Linux
Professional
Institute

Linux Essentials

Version 1.6
English

010

Table of Contents

Linux Professional Institute

Topic 1: The Linux Community and a Career in Open Source

1.1 Linux Evolution and Popular Operating Systems

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 1.1

Weight

2

Key knowledge areas

- Distributions
- Embedded Systems
- Linux in the Cloud

Partial list of the used files, terms and utilities

- Debian, Ubuntu (LTS)
- CentOS, openSUSE, Red Hat, SUSE
- Linux Mint, Scientific Linux
- Raspberry Pi, Raspbian
- Android

1.1 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	1 The Linux Community and a Career in Open Source
Objective:	1.1 Linux Evolution and Popular Operating Systems
Lesson:	1 of 1

Introduction

Linux is one of the most popular operating systems. Its development was started in 1991 by Linus Torvalds. The operating system was inspired by Unix, another operating system developed in the 1970s by AT&T Laboratories. Unix was geared towards small computers. At the time, “small” computers were considered machines that don’t need an entire hall with air conditioning and cost less than one million dollars. Later, they were considered as the machines that can be lifted by two people. At that time, an affordable Unix system was not readily available on computers such as office computers, which were tended to be based on the x86 platform. Therefore Linus, who was a student by that time, started to implement a Unix-like operating system which was supposed to run on this platform.

Mostly, Linux uses the same principles and basic ideas of Unix, but Linux itself doesn’t contain Unix code, as it is an independent project. Linux is not backed by an individual company but by an international community of programmers. As it is freely available, it can be used by anyone without restrictions.

Distributions

A Linux *distribution* is a bundle that consists of a Linux *kernel* and a selection of applications that are maintained by a company or user community. A distribution's goal is to optimize the kernel and the applications that run on the operating system for a certain use case or user group. Distributions often include distribution-specific tools for software installation and system administration. This is why some distributions are mainly used for desktop environments where they need to be easy to use while others are mainly used to run on servers to use the available resources as efficiently as possible.

Another way to classify distributions is by referring to the *distribution family* they belong to. Distributions of the Debian distribution family use the package manager `dpkg` to manage the software that is run on the operating system. Packages that can be installed with the package manager are maintained by voluntary members of the distribution's community. Maintainers use the `deb` package format to specify how the software is installed on the operating system and how it is configured by default. Just like a distribution, a package is a bundle of software and a corresponding configuration and documentation that makes it easier for the user to install, update and use the software.

The *Debian GNU/Linux* distribution is the biggest distribution of the Debian distribution family. The Debian GNU/Linux Project was launched by Ian Murdock in 1993. Today thousands of volunteers are working on the project. Debian GNU/Linux aims to provide a very reliable operating system. It also promotes Richard Stallman's vision of an operating system that respects the freedoms of the user to run, study, distribute and improve the software. This is why it does not provide any proprietary software by default.

Ubuntu is another Debian-based distribution worth mentioning. Ubuntu was created by Mark Shuttleworth and his team in 2004, with the mission to bring an easy to use Linux desktop environment. Ubuntu's mission is to provide a free software to everyone across the world as well as to cut the cost of professional services. The distribution has a scheduled release every six months with a long-term support release every 2 years.

Red Hat is a Linux distribution developed and maintained by the identically named software company, which was acquired by IBM in 2019. The Red Hat Linux distribution was started in 1994 and re-branded in 2003 to *Red Hat Enterprise Linux*, often abbreviated as RHEL. It is provided to companies as a reliable enterprise solution that is supported by Red Hat and comes with software that aims to ease the use of Linux in professional server environments. Some of its components require fee-based subscriptions or licenses. The *CentOS* project uses the freely available source code of Red Hat Enterprise Linux and compiles it to a distribution which is available completely free of charge, but in return does not come with commercial support.

Both RHEL and CentOS are optimized for use in server environments. The *Fedora* project was founded in 2003 and creates a Linux distribution which is aimed at desktop computers. Red Hat initiated and maintains the Fedora distribution ever since. Fedora is very progressive and adopts new technologies very quickly and is sometimes considered a test-bed for new technologies which later might be included in RHEL. All Red Hat based distributions use the package format `rpm`.

The company SUSE was founded in 1992 in Germany as a Unix service provider. The first version of *SUSE Linux* was released in 1994. Over the years SUSE Linux became mostly known for its YaST configuration tool. This tool allows administrators to install and configure software and hardware, set up servers and networks. Similar to RHEL, SUSE releases *SUSE Linux Enterprise Server*, which is their commercial edition. This is less frequently released and is suitable for enterprise and production deployment. It is distributed as a server as well as a desktop environment, with fit-for-purpose packages. In 2004, SUSE released the *openSUSE* project, which opened opportunities for developers and users to test and further develop the system. The openSUSE distribution is freely available to download.

Independent distributions have been released over the years. Some of them are based on either Red Hat or Ubuntu, some are designed to improve a specific propriety of a system or hardware. There are distributions built with specific functionalities like *QubesOS*, a very secure desktop environment, or *Kali Linux*, which provides an environment for exploiting software vulnerabilities, mainly used by penetration testers. Recently various super small Linux distributions are designed to specifically run in Linux containers, such as Docker. There are also distributions built specifically for components of embedded systems and even smart devices.

Embedded Systems

Embedded systems are a combination of computer hardware and software designed to have a specific function within a larger system. Usually they are part of other devices and help to control these devices. Embedded systems are found in automotive, medical and even military applications. Due to its wide variety of applications, a variety of operating systems based on the Linux kernel was developed in order to be used in embedded systems. A significant part of smart devices have a Linux kernel based operating system running on it.

Therefore, with embedded systems comes embedded software. The purpose of this software is to access the hardware and make it usable. The major advantages of Linux over any proprietary embedded software include cross vendor platform compatibility, development, support and no license fees. Two of the most popular embedded software projects are Android, that is mainly used on mobile phones across a variety of vendors and Raspbian, which is used mainly on Raspberry Pi.

Android

Android is mainly a mobile operating system developed by Google. Android Inc. was founded in 2003 in Palo Alto, California. The company initially created an operating system meant to run on digital cameras. In 2005, Google bought Android Inc. and developed it to be one of the biggest mobile operating systems.

The base of Android is a modified version of the Linux kernel with additional open source software. The operating system is mainly developed for touchscreen devices, but Google has developed versions for TV and wrist watches. Different versions of Android have been developed for game consoles, digital cameras, as well as PCs.

Android is freely available in open source as *Android Open Source Project* (AOSP). Google offers a series of proprietary components in addition to the open source core of Android. These components include applications such as Google Calendar, Google Maps, Google Mail, the Chrome browser as well as the Google Play Store which facilitates the easy installation of apps. Most users consider these tools an integral part of their Android experience. Therefore almost all mobile devices shipped with Android in Europe and America include proprietary Google software.

Android on embedded devices has many advantages. The operating system is intuitive and easy to use with a graphical user interface, it has a very wide developer community, therefore it is easy to find help for development. It is also supported by the majority of the hardware vendors with an Android driver, therefore it is easy and cost effective to prototype an entire system.

Raspbian and the Raspberry Pi

Raspberry Pi is a low cost, credit-card sized computer that can function as a full-functionality desktop computer, but it can be used within an embedded Linux system. It is developed by the Raspberry Pi Foundation, which is an educational charity based in UK. It mainly has the purpose to teach young people to learn to program and understand the functionality of computers. The Raspberry Pi can be designed and programmed to perform desired tasks or operations that are part of a much more complex system.

The specialties of the Raspberry Pi include a set of General Purpose Input-Output (GPIO) pins which can be used to attach electronic devices and extension boards. This allows using the Raspberry Pi as a platform for hardware development. Although it was intended for educational purposes, Raspberry Pis are used today in various DIY projects as well as for industrial prototyping when developing embedded systems.

The Raspberry Pi uses ARM processors. Various operating systems, including Linux, run on the Raspberry Pi. Since the Raspberry Pi does not contain a hard disk, the operating system is started from an SD memory card. One of the most prominent Linux distributions for the Raspberry Pi is

Raspbian. As the name suggests, it belongs to the Debian distribution family. It is customized to be installed on the Raspberry Pi hardware and provides more than 35000 packages optimized for this environment. Besides Raspbian, numerous other Linux distributions exist for the Raspberry Pi, like, for example, Kodi, which turns the Raspberry Pi into a media center.

Linux and the Cloud

The term *cloud computing* refers to a standardized way of consuming computing resources, either by buying them from a public cloud provider or by running a private cloud. As of 2017 reports, Linux runs 90% of the public cloud workload. Every cloud provider, from *Amazon Web Services* (AWS) to *Google Cloud Platform* (GCP), offers different forms of Linux. Even Microsoft offers Linux-based virtual machines in their *Azure* cloud today.

Linux is usually offered as part of *Infrastructure as a Service* (IaaS) offering. IaaS instances are virtual machines which are provisioned within minutes in the cloud. When starting an IaaS instance, an image is chosen which contains the data that is deployed to the new instance. Cloud providers offer various images containing ready to run installations of both popular Linux distributions as well as own versions of Linux. The cloud user chooses an image containing their preferred distribution and can access a cloud instance running this distribution shortly after. Most cloud providers add tools to their images to adjust the installation to a specific cloud instance. These tools can, for example, extend the file systems of the image to fit the actual hard disk of the virtual machine.

Guided Exercises

1. How is Debian GNU/Linux different from Ubuntu? Name two aspects.

2. What are the most common environments/platforms Linux is used for? Name three different environments/platforms and name one distribution you can use for each.

3. You are planning to install a Linux distribution in a new environment. Name four things that you should consider when choosing a distribution.

4. Name three devices that the Android OS runs on, other than smartphones.

5. Explain three major advantages of cloud computing.

Explorational Exercises

1. Considering cost and performance, which distributions are mostly suitable for a business that aims to reduce licensing costs, while keeping performance at its highest? Explain why.

2. What are the major advantages of the Raspberry Pi and which functions can they take in business?

3. What range of distributions does Amazon Cloud Services and Google Cloud offer? Name at least three common ones and two different ones.

Summary

In this lesson you learned:

- What distributions does Linux have
- What are Linux embedded systems
- How are Linux embedded systems used
- Different applicabilities of Android
- Different uses of a Raspberry Pi
- What is Cloud Computing
- What role does Linux play in cloud computing

Answers to Guided Exercises

1. How is Debian GNU/Linux different from Ubuntu? Name two aspects.

   Ubuntu is based on a snapshot of Debian, therefore there are many similarities between them. However, there are still significant differences between them. The first one would be the applicability for beginners. Ubuntu is recommended for beginners because of its ease of use and on the other hand Debian is recommended for more advanced users. The major difference is the complexity of the user configuration that Ubuntu doesn't require during the installation process.

   Another difference would be the stability of each distribution. Debian is considered to be more stable compared to Ubuntu. This is because Debian receives fewer updates that are tested in detail and the entire operating system is more stable. On the other hand, Ubuntu enables the user to use the latest releases of software and all the new technologies.

2. What are the most common environments/platforms Linux is used for? Name three different environments/platforms and name one distribution you can use for each.

   A few of the common environments/platforms would be smartphone, desktop and server. On smartphones, it can be used by distributions such as Android. On desktop and server, it can be used by any distribution that is mostly suitable with the functionality of that machine, from Debian, Ubuntu to CentOS and Red Hat Enterprise Linux.

3. You are planning to install a Linux distribution in a new environment. Name four things that you should consider when choosing a distribution.

   When choosing a distribution, a few of the main things that should be considered is cost, performance, scalability, how stable it is and the hardware demand of the system.

4. Name three devices that the Android OS runs on, other than smartphones.

   Other devices that Android runs on are smart TVs, tablet computers, Android Auto and smartwatches.

5. Explain three major advantages of cloud computing.

   The major advantages of cloud computing are flexibility, easy to recover and low use cost. Cloud based services are easy to implement and scale, depending on the business requirements. It has a major advantage in backup and recovery solutions, as it enables businesses to recover from incidents faster and with less repercussions. Furthermore, it reduces operation costs, as it allows to pay just for the resources that a business uses, on a

subscription-based model.

Answers to Explorational Exercises

1. Considering cost and performance, which distributions are mostly suitable for a business that aims to reduce licensing costs, while keeping performance at its highest? Explain why.

   One of the most suitable distributions to be used by business is CentOS. This is because it incorporates all Red Hat products, which are further used within their commercial operating system, while being free to use. Similarly, Ubuntu LTS releases guarantee support for a longer period of time. The stable versions of Debian GNU/Linux are also often used in enterprise environments.

2. What are the major advantages of the Raspberry Pi and which functions can they take in business?

   Raspberry Pi is small in size while working as a normal computer. Furthermore, it is low cost and can handle web traffic and many other functionalities. It can be used as a server, a firewall and can be used as the main board for robots, and many other small devices.

3. What range of distributions does Amazon Cloud Services and Google Cloud offer? Name at least three common ones and two different ones.

   The common distributions between Amazon and Google Cloud Services are Ubuntu, CentOS and Red Hat Enterprise Linux. Each cloud provider also offers specific distributions that the other one doesn't. Amazon has Amazon Linux and Kali Linux, while Google offers the use of FreeBSD and Windows Servers.

1.2 Major Open Source Applications

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 1.2

Weight

2

Key knowledge areas

- Desktop applications
- Server applications
- Development languages
- Package management tools and repositories

Partial list of the used files, terms and utilities

- OpenOffice.org, LibreOffice, Thunderbird, Firefox, GIMP
- Nextcloud, ownCloud
- Apache HTTPD, NGINX, MariaDB, MySQL, NFS, Samba
- C, Java, JavaScript, Perl, shell, Python, PHP
- dpkg, apt-get, rpm, yum

1.2 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	1 The Linux Community and a Career in Open Source
Objective:	1.2 Major Open Source Applications
Lesson:	1 of 1

Introduction

An application is a computer program whose purpose is not directly tied to the inner workings of the computer, but with tasks performed by the user. Linux distributions offer many application options to perform a variety of tasks, such as office applications, web browsers, multimedia players and editors, etc. There is often more than one application or tool to perform a particular job. It is up to the user to choose the application which best fits their needs.

Software Packages

Almost every Linux distribution offers a pre-installed set of default applications. Besides those pre-installed applications, a distribution has a package repository with a vast collection of applications available to install through its *package manager*. Although the various distributions offer roughly the same applications, several different package management systems exist for various distributions. For instance, Debian, Ubuntu and Linux Mint use the `dpkg`, `apt-get` and `apt` tools to install software packages, generally referred as *DEB packages*. Distributions such as Red Hat, Fedora and CentOS use the `rpm`, `yum` and `dnf` commands instead, which in turn install *RPM packages*. As the application packaging is different for each distribution family, it is very

important to install packages from the correct repository designed to the particular distribution. The end user usually doesn't have to worry about those details, as the distribution's package manager will choose the right packages, the required dependencies and future updates. Dependencies are auxiliary packages needed by programs. For example, if a library provides functions for dealing with JPEG images which are used by multiple programs, this library is likely packaged in its own package on which all applications using the library depend.

The commands `dpkg` and `rpm` operate on individual package files. In practice, almost all package management tasks are performed by the commands `apt-get` or `apt` on systems that use DEB packages or by `yum` or `dnf` on systems that use RPM packages. These commands work with catalogues of packages, can download new packages and their dependencies, and check for newer versions of the installed packages.

Package Installation

Suppose you have heard about a command called `figlet` which prints enlarged text on the terminal and you want to try it. However, you get the following message after executing the command `figlet`:

```
$ figlet
-bash: figlet: command not found
```

That probably means the package is not installed on your system. If your distribution works with DEB packages, you can search its repositories using `apt-cache search package_name` or `apt search package_name`. The `apt-cache` command is used to search for packages and to list information about available packages. The following command looks for any occurrences of the term "figlet" in the package's names and descriptions:

```
$ apt-cache search figlet
figlet - Make large character ASCII banners out of ordinary text
```

The search identified a package called *figlet* that corresponds to the missing command. The installation and removal of a package require special permissions granted only to the system's administrator: the user named `root`. On desktop systems, ordinary users can install or remove packages by prepending the command `sudo` to the installation/removal commands. That will require you to type your password to proceed. For DEB packages, the installation is performed with the command `apt-get install package_name` or `apt install package_name`:

```
$ sudo apt-get install figlet
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  figlet
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
```

At this point the package will be downloaded and installed on the system. Any dependencies that the package eventually needs will also be downloaded and installed:

```
Need to get 184 kB of archives.
After this operation, 741 kB of additional disk space will be used.
Get:1 http://archive.raspbian.org/raspbian stretch/main armhf figlet armhf 2.2.5-2 [184 kB]
Fetched 184 kB in 0s (213 kB/s)
Selecting previously unselected package figlet.
(Reading database ... 115701 files and directories currently installed.)
Preparing to unpack .../figlet_2.2.5-2_armhf.deb ...
Unpacking figlet (2.2.5-2) ...
Setting up figlet (2.2.5-2) ...
update-alternatives: using /usr/bin/figlet-figlet to provide /usr/bin/figlet (figlet) in
auto mode
Processing triggers for man-db (2.7.6.1-2) ...
```

After the download is finished, all files are copied to the proper locations, any additional configuration will be performed and the command will become available:

```
$ figlet Awesome!
    _                                          _
   / \__      _____  ___  ___  _ __ ___   ___| |
  / _ \ \ /\ / / _ \/ __|/ _ \| '_ ` _ \ / _ \ |
 / ___ \ V  V /  __/\__ \ (_) | | | | | |  __/_|
/_/   \_\_/\_/ \___||___/\___/|_| |_| |_|\___(_)
```

In distributions based on RPM packages, searches are performed using `yum search package_name` or `dnf search package_name`. Let's say you want to display some text in a more irreverent way, followed by a cartoonish cow, but you are not sure about the package that can perform that task. As with the DEB packages, the RPM search commands accept descriptive terms:

```
$ yum search speaking cow
Last metadata expiration check: 1:30:49 ago on Tue 23 Apr 2019 11:02:33 PM -03.
==================== Name & Summary Matched: speaking, cow ====================
```

```
cowsay.noarch : Configurable speaking/thinking cow
```

After finding a suitable package at the repository, it can be installed with `yum install package_name` or `dnf install package_name`:

```
$ sudo yum install cowsay
Last metadata expiration check: 2:41:02 ago on Tue 23 Apr 2019 11:02:33 PM -03.
Dependencies resolved.
================================================================================
 Package         Arch           Version               Repository         Size
================================================================================
Installing:
 cowsay          noarch         3.04-10.fc28          fedora             46 k

Transaction Summary
================================================================================
Install  1 Package

Total download size: 46 k
Installed size: 76 k
Is this ok [y/N]: y
```

Once again, the desired package and all its possible dependencies will be downloaded and installed:

```
Downloading Packages:
cowsay-3.04-10.fc28.noarch.rpm                   490 kB/s |  46 kB     00:00
================================================================================
Total                                             53 kB/s |  46 kB     00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
Preparing        :                                                        1/1
Installing       : cowsay-3.04-10.fc28.noarch                             1/1
Running scriptlet: cowsay-3.04-10.fc28.noarch                             1/1
Verifying        : cowsay-3.04-10.fc28.noarch                             1/1

Installed:
cowsay.noarch 3.04-10.fc28
```

```
Complete!
```

The command `cowsay` does exactly what its name implies:

```
$ cowsay "Brought to you by yum"
 _________________________
< Brought to you by yum >
 -------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Although they may seem useless, commands `figlet` and `cowsay` provide a way to draw the attention of other users to relevant information.

Package Removal

The same commands used to install packages are used to remove them. All the commands accept the `remove` keyword to uninstall an installed package: `apt-get remove package_name` or `apt remove package_name` for DEB packages and `yum remove package_name` or `dnf remove package_name` for RPM packages. The `sudo` command is also needed to perform the removal. For example, to remove the previously installed package *figlet* from a DEB-based distribution:

```
$ sudo apt-get remove figlet
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be REMOVED:
  figlet
0 upgraded, 0 newly installed, 1 to remove and 0 not upgraded.
After this operation, 741 kB disk space will be freed.
Do you want to continue? [Y/n] Y
```

After confirming the operation, the package is erased from the system:

```
(Reading database ... 115775 files and directories currently installed.)
Removing figlet (2.2.5-2) ...
Processing triggers for man-db (2.7.6.1-2) ...
```

A similar procedure is performed on an RPM-based system. For example, to remove the previously installed package *cowsay* from an RPM-based distribution:

```
$ sudo yum remove cowsay
Dependencies resolved.
================================================================================
 Package          Arch             Version                Repository         Size
================================================================================
Removing:
 cowsay           noarch           3.04-10.fc28           @fedora            76 k

Transaction Summary
================================================================================
Remove  1 Package

Freed space: 76 k
Is this ok [y/N]: y
```

Likewise, a confirmation is requested and the package is erased from the system:

```
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :                                                        1/1
  Erasing          : cowsay-3.04-10.fc28.noarch                             1/1
  Running scriptlet: cowsay-3.04-10.fc28.noarch                             1/1
  Verifying        : cowsay-3.04-10.fc28.noarch                             1/1

Removed:
  cowsay.noarch 3.04-10.fc28

Complete!
```

The configuration files of the removed packages are kept on the system, so they can be used again if the package is reinstalled in the future.

Office Applications

Office applications are used for editing files such as texts, presentations, spreadsheets and other

formats commonly used in an office environment. These applications are usually organised in collections called *office suites*.

For a long time, the most used office suite in Linux was the *OpenOffice.org* suite. OpenOffice.org was an open source version of the *StarOffice suite* released by *Sun Microsystems*. A few years later Sun was acquired by *Oracle Corporation*, which in turn transferred the project to the *Apache Foundation* and OpenOffice.org was renamed to *Apache OpenOffice*. Meanwhile, another office suite based on the same source code was released by the *Document Foundation*, which named it *LibreOffice*.

The two projects have the same basic features and are compatible with the document formats from *Microsoft Office*. However, the preferred document format is the *Open Document Format*, a fully open and ISO standardized file format. The use of ODF files ensures that documents can be transferred between operating systems and applications from different vendors, such as Microsoft Office. The main applications offered by OpenOffice/LibreOffice are:

Writer
: Text editor

Calc
: Spreadsheets

Impress
: Presentations

Draw
: Vector drawing

Math
: Math formulas

Base
: Database

Both LibreOffice and Apache OpenOffice are open source software, but LibreOffice is licensed under LGPLv3 and Apache OpenOffice is licensed under Apache License 2.0. The licensing distinction implies that LibreOffice can incorporate improvements made by Apache OpenOffice, but Apache OpenOffice cannot incorporate improvements made by LibreOffice. That, and a more active community of developers, are the reason most distributions adopt LibreOffice as their default office suite.

Web Browsers

For most users, the main purpose of a computer is to provide access to the Internet. Nowadays, web pages can act as a full featured app, with the advantage of being accessible from anywhere, without the need of installing extra software. That makes the web browser the most important application of the operating system, at least for the average user.

TIP One of the best sources for learning about web development is the MDN Web Docs, available at https://developer.mozilla.org/. Maintained by Mozilla, the site is full of tutorials for beginners and reference materials on most modern web technologies.

The main web browsers in the Linux environment are *Google Chrome* and *Mozilla Firefox*. Chrome is a web browser maintained by Google but is based on the open source browser named *Chromium*, which can be installed using the distribution's package manager and is fully compatible with Chrome. Maintained by Mozilla, a non-profit organization, Firefox is a browser whose origins are linked to Netscape, the first popular web browser to adopt the open source model. The Mozilla Foundation is deeply involved with the development of the open standards underlying the modern web.

Mozilla also develops other applications, like the e-mail client *Thunderbird*. Many users opt to use webmail instead of a dedicated email application, but a client like Thunderbird offers extra features and integrates best with other applications on the desktop.

Multimedia

Compared to the available web applications, desktop applications are still the best option for the creation of multimedia content. Multimedia related activities like video rendering often require high amounts of system resources, which are best managed by a local desktop application. Some of the most popular multimedia applications for the Linux environment and their uses are listed below.

Blender

A 3D renderer to create animations. Blender can also be used to export 3D objects to be printed by a 3D printer.

GIMP

A full-featured image editor, which can be compared with *Adobe Photoshop*, but has its own concepts and tools to work with images. GIMP can be used to create, edit and save most bitmap files, like JPEG, PNG, GIF, TIFF and many others.

Inkscape

A vector graphics editor, similar to *Corel Draw* or *Adobe Illustrator*. Inkscape's default format is SVG, which is an open standard for vector graphics. SVG files can be opened by any web browser and, due to its nature as a vector graphic, it can be used in flexible layouts of web pages.

Audacity

An audio editor. Audacity can be used to filter, to apply effects and to convert between many different audio formats, like MP3, WAV, OGG, FLAC, etc.

ImageMagick

ImageMagick is a command line tool to convert and edit most image file types. It can also be used to create PDF documents from image files and vice versa.

There are also many applications dedicated to multimedia playback. The most popular application for video playback is *VLC*, but some users prefer other alternatives, like *smplayer*. Local music playback also has many options, like *Audacious*, *Banshee* and *Amarok*, which can also manage a collection of local sound files.

Server Programs

When a web browser loads a page from a website, it actually connects to a remote computer and asks for a specific piece of information. In that scenario, the computer running the web browser is called the *client* and the remote computer is called the *server*.

The server computer, which can be an ordinary desktop computer or specialized hardware, needs a specific program to manage each type of information it will provide. Regarding serving web pages, most servers around the world deploy open source server programs. This particular server program is called an *HTTP server* (HTTP stands for *Hyper Text Transfer Protocol*) and the most popular ones are *Apache*, *Nginx* and *lighttpd*.

Even simple web pages may require many requests, which can be ordinary files — called static content — or dynamic content rendered from various sources. The role of an HTTP server is to collect and send all the requested data back to the browser, which then arranges the content as defined by the received HTML document (HTML stands for *Hyper Text Markup Language*) and other supporting files. Therefore, the rendering of a web page involves operations performed on the server side and operations performed on the client side. Both sides can use custom scripts to accomplish specific tasks. On the HTTP server side, it is quite common to use the PHP scripting language. JavaScript is the scripting language used on the client side (the web browser).

Server programs can provide all kinds of information. It's not uncommon to have a server

program requesting information provided by other server programs. That is the case when an HTTP server requires information provided by a database server.

For instance, when a dynamic page is requested, the HTTP server usually queries a database to collect all the required pieces of information and sends the dynamic content back to the client. In a similar way, when a user registers on a website, the HTTP server gathers the data sent by the client and stores it in a database.

A database is an organized set of information. A database server stores contents in a formatted fashion, making it possible to read, write and link large amounts of data reliably and at great speed. Open source database servers are used in many applications, not only on the Internet. Even local applications can store data by connecting to a local database server. The most common type of database is the *relational database*, where the data is organized in predefined tables. The most popular open source relational databases are *MariaDB* (originated from *MySQL*) and *PostgreSQL*.

Data Sharing

In local networks, like the ones found in offices and homes, it is desirable that computers not only should be able to access the Internet, but also should be able to communicate with each other. Sometimes a computer acts as a server, sometimes the same computer acts as a client. That is necessary when one wants to access files on another computer in the network — for instance, access a file stored on a desktop computer from a portable device — without the hassle of copying it to a USB drive or the like.

Between Linux machines, *NFS* (*Network File System*) is often used. The NFS protocol is the standard way to share file systems in networks equipped only with Unix/Linux machines. With NFS, a computer can share one or more of its directories with specific computers on the network, so they can read and write files in these directories. NFS can even be used to share an entire operating system's directory tree with clients that will use it to boot from. These computers, called *thin clients*, are mostly often used in large networks to avoid the maintenance of each individual operating system on each machine.

If there are other types of operating systems attached to the network, it is recommended to use a data sharing protocol that can be understood by all of them. This requirement is fulfilled by *Samba*. Samba implements a protocol for sharing files over the network originally made for the Windows operating system, but today is compatible with all major operating systems. With Samba, computers in the local network not only can share files, but also printers.

On some local networks, the authorization given upon login on a workstation is granted by a central server, called the *domain controller*, which manages the access to various local and remote resources. The domain controller is a service provided by Microsoft's *Active Directory*. Linux

workstations can associate with a domain controller by using Samba or an authentication subsystem called *SSSD*. As of version 4, Samba can also work as a domain controller on heterogeneous networks.

If the goal is to implement a cloud computing solution able to provide various methods of web based data sharing, two alternatives should be considered: *ownCloud* and *Nextcloud*. The two projects are very similar because Nextcloud is a spin-off of ownCloud, which is not unusual among open source projects. Such spin-offs are usually called a *fork*. Both provide the same basic features: file sharing and sync, collaborative workspaces, calendar, contacts and mail, all through desktop, mobile and web interfaces. Nextcloud also provides private audio/video conferencing, whilst ownCloud is more focused on file sharing and integration with third-party software. Many more features are provided as plugins which can be activated later as needed.

Both ownCloud and Nextcloud offer a paid version with extra features and extended support. What makes them different from other commercial solutions is the ability to install Nextcloud or ownCloud on a private server, free of charge, avoiding keeping sensitive data on an unknown server. As all the services depend on HTTP communication and are written in PHP, the installation must be performed on a previous configured web server, like Apache. If you consider installing ownCloud or Nextcloud on your own server, make sure to also enable HTTPS to encrypt all connections to your cloud.

Network Administration

Communication between computers is only possible if the network is working correctly. Normally, the network configuration is done by a set of programs running on the router, responsible for setting up and checking the network availability. In order to achieve this, two basic network services are used: *DHCP* (*Dynamic Host Configuration Protocol*) and *DNS* (*Domain Name System*).

DHCP is responsible for assigning an IP address to the host when a network cable is connected or when the device enters a wireless network. When connecting to the Internet, the ISP's DHCP server will provide an IP address to the requesting device. A DHCP server is very useful in local area networks also, to automatically provide IP addresses to all connected devices. If DHCP is not configured or if it's not working properly, it would be necessary to manually configure the IP address of each device connected to the network. It is not practical to manually set the IP addresses on large networks or even in small networks, that's why most network routers come with a DHCP server pre-configured by default.

The IP address is required to communicate with another device on an IP network, but domain names like `www.lpi.org` are much more likely to be remembered than an IP number like `203.0.113.165`. The domain name by itself, however, is not enough to establish the

communication through the network. That is why the domain name needs to be translated to an IP address by a DNS server. The IP address of the DNS server is provided by the ISP's DHCP server and it's used by all connected systems to translate domain names to IP addresses.

Both DHCP and DNS settings can be modified by entering the web interface provided by the router. For instance, it is possible to restrict the IP assignment only to known devices or associate a fixed IP address to specific machines. It's also possible to change the default DNS server provided by the ISP. Some third-party DNS servers, like the ones provided by Google or OpenDNS, can sometimes provide faster responses and extra features.

Programming Languages

All computer programs (client and server programs, desktop applications and the operating system itself) are made using one or more programming languages. Programs can be a single file or a complex system of hundreds of files, which the operating system treats as an instruction sequence to be interpreted and performed by the processor and other devices.

There are numerous programming languages for very different purposes and Linux systems provide a lot of them. Since open source software also includes the sources of the programs, Linux systems offer developers perfect conditions to understand, modify or create software according to their own needs.

Every program begins as a text file, called *source code*. This source code is written in a more or less human-friendly language that describes what the program is doing. A computer processor can not directly execute this code. In *compiled languages*, the source code is therefore be converted to a *binary file* which can then be executed by the computer. A program called *compiler* is responsible for doing the conversion from source code to executable form. Since the compiled binary is specific to one kind of processor, the program might have to be re-compiled to run on another type of computer.

In *interpreted languages*, the program does not need to be previously compiled. Instead, an *interpreter* reads the source code and executes its instruction every time the program is run. This makes the development easier and faster, but at the same time interpreted programs tend to be slower than compiled programs.

Here some of the most popular programming languages:

JavaScript

JavaScript is a programming language mostly used in web pages. In its origins, JavaScript applications were very simple, like form validation routines. As for today, JavaScript is considered a first class language and it is used to create very complex applications not only on

the web, but on servers and mobile devices.

C

The C programming language is closely related with operating systems, particularly Unix, but it is used to write any kind of program to almost any kind of device. The great advantages of C are flexibility and speed. The same source code written in C can be compiled to run in different platforms and operating systems, with little or no modification. After being compiled, however, the program will run only in the targeted system.

Java

The main aspect of Java is that programs written in this language are portable, which means that the same program can be executed in different operating systems. Despite the name, Java is not related to JavaScript.

Perl

Perl is a programming language most used to process text content. It has a strong regular expressions emphasis, which makes Perl a language suited for text filtering and parsing.

Shell

The shell, particularly the Bash shell, is not just a programming language, but an interactive interface to run other programs. Shell programs, known as *shell scripts*, can automate complex or repetitive tasks on the command line environment.

Python

Python is a very popular programming language among students and professionals not directly involved with computer science. Whilst having advanced features, Python is a good way to start learning programming for its easy to use approach.

PHP

PHP is most used as a server side scripting language for generating content for the web. Most online HTML pages are not static files, but dynamic content generated by the server from various sources, like databases. PHP programs — sometimes just called PHP pages or PHP scripts — are often used to generate this kind of content. The term LAMP comes from the combination of a Linux operating system, an Apache HTTP server, a MySQL (or MariaDB) database and PHP programming. LAMP servers are a very popular solution for running web servers. Besides PHP, all of the programming languages described before can be used to implement such applications too.

C and Java are compiled languages. In order to be executed by the system, source code written in C is converted to binary machine code, whereas Java source code is converted to *bytecode* executed in a special software environment called *Java Virtual Machine*. JavaScript, Perl, Shell

script, Python and PHP are all interpreted languages, which are also called *scripting languages*.

Guided Exercises

1. For each of the following commands, identify whether it is associated with the *Debian packaging system* or the *Red Hat packaging system*:

`dpkg`	
`rpm`	
`apt-get`	
`yum`	
`dnf`	

2. Which command could be used to install Blender on Ubuntu? After installation, how can the program be executed?

3. Which application from the LibreOffice suite can be used to work with electronic spreadsheets?

4. Which open-source web browser is used as the basis for the development of Google Chrome?

5. SVG is an open standard for vector graphics. Which is the most popular application for editing SVG files in Linux systems?

6. For each of the following file formats, write the name of an application able to open and edit the corresponding file:

`png`	
`doc`	
`xls`	
`ppt`	
`wav`	

7. Which software package allows file sharing between Linux and Windows machines over the local network?

Explorational Exercises

1. You know that configuration files are kept even if the associated package is removed from the system. How could you automatically remove the package named *cups* and its configuration files from a DEB based system?

2. Suppose you have many TIFF image files and want to convert them to JPEG. Which software package could be used to convert those files directly at the command line?

3. Which software package do you need to install in order to be able to open Microsoft Word documents sent to you by a Windows user?

4. Every year, linuxquestions.org promotes a survey about the most popular Linux applications. Visit https://www.linuxquestions.org/questions/2018-linuxquestions-org-members-choice-awards-128/ and find out which desktop applications are most popular among experienced Linux users.

Summary

In this lesson, you learned:

- The package management systems used in major Linux distributions
- Open source applications that can edit popular file formats
- The server programs underlying many important Internet and local network services
- Common programming languages and their uses

Answers to Guided Exercises

1. For each of the following commands, identify whether it is associated with the *Debian packaging system* or the *Red Hat packaging system*:

`dpkg`	Debian packaging system
`rpm`	Red Hat packaging system
`apt-get`	Debian packaging system
`yum`	Red Hat packaging system
`dnf`	Red Hat packaging system

2. Which command could be used to install Blender on Ubuntu? After installation, how can the program be executed?

   The command `apt-get install blender`. The package name should be specified in lowercase. The program can be executed directly from the terminal with the command `blender` or by choosing it on the applications menu.

3. Which application from the LibreOffice suite can be used to work with electronic spreadsheets?

   Calc

4. Which open-source web browser is used as the basis for the development of Google Chrome?

   Chromium

5. SVG is an open standard for vector graphics. Which is the most popular application for editing SVG files in Linux systems?

   Inkscape

6. For each of the following file formats, write the name of an application able to open and edit the corresponding file:

`png`	Gimp
`doc`	LibreOffice Writer
`xls`	LibreOffice Calc
`ppt`	LibreOffice Impress

`wav`	Audacity

7. Which software package allows file sharing between Linux and Windows machines over the local network?

   Samba

Answers to Explorational Exercises

1. You know that configuration files are kept even if the associated package is removed from the system. How could you automatically remove the package named *cups* and its configuration files from a DEB based system?

   `apt-get purge cups`

2. Suppose you have many TIFF image files and want to convert them to JPEG. Which software package could be used to convert those files directly at the command line?

   ImageMagick

3. Which software package do you need to install in order to be able to open Microsoft Word documents sent to you by a Windows user?

   LibreOffice or OpenOffice

4. Every year, linuxquestions.org promotes a survey about the most popular Linux applications. Visit https://www.linuxquestions.org/questions/2018-linuxquestions-org-members-choice-awards-128/ and find out which desktop applications are most popular among experienced Linux users.

   Browser: Firefox. Email client: Thunderbird. Media player: VLC. Raster graphics editor: GIMP.

1.3 Open Source Software and Licensing

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 1.3

Weight

1

Key knowledge areas

- Open source philosophy
- Open source licensing
- Free Software Foundation (FSF), Open Source Initiative (OSI)

Partial list of the used files, terms and utilities

- Copyleft, Permissive
- GPL, BSD, Creative Commons
- Free Software, Open Source Software, FOSS, FLOSS
- Open source business models

1.3 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	1 The Linux Community and a Career in Open Source
Objective:	1.3 Open Source Software and Licensing
Lesson:	1 of 1

Introduction

While the terms *free software* and *open source software* are widely used, there are still some misconceptions about their meaning. In particular, the concept of “freedom” needs closer examination. Let’s start with the definition of the two terms.

Definition of Free and Open Source Software

Criteria of Free Software

First of all, “free” in the context of free software has nothing to do with “free of charge”, or as the founder of the *Free Software Foundation* (FSF), Richard Stallman, succinctly puts it:

> To understand the concept, you should think of “free” as in “free speech,” not as in “free beer”.
>
> — Richard Stallman, What is free software?

Regardless of whether you have to pay for the software or not, there are four criteria which constitute free software. Richard Stallman describes these criteria as "the four essential freedoms", the counting of which he starts from zero:

- "The freedom to run the program as you wish, for any purpose (freedom 0)."

  Where, how and for what purpose the software is used can neither be prescribed nor restricted.

- "The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this."

  Everyone may change the software according to their ideas and needs. This in turn presupposes that the so-called *source code*, i.e. all files of which a software consists, must be available in a form readable by programmers. And, of course, this right applies to a single user who may want to add a single feature, as well as to software companies that build complex systems such as smartphone operating systems or router firmware.

- "The freedom to redistribute copies so you can help others (freedom 2)."

  This freedom explicitly encourages each user to share the software with others. It is therefore a matter of the widest possible distribution and thus the widest possible community of users and developers who, on the basis of these freedoms, further develop and improve the software for the benefit of all.

- "The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this."

  This is not only about the distribution of free software, but about the distribution of *modified* free software. Anyone who makes changes to free software has the right to make the changes available to others. If they do so, they are obliged to do so freely as well, i.e. they must not restrict the original freedoms when distributing the software, even if they modified or extended it. For example, if a group of developers has different ideas about the direction of a specific software than the original authors, it can split off its own development branch (called a *fork*) and continue it as a new project. But, of course, all obligations associated with these freedoms remain.

The emphasis on the idea of freedom is also consistent insofar as every freedom movement is directed *against* something, namely an opponent who suppresses the postulated freedoms, who regards software as property and wants to keep it under lock and key. In contrast to free software, such software is called *proprietary*.

Open Source Software vs. Free Software

For many, *free software* and *open source software* are synonyms. The frequently used abbreviation *FOSS* for *Free and Open Source Software* emphasizes this commonality. FLOSS for *Free/Libre and Open Source Software* is another popular term, which unmistakably emphasizes the idea of freedom also for other languages other than English. However, if one considers the origin and development of both terms, a differentiation is worthwhile.

