/bin/perl

-rwxr-xr-x 2 root root 11K Feb 4 2018 /usr/bin/perl

Options are often single letters; however, sometimes they are words or phrases as well. Typically, older commands use single letters while newer commands use complete words for options. Single-letter options are preceded by a single dash - character, like the -h option. Full-word options are preceded by two dash -- characters. The -h option also has an equivalent full-word form; the --human-readable option.

**sysadmin@localhost:~$** ls -l --human-readable /usr/bin/perl

-rwxr-xr-x 2 root root 11K Feb 4 2018 /usr/bin/perl

## 5.3.3 History

When a command is executed in the terminal, it is stored in a history list. This is designed to make it easy to execute the same command, later eliminating the need to retype the entire command.

Pressing the **Up Arrow ↑** key displays the previous command on the prompt line. The entire history of commands run in the current session can be displayed by pressing **Up** repeatedly to move back through the history of commands that have been run. Pressing the Enter key runs the displayed command again.

When the desired command is located, the **Left Arrow ←** and **Right Arrow →** keys can position the cursor for editing. Other useful keys for editing include the **Home**, **End**, **Backspace** and **Delete** keys.

To view the history list of a terminal, use the history command:

**sysadmin@localhost:~$** date

Wed Dec 12 04:28:12 UTC 2018

**sysadmin@localhost:~$** ls

Desktop Documents Downloads Music Pictures Public Templates Videos

**sysadmin@localhost:~$** cal 5 2030

May 2030

Su Mo Tu We Th Fr Sa

1 2 3 4

5 6 7 8 9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

26 27 28 29 30 31

**sysadmin@localhost:~$** history

1 date

2 ls

3 cal 5 2030

4 history

If the desired command is in the list that the history command generates, it can be executed by typing an exclamation point ! character and then the number next to the command, for example, to execute the cal command again:

**sysadmin@localhost:~$** history

1 date

2 ls

3 cal 5 2030

4 history

**sysadmin@localhost:~$** !3

cal 5 2030

May 2030

Su Mo Tu We Th Fr Sa

1 2 3 4

5 6 7 8 9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

If the history command is passed a number as an argument, it outputs that number of previous commands from the history list. For example, to show the last three commands:

**sysadmin@localhost:~$** history 3

6 date

7 ls /home

8 history 3

‌⁠​​⁠​ To execute the nth command from the bottom of the history list, type !-n and hit Enter. For example, to execute the third command from the bottom of the history list execute the following:

**sysadmin@localhost:~$** !-3

date

Wed Dec 12 04:31:55 UTC 2018

To execute the most recent command type !! and hit **Enter**:

**sysadmin@localhost:~$** date

Wed Dec 12 04:32:36 UTC 2018

**sysadmin@localhost:~$** !!

date

Wed Dec 12 04:32:38 UTC 2018

To execute the most recent iteration of a specific command, type ! followed by the name of the command and hit **Enter**. For example, to execute the most recent ls command:

**sysadmin@localhost:~$** !ls

ls /home

sysadmin

## 5.4 Variables

A variable is a feature that allows the user or the shell to store data. This data can be used to provide critical system information or to change the behavior of how the Bash shell (or other commands) work. Variables are given names and stored temporarily in memory. There are two types of variables used in the Bash shell: local and environment.

## 5.4.1 Local Variables

Local or shell variables exist only in the current shell, and cannot affect other commands or applications. When the user closes a terminal window or shell, all of the variables are lost. They are often associated with user-based tasks and are lowercase by convention.

To set the value of a variable, use the following assignment expression. If the variable already exists, the value of the variable is modified. If the variable name does not already exist, the shell creates a new local variable and sets the value:

variable=value

The following example creates a local variable named variable1 and assigns it a value of Something:

**sysadmin@localhost:~$** variable1='Something'

The echo command is used to display output in the terminal. To display the value of the variable, use a dollar sign $ character followed by the variable name as an argument to the echo command:

**sysadmin@localhost:~$** echo $variable1

Something

5.4.2 Environment Variables

*Environment variables*, also called *global variables*, are available system-wide, in all shells used by Bash when interpreting commands and performing tasks. The system automatically recreates environment variables when a new shell is opened. Examples include the PATH, HOME, and HISTSIZE variables. The HISTSIZE variable defines how many previous commands to store in the history list. The command in the example below displays the value of the HISTSIZE variable:

**sysadmin@localhost:~$** echo $HISTSIZE

1000

To modify the value of an existing variable, use the assignment expression:

**sysadmin@localhost:~$** HISTSIZE=500

**sysadmin@localhost:~$** echo $HISTSIZE

500

Many variables are available for the Bash shell, as well as variables that affect different Linux commands. A discussion of all variables is beyond the scope of this chapter; however, more shell variables will be covered as this course progresses.

When run without arguments, the env command outputs a list of the environment variables. Because the output of the env command can be quite long, the following examples use a text search to filter that output.

In a previous example variable1 was created as a local variable, so the following search in the environment variables results in no output:

**sysadmin@localhost:~$** env | grep variable1

The pipe | character passes the output of the env command to the grep command, which searches the output.

*This text filtering technique will be covered in detail later in the course.*

The export command is used to turn a local variable into an environment variable.

export *variable*

After exporting variable1, it is now an environment variable. It is now found in the search through the environment variables:

**sysadmin@localhost:~$** export variable1

**sysadmin@localhost:~$** env | grep variable1

variable1=Something

The export command can also be used to make a variable an environment variable upon its creation by using the assignment expression as the argument:

**sysadmin@localhost:~$** export variable2='Else'

**sysadmin@localhost:~$** env | grep variable2

variable2=Else

To change the value of an environment variable, use the assignment expression:

**sysadmin@localhost:~$** variable1=$variable1' '$variable2

**sysadmin@localhost:~$** echo $variable1

Something Else

Exported variables can be removed using the unset command:

**sysadmin@localhost:~$** unset variable2

5.4.3 Path Variable

One of the most important Bash shell variables to understand is the PATH variable. It contains a list that defines which directories the shell looks in to find commands. If a valid command is entered and the shell returns a "command not found" error, it is because the Bash shell was unable to locate a command by that name in any of the directories included in the path. The following command displays the path of the current shell:

**sysadmin@localhost:~$** echo $PATH

/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

**sysadmin@localhost:~$**

Each directory in the list is separated by a colon : character. Based on the preceding output, the path contains the following directories. The shell will check the directories in the order they are listed:

/home/sysadmin/bin

/usr/local/sbin

/usr/local/bin

/usr/sbin

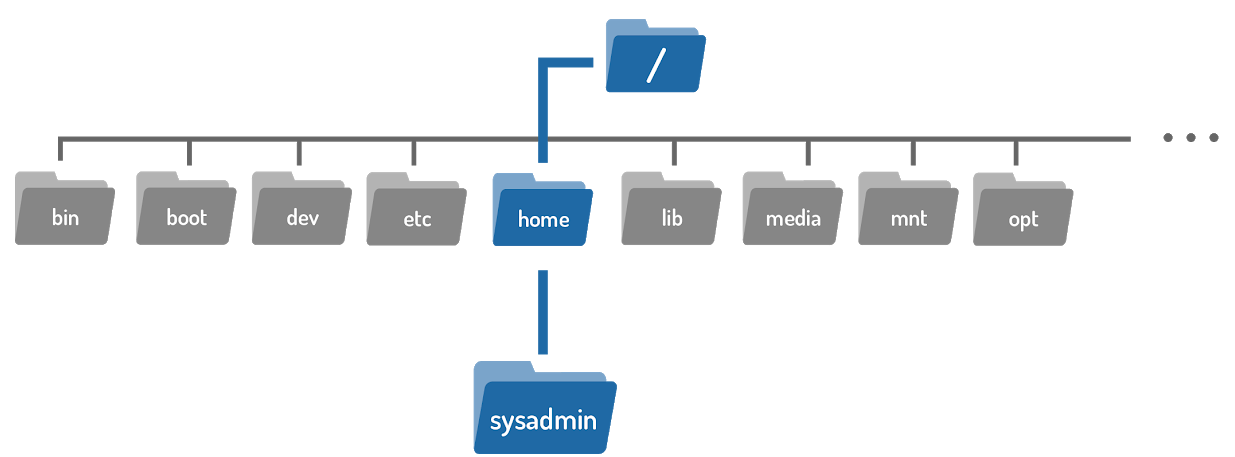
/usr/bin

/sbin

/bin

/usr/games

Each of these directories is represented by a *path*. A path is a list of directories separated by the / character. If you think of the filesystem as a map, paths are the directory addresses, which include step-by-step navigation directions; they can be used to indicate the location of any file within the filesystem. For example, /home/sysadmin is a path to the home directory:



*Directories and paths will be covered in detail later in the course.*

If the command is not found in any directory listed in the PATH variable, then the shell returns an error:

**sysadmin@localhost:~$** zed

-bash: zed: command not found

**sysadmin@localhost:~$**

‌⁠​​⁠​ If custom software is installed on the system it may be necessary to modify the PATH to make it easier to execute these commands. For example, the following will add and verify the /usr/bin/custom directory to the PATH variable:

**sysadmin@localhost:~$** PATH=/usr/bin/custom:$PATH

**sysadmin@localhost:~$** echo $PATH

/usr/bin/custom:/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

When updating the PATH variable, always include the current path, so as not to lose access to commands located in those directories. This can be accomplished by appending $PATH to the value in the assignment expression. Recall that a variable name preceded by a dollar sign represents the value of the variable.

## 5.5 Command Types

One way to learn more about a command is to look at where it comes from. The type command can be used to determine information about command type.

type command

There are several different sources of commands within the shell of your CLI including internal commands, external commands, aliases, and functions.

## 5.5.1 Internal Commands

Also called built-in commands, internal commands are built into the shell itself. A good example is the cd (change directory) command as it is part of the Bash shell. When a user types the cd command, the Bash shell is already executing and knows how to interpret it, requiring no additional programs to be started.