The term *free software* with the definition of the described four freedoms goes back to Richard Stallman and the GNU project founded by him in 1985 — almost 10 years before the emergence of Linux. The name “GNU is not Unix” describes the intention with a wink of the eye: GNU started as an initiative to develop a technically convincing solution — namely the operating system Unix — from scratch, to make it available to the general public and to improve it continuously with the general public. The openness of the source code was merely a technical and organizational necessity for this, but in its self-image the free software movement is still a *social* and *political* — some also say ideological — movement.

With the success of Linux, the collaborative possibilities of the Internet, and the thousands of projects and companies that emerged in this new software cosmos, the social aspect increasingly receded into the background. The openness of the source code itself changed from a technical requirement to a defining feature: as soon as the source code was visible, the software was considered “open source”. The social motives gave way to a more pragmatic approach to software development.

Free software and open source software work on the same thing, with the same methods and in a worldwide community of individuals, projects and companies. But since they have come together from different directions — one social and one pragmatic-technical — there are sometimes conflicts. These conflicts arise when the results of the joint work do not correspond to the original goals of both movements. This happens especially when software reveals its sources but does not respect the four freedoms of free software at the same time, for example when there are restrictions on disclosure, change, or connections with other software components.

The *license* under which the software is available determines which conditions a software is subject to with regard to use, distribution and modification. And because requirements and motives can be very different, countless different licenses have been created in the FOSS area. Due to the much more fundamental approach of the free software movement, it is not surprising that it does not recognize many open source licenses as “free” and therefore rejects them. Conversely, this is hardly the case due to the much more pragmatic open source approach.

Let’s take a brief look at the actually very complex area of licenses below.

Licenses

Unlike a refrigerator or a car, software is not a *physical* product, but a *digital* product. Thus, a company cannot actually transfer ownership of such a product by selling it and changing the physical possession — rather, it transfers the rights of use to that product, and the user contractually agrees to those rights of use. Which rights of use these are and above all are *not* is recorded in the software license, and thus it becomes understandable how important the regulations contained therein are.

While large vendors of proprietary software, such as Microsoft or SAP, have their own licenses that are precisely tailored to their products, the advocates of free and open source software have from the outset striven for clarity and general validity of their licenses, because after all, every user should understand them and, if necessary, use them personally for their own developments.

However, it should not be concealed that this ideal of simplicity can hardly be achieved because too many specific requirements and internationally not always compatible legal understandings stand in the way of this. To give just one example: German and American copyright law are fundamentally different. According to German law there is one *person* as *author* (more precisely: *Urheber*), whose work is their *intellectual property*. While the author can grant permission to use their work, they cannot assign or give up their authorship. The latter is alien to American law. Here, too, there is an author (who, however, can also be a company or an institution), but the author only has exploitation rights that they can transfer in part or in full and thus completely detach the author from the work. An internationally valid licence must be interpreted with respect of different legislation.

The consequences are numerous and sometimes very different FOSS licenses. Worse, still, are different versions of a license, or a mix of licenses (within a project, or even when connecting multiple projects) which can cause confusion or even legal disputes.

Both the representatives of free software and the advocates of the clearly economically oriented open source movement created their own organizations, which today are decisively responsible for the formulation of software licenses according to their principles and support their members in their enforcement.

Copyleft

The already mentioned *Free Software Foundation* (FSF) has formulated the *GNU General Public License* (GPL) as one of the most important licenses for free software, which is used by many projects, e.g. the Linux kernel. In addition, it has released licenses with case-specific customizations, such as the *GNU Lesser General Public License* (LGPL), which governs the combination of free software with modifications made to code where the source code for the

modifications do not have to be released to the public, the *GNU Affero General Public License* (AGPL), which covers selling access to hosted software, or the *GNU Free Documentation License* (FDL), which extends freedom principles to software documentation. In addition, the FSF makes recommendations for or against third-party licenses, and affiliated projects such as GPL-Violations.org investigate suspected violations of free licenses.

The FSF calls the principle according to which a free license also applies to modified variants of the software *copyleft* — in contrast to the principle of restrictive copyright which it rejects. The idea, therefore, is to transfer the liberal principles of a software license as unrestrictedly as possible to future variants of the software in order to prevent subsequent restrictions.

What sounds obvious and simple, however, leads to considerable complications in practice, which is why critics often call the copyleft principle “viral”, since it is transmitted to subsequent versions.

From what has been said it follows, for example, that two software components that are licensed under different copyleft licenses might not be combinable with each other, since both licenses cannot be transferred to the subsequent product at the same time. This can even apply to different versions of the same license!

For this reason, newer licenses or license versions often no longer grasp the copyleft so rigorously. Already the mentioned *GNU Lesser General Public License* (LGPL) is in this sense a concession to be able to connect free software with “non-free” components, as it is frequently done with so-called *libraries*. Libraries contain subroutines or routines, which in turn are used by various other programs. This leads to the common situation where proprietary software calls such a subroutine from a free library.

Another way to avoid license conflicts is *dual licensing*, where one software is licensed under different licenses, e.g. a free license and a proprietary license. A typical use case is a free version of a software which might only be used when respecting the copyleft restrictions and the alternative offering to obtain the software under a different license which frees the licensee from certain restriction in return for a fee which could be used to fund the development of the software.

It should therefore become clear that the choice of license for software projects should be made with much caution, since the cooperation with other projects, the combinability with other components and also the future design of the own product depend on it. The copyleft presents developers with special challenges in this respect.

Open Source Definition and Permissive Licenses

On the open source side, it is the *Open Source Initiative* (OSI), founded in 1998 by Eric S. Raymond and Bruce Perens, which is mainly concerned with licensing issues. It has also developed a standardized procedure for checking software licenses for compliance with its *Open Source Definition*. More than 80 recognized open source licenses can currently be found on the OSI website.

Here they also list licenses as "OSI-approved" that explicitly contradict the copyleft principle, especially the *BSD licenses* group. The *Berkeley Software Distribution* (BSD) is a variant of the Unix operating system originally developed at the University of Berkeley, which later gave rise to free projects such as *NetBSD*, *FreeBSD* and *OpenBSD*. The licenses underlying these projects are often referred to as *permissive*. In contrast to copyleft licenses, they do not have the aim of establishing the terms of use of modified variants. Rather, the maximum freedom should help the software to be as widely distributed as possible by leaving the editors of the software alone to decide how to proceed with the edits — whether, for example, they also release them or treat them as closed source and distribute them commercially.

The *2-Clause BSD License*, also called *Simplified BSD License* or *FreeBSD License*, proves how reduced such a permissive license can be. In addition to the standardized liability clause, which protects developers from liability claims arising from damage caused by the software, the license consists of only the following two rules:

> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
> 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Creative Commons

The successful development concept of FLOSS and the associated technological progress led to attempts to transfer the open source principle to other, non-technical areas. The preparation and provision of knowledge, as well as the creative cooperation in solving complex tasks, are now regarded as evidence of the extended, content-related open source principle.

This led to the need to create reliable foundations in these areas as well, according to which work results can be shared and processed. Since the available software licenses were hardly suitable for

this, there were numerous attempts to convert the specific requirements from scientific work to digitized works of art “in the spirit of open source” into similarly handy licenses.

By far the most important initiative of this kind today is *Creative Commons* (CC), which summarizes its concerns as follows:

> Creative Commons is a global nonprofit organization that enables sharing and reuse of creativity and knowledge through the provision of free legal tools.
>
> — https://creativecommons.org/faq/#what-is-creative-commons-and-what-do-you-do

With Creative Commons, the focus of rights assignment goes back from the distributor to the author. An example: In traditional publishing, an author usually transfers all publishing rights (printing, translation, etc.) to a publisher, who in turn ensures the best possible distribution of the work. The significantly changed distribution channels of the Internet now put the author in a position to exercise many of these publishing rights herself and to decide for herself how her work may be used. Creative Commons gives the opportunity to determine this simply and legally reliably, but Creative Commons wants more: authors are encouraged to make their works available as a contribution to a general process of exchange and cooperation. Unlike traditional copyright, which gives the author all the rights that they can transfer to others as needed, the Creative Commons approach takes the opposite approach: the author makes her work available to the community, but can choose from a set of features those that need to be considered when using the work — the more features she chooses, the more restrictive the license.

And so the “Choose a License” principle of CC asks an author step by step for the individual properties and generates the recommended license, which the author can last assign to the work as text and icon.

For a better understanding, here is an overview of the six possible combinations and licenses offered by CC:

CC BY (“Attribution”)

The free license that allows anyone to edit and distribute the work as long as they name the author.

CC BY-SA (“Attribution-ShareAlike”)

As CC BY, except that the modified work may only be distributed under the same license. The principle reminds of the copyleft, because the license is “inherited” here as well.

CC BY-ND (“Attribution-NoDerivatives”)

Like CC BY, except that the work may only be passed on unmodified.

CC BY-NC ("Attribution-NonCommercial")

The work may be edited and distributed by naming the author, but only under non-commercial conditions.

CC BY-NC-SA ("Attribution-NonCommercial-ShareAlike")

As BY-NC, except that the work may only be shared under the same conditions (i.e. a copyleft-like license).

CC BY-NC-ND ("Attribution-NonCommercial-NoDerivatives")

The most restrictive license: the distribution is allowed with attribution of the author, but only unchanged and under non-commercial conditions.

Business Models in Open Source

In retrospect, the triumph of FLOSS acts like a grassroots movement of technophile idealists who, independent of economic constraints and free of monetary dependencies, put their work at the service of the general public. At the same time, companies worth billions have been created in the FLOSS environment; to name just one, the US company *Red Hat* founded in 1993 with annual sales of over 3 billion USD (2018), which was taken over by the IT giant IBM in 2018.

So let's take a look at the tension between the free and mostly free-of-charge distribution of high-quality software and the business models for its creators, because one thing should be clear: The countless highly qualified developers of free software must also earn money, and the originally purely non-commercial FLOSS environment must therefore develop sustainable business models in order to preserve its own cosmos.

A common approach, especially for larger projects in the initial phase, is the so-called *crowdfunding*, i.e. the collection of money donations via a platform like *Kickstarter*. In return, the donors receive a pre-defined bonus from the developers in the event of success, i.e. if previously defined goals are achieved, be it unlimited access to the product or special features.

Another approach is *dual licensing*: free software is offered in parallel under a more restrictive or even proprietary license, which in turn guarantees the customer more extensive services (response times in the event of errors, updates, versions for certain platforms, etc.). One example among many is *ownCloud*, which is being developed under the GPL and offers business customers a "Business Edition" under a proprietary license.

Let us also take ownCloud as an example of another widespread FLOSS business model: professional services. Many companies lack the necessary in-house technical knowledge to set up and operate complex and critical software reliably and, above all, securely. That's why they buy professional services such as consulting, maintenance or helpdesk directly from the

manufacturer. Liability issues also play a role in this decision, as the company transfers the risks of operation to the manufacturer.

If a software manages to become successful and popular in its field, it is peripheral monetization possibilities such as merchandising or certificates that customers acquire and thus point out its special status when using this software. The learning platform *Moodle* offers the certification of trainers, who document their knowledge to potential clients, for example, and this is just one example among countless others.

Software as a Service (SaaS) is another business model, especially for web-based technologies. Here, a cloud provider runs a software like a Customer Relationship Management (CRM) or a Content Management System (CMS) on their servers and grant their customers access to the installed application. This saves the customer installation and maintenance of the software. In return, the customer pays for the use of the software according to various parameters, for example the number of users. Availability and security play an important role as business-critical factors.

Last but not least, the model of developing customer-specific extensions into free software by order is particularly common in smaller projects. It is then usually up to the customer to decide how to proceed with these extensions, i.e. whether they also release the extensions or keep them under lock and key as part of their own business model.

One thing should have become clear: Although free software is usually available free of charge, numerous business models have been created in their environment, which are constantly modified and extended by countless freelancers and companies worldwide in a very creative form, which ultimately also ensures the continued existence of the entire FLOSS movement.

Guided Exercises

1. What are — in a nutshell — the “four freedoms” as defined by Richard Stallman and the Free Software Foundation?

freedom 0	
freedom 1	
freedom 2	
freedom 3	

2. What does the abbreviation FLOSS stand for?

3. You have developed free software and want to ensure that the software itself, but also all future works based on it, remain free as well. Which license do you choose?

CC BY	
GPL version 3	
2-Clause BSD License	
LGPL	

4. Which of the following licenses would you call permissive, which would you call copyleft?

Simplified BSD License	
GPL version 3	
CC BY	
CC BY-SA	

5. You have written a web application and published it under a free license. How can you earn money with your product? Name three possibilities.

Explorational Exercises

1. Under which license (including version) are the following applications available?

Apache HTTP Server	
MySQL Community Server	
Wikipedia articles	
Mozilla Firefox	
GIMP	

2. You want to release your software under the GNU GPL v3. What steps should you follow?

3. You have written proprietary software and would like to combine it with free software under the GPL version 3. Are you allowed to do this or what do you have to consider?

4. Why did the Free Software Foundation release the *GNU Affero General Public License* (GNU AGPL) as a supplement to the GNU GPL?

5. Name three examples of free software, which are also offered as “Business Edition”, i.e. in a chargeable version.

Summary

In this lesson you have learned:

- Similarities and differences between free and open source software (FLOSS)
- FLOSS licenses, their importance and problems
- Copyleft vs. permissive licences
- FLOSS business models

Answers to Guided Exercises

1. What are — in a nutshell — the “four freedoms” as defined by Richard Stallman and the Free Software Foundation?

freedom 0	run the software
freedom 1	study and modify the software (source code)
freedom 2	distribute the software
freedom 3	distribute the modified software

2. What does the abbreviation FLOSS stand for?

   Free/Libre Open Source Software

3. You have developed free software and want to ensure that the software itself, but also all future results based on it, remain free as well. Which license do you choose?

CC BY	
GPL version 3	X
2-Clause BSD License	
LGPL	

4. Which of the following licenses would you call permissive, which would you call copyleft?

Simplified BSD License	permissive
GPL version 3	copyleft
CC BY	permissive
CC BY-SA	copyleft

5. You have written a web application and published it under a free license. How can you earn money with your product? Name three possibilities.
   - Dual licensing, e.g. by offering a chargeable “Business Edition”
   - Offering hosting, service, and support
   - Developing proprietary extensions for customers

Answers to Explorational Exercises

1. Under which license (including version) are the following applications available?

Apache HTTP Server	Apache License 2.0
MySQL Community Server	GPL 2
Wikipedia articles (English)	Creative Commons Attribution Share-Alike license (CC-BY-SA)
Mozilla Firefox	Mozilla Public License 2.0
GIMP	GPL 3

2. You want to release your software under the GNU GPL v3. What steps should you follow?
   - If necessary, secure yourself against the employer with a copyright waiver, for example, so that you can specify the license.
   - Add a copyright notice to each file.
   - Add a file called `COPYING` with the full license text to your software.
   - Add a reference to the license in each file.
3. You have written proprietary software and would like to combine it with free software under the GPL version 3. Are you allowed to do this or what do you have to consider?

   The FAQs of the Free Software Foundation provide information here: Provided that your proprietary software and the free software remain separate from each other, the combination is possible. However, you have to make sure that this separation is technically guaranteed and recognizable for the users. If you integrate the free software in such a way that it becomes part of your product, you must also publish the product under the GPL according to the copyleft principle.

4. Why did the Free Software Foundation release the *GNU Affero General Public License* (GNU AGPL) as a supplement to the GNU GPL?

   The GNU AGPL closes a license gap that arises especially with free software hosted on a server: If a developer makes changes to the software, they is not obliged under the GPL to make these changes accessible, since they allow access to the program, but do not “redistribute” the program in the GPL sense. The GNU AGPL, on the other hand, stipulates that the software must be made available for download with all changes.

5. Name three examples of free software, which are also offered as “Business Edition”, e.g. in a

chargeable version.

MySQL, Zammad, Nextcloud

1.4 ICT Skills and Working in Linux

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 1.4

Weight

2

Key knowledge areas

- Desktop skills
- Getting to the command line
- Industry uses of Linux, cloud computing and virtualization

Partial list of the used files, terms and utilities

- Using a browser, privacy concerns, configuration options, searching the web and saving content
- Terminal and console
- Password issues
- Privacy issues and tools
- Use of common open source applications in presentations and projects

1.4 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	1 The Linux Community and a Career in Open Source
Objective:	1.4 ICT Skills and Working in Linux
Lesson:	1 of 1

Introduction

There was a time when working with Linux on the desktop was considered hard since the system lacked many of the more polished desktop applications and configuration tools that other operating systems had. Some of the reasons for that were that Linux was a lot younger than many other operating systems. That being said, it was easier to start by developing more essential command line applications and just leave the more complex graphical tools for later. In the beginning, since Linux was first targeted to more advanced users, that should not have been a problem. But those days are long gone. Today, Linux desktop environments are very mature, leaving nothing to be desired as regards to features and ease of use. Nevertheless, the command line is still considered a powerful tool used each and every day by advanced users. In this lesson we'll take a look at some of the basic desktop skills you will need in order to choose the best tool for the right job, including getting to the command line.

Linux User Interfaces

When using a Linux system, you either interact with a command line or with a graphical user interfaces. Both ways grant you access to numerous applications that support performing almost

any task with the computer. While objective 1.2 already introduced you to a series of commonly used applications, we will start this lesson with a closer look at desktop environments, ways to access the terminal and tools used for presentations and project management.

Desktop Environments

Linux has a modular approach where different parts of the system are developed by different projects and developers, each one filling a specific need or objective. Because of that, there are several options of desktop environments to choose from and together with package managers, the default desktop environment is one of the main differences among the many distributions out there. Unlike proprietary operating systems like Windows and macOS, where the users are restricted to the desktop environment that comes with their OS, there is the possibility to install multiple environments and pick the one that adapts the most to you and your needs.

Basically, there are two major desktop environments in the Linux world: *Gnome* and *KDE*. They are both very complete, with a large community behind them and aim for the same purpose but with slightly divergent approaches. In a nutshell, Gnome tries to follow the KISS ("keep it simple stupid") principle, with very streamlined and clean applications. On the other hand, KDE has another perspective with a larger selection of applications and giving the user the opportunity to change every configuration setting in the environment.

While Gnome applications are based on the GTK toolkit (written in the C language), KDE applications make use of the Qt library (written in C++). One of the most practical aspects of writing applications with the same graphical toolkit is that applications will tend to share a similar look and feel, which is responsible for giving the user a sense of unity during their experience. Another important characteristic is that having the same shared graphical library for many frequently used applications may save some memory space at the same time that it will speed up loading time once the library has been loaded for the first time.

Getting to the Command Line

For us, one of the most important applications is the graphical terminal emulator. Those are called terminal emulators because they really emulate, in a graphical environment, the old style serial terminals (often Teletype machines) that were in fact clients that used to be connected to a remote machine where the computing actually happened. Those machines were really simple computers with no graphics at all that were used on the first versions of Unix.

In Gnome, such an application is called *Gnome Terminal*, while in KDE it can be found as *Konsole*. But there are many other choices available, such as *Xterm*. Those applications are a way for us to have access to a command line environment in order to be able to interact with a shell.

So, you should look at the application menu of your distribution of choice for a terminal application. Besides any difference between them, every application will offer you what is necessary to gain confidence in using the command line.

Another way to get into the terminal is to use the virtual TTY. You can get into them by pressing `Ctrl` + `Alt` + `F#`. Read `F#` as one of the function keys from 1 to 7, for example. Probably, some of the initial combinations might be running your session manager or your graphical environment. The others will show a prompt asking for your login name like the one below:

```
Ubuntu 18.10 arrelia tty3
arrelia login:
```

`arrelia` in this case, is the hostname of the machine and `tty3` is the terminal available after using the key combination above, plus the `F3` key, like in `Ctrl` + `Alt` + `F3`.

After providing your login and password, you will finally get into a terminal, but there is no graphical environment in here, so, you won't be able to use the mouse or run graphical applications without first starting an X, or Wayland, session. But that's beyond the scope of this lesson.

Presentations and Projects

The most important tool for presentations on Linux is *LibreOffice Impress*. It's part of the open source office suite called *LibreOffice*. Think about LibreOffice as an open source replacement for the equivalent *Microsoft Office*. It can even open and save the PPT and PPTX files that are native to *Powerpoint*. But in spite of that, I really recommend you to use the native ODP Impress format. The ODP is part of the larger *Open Document Format*, which is a international standard for this kind of file. This is especially important if you want to keep your documents accessible for many years and worry less about compatibility problems. Because they are an open standard, it's possible for anyone to implement the format without paying any royalties or licenses. This also makes you free to try other presentations software that you may like better and take your files with you, as it's very likely they will be compatible with those newer softwares.

But if you prefer code over graphical interfaces, there are a few tools for you to choose. *Beamer* is a *LaTeX* class that can create slide presentations from LaTeX code. LaTeX itself is a typesetting system largely used for writing scientific documents at the academy, specially for its capacity to handle complex math symbols, which other softwares have difficulty to deal with. If you are at the university and need to deal with equations and other math related problems, Beamer can save you a lot of time.

The other option is *Reveal.js*, an awesome NPM package (NPM is the default NodeJS package

manager) which allows you to create beautiful presentations by using the web. So, if you can write HTML and CSS, Reveal.js will bring most of the JavaScript necessary to create pretty and interactive presentations that will adapt well on any resolution and screen size.

Lastly, if you want a replacement for *Microsoft Project*, you can try *GanttProject* or *ProjectLibre*. Both are very similar to their proprietary counterpart and compatible with Project files.

Industry Uses of Linux

Linux is heavily used among the software and Internet industries. Sites like W3Techs report that about 68% of the website servers on the Internet are powered by Unix and the biggest portion of those are known to be Linux.

This large adoption is given not only for the free nature of Linux (as both in free beer and in freedom of speech) but also for its stability, flexibility and performance. These characteristics allow vendors to offer their services with a lower cost and a greater scalability. A significant portion of Linux systems nowadays run in the cloud, either on a IaaS (Infrastructure as a service), PaaS (Platform as a Service) or SaaS (Software as a Service) model.

IaaS is a way to share the resources of a large server by offering them access to virtual machines that are, in fact, multiple operating systems running as guests on a host machine over an important piece of software that is called a *hypervisor*. The hypervisor is responsible for making it possible for these guest OSs to run by segregating and managing the resources available on the host machine to those guests. That's what we call *virtualization*. In the IaaS model, you pay only for the fraction of resources your infrastructure uses.

Linux has three well know open source hypervisors: *Xen*, *KVM* and *VirtualBox*. Xen is probably the oldest of them. KVM ran out Xen as the most prominent Linux Hypervisor. It has its development sponsored by RedHat and it is used by them and other players, both in public cloud services and in private cloud setups. VirtualBox belongs to Oracle since its acquisition of Sun Microsystems and is usually used by end users because of its easiness of use and administration.

PaaS and SaaS, on the other hand, build up on the IaaS model, both technically and conceptually. In PaaS instead of a virtual machine, the users have access to a platform where it will be possible to deploy and run their application. The goal here is to ease the burden of dealing with system administration tasks and operating systems updates. Heroku is a common PaaS example where program code can just be run without taking care of the underlying containers and virtual machines.

Lastly, SaaS is the model where you usually pay for a subscription in order to just use a software without worrying about anything else. *Dropbox* and *Salesforce* are two good examples of SaaS.

Most of these services are accessed through a web browser.

A project like *OpenStack* is a collection of open source software that can make use of different hypervisors and other tools in order to offer a complete IaaS cloud environment on premise, by leveraging the power of computer cluster on your own datacenter. However, the setup of such infrastructure is not trivial.

Privacy Issues when using the Internet

The web browser is a fundamental piece of software on any desktop these days, but some people still lack the knowledge to use it securely. While more and more services are accessed through a web browser, almost all actions done through a browser are tracked and analyzed by various parties. Securing access to internet services and preventing tracking is an important aspect of using the internet in a safe manner.

Cookie Tracking

Let's assume you have browsed an e-commerce website, selected a product you wanted and placed that in the shopping cart. But at the last second, you have decided to give it a second thought and think a little longer if you really needed that. After a while, you start seeing ads of that same product following you around the web. When clicking on the ads, you are immediately sent to the product page of that store again. It's not uncommon that the products you placed in the shopping cart are still there, just waiting for you to decide to check them out. Have you ever wondered how they do that? How they show you the right ad at another web page? The answer for these questions is called *cookie tracking*.

Cookies are small files a website can save on your computer in order to store and retrieve some kind of information that can be useful for your navigation. They have been in use for many years and are one of the oldest ways to store data on the client side. One good example of their use are unique shopping card IDs. That way, if you ever come back to the same website in a few days, the store can remember you the products you've placed in your cart during your last visit and save you the time to find them again.

That's usually okay, since the website is offering you a useful feature and not sharing any data with third parties. But what about the ads that are shown to you while you surf on other web pages? That's where the ad networks come in. Ad networks are companies that offer ads for e-commerce sites like the one in our example on one side, and monetization for websites, on the other side. Content creators like bloggers, for example, can make some space available for those ad networks on their blog, in exchange for a commission related to the sales generated by that ad.

But how do they know what product to show you? They usually do that by saving also a cookie

from the ad network at the moment you visited or searched for a certain product on the e-commerce website. By doing that, the network is able to retrieve the information on that cookie wherever the network has ads, making the correlation with the products you were interested. This is usually one of the most common ways to track someone over the Internet. The example we gave above makes use of an e-commerce to make things more tangible, but social media platforms do the same with their "Like" or "Share" buttons and their social login.

One way you can get rid of that is by not allowing third party websites to store cookies on your browser. This way, only the website you visit can store their cookies. But be aware that some "legitimate" features may not work well if you do that, because many sites today rely on third party services to work. So, you can look for a cookie manager at your browser's add-on repository in order to have a fine-grained control of which cookies are being stored on your machine.

Do Not Track (DNT)

Another common misconception is related to a certain browser configuration better known as DNT. That's an acronym for "Do Not Track" and it can be turned on basically on any current browser. Similarly to the private mode, it's not hard to find people that believe they will not be tracked if they have this configuration on. Unfortunately, that's not always true. Currently, DNT is just a way for you to tell the websites you visit that you do not want them to track you. But, in fact, they are the ones who will decide if they will respect your choice or not. In other words, DNT is a way to opt-out from website tracking, but there is no guarantee on that choice.

Technically, this is done by simply sending an extra flag on the header of the HTTP request protocol (`DNT: 1`) upon requesting data from a web server. If you want to know more about this topic, the website https://allaboutdnt.com is good starting point.

"Private" Windows

You might have noticed the quotes in the heading above. This is because those windows are not as private as most people think. The names may vary but they can be called "private mode", "incognito" or "anonymous" tab, depending on which browser you are using.

In Firefox, you can easily use it by pressing `Ctrl` + `Shift` + `P` keys. In Chrome, just press `Ctrl` + `Shift` + `N`. What it actually does is open a brand new session, which usually doesn't share any configuration or data from your standard profile. When you close the private window, the browser will automatically delete all the data generated by that session, leaving no trace on the computer used. This means that no personal data, like history, passwords or cookies are stored on that computer.

Thus, many people misunderstand this concept by believing that they can browse anonymous on the Internet, which is not completely true. One thing that the privacy or incognito mode does is

avoid what we call cookie tracking. When you visit a website, it can store a small file on your computer which may contain an ID that can be used to track you. Unless you configure your browser to not accept third-party cookies, ad networks or other companies can store and retrieve that ID and actually track your browsing across websites. But since the cookies stored on a private mode session are deleted right after you close that session, that information is forever lost.

Besides that, websites and other peers on the Internet can still use plenty other techniques in order to track you. So, private mode brings you some level of anonymity but it's completely private only on the computer you are using. If you are accessing your email account or banking website from a public computer, like in an airport or a hotel, you should definitely access those using your browser's private mode. In other situations, there can be benefits but you should know exactly what risks you are avoiding and which ones have no effect. Whenever you use a public accessible computer, be aware that other security threats such as malware or key loggers might exist. Be careful whenever you enter personal information, including usernames and passwords, on such computers or when you download or copy confidential data.

Choosing the Right Password

One of the most difficult situations any user faces is choosing a secure password for the services they make use of. You have certainly heard before that you should not use common combinations like `qwerty`, `123456` or `654321`, nor easily guessable numbers like your (or a relative's) birthday or zip code. The reason for that is because those are all very obvious combinations and the first attempts an invader will try in order to gain access to your account.

There are known techniques for creating a safe password. One of the most famous is making up a sentence which reminds you of that service and picking the first letters of each word. Let's assume I want to create a good password for Facebook, for example. In this case, I could come up with a sentence like "I would be happy if I had a 1000 friends like Mike". Pick the first letter of each word and the final password would be `IwbhiIha1000flM`. This would result in a 15 characters password which is long enough to be hard to guess and easy to remember at the same time (as long as I can remember the sentence and the "algorithm" for retrieving the password).

Sentences are usually easier to remember than the passwords but even this method has its limitations. We have to create passwords for so many services nowadays and as we use them with different frequencies, it will eventually be very difficult to remember all the sentences at the time we need them. So what can we do? You may answer that the wisest thing to do in this case is creating a couple good passwords and reuse them on similar services, right?

Unfortunately, that's also not a good idea. You probably also heard you should not reuse the same password among different services. The problem of doing such a thing is that a specific service may leak your password (yes, it happens all the time) and any person who have access to it will

try to use the same email and password combination on other popular services on the Internet in hope you have done exactly that: recycled passwords. And guess what? In case they are right you will end up having a problem not only on just one service but on several of them. And believe me, we tend to think it's not going to happen to us until it's too late.

So, what can we do in order to protect ourselves? One of the most secure approaches available today is using what is called a *password manager*. Password managers are a piece of software that will essentially store all your passwords and usernames in an encrypted format which can be decrypted by a master password. This way you only need to remember one good password since the manager will keep all the others safe for you.

KeePass is one of the most famous and feature rich open source password managers available. It will store your passwords in an encrypted file within your file system. The fact it's open source is an important issue for this kind of software since it guarantees they will not make any use of your data because any developer can audit the code and know exactly how it works. This brings a level of transparency that's impossible to reach with proprietary code. KeePass has ports for most operating systems, including Windows, Linux and macOS; as well as mobile ones like iOS and Android. It also includes a plugin system that is able to extend it's functionality far beyond the defaults.

Bitwarden is another open source solution that has a similar approach but instead of storing your data in a file, it will make use of a cloud server. This way, it's easier to keep all your devices synchronized and your passwords easily accessible through the web. *Bitwarden* is one of the few projects that will make not only the clients, but also the cloud server available as an open source software. This means you can host your own version of Bitwarden and make it available to anyone, like your family or your company employees. This will give you flexibility but also total control over how their passwords are stored and used.

One of the most important things to keep in mind when using a password manager is creating a random password for each different service since you will not need to remind them anyway. It would be worthless if you use a password manager to store recycled or easily guessable passwords. Thus, most of them will offer you a random password generator you can use to create those for you.

Encryption

Whenever data is transferred or stored, precautions need to be taken to ensure that third parties may not access the data. Data transferred over the internet passes by a series of routers and networks where third parties might be able to access the network traffic. Likewise, data stored on physical media might be read by anyone who comes into possession of that media. To avoid this kind of access, confidential information should be encrypted before it leaves a computing device.

TLS

Transport Layer Security (TLS) is a protocol to offer security over network connections by making use of cryptography. TLS is the successor of the *Secure Sockets Layer* (SSL) which has been deprecated because of serious flaws. TLS has also evolved a couple of times in order to adapt itself and become more secure, thus it's current version is 1.3. It can provide both privacy, and authenticity by making use of what is called symmetric and public-key cryptography. By saying that, we mean that once in use, you can be sure that nobody will be able to eavesdrop or alter your communication with that server during that session.

The most important lesson here is recognizing that a website is trustworthy. You should look for the "lock" symbol on the browser's address bar. If you desire, you can click on it to inspect the certificate that plays an important role in the HTTPS protocol.

TLS is what is used on the HTTPS protocol (*HTTP over TLS*) in order to make it possible to send sensitive data (like your credit card number) through the web. Explaining how TLS works goes way beyond the purpose of this article, but you can find more information on the Wikipedia and at the Mozilla wiki.

File and E-mail Encryption With GnuPG

There are plenty of tools for securing emails but one of the most important of them is certainly *GnuPG*. GnuPG stands for *GNU Privacy Guard* and it is an open source implementation of *OpenPGP* which is an international standard codified within RFC 4880.

GnuPG can be used to sign, encrypt, and decrypt texts, e-mails, files, directories, and even whole disk partitions. It works with public-key cryptography and is widely available. In a nutshell GnuPG creates a pair of files which contain your public and private keys. As the name implies, the public key can be available to anyone and the private key needs to be kept in secret. People will use your public key to encrypt data which only your private key will be able to decrypt.

You can also use your private key to sign any file or e-mail which can be validated against the corresponding public key. This digital signage works analogous to the real world signature. As long as you are the only one who posses your private key, the receiver can be sure that it was you who have authored it. By making use of the cryptographic hash functionality GnuPG will also guarantee no changes have been made after the signature because any changes to the content would invalidate the signature.

GnuPG is a very powerful tool and, in a certain extent, also a complex one. You can find more information on its website and on Archlinux wiki (Archlinux wiki is a very good source of information, even though you don't use Archlinux).

Disk Encryption

A good way to secure your data is to encrypt your whole disk or partition. There are many open source softwares you can use to achieve such a purpose. How they work and what level of encryption they offer also varies significantly. There are basic two methods available: *stacked* and *block* device encryption.

Stacked filesystem solutions are implemented on top of existing filesystem. When using this method, the files and directories are encrypted before being stored on the filesystem and decrypted after reading them. This means the files are stored on the host filesystem in an encrypted form (meaning that their contents, and usually also their file/folder names, are replaced by random-looking data), but other than that, they still exist in that filesystem as they would without encryption, as normal files, symlinks, hardlinks, etc.

On the other hand, block device encryption happens below the filesystem layer, making sure everything that is written to a block device is encrypted. If you look to the block while it's offline, it will look like a large section of random data and you won't even be able to tell what type of filesystem is there without decrypting it first. This means you can't tell what is a file or directory; how big they are and what kind of data it is, because metadata, directory structure and permissions are also encrypted.

Both methods have their own pros and cons. Among all the options available, you should take a look at *dm-crypt*, which is the de-facto standard for block encryption for Linux systems, since it's native in the kernel. It can be used with *LUKS* (*Linux Unified Key Setup*) extension, which is a specification that implements a platform-independent standard for use with various tools.

If you want to try a stackable method, you should take a look at *EncFS*, which is probably the easiest way to secure data on Linux because it does not require root privileges to implement and it can work on an existing filesystem without modifications.

Finally, if you need to access data on various platforms, check out Veracrypt. It is the successor of a Truecrypt and allows the creation of encrypted media and files, which can be used on Linux as well as on macOS and Windows.

Guided Exercises

1. You should use a “private window” in your browser if you want:

To browse completely anonymous on the Internet	
To leave no trace on the computer you’re using	
To activate TLS to avoid cookie tracking	
In order to use DNT	
To use cryptography during data transmission	

2. What is OpenStack?

A project that allows the creation of private IaaS	
A project that allows the creation of private PaaS	
A project that allows the creation of private SaaS	
A hypervisor	
An open source password manager	

3. Which of the below options are valid disk encryption softwares?

RevealJS, EncFS and dm-crypt	
dm-crypt and KeePass	
EncFS and Bitwarden	
EncFS and dm-crypt	
TLS and dm-crypt	

4. Select true or false for dm-crypt device encryption:

Files are encrypted before being written to the disk	

The entire filesystem is an encrypted blob	
Only files and directories are encrypted, not symlinks	
Don’t require root access	
Is a block device encryption	

5. Beamer is:

An encryption mechanism	
A hypervisor	
A virtualization software	
An OpenStack component	
A LaTeX presentation tool	

Explorational Exercises

1. Most distributions come with Firefox installed by default (if yours doesn't, you will have to install it first). We are going to install a Firefox extension called *Lightbeam*. You can do that by either pressing `Ctrl` + `Shift` + `A` and searching for "Lightbeam" on the search field that will be shown on the tab opened, or by visiting the extension page with Firefox and clicking on the "Install" button: https://addons.mozilla.org/en-GB/firefox/addon/lightbeam-3-0/. After doing this, start the extension by clicking on its icon and start visiting some webpages on other tabs to see what happens.
2. What is the most important thing when using a password manager?

3. Use your web browser to navigate to https://haveibeenpwned.com/. Find out the purpose of the website and check if your email address was included in some data leaks.

Summary

The terminal is a powerful way to interact with the system and there are lots of useful and very mature tools to use in this kind of environment. You can get to the terminal by looking for a graphical one at your desktop environment menu or pressing `Ctrl` + `Alt` + `F#`.

Linux is largely used in the tech industry to offer IaaS, PaaS and SaaS services. There are three main hypervisors which play an important role in supporting those: Xen, KVM and Virtualbox.

The browser is an essential piece of software in computing nowadays, but it's necessary to understand some things to use it with safety. DNT is just a way to tell the website that you do not want to be tracked, but there is no guarantee on that. Private windows are private only to the computer you're using but this can allow you to escape from cookie tracking exactly because of that.

TLS is able to encrypt your communication on the Internet, but you have to be able to recognize when it's in use. Using strong passwords is also very important to keep you safe, so the best idea is to delegate that responsibility to a password manager and allow the software to create random passwords to every site you log into.

Another way to secure your communication is to sign and encrypt your files folders and emails with GnuPG. dm-crypt and EncFS are two alternatives to encrypt whole disks or partitions that use respectively block and stack encryption methods.

Finally, LibreOffice Impress is a very complete open source alternative to Microsoft Powerpoint but there are Beamer and RevealJS if you prefer to create presentations using code instead of GUIs. ProjectLibre and GanttProject can be the right choice if you need a Microsoft Project replacement.

Answers to Guided Exercises

1. You should use a “private window” in your browser if you want:

To browse completely anonymous on the Internet	
To leave no trace on the computer you’re using	X
To activate TLS to avoid cookie tracking	
In order to use DNT	
To use cryptography during data transmission	

2. What is OpenStack?

A project that allows the creation of private IaaS	X
A project that allows the creation of PaaS	
A project that allows the creation of SaaS	
A hypervisor	
An open source password manager	

3. Which of the below options are valid disk encryption softwares?

RevealJS, EncFS and dm-crypt	
dm-crypt and KeePass	
EncFS and Bitwarden	
EncFS and dm-crypt	X
TLS and dm-crypt	

4. Select true or false for dm-crypt device encryption:

Files are encrypted before being written to the disk	T
The entire filesystem is an encrypted blob	T

Only files and directories are encrypted, not symlinks	F
Don’t require root access	F
Is a block device encryption	T

5. Beamer is:

An encryption mechanism	
A hypervisor	
A virtualization software	
An OpenStack component	
A LaTeX presentation tool	X

Answers to Explorational Exercises

1. Most distributions come with Firefox installed by default (if yours doesn't, you will have to install it first). We are going to install a Firefox extension called *Lightbeam*. You can do that by either pressing `Ctrl` + `Shift` + `A` and searching for "Lightbeam" on the search field that will be shown on the tab opened, or by visiting the extension page with Firefox and clicking on the "Install" button: https://addons.mozilla.org/en-US/firefox/addon/lightbeam. After doing this, start the extension by clicking on it's icon and start visiting some webpages on other tabs to see what happens.

   Remember those cookies we said that can share your data with different services when you visit a website? That's exactly what this extension is going to show you. Lightbeam is a Mozilla experiment that tries to reveal the first and third party sites you interact with upon visiting a single URL. This content is usually not visible to the average user and it can show that sometimes a single website is able to interact with a dozen or more services.

2. What is the most important thing when using a password manager?

   When using a password manager, the most important thing to have in mind is memorizing your master password and use a unique random password for each different service.

3. Use your web browser to navigate to https://haveibeenpwned.com/. Find out the purpose of the website and check if your email address was included in some data leaks.

   The website maintains a database of login information whose passwords were affected by a password leak. It allows searching for an email address and shows if that email address was included in a public database of stolen credentials. Chances are that your email address is affected by one or the other leak, too. If that is the case, make sure you have updated your passwords recently. If you don't already use a password manager, take a look at the ones recommended in this lesson.

Linux Professional Institute

Topic 2: Finding Your Way on a Linux System

2.1 Command Line Basics

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 2.1

Weight

3

Key knowledge areas

- Basic shell
- Command line syntax
- Variables
- Quoting

Partial list of the used files, terms and utilities

- Bash
- `echo`
- `history`
- `PATH` environment variable
- `export`
- `type`

2.1 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	2 Finding Your Way on a Linux System
Objective:	2.1 Command Line Basics
Lesson:	1 of 2

Introduction

Modern Linux distributions have a wide range of graphical user interfaces but an administrator will always need to know how to work with the command line, or *shell* as it is called. The shell is a program that enables text based communication between the operating system and the user. It is usually a text mode program that reads the user's input and interprets it as commands to the system.

There are several different shells on Linux, these are just a few:

- Bourne-again shell (Bash)
- C shell (csh or tcsh, the enhanced csh)
- Korn shell (ksh)
- Z shell (zsh)

On Linux the most common one is the Bash shell. This is also the one that will be used in examples or exercises here.

When using an interactive shell, the user inputs commands at a so-called prompt. For each Linux

distribution, the default prompt may look a little different, but it usually follows this structure:

```
username@hostname current_directory shell_type
```

On Ubuntu or Debian GNU/Linux, the prompt for a regular user will likely look like this:

```
carol@mycomputer:~$
```

The superuser's prompt will look like this:

```
root@mycomputer:~#
```

On CentOS or Red Hat Linux, the prompt for a regular user will instead look like this:

```
[dave@mycomputer ~]$
```

And the superuser's prompt will look like this:

```
[root@mycomputer ~]#
```

Let's explain each component of the structure:

`username`

Name of the user that runs the shell

`hostname`

Name of the host on which the shell runs. There is also a command `hostname`, with which you can show or set the system's host name.

`current_directory`

The directory that the shell is currently in. A `~` means that the shell is in the current user's home directory.

`shell_type`

`$` indicates the shell is run by a regular user.

`#` indicates the shell is run by the superuser `root`.

As we do not need any special privileges, we will use an unprivileged prompt in the following examples. For brevity, we will just use the `$` as prompt.

Command Line Structure

Most commands at the command line follow the same basic structure:

```
command  [option(s)/parameter(s)...]  [argument(s)...]
```

Take the following command as an example:

```
$ ls -l /home
```

Let's explain the purpose of each component:

Command

Program that the user will run – `ls` in the above example.

Option(s)/Parameter(s)

A "switch" that modifies the behavior of the command in some way, such as `-l` in the above example. Options can be accessed in a short and in a long form. For example, `-l` is identical to `--format=long`.

Multiple options can be combined as well and for the short form, the letters can usually be typed together. For example, the following commands all do the same:

```
$ ls -al
$ ls -a -l
$ ls --all --format=long
```

Argument(s)

Additional data that is required by the program, like a filename or path, such as `/home` in the above example.

The only mandatory part of this structure is the command itself. In general, all other elements are optional, but a program may require certain options, parameters or arguments to be specified.

NOTE

Most commands display a short overview of available options when they are run with the `--help` parameter. We will learn additional ways to learn more about

> Linux commands soon.

Command Behavior Types

The shell supports two types of commands:

Internal

These commands are part of the shell itself and are not separate programs. There are around 30 such commands. Their main purpose is executing tasks inside the shell (e.g. `cd`, `set`, `export`).

External

These commands reside in individual files. These files are usually binary programs or scripts. When a command which is not a shell builtin is run, the shell uses the `PATH` variable to search for an executable file with same name as the command. In addition to programs which are installed with the distribution's package manager, users can create their own external commands as well.

The command `type` shows what type a specific command is:

```
$ type echo
echo is a shell builtin
$ type man
man is /usr/bin/man
```

Quoting

As a Linux user, you will have to create or manipulate files or variables in various ways. This is easy when working with short filenames and single values, but it becomes more complicated when, for example, spaces, special characters and variables are involved. Shells provide a feature called quoting which encapsulates such data using various kinds of quotes (" ", ' '). In Bash, there are three types of quotes:

- Double quotes
- Single quotes
- Escape characters

For example, the following commands do not act in the same way due to quoting:

```
$ TWOWORDS="two words"
$ touch $TWOWORDS
$ ls -l
-rw-r--r-- 1 carol carol     0 Mar 10 14:56 two
-rw-r--r-- 1 carol carol     0 Mar 10 14:56 words
$ touch "$TWOWORDS"
$ ls -l
-rw-r--r-- 1 carol carol     0 Mar 10 14:56  two
-rw-r--r-- 1 carol carol     0 Mar 10 14:58 'two words'
-rw-r--r-- 1 carol carol     0 Mar 10 14:56  words
$ touch '$TWOWORDS'
$ ls -l
-rw-r--r-- 1 carol carol     0 Mar 10 15:00 '$TWOWORDS'
-rw-r--r-- 1 carol carol     0 Mar 10 14:56  two
-rw-r--r-- 1 carol carol     0 Mar 10 14:58 'two words'
-rw-r--r-- 1 carol carol     0 Mar 10 14:56  words
```

NOTE The line with `TWOWORDS=` is a Bash variable that we have created ourselves. We will introduce variables later. This is just meant to show you how quoting affects the output of variables.

Double Quotes

Double quotes tell the shell to take the text in between the quote marks ("...") as regular characters. All special characters lose their meaning, except the `$` (dollar sign), `\` (backslash) and `` ` `` (backquote). This means that variables, command substitution and arithmetic functions can still be used.

For example, the substitution of the `$USER` variable is not affected by the double quotes:

```
$ echo I am $USER
I am tom
$ echo "I am $USER"
I am tom
```

A space character, on the other hand, loses its meaning as an argument separator:

```
$ touch new file
$ ls -l
-rw-rw-r-- 1 tom students 0 Oct 8 15:18 file
-rw-rw-r-- 1 tom students 0 Oct 8 15:18 new
```

```
$ touch "new file"
$ ls -l
-rw-rw-r-- 1 tom students 0 Oct 8 15:19 new file
```

As you can see, in the first example, the `touch` command creates two individual files, the command interprets the two strings as individual arguments. In the second example, the command interprets both strings as one argument, therefore it only creates one file. It is, however, best practice to avoid the space character in filenames. Instead, an underscore (`_`) or a dot (`.`) could be used.

Single Quotes

Single quotes don't have the exceptions of the double quotes. They revoke any special meaning from each character. Let's take one of the first examples from above:

```
$ echo I am $USER
I am tom
```

When applying the single quotes you see a different result:

```
$ echo 'I am $USER'
I am $USER
```

The command now displays the exact string without substituting the variable.

Escape Characters

We can use *escape characters* to remove special meanings of characters from Bash. Going back to the `$USER` environment variable:

```
$ echo $USER
carol
```

We see that by default, the contents of the variable are displayed in the terminal. However, if we were to precede the dollar sign with a backslash character (`\`) then the special meaning of the dollar sign will be negated. This in turn will not let Bash expand the variable's value to the username of the person running the command, but will instead interpret the variable name literally:

```
$ echo \$USER
$USER
```

If you recall, we can get similar results to this using the single quote, which prints the literal contents of whatever is between the single quotes. However the escape character works differently by instructing Bash to ignore whatever special meaning the character it precedes may possess.

Guided Exercises

1. Split the lines below into the components of command, option(s)/parameter(s) and argument(s):
   - Example: `cat -n /etc/passwd`

Command:	`cat`
Option:	`-n`
Argument:	`/etc/passwd`

   - `ls -l /etc`

Command:	
Option:	
Argument:	

   - `ls -l -a`

Command:	
Option:	
Argument:	

   - `cd /home/user`

Command:	
Option:	
Argument:	

2. Find what type the following commands are:

   Example:

`pwd`	Shell builtin
`mv`	External command

`cd`	

`cat`	
`exit`	

3. Resolve the following commands that use quotes:

   Example:

`echo "$HOME is my home directory"`	`echo /home/user is my home directory`

`touch "$USER"`	
`touch 'touch'`	

Explorational Exercises

1. With one command and using brace expansion in Bash (review the man page for Bash), create 5 files numbered 1 to 5 with the prefix `game` (`game1`, `game2`, ...).

2. Delete all 5 files that you just created with just one command, using a different special character (review *Pathname Expansion* in the Bash man pages).

3. Is there any other way to make two commands interact with each other? What are those?

Summary

In this lab you learned:

- Concepts of the Linux shell
- What is the Bash shell
- The structure of the command line
- An introduction to quoting

Commands used in the exercises:

`bash`
: The most popular shell on Linux computers.

`echo`
: Output text on the terminal.

`ls`
: List the contents of a directory.

`type`
: Show how a specific command is executed.

`touch`
: Create an empty file or update an existing file's modification date.

`hostname`
: Show or change a system's hostname.

Answers to Guided Exercises

1. Split the lines below into the components of command, option(s)/parameter(s) and argument(s):
   - `ls -l /etc`

Command:	`ls`
Option:	`-l`
Argument:	`/etc`

   - `ls -l -a`

Command:	`ls`
Option:	`-l -a`
Argument:	

   - `cd /home/user`

Command:	`cd`
Option:	
Argument:	`/home/user`

2. Find what type the following commands are:

`cd`	Shell builtin
`cat`	External command
`exit`	Shell builtin

3. Resolve the following commands that use quotes:

`touch "$USER"`	`tom`
`touch 'touch'`	Creates a file named `touch`

Answers to Explorational Exercises

1. With one command and using brace expansion in Bash (review the man page for Bash), create 5 files numbered 1 to 5 with the prefix `game` (`game1`, `game2`, ...).

   Ranges can be used to express the numbers from 1 to 5 within one command:

   ```
   $ touch game{1..5}
   $ ls
   game1  game2  game3  game4  game5
   ```

2. Delete all 5 files that you just created with just one command, using a different special character (review *Pathname Expansion* in the Bash man pages).

   Since all files start with `game` and end in a single character (a number from 1 to 5 in this case), `?` can be used as a special character for the last character in the filename:

   ```
   $ rm game?
   ```

3. Is there any other way to make two commands interact with each other? What are those?

   Yes, one command could, for example, write data to a file which is then processed by another command. Linux can also collect the output of one command and use it as input for another command. This is called *piping* and we will learn more about it in a future lesson.

2.1 Lesson 2

Certificate:	Linux Essentials
Version:	1.6
Topic:	2 Finding Your Way on a Linux System
Objective:	2.1 Command Line Basics
Lesson:	2 of 2

Introduction

All shells manage a set of status information throughout the shell sessions. This runtime information may change during the session and influences how the shell behaves. This data is also used by programs to determinate aspects of the system's configuration. Most of this data is stored in so-called *variables*, which we will cover in this lesson.

Variables

Variables are pieces of storage for data, such as text or numbers. Once set, a variable's value can be accessed at a later time. Variables have a name which allows accessing a specific variable, even when the variable's content changes. They are a very common tool in most programming languages.

In most Linux shells, there are two types of variables:

Local variables

These variables are available to the current shell process only. If you create a local variable and then start another program from this shell, the variable is not accessible to that program

anymore. Because they are not inherited by sub processes, these variables are called *local variables*.

Environment variables

These variables are available both in a specific shell session and in sub processes spawned from that shell session. These variables can be used to pass configuration data to commands which are run. Because these programs can access these variables, they are called *environment variables*. The majority of the environment variables are in capital letters (e.g. `PATH`, `DATE`, `USER`). A set of default environment variables provide, for example, information about the user's home directory or terminal type. Sometimes the complete set of all environment variables is referred to as the *environment*.

NOTE Variables are not persistent. When the shell in which they were set is closed, all variables and their contents are lost. Most shells provide configuration files that contain variables which are set whenever a new shell is started. Variables that should be set permanently must be added to one of these configuration files.

Manipulating Variables

As a system administrator, you will need to create, modify or remove both local and environment variables.

Working with Local Variables

You can set up a local variable by using the `=` (equal) operator. A simple assignment will create a local variable:

```
$ greeting=hello
```

NOTE Don't put any space before or after the `=` operator.

You can display any variable using the `echo` command. The command usually displays the text in the argument section:

```
$ echo greeting
greeting
```

In order to access the value of the variable you will need to use `$` (dollar sign) in front of the variable's name.

```
$ echo $greeting
hello
```

As it can be seen, the variable has been created. Now open another shell and try to display the contents of the variable created.

```
$ echo $greeting
```

Nothing is displayed. This illustrates, that variables always exist in a specific shell only.

To verify that the variable is actually a local variable, try to spawn a new process and check if this process can access the variable. We can do so by starting another shell and let this shell run the `echo` command. As the new shell is run in a new process, it won't inherit local variables from its parent process:

```
$ echo $greeting world
hello world
$ bash -c 'echo $greeting world'
world
```

NOTE Make sure to use single quotes in the example above.

In order to remove a variable, you will need to use the command `unset`:

```
$ echo $greeting
hey
$ unset greeting
$ echo $greeting
```

NOTE `unset` requires the name of the variable as an argument. Therefore you may not add `$` to the name, as this would resolve the variable and pass the variable's value to `unset` instead of the variable's name.

Working with Global Variables

To make a variable available to subprocesses, turn it from a local into an environment variable. This is done by the command `export`. When it is invoked with the variable name, this variable is added to the shell's environment:

```
$ greeting=hello
$ export greeting
```

NOTE Again, make sure to not use `$` when running `export` as you want to pass the name of the variable instead of its contents.

An easier way to create the environment variable is to combine both of the above methods, by assigning the variable value in the argument part of the command.

```
$ export greeting=hey
```

Let's check again if the variable is accessible to subprocesses:

```
$ export greeting=hey
$ echo $greeting world
hey world
$ bash -c 'echo $greeting world'
hey world
```

Another way to use environment variables is to use them in front of commands. We can test this with the environment variable `TZ` which holds the timezone. This variable is used by the command `date` to determine which timezone's time to display:

```
$ TZ=EST date
Thu 31 Jan 10:07:35 EST 2019
$ TZ=GMT date
Thu 31 Jan 15:07:35 GMT 2019
```

You can display all environment variables using the `env` command.

The PATH Variable

The `PATH` variable is one of the most important environment variables in a Linux system. It stores a list of directories, separated by a colon, that contain executable programs eligible as commands from the Linux shell.

```
$ echo $PATH
/home/user/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

To append a new directory to the variable, you will need to use the colon sign (`:`).

```
$ PATH=$PATH:new_directory
```

Here an example:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
$ PATH=$PATH:/home/user/bin
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/home/user/bin
```

As you see, `$PATH` is used in the new value assigned to `PATH`. This variable is resolved during the command execution and makes sure that the original content of the variable is preserved. Of course, you can use other variables in the assignment as well:

```
$ mybin=/opt/bin
$ PATH=$PATH:$mybin
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/home/user/bin:/opt/bin
```

The `PATH` variable needs to be handled with caution, as it is crucial for working on the command line. Let's consider the following `PATH` variable:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

To find out how the shell invokes a specific command, `which` can be run with the command's name as argument. We can, for example, try to find out where `nano` is stored:

```
$ which nano
/usr/bin/nano
```

As it can be seen, the `nano` executable is located within the `/usr/bin` directory. Let's remove the directory from the variable and check to see if the command still works:

```
$ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/sbin:/bin:/usr/games
$ echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/sbin:/bin:/usr/games
```

Let's look up the `nano` command again:

```
$ which nano
which: no nano in (/usr/local/sbin:/usr/local/bin:/usr/sbin:/sbin:/bin:/usr/games)
```

As it can be seen, the command is not found, therefore not executed. The error message also explains the reason why the command was not found and in what locations it was searched.

Let's add back the directories and try running the command again.

```
$ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
$ which nano
/usr/bin/nano
```

Now our command works again.

TIP The order of elements in `PATH` also defines the lookup order. The first matching executable found while going through the paths is executed.

Guided Exercises

1. Create a local variable `number`.

2. Create an environment variable `ORDER`, using one of the two above methods.

3. Display both the variable names and their contents.

4. What are the scopes of the previously created variables?

Explorational Exercises

1. Create a local variable `nr_files` and assign the number of lines found in the `/etc/passwd` file. Hint: Look into the command `wc` and command substitution and don't forget about quotation marks.

2. Create an environment variable `ME`. Assign the `USER` variable's value to it.

3. Append the `HOME` variable's value to `ME`, having the `:` delimiter. Display the contents of the `ME` variable.

4. Using the date example above, create a variable called `today` and assign the date for one of the time zones.

5. Create another variable called `today1` and assign the system's date to it.

Summary

In this lab you learned:

- Types of variables
- How to create variables
- How to manipulate variables

Commands used in the exercises:

`env`
: Display the current environment.

`echo`
: Output text.

`export`
: Make local variables available to subprocesses.

`unset`
: Remove a variable.

Answers to Guided Exercises

1. Create a local variable `number`.

```
$ number=5
```

2. Create an environment variable `ORDER`, using one of the two above methods.

```
$ export ORDER=desc
```

3. Display both the variable names and their contents.

```
$ echo number
number
$ echo ORDER
ORDER
$ echo $number
5
$ echo $ORDER
desc
```

4. What are the scopes of the previously created variables?
   - The scope of the local variable `number` is the current shell only.
   - The scope of the environment variable `ORDER` is the current shell and all the subshells generated by it.

Answers to Explorational Exercises

1. Create a local variable `nr_files` and assign the number of lines found in the `/etc/passwd` file. Hint: Look into the command `wc` and command substitution and don't forget about quotation marks.

```
$ nr_files=`wc -l /etc/passwd`
```

2. Create an environment variable `ME`. Assign the `USER` variable's value.

```
$ export ME=$USER
```

3. Append the `HOME` variable value to `ME`, having the `:` delimiter. Display the contents of the `ME` variable.

```
$ ME=$ME:$HOME
$ echo $ME
user:/home/user
```

4. Using the date example above, create a variable called `today` and assign the date for one of the time zones.

   The following use the GMT and EST time zones as an example, but any time zone selection is valid.