The type command identifies the cd command as an internal command:

**sysadmin@localhost:~$** type cd

cd is a shell builtin

## 5.5.2 External Commands

External commands are binary executables stored in directories that are searched by the shell. If a user types the ls command, then the shell searches through the directories that are listed in the PATH variable to try to find a file named ls that it can execute.

If a command does not behave as expected or if a command is not accessible that should be, it can be beneficial to know where the shell is finding the command or which version it is using. It would be tedious to have to manually look in each directory that is listed in the PATH variable. Instead, use the which command to display the full path to the command in question:

which command

The which command searches for the location of a command by searching the PATH variable.

**sysadmin@localhost:~$** which ls

/bin/ls

**sysadmin@localhost:~$** which cal

/usr/bin/cal

External commands can also be executed by typing the complete path to the command. For example, to execute the ls command:

**sysadmin@localhost:~$** /bin/ls

Desktop Documents Downloads Music Pictures Public Templates Videos

For external commands, the type command displays the location of the command:

**sysadmin@localhost:~$** type cal

cal is /usr/bin/cal

In some cases the output of the type command may differ significantly from the output of the which command:

**sysadmin@localhost:~$** type echo

echo is a shell builtin

**sysadmin@localhost:~$** which echo

/bin/echo

Using the -a option of the type command displays all locations that contain the command named:

**sysadmin@localhost:~$** type -a echo

echo is a shell builtin

echo is /bin/echo

## 5.5.3 Aliases

An alias can be used to map longer commands to shorter key sequences. When the shell sees an alias being executed, it substitutes the longer sequence before proceeding to interpret commands.

For example, the command ls -l is commonly aliased to l or ll. Because these smaller commands are easier to type, it becomes faster to run the ls -l command line.

To determine what aliases are set on the current shell use the alias command:

**sysadmin@localhost:~$** alias

alias egrep='egrep --color=auto'

alias fgrep='fgrep --color=auto'

alias grep='grep --color=auto'

alias l='ls -CF'

alias la='ls -A'

alias ll='ls -alF'

alias ls='ls --color=auto'

The aliases from the previous examples were created by initialization files. These files are designed to make the process of creating aliases automatic.

New aliases can be created using the following format, where name is the name to be given the alias and command is the command to be executed when the alias is run.

alias name=command

For example, the cal 2019 command displays the calendar for the year 2019. Suppose you end up running this command often. Instead of executing the full command each time, you can create an alias called mycal and run the alias, as demonstrated in the following graphic:

**sysadmin@localhost:~$** alias mycal="cal 2019"

**sysadmin@localhost:~$** mycal

2019

January February March

Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa

1 2 3 4 5 1 2 1 2

6 7 8 9 10 11 12 3 4 5 6 7 8 9 3 4 5 6 7 8 9

13 14 15 16 17 18 19 10 11 12 13 14 15 16 10 11 12 13 14 15 16

20 21 22 23 24 25 26 17 18 19 20 21 22 23 17 18 19 20 21 22 23

27 28 29 30 31 24 25 26 27 28 24 25 26 27 28 29 30

31

Aliases created this way only persist while the shell is open. Once the shell is closed, the new aliases are lost. Additionally, each shell has its own aliases, so aliases created in one shell won’t be available in a new shell that’s opened.

The type command can identify aliases to other commands:

**sysadmin@localhost:~$** type ll

ll is aliased to `ls -alF'

**sysadmin@localhost:~$** type -a ls

ls is aliased to `ls --color=auto'

ls is /bin/ls

The output of these commands indicates that ll is an alias for ls -alF, and even ls is an alias for ls --color=auto.

## 5.5.4 Functions

Functions can also be built using existing commands to either create new commands, or to override commands built-in to the shell or commands stored in files. Aliases and functions are normally loaded from the initialization files when the shell first starts.

Functions are more advanced than aliases and typically are used in Bash shell scripts. Typically, functions are used to execute multiple commands. To create a function, the following syntax is used:

function\_name ()

{

commands

}

In the format above, function\_name can be anything that the administrator wants to call the function. The commands that the administrator wants to execute can replace the commands placeholder. Note the formatting, in particular, the location of the parenthesis () and braces {}, as well as the convention of using tabs to make the function more easily readable.

Functions are useful as they allow for a set of commands to be executed one at a time instead of typing each command repeatedly. In the example below, a function called my\_report is created to execute the ls, date, and echo commands.

**sysadmin@localhost:~$** my\_report () {

> ls Documents

> date

> echo "Document directory report"

> }

When creating a function, a > character will appear as a prompt to enter the commands for the function. The curly braces {} are used to let the shell know when a function begins and ends so as to exit the > prompt.