```
$ today=$(TZ=GMT date)
$ echo $today
Thu 31 Jan 15:07:35 GMT 2019
```

   or

```
$ today=$(TZ=EST date)
$ echo $today
Thu 31 Jan 10:07:35 EST 2019
```

5. Create another variable called `today1` and assign the system's date to it.

   Assuming that you are in GMT:

```
$ today1=$(date)
$ echo $today1
Thu 31 Jan 10:07:35 EST 2019
```

2.2 Using the Command Line to Get Help

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 2.2

Weight

2

Key knowledge areas

- Man pages
- Info pages

Partial list of the used files, terms and utilities

- `man`
- `info`
- `/usr/share/doc/`
- `locate`

2.2 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	2 Finding Your Way on a Linux System
Objective:	2.2 Using the Command Line to Get Help
Lesson:	1 of 1

Introduction

The command line is a very complex tool. Each command has its own unique options, therefore documentation is key when working with a Linux system. Besides the `/usr/share/doc/` directory, which stores most of the documentation, various other tools provide information on using Linux commands. This chapter focuses on methods to access that documentation, with the purpose of getting help.

There are a multitude of methods to get help within the Linux command line. `man`, `help` and `info` are just a few of them. For Linux Essentials, we will be focusing on `man` and `info` as they are the most commonly used tools for obtaining help.

Another topic of this chapter will be locating files. You will mainly work with the `locate` command.

Getting Help on the Command Line

Built-in Help

When started with the `--help` parameter, most commands display some brief instructions about their usage. Although not all commands provide this switch, it is still a good first try to learn more about the parameters of a command. Be aware that the instructions from `--help` often are rather brief compared to the other sources of documentation which we will discuss in the rest of this lesson.

Man Pages

Most commands provide a manual page or "man" page. This documentation is usually installed with the software and can be accessed with the `man` command. The command whose man page should be displayed is added to `man` as an argument:

```
$ man mkdir
```

This command opens the man page for `mkdir`. You can use the up and down arrow keys or the space bar to navigate through the man page. To exit the man page, press `Q`.

Each man page is divided in maximum of 11 sections, though many of these sections are optional:

Section	Description
NAME	Command name and brief description
SYNOPSIS	Description of the command's syntax
DESCRIPTION	Description of the effects of the command
OPTIONS	Available options
ARGUMENTS	Available arguments
FILES	Auxiliary files
EXAMPLES	A sample of the command line
SEE ALSO	Cross-references to the related topics
DIAGNOSTICS	Warning and Error messages
COPYRIGHT	Author(s) of the command
BUGS	Any known limitations of the command

In practice, most man pages don't contain all of these parts.

Man pages are organized in eight categories, numbered from 1 to 8:

Category	Description
1	User command
2	System calls
3	Functions of the C library
4	Drivers and device files
5	Configuration files and file formats
6	Games
7	Miscellaneous
8	System administrator commands
9	Kernel functions (not standard)

Each man page belongs to exactly one category. However, multiple categories can contain man pages with the same name. Let's take the `passwd` command as an example. This command can be used to change a user's password. Since `passwd` is a user command, its man page resides in category 1. In addition to the `passwd` command, the password database file `/etc/passwd` also has a man page which is called `passwd`, too. As this file is a configuration file, it belongs to category 5. When referring to a man page, the category is often added to the name of the man page, as in `passwd(1)` or `passwd(5)` to identify the respective man page.

By default, `man passwd` displays the first available man page, in this case `passwd(1)`. The category of the desired man page can be specified in a command such as `man 1 passwd` or `man 5 passwd`.

We have already discussed how to navigate through a man page and how to return to the command line. Internally, `man` uses the `less` command to display the man page's content. `less` lets you search for text within a man page. To search for the word `linux` you can just use `/linux` for forward searching from the point that you are on the page, or `?linux` to start a backward search. This action highlights the all the matching results and moves the page to the first highlighted match. In both cases you can type `N` to jump to the next match. In order to find more information about these additional features, press `H` and a menu with all the information will be displayed.

Info Pages

Another tool that will help you while working with the Linux system are the info pages. The info

pages are usually more detailed than the man pages and are formatted in hypertext, similar to web pages on the Internet.

The info pages can be displayed like so:

```
$ info mkdir
```

For each info page, `info` reads an info file that is structured into individual nodes within a tree. Each node contains a simple topic and the `info` command contains hyperlinks that can help you move from one to the other. You can access the link by pressing enter while placing the cursor on one of the leading asterisks.

Similar to `man`, the `info` tool also has page navigation commands. You can find out more about these command by pressing `?` while being on the info page. These tools will help you navigate the page easier as well as understand how to access the nodes and move within the node tree.

The `/usr/share/doc/` directory

As mentioned before, the `/usr/share/doc/` directory stores most documentation of the commands that the system is using. This directory contains a directory for most packages installed on the system. The name of the directory is usually the name of the package and occasionally its version. These directories include a `README` or `readme.txt` file that contains the package's basic documentation. Alongside the `README` file, the folder can also contain other documentation files, such as the changelog which includes the history of the program in detail, or examples of configuration files for the specific package.

The information within the `README` file varies from one package to another. All files are written in plain text, therefore they can be read with any preferred text editor. The exact number and kinds of files depend on the package. Check some of the directories to get an overview of their contents.

Locating files

The `locate` command

A Linux system is built from numerous directories and files. Linux has many tools to locate a particular file within a system. The quickest one is the command `locate`.

`locate` searches within a database and then outputs every name that matches the given string:

```
$ locate note
```

```
/lib/udev/keymaps/zepto-znote
/usr/bin/zipnote
/usr/share/doc/initramfs-tools/maintainer-notes.html
/usr/share/man/man1/zipnote.1.gz
```

The `locate` command supports the usage of wildcards and regular expressions as well, therefore the search string does not have to match the entire name of the desired file. You will learn more about regular expressions in a later chapter.

By default, `locate` behaves as if the pattern would be surrounded by asterisks, so `locate PATTERN` is the same as `locate *PATTERN*`. This allows you just provide substrings instead of the exact filename. You can modify this behavior with the different options that you can find explained in the `locate` man page.

Because `locate` is reading from a database, you may not find a file that you recently created. The database is managed by a program named `updatedb`. Usually it is run periodically, but if you have root privileges and you need the database to be updated immediately, you can run the `updatedb` command yourself at any time.

The `find` command

`find` is another very popular tool that is used to search for files. This command has a different approach, compared to the `locate` command. `find` command searches a directory tree recursively, including its subdirectories. `find` does such a search at each invocation, it does not maintain a database like `locate`. Similar to `locate`, `find` also supports wildcards and regular expressions.

`find` requires at least the path it should search. Furthermore, so-called expressions can be added to provide filter criteria for which files to display. An example is the `-name` expression, which looks for files with a specific name:

```
~$ cd Downloads
~/Downloads
$ find . -name thesis.pdf
./thesis.pdf
~/Downloads
$ find ~ -name thesis.pdf
/home/carol/Downloads/thesis.pdf
```

The first `find` command searches for the file within the current `Downloads` directory, whereas the second one searches for the file in the user's home directory.

The `find` command is very complex, therefore it will not be covered in the Linux Essentials exam. However, it is a powerful tool which is particularly handy in practice.

Guided Exercises

1. Use the `man` command to find out what each command does:

Command	Description
`ls`	Display the contents of a directory.
`cat`	
`cut`	
`cd`	
`cp`	
`mv`	
`mkdir`	
`touch`	
`wc`	
`passwd`	
`rm`	
`rmdir`	
`more`	
`less`	
`whereis`	
`head`	
`tail`	
`sort`	
`tr`	
`chmod`	
`grep`	

2. Open the `ls` info page and identify the MENU.
   - What options do you have?
   - Find the option that allows you to sort the output by modification time.

3. Display the path to the first 3 `README` files. Use the `man` command to identify the correct option for `locate`.

4. Create a file called `test` in your home directory. Find its absolute path with the `locate` command.

5. Did you find it immediately? What did you have to do in order for `locate` to find it?

6. Search for the test file that you previously created, using the `find` command. What syntax did you use and what is the absolute path ?

Explorational Exercises

1. There is one command in the table above that doesn't have a `man` page. Which one is it and why do you think that the command doesn't have a man page?

2. Using the commands in the table above, create the following file tree. The names that start with a capital are Directories and the ones in lower case are files.

```
User
├── Documents
|   ├──Hello
|   |   ├──hey2
|   |   ├──helloa
|   |   └──ola5
|   └──World
|       └──earth9
├── Downloads
|   ├──Music
|   └──Songs
|       ├──collection1
|       └──collection2
├── Test
|   └── passa
└── test
```

3. Display on the screen the present working directory, including the subfolders.

4. Search within the tree for all files that end with a number.

5. Remove the entire directory tree with a single command.

Summary

In this lesson you learned:

- How to get help
- How to use the `man` command
- How to navigate the `man` page
- Different sections of the `man` page
- How to use the `info` command
- How to navigate between different nodes
- How to search for files within the system

Commands used in the exercises:

`man`
Display a man page.

`info`
Display an info page.

`locate`
Search the `locate` database for files with a specific name.

`find`
Search the file system for names matching a set of selection criteria.

`updatedb`
Update the `locate` database.

Answers to Guided Exercises

1. Use the `man` command to find out what each command does:

Command	Description
`ls`	Display the contents of a directory.
`cat`	Concatenates or views text files
`cut`	Removes sections from a text file
`cd`	Changes to a different directory
`cp`	Copies a file
`mv`	Moves a file (it can also be used to rename)
`mkdir`	Creates a new directory
`touch`	Creates a file or modifies an existing file's last modified time and date
`wc`	Counts the number of words, lines or bytes of a file
`passwd`	Changes a user's password
`rm`	Deletes a file
`rmdir`	Deletes a directory
`more`	Views text files one screen at a time
`less`	Views text files, allows scrolling up and down a line or page at a time
`whereis`	Displays the file path to a specified program and related manual files
`head`	Displays the first few lines of a file
`tail`	Displays the last few lines of a file
`sort`	Orders a file numerically or alphabetically
`tr`	Translates or removes characters of a file
`chmod`	Changes a file's permissions
`grep`	Searches within a file

2. Open the `ls` info page and identify the MENU.*

- What options do you have?
  - Which files are listed
  - What information is listed
  - Sorting the output
  - Details about version sort
  - General output formatting
  - Formatting file timestamps
  - Formatting the file names
- Find the option that allows you to sort the output by modification time.

  `-t` or `--sort=time`

3. Display the path to the first 3 `README` files. Use the `man` command to identify the correct option for `locate`.

```
$ locate -l 3 README
/etc/alternatives/README
/etc/init.d/README
/etc/rc0.d/README
```

4. Create a file called `test` in your home directory. Find its absolute path with the `locate` command.

```
$ touch test
$ locate test
/home/user/test
```

5. Did you find it immediately? What did you have to do in order for `locate` to find it?

```
$ sudo updatedb
```

   The file is newly created, therefore there is no record of it in the database.

6. Search for the test file that you previously created, using the `find` command. What syntax did you use and what is the absolute path ?

```
$ find ~ -name test
```

or

```
$ find . -name test
/home/user/test
```

Answers to Explorational Exercises

1. There is one command in the table above that doesn't have a `man` page. Which one is it and why do you think that the command doesn't have a man page?

   The `cd` command. It doesn't have a man page because it is a built-in shell command.

2. Using the commands in the table above, create the following file tree. The names that start with a capital are Directories and the ones in lower case are files.

```
User
├── Documents
|   ├──Hello
|   |   ├──hey2
|   |   ├──helloa
|   |   └──ola5
|   └──World
|       └──earth9
├── Downloads
|   ├──Music
|   └──Songs
|       ├──collection1
|       └──collection2
├── Test
|   └── passa
└── test
```

   The solution is a combination of `mkdir` and `touch` commands.

3. Display on the screen the present working directory, including the subfolders.

```
$ ls -R
```

4. Search within the tree for all files that end with a number.

```
$ find ~ -name "*[0-9]"
$ locate "*[0-9]"
```

5. Remove the entire directory tree with a single command.

```
$ rm -r Documents Downloads Test test
```

2.3 Using Directories and Listing Files

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 2.3

Weight

2

Key knowledge areas

- Files, directories
- Hidden files and directories
- Home directories
- Absolute and relative paths

Partial list of the used files, terms and utilities

- Common options for `ls`
- Recursive listings
- `cd`
- `.` and `..`
- `home` and `~`

2.3 Lesson 1

Introduction

Certificate:	Linux Essentials
Version:	1.6
Topic:	2 Finding Your Way on a Linux System
Objective:	2.3 Using Directories and Listing Files
Lesson:	1 of 2

Files and Directories

The Linux filesystem is similar to other operating system's filesystems in that it contains *files* and *directories*. Files contain data such as human-readable text, executable programs, or binary data that is used by the computer. Directories are used to create organization within the filesystem. Directories can contain files and other directories.

```
$ tree

Documents
├── Mission-Statement.txt
└── Reports
    └── report2018.txt

1 directory, 2 files
```

In this example, `Documents` is a directory that contains one file (`Mission-Statement.txt`) and one *subdirectory* (`Reports`). The `Reports` directory in turn contains one file called `report2018.txt`. The `Documents` directory is said to be the *parent* of the `Reports` directory.

TIP If the command `tree` is not available on your system, install it using your Linux distribution's package manager. Refer to the lesson on package management to learn how to do so.

File and Directory Names

File and directory names in Linux can contain lower case and upper case letters, numbers, spaces and special characters. However, since many special characters have a special meaning in the Linux shell, it is good practice to not use spaces or special characters when naming files or directories. Spaces, for example, need the *escape character* `\` to be entered correctly:

```
$ cd Mission\ Statements
```

Also, refer to the filename `report2018.txt`. Filenames can contain a *suffix* which comes after the period (`.`). Unlike Windows, this suffix has no special meaning in Linux; it is there for human understanding. In our example `.txt` indicates to us that this is a plaintext file, although it could technically contain any kind of data.

Navigating the Filesystem

Getting Current Location

Since Linux shells such as Bash are text-based, it is important to remember your current location when navigating the filesystem. The *command prompt* provides this information:

```
user@hostname ~/Documents/Reports $
```

Note that information such as `user` and `hostname` will be covered in future sections. From the prompt, we now know that our current location is in the `Reports` directory. Similarly, the command `pwd` will *print working directory*:

```
user@hostname ~/Documents/Reports $ pwd
/home/user/Documents/Reports
```

The relationship of directories is represented with a forward slash (`/`). We know that `Reports` is a subdirectory of `Documents`, which is a subdirectory of `user`, which is located in a directory called `home`. `home` doesn't seem to have a parent directory, but that is not true at all. The parent of `home` is called *root*, and is represented by the first slash (`/`). We will discuss the root directory in a later section.

Notice that the output of the `pwd` command differs slightly from the path given on the command prompt. Instead of `/home/user`, the command prompt contains a tilde (`~`). The tilde is a special character that represents the user's home directory. This will be covered in more detail in the next lesson.

Listing Directory Contents

The contents of the current directory are listed with the `ls` command:

```
user@hostname ~/Documents/Reports $ ls
report2018.txt
```

Note that `ls` provides no information about the parent directory. Similarly, by default `ls` does not show any information about contents of subdirectories. `ls` can only "see" what is in the current directory.

Changing Current Directory

Navigation in Linux is primarily done with the `cd` command. This *changes directory*. Using the `pwd` command from before, we know our current directory is `/home/user/Documents/Reports`. We can change our current directory by entering a new path:

```
user@hostname ~ $ cd /home/user/Documents
user@hostname ~/Documents $ pwd
/home/user/Documents
user@hostname ~/Documents $ ls
Mission-Statement.txt Reports
```

From our new location, we can "see" `Mission-Statement.txt` and our subdirectory `Reports`, but not the contents of our subdirectory. We can navigate back into `Reports` like this:

```
user@hostname ~/Documents $ cd Reports
user@hostname ~/Documents/Reports $ pwd
/home/user/Documents/Reports
```

```
user@hostname ~/Documents/Reports $ ls
report2018.txt
```

We are now back where we started.

Absolute and Relative Paths

The `pwd` command always prints an *absolute path*. This means that the path contains every step of the path, from the top of the filesystem (`/`) to the bottom (`Reports`). Absolute paths always begin with a `/`.

```
/
└── home
    └── user
        └── Documents
            └── Reports
```

The absolute path contains all the information required to get to `Reports` from anywhere in the filesystem. The drawback is that it is tedious to type.

The second example (`cd Reports`) was much easier to type. This is an example of a *relative path*. Relative paths are shorter but only have meaning in relation to your current location. Consider this analogy: I am visiting you at your house. You tell me that your friend lives next door. I will understand that location because it is relative to my current location. But if you tell me this over the phone, I will not be able to find your friend's house. You will need to give me the complete street address.

Special Relative Paths

The Linux shell gives us ways to shorten our paths when navigating. To reveal the first special paths, we enter the `ls` command with the flag `-a`. This flag modifies the `ls` command so that *all* files and directories are listed, including hidden files and directories:

```
user@hostname ~/Documents/Reports $ ls -a
.
..
report2018.txt
```

NOTE You can refer to the `man` page for `ls` to understand what `-a` is doing here.

This command has revealed two additional results: These are special paths. They do not represent new files or directories, but rather they represent directories that you already know:

`.`
: Indicates the *current location* (in this case, `Reports`).

`..`
: Indicates the *parent directory* (in this case, `Documents`).

It is usually unnecessary to use the special relative path for the current location. It is easier and more understandable to type `report2018.txt` than it is to type `./report2018.txt`. But the `.` has uses that you will learn in future sections. For now, we will focus on the relative path for the parent directory:

```
user@hostname ~/Documents/Reports $ cd ..
user@hostname ~/Documents $ pwd
/home/user/Documents
```

The example of `cd` is much easier when using `..` instead of the absolute path. Additionally, we can combine this pattern to navigate up the file tree very quickly.

```
user@hostname ~/Documents $ cd ../..
$ pwd
/home
```

Guided Exercises

1. For each of the following paths, identify whether it is *absolute* or *relative*:

`/home/user/Downloads`	
`../Reports`	
`/var`	
`docs`	
`/`	

2. Observe the following file structure. Note: Directories end with a slash (`/`) when `tree` is invoked with the `-F` option. You will need elevated privileges in order to run the `tree` command on the root (`/`) directory. The following is example output and is not indicative of a full directory structure. Use it to answer the following questions:

```
$ sudo tree -F /

/
├── etc/
│   ├── network/
│   │   └── interfaces
│   ├── systemd/
│   │   ├── resolved.conf
│   │   ├── system/
│   │   ├── system.conf
│   │   ├── user/
│   │   └── user.conf
│   └── udev/
│       ├── rules.d/
│       └── udev.conf
└── home/
    ├── lost+found/
    └── user/
        └── Documents/

12 directories, 5 files
```

   Use this structure to answer the following questions.

   A user enters the following commands:

```
$ cd /etc/udev
$ ls -a
```

What will be the output of the `ls -a` command?

3. Enter the shortest possible command for each of the following:
   - Your current location is root (`/`). Enter the command to navigate to `lost+found` within the `home` directory (example):

     ```
     $ cd home/lost+found
     ```

   - Your current location is root (`/`). Enter the command to navigate to the directory named `/etc/network/`.
   - Your current location is `/home/user/Documents/`. Navigate to the directory named `/etc/`.
   - Your current location is `/etc/systemd/system/`. Navigate to the directory named `/home/user/`.

4. Consider the following commands:

```
$ pwd
/etc/udev/rules.d
$ cd ../../systemd/user
$ cd ..
$ pwd
```

What is the output of the final `pwd` command?

Explorational Exercises

1. Suppose a user has entered the following commands:

```
$ mkdir "this is a test"
$ ls
this is a test
```

   What `cd` command would allow you to enter this directory?

2. Try this again, but after typing in `cd this`, press the TAB key. What is now displayed on the prompt?

   This is an example of *autocompletion*, which is an invaluable tool not only for saving time, but for preventing spelling errors.

3. Try to create a directory whose name contains a `\` character. Display the directory's name with `ls` and delete the directory.

Summary

In this lesson, you learned:

- The fundamentals of the Linux filesystem
- The difference between *parent* directories and *subdirectories*
- The difference between *absolute* file paths and *relative* file paths
- The special relative paths `.` and `..`
- Navigate the filesystem using `cd`
- Show your current location using `pwd`
- List *all* files and directories using `ls -a`

The following commands were discussed in this lesson:

`cd`
: Change the current directory.

`pwd`
: Print the current working directory's path

`ls`
: List the contents of a directory and display properties of files

`mkdir`
: Create a new directory

`tree`
: Display a hierarchical listing of a directory tree

Answers to Guided Exercises

1. For each of the following paths, identify whether it is *absolute* or *relative*:

`/home/user/Downloads`	absolute
`../Reports`	relative
`/var`	absolute
`docs`	relative
`/`	absolute

2. Observe the following file structure. Note: Directories end with a slash (`/`) when `tree` is invoked with the `-F` option. You will need elevated privileges in order to run the `tree` command on the root (`/`) directory. The following is example output and is not indicative of a full directory structure. Use it to answer the following questions:

```
$ sudo tree -F /

/
├── etc/
│   ├── network/
│   │   └── interfaces
│   ├── systemd/
│   │   ├── resolved.conf
│   │   ├── system/
│   │   ├── system.conf
│   │   ├── user/
│   │   └── user.conf
│   └── udev/
│       ├── rules.d/
│       └── udev.conf
└── home/
    ├── lost+found/
    └── user/
        └── Documents/

12 directories, 5 files
```

A user enters the following commands:

```
$ cd /etc/udev
$ ls -a
```

What will be the output of the `ls -a` command?