Once a function is created, the function name may be invoked from the BASH prompt to execute the function:

**sysadmin@localhost:~$** my\_report

School alpha-third.txt hidden.txt numbers.txt spelling.txt

Work alpha.txt letters.txt os.csv words

adjectives.txt animals.txt linux.txt people.csv

alpha-first.txt food.txt longfile.txt profile.txt

alpha-second.txt hello.sh newhome.txt red.txt

Wed Oct 13 06:54:04 UTC 2021

Document directory report

**sysadmin@localhost:~$**

## 5.6 Quoting

Quotation marks are used throughout Linux administration and most computer programming languages to let the system know that the information contained within the quotation marks should either be ignored or treated in a way that is very different than it would normally be treated. There are three types of quotes that have special significance to the Bash shell: double quotes ", single quotes ', and back quotes `. Each set of quotes alerts the shell not to treat the text within the quotes in the normal way.

## 5.6.1 Double Quotes

Double quotes stop the shell from interpreting some metacharacters (special characters), including glob characters.

Glob characters, also called wild cards, are symbols that have special meaning to the shell; they are interpreted by the shell itself before it attempts to run any command. Glob characters include the asterisk \* character, the question ? mark character, and the brackets [ ], among others.

Globbing will be covered in greater detail later in the course.

Within double quotes an asterisk is just an asterisk, a question mark is just a question mark, and so on, which is useful when you want to display something on the screen that is normally a special character to the shell. In the echo command below, the Bash shell doesn't convert the glob pattern into filenames that match the pattern:

**sysadmin@localhost:~$** echo "The glob characters are \*, ? and [ ]"

The glob characters are \*, ? and [ ]

Double quotes still allow for command substitution, variable substitution, and permit some other shell metacharacters that haven't been discussed yet. The following demonstration shows that the value of the PATH variable is still displayed:

**sysadmin@localhost:~$** echo "The path is $PATH"

The path is /usr/bin/custom:/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

## 5.6.2 Single Quotes

Single quotes prevent the shell from doing any interpreting of special characters, including globs, variables, command substitution and other metacharacters that have not been discussed yet.

For example, to make the $ character simply mean a $, rather than it acting as an indicator to the shell to look for the value of a variable, execute the second command displayed below:

**sysadmin@localhost:~$** echo The car costs $100

The car costs 00

**sysadmin@localhost:~$** echo 'The car costs $100'

The car costs $100

## 5.6.3 Backslash Character

There is also an alternative technique to essentially single quote a single character. Consider the following message:

The service costs $1 and the path is $PATH

If this sentence is placed in double quotes, $1 and $PATH are considered variables.

**sysadmin@localhost:~$** echo "The service costs $1 and the path is $PATH"

‌⁠​​⁠​The service costs and the path is /usr/bin/custom:/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

If it is placed in single quotes, $1 and $PATH are not considered variables.

**sysadmin@localhost:~$** echo 'The service costs $1 and the path is $PATH'

The service costs $1 and the path is $PATH

But what if you want to have $PATH treated as a variable and $1 not?

In this case, use a backslash \ character in front of the dollar sign $ character to prevent the shell from interpreting it. The command below demonstrates using the \ character:

**sysadmin@localhost:~$** echo The service costs \$1 and the path is $PATH

The service costs $1 and the path is /usr/bin/custom:/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

## 5.6.4 Backquotes

Backquotes, or backticks, are used to specify a command within a command, a process called command substitution. This allows for powerful and sophisticated use of commands.

While it may sound confusing, an example should make things more clear. To begin, note the output of the date command:

**sysadmin@localhost:~$** date

Mon Nov 4 03:35:50 UTC 2018

Now, note the output of the echo command:

**sysadmin@localhost:~$** echo Today is date

Today is date

In the previous command, the word date is treated as regular text, and the shell passes date to the echo command. To execute the date command and have the output of that command sent to the echo command, put the date command in between two backquote characters:

**sysadmin@localhost:~$** echo Today is `date`

Today is Mon Nov 4 03:40:04 UTC 2018

## 5.7 Control Statements

Control statements allow you to use multiple commands at once or run additional commands, depending on the success of a previous command. Typically these control statements are used within scripts, but they can also be used on the command line as well.

## 5.7.1 Semicolon

command1; command2; command3

The semicolon ; character can be used to run multiple commands, one after the other. Each command runs independently and consecutively; regardless of the result of the first command, the second command runs once the first has completed, then the third and so on.

For example, to print the months of January, February and March of 2030, execute the following command:

**sysadmin@localhost:~$** cal 1 2030; cal 2 2030; cal 3 2030

January 2030

Su Mo Tu We Th Fr Sa

1 2 3 4 5

6 7 8 9 10 11 12

13 14 15 16 17 18 19

20 21 22 23 24 25 26

27 28 29 30 31

February 2030

Su Mo Tu We Th Fr Sa

1 2

3 4 5 6 7 8 9

10 11 12 13 14 15 16

17 18 19 20 21 22 23

24 25 26 27 28

March 2030

Su Mo Tu We Th Fr Sa

1 2

3 4 5 6 7 8 9

10 11 12 13 14 15 16

17 18 19 20 21 22 23

24 25 26 27 28 29 30

31

## 5.7.2 Double Ampersand

command1 && command2

The double ampersand && acts as a logical "and"; if the first command is successful, then the second command will also run. If the first command fails, then the second command will not run.

To better understand how this works, consider first the concept of failure and success for commands. Commands succeed when they work properly and fail when something goes wrong. For example, consider the ls command. The command succeeds if the given directory is accessible and fails if it isn't.

In the following example, the first command succeeds because the /etc/ppp directory exists and is accessible while the second command fails because there is no /junk directory:

**sysadmin@localhost:~$** ls /etc/ppp

ip-down.d ip-up.d

**sysadmin@localhost:~$** ls /etc/junk

ls: cannot access /etc/junk: No such file or directory

To use the success or failure of the ls command in conjunction with && execute commands like the following. In the first example, the echo command is executed because the ls command succeeds:

**sysadmin@localhost:~$** ls /etc/ppp && echo success

ip-down.d ip-up.d

success

In the second example, the echo command isn't executed because the ls command fails:

**sysadmin@localhost:~$** ls /etc/junk && echo success

ls: cannot access /etc/junk: No such file or directory

## 5.7.3 Double Pipe

command1 || command2

The double pipe || is a logical "or". Depending on the result of the first command, the second command will either run or be skipped.

With the double pipe, if the first command runs successfully, the second command is skipped; if the first command fails, then the second command is run. In other words, you are essentially telling the shell, "Either run this first command or the second one”.

In the following example, the echo command only executes if the ls command fails:

**sysadmin@localhost:~$** ls /etc/ppp || echo failed

ip-down.d ip-up.d

**sysadmin@localhost:~$** ls /etc/junk || echo failed

ls: cannot access /etc/junk: No such file or directory

failed

5.1 Introduction

This is Lab 5: Command Line Skills. By performing this lab, students will learn how to use basic features of the shell.

In this lab, you will perform the following tasks:

* Explore Bash features
* Use shell variables
* Be able to make use of quoting

## 5.2 Files and Directories

In this task, we will access the Command Line Interface (CLI) for Linux to explore how to execute basic commands and what affects how they can be executed.

Most users are probably more familiar with how commands are executed using a Graphical User Interface (GUI). Therefore, this task will likely present some new concepts to you if you have not previously worked with a CLI. To use a CLI, you will need to type the command that you want to run.

The window where you will type your command is known as a terminal emulator application. Inside of the Terminal window the system is displaying a prompt, which currently contains a prompt followed by a blinking cursor:

**sysadmin@localhost:~$**

**Remember**

You may need to press **Enter** in the window to display the prompt.

The prompt tells you that you are user sysadmin; the host or computer you are using: localhost; and the directory where you are at: ~, which represents your home directory.

When you type a command, it will appear at the text cursor. You can use keys such as the **home**, **end**, **backspace**, and **arrow keys** for editing the command you are typing.

Equally important is the command line syntax, which is the order in which the command, the option(s), and argument(s) must be entered into the prompt so the shell recognizes how to properly execute the command. Proper command line syntax looks like the following:

command [options] [arguments]

Type the ls command at the prompt, which will list the files and directories contained in your current working directory. In this example, no options or arguments will be used.

Once you have typed the command correctly, press **Enter** to execute it.

**sysadmin@localhost:~$** ls

**Desktop Documents Downloads Music Pictures Public Templates Videos**

## 5.2.1 Step 1

The ls command is used to list information about directories and files and by default it displays information for the current directory. Use the -l option to display this information in the long format, which gives additional information about files located in the current working directory:

ls -l

Your output should be similar to the following:

**sysadmin@localhost:~$** ls -l

total 32

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 19:52 **Desktop**

drwxr-xr-x 4 sysadmin sysadmin 4096 Oct 31 19:52 **Documents**

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 19:52 **Downloads**

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 19:52 **Music**

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 19:52 **Pictures**

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 19:52 **Public**

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 19:52 **Templates**

drwxr-xr-x 2 sysadmin sysadmin 4096 Oct 31 19:52 **Videos**

Note that directories are considered a type of file in the Linux file system.

## 5.2.2 Step 2

Arguments can be added to commands as well. Adding the location of a specific directory to the ls command will list information for that directory. Use the argument /tmp to display detailed information about files in the /tmp directory.

ls -l /tmp

Your output should be similar to the following:

**sysadmin@localhost:~$** ls -l /tmp

total 0

-rw-rw-r-- 1 root root 1863 Oct 31 19:47 inside\_setup.sh

By using the option -l and the argument /tmp, we can now see that the /tmp directory has a file called inside\_setup.sh located inside it.

## 5.2.3 Step 3

The following command will display the same information that you see in the first part of the prompt. Make sure that you have selected (clicked on) the **Terminal** window first and then type the following command followed by the **Enter** key:

whoami

Your output should be similar to the following:

**sysadmin@localhost:~$** whoami

sysadmin

**sysadmin@localhost:~$**

The output of the whoami command, sysadmin, displays the user name of the current user. Although in this case your username is displayed in the prompt, this command could be used to obtain this information in a situation when the prompt did not contain this information.

## 5.2.4 Step 4

The next command displays information about the current system. To be able to see the name of the kernel you are using, type the following command into the terminal:

uname

Your output should be similar to the following:

**sysadmin@localhost:~$** uname

Linux

Many commands that are executed produce text output like this. You can change what output is produced by a command by using options after the name of the command.

Options for a command can be specified in several ways. Traditionally in UNIX, options were expressed by a hyphen followed by another character; for example: -n.

In Linux, options can sometimes also be given by two hyphen characters followed by a word, or hyphenated word; for example: --nodename.

Execute the uname command again twice in the terminal, once with the option -n and again with the option --nodename. This will display the network node hostname, also found in the prompt.

uname -n

uname --nodename

Your output should be similar to the following:

**sysadmin@localhost:~$** uname -n

localhost

**sysadmin@localhost:~$** uname --nodename

localhost

## 5.2.5 Step 5

The pwd command is used to display your current "location" or current "working" directory. Type the following command to display the working directory:

pwd

Your output should be similar to the following:

**sysadmin@localhost:~$** pwd

/home/sysadmin

**sysadmin@localhost:~$**

The current directory in the example above is /home/sysadmin. This is also referred to as your home directory, a special place where you have control of files and other users normally have no access. By default, this directory is named the same as your username and is located underneath the /home directory.

As you can see from the output of the command, /home/sysadmin, Linux uses the forward slash / to separate directories to make what is called a path. The initial forward slash represents the top-level directory, known as the root directory. More information regarding files, directories and paths will be presented in later labs.

The tilde ~ character that you see in your prompt is also indicating what the current directory is. This character is a "shortcut" way to represent your home.

**Consider This**

pwd stands for "print working directory". While it doesn't actually "print" in modern versions, older UNIX machines didn't have monitors so the output of commands went to a printer, hence the somewhat misleading name of pwd.

## 5.3 Command History

The Bash shell maintains a history of the commands that you type. Previous commands can be easily accessed in this history in several ways.

The first and easiest way to recall a previous command is to use the **up arrow key**. Each press of the **up arrow key** goes backwards one command through your command history. If you accidentally go back too far, then the **down arrow key** will go forwards through the history of commands.

When you find the command that you want to execute, you can use the **left arrow keys** and **right arrow keys** to position the cursor for editing. Other useful keys for editing include the **Home**, **End**, **Backspace** and **Delete** keys.

Another way to use your command history is to execute the history command to be able to view a numbered history list. The number listed to the left of the command can be used to execute the command again. The history command also has a number of options and arguments which can manipulate which commands will be stored or displayed.

5.3.1 Step 1

Execute a new command and then execute the history command:

echo Hi

history

**Remember**

The date command will print the time and date on the system. The clear command clears the screen.

Your output should be similar to the following:

**sysadmin@localhost:~$** history

1 ls

2 ls -l

3 ls -l /tmp

4 whoami

5 uname

6 uname -n

7 uname --nodename

8 pwd

9 echo Hi

10 history

**sysadmin@localhost:~$**

Your command numbers may differ from those provided above. This is because you may have executed a different number of commands since opening the virtual terminal.

## 5.3.2 Step 2

To view a limited number of commands, the history command can take a number as a parameter to display exactly that many recent entries. Type the following command to display the last five commands from your history:

history 5

Your output should be similar to the following:

**sysadmin@localhost:~$** history 5

7 uname --nodename

8 pwd

9 echo Hi

10 history

11 history 5

## 5.3.3 Step 3

To execute a command again, type the exclamation point and the history list number. For example, to execute the 9th command in your history list, you would execute the following:

!9

**sysadmin@localhost:~$** !9

echo Hi

Hi

## 5.3.4 Step 4

Next, experiment with accessing your history using the **up arrow keys** and **down arrow keys**. Keep pressing the **up arrow key** until you find a command you want to execute. If necessary, use other keys to edit the command and then press **Enter** to execute the command.

## 5.4 Shell Variables

Shell variables are used to store data in Linux. This data is used by the shell itself as well as by programs and users.

The focus of this section is to learn how to display the values of shell variables.

## 5.4.1 Step 1

The echo command can be used to print text and the value of a variable, and to show how the shell environment expands metacharacters (more on metacharacters later in this lab). Type the following command to have it output literal text:

echo Hello Student

Your output should be similar to the following:

**sysadmin@localhost:~$** echo Hello Student

Hello Student

**sysadmin@localhost:~$**

## 5.4.2 Step 2

Environment variables are available system-wide. The system automatically recreates environment variables when a new shell is opened. Examples include the PATH, HOME, and HISTSIZE variables. The HISTSIZE variable defines how many previous commands to store in the history list. In the example below, the command will display the value of the HISTSIZE variable:

**sysadmin@localhost:~$** echo $HISTSIZE‌

1000

**sysadmin@localhost:~$**

## 5.4.3 Step 3

Type the following command to display the value of the PATH variable:

echo $PATH

Your output should be similar to the following:

**sysadmin@localhost:~$** echo $PATH

/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

**sysadmin@localhost:~$**

The PATH variable is displayed by placing a $ character in front of the name of the variable.

This variable is used to find the location of commands. Each of the directories listed above are searched when you run a command. For example, if you try to run the date command, the shell will first look for the command in the /home/sysadmin/bin directory and then in the /usr/local/sbin directory and so on. Once the date command is found, the shell "runs it”.

## 5.4.4 Step 4

Use the which command to determine if there is an executable file, in this case named date, that is located within a directory listed in the PATH value:

which date

Your output should be similar to the following:

**sysadmin@localhost:~$** which date

/bin/date

**sysadmin@localhost:~$**

The output of the which command tells you that when you execute the date command, the system will run the command /bin/date. The which command makes use of the PATH variable to determine the location of the date command.

## 5.5 Command Types

In this section we will learn about the four types of commands used in Linux. Understanding where these commands come from and how they differ allows an administrator to manage the system more effectively.

## 5.5.1 Step 1

One way to learn more about a command is to look at where it comes from. The type command can be used to determine information about command type.

type command

There are several different sources of commands within the shell of your CLI:

Internal commands are built into the shell itself. A good example is the cd (change directory) command as it is part of the Bash shell. When a user types the cd command, the Bash shell is already executing and knows how to interpret it, requiring no additional programs to be started.

The type command identifies the cd command as an internal command:

**sysadmin@localhost:~$** type cd

cd is a shell builtin

## 5.5.2 Step 2

External commands are binary executables stored in directories that are searched by the shell. If a user types the ls command, the shell searches through the directories that are listed in the PATH variable to try to find a file named ls that it can execute. Use the which command to display the full path to the ls command.

which ls

**sysadmin@localhost:~$** which ls

/bin/ls

For external commands, the type command displays the location of the command:

**sysadmin@localhost:~$** type cp

cp is /bin/cp

In some cases the output of the type command may differ significantly from the output of the which command:

**sysadmin@localhost:~$** which cp

/bin/cp

Using the -a option of the type command displays all locations that contain the command:

**sysadmin@localhost:~$** type -a ls

ls is aliased to `ls --color=auto'

ls is /bin/ls

## 5.5.3 Step 3

Aliases can be used to map longer commands to shorter key sequences. When the shell sees an alias being executed, it substitutes the longer sequence before proceeding to interpret commands.

To determine what aliases are set on the current shell, use the alias command:

**sysadmin@localhost:~$** alias

alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo

error)" "$(history|tail -n1|sed -e '\''s/^\s\*[0-9]\+\s\*//;s/[;&|]\s\*alert$//'\'')"'

alias egrep='egrep --color=auto'

alias fgrep='fgrep --color=auto'

alias grep='grep --color=auto'

alias l='ls -CF'

alias la='ls -A'

alias ll='ls -alF'

alias ls='ls --color=auto'

## 5.5.4 Step 4

The final command type is the executable program. These commands invoke programs installed on the system which perform specific tasks. When a user types the vi command, the shell uses the PATH file to locate and execute the program. Programs like vi are available on just about every Linux distribution; other programs, like vlc (an open source media player often used on Linux desktops), are installed by users or administrators for a specific purpose and will not be listed in the PATH unless they have been installed separately.

type vi

cd /bin

type vlc

cd

**sysadmin@localhost:~$** type vi

vi is /usr/bin/vi

**sysadmin@localhost:~$** cd /bin

**sysadmin@localhost:/bin$** type vlc

-bash: type: vlc: not found

**sysadmin@localhost:/bin$** cd

**sysadmin@localhost:~$**

## 5.6 Quoting

There are three types of quotes used by the Bash shell: single quotes ('), double quotes (") and back quotes (`). These quotes have special features in the Bash shell as described below.

To understand single and double quotes, consider that there are times that you don't want the shell to treat some characters as special. For example, the \* character is used as a wildcard. What if you wanted the \* character to just mean a literal asterisk?

* Single ' quotes prevent the shell from "interpreting" or expanding all special characters. Often single quotes are used to protect a string (a sequence of characters) from being changed by the shell, so that the string can be interpreted by a command as a parameter to affect the way the command is executed.
* Double " quotes stop the expansion of glob characters like the asterisk (\*), question mark (?), and square brackets ( [] ). Double quotes do allow for both variable expansion and command substitution (see back quotes) to take place.
* Back ` quotes cause command substitution which allows for a command to be executed within the line of another command.

When using quotes, they must be entered in pairs or else the shell will not consider the command complete.

While single quotes are useful for blocking the shell from interpreting one or more characters, the shell also provides a way to block the interpretation of just a single character called "escaping" the character. To escape the special meaning of a shell metacharacter, the backslash \ character is used as a prefix to that one character.

## 5.6.1 Step 1

Execute the following command to use back quotes ` (found under the ~ character on some keyboards) to execute the date command within the line of the echo command:

echo Today is `date`

Your output should be similar to the following:

**sysadmin@localhost:~$** echo Today is `date`

Today is Mon Dec 3 21:29:45 UTC 2018

## 5.6.2 Step 2

You can also place $( before the command and ) after the command to accomplish command substitution:

echo Today is $(date)

Your output should be similar to the following:

**sysadmin@localhost:~$** echo Today is $(date)

Today is Mon Dec 3 21:33:41 UTC 2018

Why two different methods that accomplish the same thing? Backquotes look very similar to single quotes, making it harder to "see" what a command is supposed to do. Originally, shells used backquotes; the $(command) format was added in a later version of the Bash shell to make the statement more visually clear.

## 5.6.3 Step 3

If you don't want the backquotes to be used to execute a command, place single quotes around them. Execute the following:

echo This is the command '`date`'

Your output should be similar to the following:

**sysadmin@localhost:~$** echo This is the command '`date`'

This is the command `date`

**sysadmin@localhost:~$**

## 5.6.4 Step 4

Note that you could also place a backslash character in front of each backquote character. Execute the following:

echo This is the command \`date\`

Your output should be similar to the following:

**sysadmin@localhost:~$** echo This is the command \`date\`

This is the command `date`

**sysadmin@localhost:~$**

## 5.6.5 Step 5

Double quote characters don't have any effect on backquote characters. The shell will still use them as command substitution. Execute the following to see a demonstration:

echo This is the command "`date`"

Your output should be similar to the following:

**sysadmin@localhost:~$** echo This is the command "`date`"

This is the command Mon Dec 3 21:37:33 UTC 2018

## 5.6.6 Step 6

Double quote characters will have an effect on wildcard characters, disabling their special meaning. Execute the following:

echo D\*

echo "D\*"

Your output should be similar to the following:

**sysadmin@localhost:~$** echo D\*

Desktop Documents Downloads

**sysadmin@localhost:~$** echo "D\*"

D\*

**sysadmin@localhost:~$**

**Important**

Quoting may seem trivial and weird at the moment, but as you gain more experience working in the command shell, you will discover that having a good understanding of how different quotes work is critical to using the shell.

## 5.7 Control Statements

Typically, you type a single command and you execute it when you press **Enter**. The Bash shell offers three different statements that can be used to separate multiple commands typed together.

The simplest separator is the semicolon (;). Using the semicolon between multiple commands allows for them to be executed one right after another, sequentially from left to right.

The && characters create a logical "and" statement. Commands separated by && are conditionally executed. If the command on the left of the && is successful, then the command to the right of the && will also be executed. If the command to the left of the && fails, then the command to the right of the && is not executed.

The || characters create a logical "or" statement, which also causes conditional execution. When commands are separated by ||, then only if the command to the left fails, does the command to the right of the || execute. If the command to the left of the || succeeds, then the command to the right of the || will not execute.

To see how these control statements work, you will be using two special executables: true and false. The true executable always succeeds when it executes, whereas, the false executable always fails. While this may not provide you with realistic examples of how && and || work, it does provide a means to demonstrate how they work without having to introduce new commands.

## 5.7.1 Step 1

Execute the following three commands together separated by semicolons:

echo Hello; echo Linux; echo Student

As you can see the output shows all three commands executed sequentially:

**sysadmin@localhost:~$** echo Hello; echo Linux; echo Student

Hello

Linux

Student

**sysadmin@localhost:~$**

## 5.7.2 Step 2

Now, put three commands together separated by semicolons, where the first command executes with a failure result:

false; echo Not; echo Conditional

Your output should be similar to the following:

**sysadmin@localhost:~$** false; echo Not; echo Conditional

Not

Conditional

**sysadmin@localhost:~$**

Note that in the previous example, all three commands still executed even though the first one failed. While you can't see from the output of the false command, it did execute. However, when commands are separated by the ; character, they are completely independent of each other.

## 5.7.3 Step 3

Next, use logical "and" to separate the commands:

echo Start && echo Going && echo Gone

Your output should be similar to the following:

**sysadmin@localhost:~$** echo Start && echo Going && echo Gone

Start

Going

Gone

**sysadmin@localhost:~$**

Because each echo statement executes correctly, a return value of success is provided, allowing the next statement to also be executed.

## 5.7.4 Step 4

Use logical "and" with a command that fails as shown below:

echo Success && false && echo Bye

Your output should be similar to the following:

**sysadmin@localhost:~$** echo Success && false && echo Bye

Success

**sysadmin@localhost:~$**

The first echo command succeeds and we see its output. The false command executes with a failure result, so the last echo statement is not executed.

## 5.7.5 Step 5

The "or" characters separating the following commands demonstrates how the failure before the "or" statement causes the command after it to execute; however, a successful first statement causes the command to not execute:

false || echo Fail Or

true || echo Nothing to see here

Your output should be similar to the following:

**sysadmin@localhost:~$** false || echo Fail Or

Fail Or

**sysadmin@localhost:~$** true || echo Nothing to see here

**sysadmin@localhost:~$**