```
. .. rules.d udev.conf
```

3. Enter the shortest possible command for each of the following:
   - Your current location is root (`/`). Enter the command to navigate to `lost+found` within the home directory (example):

     ```
     $ cd home/lost+found
     ```

   - Your current location is root (`/`). Enter the command to navigate to the directory named `network`:

     ```
     $ cd etc/network
     ```

   - Your current location is `Documents`. Navigate to the directory named `etc`:

     ```
     $ cd /etc
     ```

   - Your current location is `system`. Navigate to the directory named `user`:

     ```
     $ cd /home/user
     ```

4. Consider the following commands:

```
$ pwd
/etc/udev/rules.d
$ cd ../../systemd/user
$ cd ..
$ pwd
```

What is the output of the final `pwd` command?

```
/etc/systemd
```

Answers to Explorational Exercises

1. Suppose a user has entered the following commands:

   ```
   $ mkdir "this is a test"
   $ ls
   this is a test
   ```

   What `cd` command would allow you to enter this directory?

   ```
   $ cd this\ is\ a\ test
   ```

2. Try this again, but after typing in `cd this`, press the TAB key. What is now displayed on the prompt?

   ```
   $ cd this\ is\ a\ test
   ```

   This is an example of autocompletion, which is an invaluable tool not only for saving time, but for preventing spelling errors.

3. Try to create a directory whose name contains a `\` character. Display the directory's name with `ls` and delete the directory.

   You can either escape the backslash using another backslash (`\\`) or use single or double quotes around the whole directory name:

   ```
   $ mkdir my\\dir
   $ ls
   'my\dir'
   $ rmdir 'my\dir'
   ```

2.3 Lesson 2

Certificate:	Linux Essentials
Version:	1.6
Topic:	2 Finding Your Way on a Linux System
Objective:	2.3 Using Directories and Listing Files
Lesson:	2 of 2

Introduction

The Unix operating system was originally designed for mainframe computers in the mid-1960s. These computers were shared among many users, who accessed the system's resources through *terminals*. These fundamental ideas carry through to Linux systems today. We still talk about using "terminals" to enter commands in the shell, and every Linux system is organized in such a way that it is easy to create many users on a single system.

Home Directories

This is an example of a normal file system in Linux:

```
$ tree -L 1 /
/
├── bin
├── boot
├── cdrom
├── dev
├── etc
```

```
├── home
├── lib
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── sys
├── tmp
├── usr
└── var
```

Most of these directories are consistent across all Linux systems. From servers to supercomputers to tiny embedded systems, a seasoned Linux user can be confident that they can find the `ls` command inside `/bin`, can change the system configuration by modifying files in `/etc`, and read system logs in `/var`. The standard location of these files and directories is defined by the Filesystem Hierarchy Standard (FHS), which will be discussed in a later lesson. You will learn more about the contents of these directories as you continue learning about Linux, but for the time being, know that:

- changes that you make in the root filesystem will affect all users, and
- changing files in the root filesystem will require administrator permissions.

This means that normal users will be prohibited from modifying these files, and may also be prohibited from even reading these files. We will cover the topic of permissions in a later section.

Now, we will focus on the directory `/home`, which should be somewhat familiar at this point:

```
$ tree -L 1 /home
/home
├── user
├── michael
└── lara
```

Our example system has three normal users, and each of our users has their own dedicated location, where they can create and modify files and directories without affecting their neighbor. For example, in the previous lesson we were working with the following file structure:

```
$ tree /home/user
```

```
user
└── Documents
    ├── Mission-Statement
    └── Reports
        └── report2018.txt
```

In actuality, the real filesystem may look like this:

```
$ tree /home
/home
├── user
│   └── Documents
│       ├── Mission-Statement
│       └── Reports
│           └── report2018.txt
├── michael
│   ├── Documents
│   │   └── presentation-for-clients.odp
│   └── Music
```

...and so on for `lara`.

In Linux, `/home` is similar to an apartment building. Many users may have their space here, separated into dedicated apartments. The utilities and maintenance of the building itself are the responsibility of the superintendent root user.

The Special Relative Path for Home

When you start a new terminal session in Linux, you see a command prompt similar to this:

```
user@hostname ~ $
```

The tilde (`~`) here represents our *home directory*. If you run the `ls` command, you will see some familiar output:

```
$ cd ~
$ ls
Documents
```

Compare that with the file system above to check your understanding.

Consider now what we know about Linux: it is similar to an apartment building, with many users residing in `/home`. So `user` 's home will be different from user `michael` 's home. To demonstrate this, we will use the `su` command to *switch user*.

```
user@hostname ~ $ pwd
/home/user
user@hostname ~ $ su - michael
Password:
michael@hostname ~ $ pwd
/home/michael
```

The meaning of `~` changes depending of who the user is. For `michael`, the absolute path of `~` is `/home/michael`. For `lara`, the absolute path of `~` is `/home/lara`, and so on.

Relative-to-Home File Paths

Using `~` for commands is very convenient, provided that you don't switch users. We will consider the following example for `user`, who has begun a new session:

```
$ ls
Documents
$ cd Documents
$ ls
Mission-Statement
Reports
$ cd Reports
$ ls
report2018.txt
$ cd ~
$ ls
Documents
```

Note that users will always begin a new session in their home directory. In this example, `user` has traveled into their `Documents/Reports` subdirectory, and with the `cd ~` command they have returned to where they started. You can perform the same action by using the `cd` command with no arguments:

```
$ cd Documents/Reports
$ pwd
/home/user/Documents/Reports
```

```
$ cd
$ pwd
/home/user
```

One last thing to note: we can specify the home directories of *other users* by specifying the username after the tilde. For example:

```
$ ls ~michael
Documents
Music
```

Note that this will only work if `michael` has given us permission to view the contents of his home directory.

Let's consider a situation where `michael` would like to view the file `report2018.txt` in the `user` home directory. Assuming that `michael` has the permission to do so, he can use the `less` command.

```
$ less ~user/Documents/Reports/report2018.txt
```

Any file path that contains the `~` character is called a *relative-to-home* path.

Hidden Files and Directories

In the previous lesson, we introduced the option `-a` for the `ls` command. We used `ls -a` to introduce the two special relative paths: `.` and `..`. The `-a` option will list *all* files and directories, including *hidden* files and directories.

```
$ ls -a ~
.
..
.bash_history
.bash_logout
.bash-profile
.bashrc
Documents
```

Hidden files and directories will always begin with a period (`.`). By default, a user's home directory will include many hidden files. These are often used to set user-specific configuration

settings, and should only be modified by an experienced user.

The Long List Option

The `ls` command has many options to change its behavior. Let's look at one of the most common options:

```
$ ls -l
-rw-r--r-- 1 user staff      3606 Jan 13  2017 report2018.txt
```

`-l` creates a *long list*. Files and directories will each occupy one line, but additional information about each file and directory will be displayed.

`-rw-r--r--`
: Type of file and permissions of the file. Note that a regular file will begin with dash, and a directory will start with `d`.

`1`
: Number of links to the file.

`user staff`
: Specifies ownership of the file. `user` is the owner of the file, and the file is also associated with the `staff` group.

`3606`
: Size of the file in bytes.

`Jan 13 2017`
: Time stamp of the last modification to the file.

`report2018.txt`
: Name of the file.

Subjects such as ownership, permissions and links will be covered in future sections. As you can see, the long list version of the `ls` is oftentimes preferable to the default.

Additional ls Options

Below are some of the ways that we most commonly use the `ls` command. As you can see, the user can combine many options together to get the desired output.

`ls -lh`

Combining *long list* with *human readable* file sizes will give us useful suffixes such as `M` for megabytes or `K` for kilobytes.

`ls -d */`

The `-d` option will list directories but not their contents. Combining this with `*/` will show only subdirectories and no files.

`ls -lt`

Combines *long list* with the option to sort by *modification time*. The files with the most recent changes will be at the top, and files with the oldest changes will be at the bottom. But this order can be reversed with:

`ls -lrt`

Combines *long list* with *sort by (modification) time*, combined with `-r` which *reverses* the sort. Now files with the most recent changes are at the bottom of the list. In addition to sorting by *modification* time, files can also be sorted by *access* time or by *status* time.

`ls -lX`

Combines *long list* with the option to sort by *file eXtension*. This will, for example, group all files ending with `.txt` together, all files ending with `.jpg` together, and so on.

`ls -S`

The `-S` sorts by file *size*, much in the same way as `-t` and `-X` sort by time and extension respectively. The largest files will come first, and smallest last. Note that the contents of subdirectories are *not* included in the sort.

`ls -R`

The `-R` option will modify the `ls` command to display a *recursive* list. What does this mean?

Recursion in Bash

Recursion refers to a situation when “something is defined in terms of itself”. Recursion is a very important concept in computer science, but here its meaning is far simpler. Let’s consider our example from before:

```
$ ls ~
Documents
```

We know from before that `user` has a home directory, and in this directory there is one

subdirectory. `ls` has up until now only shown us the files and subdirectories of a location, but cannot tell us the contents of these subdirectories. In these lessons, we have been using the `tree` command when we wanted to display the contents of many directories. Unfortunately, `tree` is not one of the core utilities of Linux and thus is not always available. Compare the output of `tree` with the output of `ls -R` in the following examples:

```
$ tree /home/user
user
└── Documents
    ├── Mission-Statement
    └── Reports
        └── report2018.txt

$ ls -R ~
/home/user/:
Documents

/home/user/Documents:
Mission-Statement
Reports

/home/user/Documents/Reports:
report2018.txt
```

As you can see, with the recursive option, we get a far longer list of files. In fact, it is as if we ran the `ls` command in `user` 's home directory, and encountered one subdirectory. Then, we entered into that subdirectory and ran the `ls` command again. We encountered the file `Mission-Statement` and another subdirectory called `Reports`. And again, we entered into the subdirectory, and ran the `ls` command again. Essentially, running `ls -R` is like telling Bash: “Run `ls` here, and repeat the command in every subdirectory that you find.”

Recursion is particularly important in file modification commands such as copying or removing directories. For example, if you wanted to copy the `Documents` subdirectory, you would need to specify a recursive copy in order to extend this command to all subdirectories.

Guided Exercises

1. Use the following file structure to answer the following three questions:

```
/
├── etc/
│   ├── network/
│   │   └── interfaces/
│   ├── systemd/
│   │   ├── resolved.conf
│   │   ├── system/
│   │   ├── system.conf
│   │   ├── user/
│   │   └── user.conf
│   └── udev/
│       ├── rules.d
│       └── udev.conf
└── home/
    ├── lost+found/
    ├── user/
    │   └── Documents/
    └── michael/
        └── Music/
```

   - What command will navigate into the `network` directory regardless of your current location?

   - What command can `user` enter to navigate into their `Documents` directory from `/etc/udev`? Use the shortest possible path.

   - What command can `user` enter to navigate into `michael` 's `Music` directory? Use the shortest possible path.

2. Consider the following output of `ls -lh` to answer the next two questions. Note that directories are indicated with a `d` at the beginning of the line.

```
drwxrwxrwx  5 eric eric  4.0K Apr 26  2011 China/
-rwxrwxrwx  1 eric eric  1.5M Jul 18  2011 img_0066.jpg
```

```
-rwxrwxrwx  1 eric eric  1.5M Jul 18  2011 img_0067.jpg
-rwxrwxrwx  1 eric eric  1.6M Jul 18  2011 img_0074.jpg
-rwxrwxrwx  1 eric eric  1.8M Jul 18  2011 img_0075.jpg
-rwxrwxrwx  1 eric eric   46K Jul 18  2011 scary.jpg
-rwxrwxrwx  1 eric eric  469K Jan 29  2018 Screenshot from 2017-08-13 21-22-24.png
-rwxrwxrwx  1 eric eric  498K Jan 29  2018 Screenshot from 2017-08-14 21-18-07.png
-rwxrwxrwx  1 eric eric  211K Jan 29  2018 Screenshot from 2018-01-06 23-29-30.png
-rwxrwxrwx  1 eric eric  150K Jul 18  2011 tobermory.jpg
drwxrwxrwx  6 eric eric  4.0K Apr 26  2011 Tokyo/
-rwxrwxrwx  1 eric eric  1.4M Jul 18  2011 Toronto 081.jpg
-rwxrwxrwx  1 eric eric  1.4M Jul 18  2011 Toronto 085.jpg
-rwxrwxrwx  1 eric eric  944K Jul 18  2011 Toronto 152.jpg
-rwxrwxrwx  1 eric eric  728K Jul 18  2011 Toronto 173.jpg
drwxrwxrwx  2 eric eric  4.0K Jun  5  2016 Wallpapers/
```

- When you run the command `ls -lrS`, what file will be at the beginning?

- Please describe what you expect to see as the output for `ls -ad */`.

Explorational Exercises

1. Run the `ls -lh` command in a directory that contains subdirectories. Note the listed size of these directories. Do these file sizes seem correct to you? Do they accurately represent the contents of all files inside that directory?

2. Here is a new command to try: `du -h`. Run this command and describe the output that it gives you.

3. On many Linux systems, you can type in `ll` and get the same output as you would if you typed `ls -l`. Please note however that `ll` is *not* a command. For example, `man ll` will give you the message that no manual entry exists for it. This is an example of an *alias*. Why might aliases be useful to a user?

Summary

In this lab, you learned:

- that each Linux user will have a home directory,
- the current user's home directory can be reached by using `~`,
- any file path that uses `~` is called a *relative-to-home* path.

You also learned about some of the most common ways of modifying the `ls command.

`-a` (all)
prints all files/directories, including hidden

`-d` (directories)
list directories, not their contents

`-h` (human readable)
prints file sizes in human readable format

`-l` (long list)
provides extra details, one file/directory per line

`-r` (reverse)
reverses the order of a sort

`-R` (recursive)
lists every file, including files in each subdirectory

`-S` (size)
sorts by file size

`-t` (time)
sorts by modification time

`-X` (eXtension)
sorts by file extension

Answers to Guided Exercises

1. Use the following file structure to answer the following three questions:

```
/
├── etc/
│   ├── network/
│   │   └── interfaces/
│   ├── systemd/
│   │   ├── resolved.conf
│   │   ├── system/
│   │   ├── system.conf
│   │   ├── user/
│   │   └── user.conf
│   └── udev/
│       ├── rules.d
│       └── udev.conf
└── home/
    ├── lost+found/
    ├── user/
    │   └── Documents/
    └── michael/
        └── Music/
```

   - What command will navigate into the `network` directory regardless of your current location?

     ```
     cd /etc/network
     ```

   - What command can `user` enter to navigate into their `Documents` directory from `/etc/udev`? Use the shortest possible path.

     ```
     cd ~/Documents
     ```

   - What command can `user` enter to navigate into `michael` 's `Music` directory? Use the shortest possible path:

     ```
     cd ~michael/Music
     ```

2. Consider the following output of `ls -lh` to answer the next two questions. Note that directories are indicated with a `d` at the beginning of the line.

```
drwxrwxrwx  5 eric eric  4.0K Apr 26  2011 China/
-rwxrwxrwx  1 eric eric  1.5M Jul 18  2011 img_0066.jpg
-rwxrwxrwx  1 eric eric  1.5M Jul 18  2011 img_0067.jpg
-rwxrwxrwx  1 eric eric  1.6M Jul 18  2011 img_0074.jpg
-rwxrwxrwx  1 eric eric  1.8M Jul 18  2011 img_0075.jpg
-rwxrwxrwx  1 eric eric   46K Jul 18  2011 scary.jpg
-rwxrwxrwx  1 eric eric  469K Jan 29  2018 Screenshot from 2017-08-13 21-22-24.png
-rwxrwxrwx  1 eric eric  498K Jan 29  2018 Screenshot from 2017-08-14 21-18-07.png
-rwxrwxrwx  1 eric eric  211K Jan 29  2018 Screenshot from 2018-01-06 23-29-30.png
-rwxrwxrwx  1 eric eric  150K Jul 18  2011 tobermory.jpg
drwxrwxrwx  6 eric eric  4.0K Apr 26  2011 Tokyo/
-rwxrwxrwx  1 eric eric  1.4M Jul 18  2011 Toronto 081.jpg
-rwxrwxrwx  1 eric eric  1.4M Jul 18  2011 Toronto 085.jpg
-rwxrwxrwx  1 eric eric  944K Jul 18  2011 Toronto 152.jpg
-rwxrwxrwx  1 eric eric  728K Jul 18  2011 Toronto 173.jpg
drwxrwxrwx  2 eric eric  4.0K Jun  5  2016 Wallpapers/
```

   - When you run the command `ls -lrS`, what file will be at the beginning?

     The three folders are all 4.0K, which is the smallest file size. `ls` will then sort the directories alphabetically by default. The correct answer is the file `scary.jpg`.

   - Please describe what you expect to see as the output for `ls -ad */`.

     This command will show all subdirectories, including hidden subdirectories.

Answers to Explorational Exercises

1. Run the `ls -lh` command in a directory that contains subdirectories. Note the listed size of these directories. Do these file sizes seem correct to you? Do they accurately represent the contents of all files inside that directory?

   No, they do not. Each directory has a listed file size of 4096 bytes. This is because directories here are an abstraction: they don't exist as a tree structure on the disk. When you see a directory listed, you are seeing a *link* to a list of files. The size of these links is 4096 bytes.

2. Here is a new command to try: `du -h`. Run this command and describe the output that it gives you.

   The `du` command will generate a list of all files and directories, and indicate the size of each. For example, `du -s` will display the file size of all files, directories, and subdirectories for a certain location.

3. On many Linux systems, you can type in `ll` and get the same output as you would if you typed `ls -l`. Please note however that `ll` is *not* a command. For example, `man ll` will give you the message that no manual entry exists for it. What does this suggest to you about a feature of the command line?

   `ll` is an *alias* of `ls -l`. In Bash, we can use aliases to simplify commonly-used commands. `ll` is often defined for you in Linux, but you can create your own as well.

2.4 Creating, Moving and Deleting Files

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 2.4

Weight

2

Key knowledge areas

- Files and directories
- Case sensitivity
- Simple globbing

Partial list of the used files, terms and utilities

- `mv`, `cp`, `rm`, `touch`
- `mkdir`, `rmdir`

2.4 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	2 Finding Your Way on a Linux System
Objective:	2.4 Creating, Moving and Deleting Files
Lesson:	1 of 1

Introduction

This lesson covers managing files and directories on Linux using command line tools.

A file is a collection of data with a name and set of attributes. If, for example, you were to transfer some photos from your phone to a computer and give them descriptive names, you would now have a bunch of image files on your computer. These files have attributes such as the time the file was last accessed or modified.

A directory is a special kind of file used to organize files. A good way to think of directories is like the file folders used to organize papers in a file cabinet. Unlike paper file folders, you can easily put directories inside of other directories.

The command line is the most effective way to manage files on a Linux system. The shell and command line tools have features that make using the command line faster and easier than a graphical file manager.

In this section you will use the commands `ls`, `mv`, `cp`, `pwd`, `find`, `touch`, `rm`, `rmdir`, `echo`, `cat`, and `mkdir` to manage and organize files and directories.

Case Sensitivity

Unlike Microsoft Windows, file and directory names on Linux systems are case sensitive. This means that the names `/etc/` and `/ETC/` are different directories. Try the following commands:

```
$ cd /
$ ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
$ cd ETC
bash: cd: ETC: No such file or directory
$ pwd
/
$ cd etc
$ pwd
/etc
```

The `pwd` shows you the directory you are currently in. As you can see, changing to `/ETC` did not work as there is no such directory. Changing into the directory `/etc` which exists, did succeed.

Creating Directories

The `mkdir` command is used to create directories.

Let's create a new directory within our home directory:

```
$ cd ~
$ pwd
/home/emma
$ ls
Desktop  Documents  Downloads
$ mkdir linux_essentials-2.4
$ ls
Desktop  Documents  Downloads  linux_essentials-2.4
$ cd linux_essentials-2.4
$ pwd
/home/emma/linux_essentials-2.4
```

For the duration of this lesson, all commands will take place within this directory or in one of its subdirectories.

To easily return to the lesson directory from any other position in your file system, you can use the command:

```
$ cd ~/linux_essentials-2.4
```

The shell interprets the `~` character as your home directory.

When you're in the lesson directory, create some more directories which we will use for the exercises. You can add all the directory names, separated by spaces, to `mkdir`:

```
$ mkdir creating moving copying/files copying/directories deleting/directories
deleting/files globs
mkdir: cannot create directory ‘copying/files’: No such file or directory
mkdir: cannot create directory ‘copying/directories’: No such file or directory
mkdir: cannot create directory ‘deleting/directories’: No such file or directory
mkdir: cannot create directory ‘deleting/files’: No such file or directory
$ ls
creating  globs  moving
```

Notice the error message and that only `moving`, `globs`, and `creating` were created. The `copying` and `deleting` directories don't exist yet. `mkdir`, by default, won't create a directory inside of a directory that does not exist. The `-p` or `--parents` option instructs `mkdir` to create parent directories if they do not exist. Try the same `mkdir` command with the `-p` option:

```
$ mkdir -p creating moving copying/files copying/directories deleting/directories
deleting/files globs
```

Now you don't get any error messages. Let's see which directories exist now:

```
$ find
.
./creating
./moving
./globs
./copying
./copying/files
./copying/directories
./deleting
./deleting/directories
./deleting/files
```

The `find` program is usually used to search for files and directories, but without any options, it will show you a listing of all the files, directories, and sub-directories of your current directory.

TIP When listing the contents of a directory with `ls`, the `-t` and `-r` options are particularly handy. They sort the output by time (`-t`) and reverse the sorting order (`-r`). In this case, the newest files will be at the bottom of the output.

Creating Files

Typically, files will be created by the programs that work with the data stored in the files. An empty file can be created using the `touch` command. If you run `touch` on an existing file, the file's contents won't be changed, but the files modification timestamp will be updated.

Run the following command to create some files for the globbing lesson:

```
$ touch globs/question1 globs/question2012 globs/question23 globs/question13
globs/question14
$ touch globs/star10 globs/star1100 globs/star2002 globs/star2013
```

Now let's verify all files exist in the `globs` directory:

```
$ cd globs
$ ls
question1   question14    question23  star1100  star2013
question13  question2012  star10      star2002
```

Notice how `touch` created the files? You can view the contents of a text file with the `cat` command. Try it on one of the files you just created:

```
$ cat question14
```

Since `touch` creates empty files, you should get no output. You can use `echo` with `>` to create simple text files. Try it:

```
$ echo hello > question15
$ cat question15
hello
```

`echo` displays text on the command line. The `>` character instructs the shell to write output of a

command to the specified file instead of your terminal. This leads to the output of `echo`, `hello` in this case, being written to the file `question15`. This isn't specific to `echo`, it can be used with any command.

WARNING Be careful when using `>`! If the named file already exists, it will be overwritten!

Renaming Files

Files are moved and renamed with the `mv` command.

Set your working directory to the `moving` directory:

```
$ cd ~/linux_essentials-2.4/moving
```

Create some files to practice with. By now, you should already be familiar with these commands:

```
$ touch file1 file22
$ echo file3 > file3
$ echo file4 > file4
$ ls
file1  file22  file3  file4
```

Suppose `file22` is a typo and should be `file2`. Fix it with the `mv` command. When renaming a file, the first argument is the current name, the second is the new name:

```
$ mv file22 file2
$ ls
file1  file2  file3  file4
```

Be careful with the `mv` command. If you rename a file to the name of an existing file, it will overwrite it. Let's test this with `file3` and `file4`:

```
$ cat file3
file3
$ cat file4
file4
$ mv file4 file3
$ cat file3
file4
```

```
$ ls
file1  file2  file3
```

Notice how the contents of `file3` is now `file4`. Use the `-i` option to make `mv` prompt you if you are about to overwrite an existing file. Try it:

```
$ touch file4 file5
$ mv -i file4 file3
mv: overwrite ‘file3’? y
```

Moving Files

Files are moved from one directory to another with the `mv` command.

Create a few directories to move files into:

```
$ cd ~/linux_essentials-2.4/moving
$ mkdir dir1 dir2
$ ls
dir1  dir2  file1  file2  file3  file5
```

Move `file1` into `dir1`:

```
$ mv file1 dir1
$ ls
dir1  dir2  file2  file3  file5
$ ls dir1
file1
```

Notice how the last argument to `mv` is the destination directory. Whenever the last argument to `mv` is a directory, files are moved into it. Multiple files can be specified in a single `mv` command:

```
$ mv file2 file3 dir2
$ ls
dir1  dir2  file5
$ ls dir2
file2  file3
```

It is also possible to use `mv` to move and rename directories. Rename `dir1` to `dir3`:

```
$ ls
dir1  dir2  file5
$ ls dir1
file1
$ mv dir1 dir3
$ ls
dir2  dir3  file5
$ ls dir3
file1
```

Deleting Files and Directories

The `rm` command can delete files and directories, while the `rmdir` command can only delete directories. Let's clean up the `moving` directory by deleting `file5`:

```
$ cd ~/linux_essentials-2.4/moving
$ ls
dir2  dir3  file5
$ rmdir file5
rmdir: failed to remove 'file5': Not a directory
$ rm file5
$ ls
dir2  dir3
```

By default `rmdir` can only delete empty directories, therefore we had to use `rm` to delete a regular file. Try to delete the `deleting` directory:

```
$ cd ~/linux_essentials-2.4/
$ ls
copying  creating  deleting  globs  moving
$ rmdir deleting
rmdir: failed to remove 'deleting': Directory not empty
$ ls -l deleting
total 0
drwxrwxr-x. 2 emma emma 6 Mar 26 14:58 directories
drwxrwxr-x. 2 emma emma 6 Mar 26 14:58 files
```

By default, `rmdir` refuses to delete a directory that is not empty. Use `rmdir` to remove one of the empty subdirectories of the `deleting` directory:

```
$ ls -a deleting/files
.  ..
$ rmdir deleting/files
$ ls -l deleting
directories
```

Deleting large numbers of files or deep directory structures with many subdirectories may seem tedious, but it is actually easy. By default, `rm` only works on regular files. The `-r` option is used to override this behavior. Be careful, `rm -r` is an excellent foot gun! When you use the `-r` option, `rm` will not only delete any directories, but everything within that directory, including subdirectories and their contents. See for yourself how `rm -r` works:

```
$ ls
copying  creating  deleting  globs  moving
$ rm deleting
rm: cannot remove ‘deleting’: Is a directory
$ ls -l deleting
total 0
drwxrwxr-x. 2 emma emma 6 Mar 26 14:58 directories
$ rm -r deleting
$ ls
copying  creating  globs  moving
```

Notice how `deleting` is gone, even though it was not empty? Like `mv`, `rm` has a `-i` option to prompt you before doing anything. Use `rm -ri` to remove directories from `moving` section that are no longer needed:

```
$ find
.
./creating
./moving
./moving/dir2
./moving/dir2/file2
./moving/dir2/file3
./moving/dir3
./moving/dir3/file1
./globs
./globs/question1
./globs/question2012
./globs/question23
./globs/question13
```

```
./globs/question14
./globs/star10
./globs/star1100
./globs/star2002
./globs/star2013
./globs/question15
./copying
./copying/files
./copying/directories
$ rm -ri moving
rm: descend into directory ‘moving’? y
rm: descend into directory ‘moving/dir2’? y
rm: remove regular empty file ‘moving/dir2/file2’? y
rm: remove regular empty file ‘moving/dir2/file3’? y
rm: remove directory ‘moving/dir2’? y
rm: descend into directory ‘moving/dir3’? y
rm: remove regular empty file ‘moving/dir3/file1’? y
rm: remove directory ‘moving/dir3’? y
rm: remove directory ‘moving’? y
```

Copying Files and Directories

The `cp` command is used to copy files and directories. Copy a few files into the `copying` directory:

```
$ cd ~/linux_essentials-2.4/copying
$ ls
directories  files
$ cp /etc/nsswitch.conf files/nsswitch.conf
$ cp /etc/issue /etc/hostname files
```

If the last argument is a directory, `cp` will create a copy of the previous arguments inside the directory. Like `mv`, multiple files can be specified at once, as long as the target is a directory.

When both operands of `cp` are files and both files exist, `cp` overwrites the second file with a copy of the first file. Let's practice this by overwrite the `issue` file with the `hostname` file:

```
$ cd ~/linux_essentials-2.4/copying/files
$ ls
hostname  issue  nsswitch.conf
$ cat hostname
mycomputer
$ cat issue
```

```
Debian GNU/Linux 9 \n \l

$ cp hostname issue
$ cat issue
mycomputer
```

Now let's try to create a copy of the `files` directory within the `directories` directory:

```
$ cd ~/linux_essentials-2.4/copying
$ cp files directories
cp: omitting directory 'files'
```

As you can see, `cp` by default only works on individual files. To copy a directory, you use the `-r` option. Keep in mind that the `-r` option will cause `cp` to also copy the contents of the directory you are copying:

```
$ cp -r files directories
$ find
.
./files
./files/nsswitch.conf
./files/fstab
./files/hostname
./directories
./directories/files
./directories/files/nsswitch.conf
./directories/files/fstab
./directories/files/hostname
```

Notice how when an existing directory was used as the destination, `cp` creates a copy of the source directory inside of it? If the destination doesn't exist, it will create it and fill it with the contents of the source directory:

```
$ cp -r files files2
$ find
.
./files
./files/nsswitch.conf
./files/fstab
./files/hostname
./directories
```

```
./directories/files
./directories/files/nsswitch.conf
./directories/files/fstab
./directories/files/hostname
./files2
./files2/nsswitch.conf
./files2/fstab
./files2/hostname
```

Globbing

What is commonly referred to as globbing is a simple pattern matching language. Command line shells on Linux systems use this language to refer to groups of files whose names match a specific pattern. POSIX.1-2017 specifies the following pattern matching characters:

`*`

Matches any number of any character, including no characters

`?`

Matches any one character

`[]`

Matches a class of characters

In English, this means you can tell your shell to match a pattern instead of a literal string of text. Usually Linux users specify multiple files with a glob instead of typing out each file name. Run the following commands:

```
$ cd ~/linux_essentials-2.4/globs
$ ls
question1   question14  question2012  star10    star2002
question13  question15  question23    star1100  star2013
$ ls star1*
star10  star1100
$ ls star*
star10  star1100  star2002  star2013
$ ls star2*
star2002  star2013
$ ls star2*2
star2002
$ ls star2013*
```

```
star2013
```

The shell expands `*` to any number of anything, so your shell interprets `star*` to mean anything in the relevant context that starts with `star`. When you run the command `ls star*`, your shell doesn't run the `ls` program with an argument of `star*`, it looks for files in the current directory that match the pattern `star*` (including just `star`), and turns each file matching the pattern into an argument to `ls`:

```
$ ls star*
```

as far as `ls` is concerned is the equivalent of

```
$ ls star10  star1100  star2002  star2013
```

The `*` character doesn't mean anything to `ls`. To prove this, run the following command:

```
$ ls star\*
ls: cannot access star*: No such file or directory
```

When you precede a character with a `\`, you are instructing your shell not to interpret it. In this case, you want `ls` to have an argument of `star*` instead of what the glob `star*` expands to.

The `?` expands to any single character. Try the following commands to see for yourself:

```
$ ls
question1   question14  question2012  star10    star2002
question13  question15  question23    star1100  star2013
$ ls question?
question1
$ ls question1?
question13  question14  question15
$ ls question?3
question13  question23
$ ls question13?
ls: cannot access question13?: No such file or directory
```

The `[]` brackets are used to match ranges or classes of characters. The `[]` brackets work like they do in POSIX regular expressions except with globs the `^` is used instead of `!`.

Create some files to experiment with:

```
$ mkdir brackets
$ cd brackets
$ touch file1 file2 file3 file4 filea fileb filec file5 file6 file7
```

Ranges within `[]` brackets are expressed using a `-`:

```
$ ls
file1  file2  file3  file4  file5  file6  file7  filea  fileb  filec
$ ls file[1-2]
file1  file2
$ ls file[1-3]
file1  file2  file3
```

Multiple ranges can be specified:

```
$ ls file[1-25-7]
file1  file2  file5  file6  file7
$ ls file[1-35-6a-c]
file1  file2  file3  file5  file6  filea  fileb  filec
```

Square brackets can also be used to match a specific set of characters.

```
$ ls file[1a5]
file1  file5  filea
```

You can also use the `^` character as the first character to match everything except certain characters.

```
$ ls file[^a]
file1  file2  file3  file4  file5  file6  file7  fileb  filec
```

The last thing we will cover in this lesson is character classes. To match a character class, you use `[:classname:]`. For example, to use the digit class, which matches numerals, you would do something like this:

```
$ ls file[[:digit:]]
```

```
file1  file2  file3  file4  file5  file6  file7
$ touch file1a file11
$ ls file[[:digit:]a]
file1  file2  file3  file4  file5  file6  file7  filea
$ ls file[[:digit:]]a
file1a
```

The glob `file[[:digit:]a]`, matches `file` followed by a digit or `a`.

The character classes supported depends on your current locale. POSIX requires the following character classes for all locales:

`[:alnum:]`
: Letters and numbers.

`[:alpha:]`
: Upper or lowercase letters.

`[:blank:]`
: Spaces and tabs.

`[:cntrl:]`
: Control characters, e.g. backspace, bell, NAK, escape.

`[:digit:]`
: Numerals (`0123456789`).

`[:graph:]`
: Graphic characters (all characters except `ctrl` and the space character)

`[:lower:]`
: Lowercase letters (`a-z`).

`[:print:]`
: Printable characters (`alnum`, `punct`, and the space character).

`[:punct:]`
: Punctuation characters, i.e. `!`, `&`, `"`.

`[:space:]`
: Whitespace characters, e.g. tabs, spaces, newlines.

`[:upper:]`

Uppercase letters (`A-Z`).

`[:xdigit:]`

Hexadecimal numerals (usually `0123456789abcdefABCDEF`).

Guided Exercises

1. Given the following, select the directories that would be created by the command `mkdir -p /tmp/outfiles/text/today /tmp/infiles/text/today`

```
$ pwd
/tmp
$ find
.
./outfiles
./outfiles/text
```

`/tmp`	
`/tmp/outfiles`	
`/tmp/outfiles/text`	
`/tmp/outfiles/text/today`	
`/tmp/infiles`	
`/tmp/infiles/text`	
`/tmp/infiles/text/today`	

2. What does `-v` do for `mkdir`, `rm`, and `cp`?

3. What happens if you accidentally attempt to copy three files on the same command line to a file that already exists instead of a directory?

4. What happens when you use `mv` to move a directory into itself?

5. How would you delete all files in your current directory that start with `old`?

6. Which of the following files would `log_[a-z]_201?_*_01.txt` match?

`log_3_2017_Jan_01.txt`	
`log_+_2017_Feb_01.txt`	

`log_b_2007_Mar_01.txt`	
`log_f_201A_Wednesday_01.txt`	

7. Create a few globs to match the following list of file names:

```
doc100
doc200
doc301
doc401
```

Explorational Exercises

1. Use the `cp` man page to find out how to make a copy of a file and have the permissions and modification time match the original.

2. What does the `rmdir -p` command do? Experiment with it and explain how it differs from `rm -r`.

3. DO NOT ACTUALLY EXECUTE THIS COMMAND: What do you think `rm -ri /*` will do? (HONESTLY, DO NOT ATTEMPT TO DO THIS!)

4. Other than using `-i`, is it possible to prevent `mv` from overwriting destination files?

5. Explain the command `cp -u`.

Summary

The Linux command line environment provides tools to manage files. Some commonly used ones are `cp`, `mv`, `mkdir`, `rm`, and `rmdir`. These tools, combined with globs, allow users to get a lot of work done very quickly.

Many commands have a `-i` option, which prompts you before doing anything. Prompting can save you a lot of hassle if you mistyped something.

A lot of commands have a `-r` option. The `-r` option usually means recursion. In mathematics and computer science, a recursive function is a function using itself in its definition. When it comes to command line tools, it usually means apply the command to a directory and everything in it.

Commands used in this lesson:

`cat`
: Read and output the contents of a file.

`cp`
: Copy files or directories.

`echo`
: Output a string.

`find`
: Traverse a file system tree and search for files matching a specific set of criteria.

`ls`
: Show properties of files and directories and list a directory's contents.

`mkdir`
: Create new directories.

`mv`
: Move or rename files or directories.

`pwd`
: Output the current working directory.

`rm`
: Delete files or directories.

`rmdir`

Delete directories.

`touch`

Create new empty files or update an existing file's modification timestamp.

Answers to Guided Exercises

1. Given the following, select the directories that would be created by the command `mkdir -p /tmp/outfiles/text/today /tmp/infiles/text/today`

```
$ pwd
/tmp
$ find
.
./outfiles
./outfiles/text
```

   The marked directories would be created. The directories `/tmp`, `/tmp/outfiles`, and `/tmp/outfiles/text` already exist, so `mkdir` will ignore them.

`/tmp`	
`/tmp/outfiles`	
`/tmp/outfiles/text`	
`/tmp/outfiles/text/today`	X
`/tmp/infiles`	X
`/tmp/infiles/text`	X
`/tmp/infiles/text/today`	X

2. What does `-v` do for `mkdir`, `rm`, and `cp`?

   Typically `-v` turns on verbose output. It causes the respective programs to output what they are doing as they are doing it:

```
$ rm -v a b
removed 'a'
removed 'b'
$ mv -v a b
'a' -> 'b'
$ cp -v b c
'b' -> 'c'
```

3. What happens if you accidentally attempt to copy three files on the same command line to a file that already exists instead of a directory?

`cp` will refuse to do anything and output an error message:

```
$ touch a b c d
$ cp a b c d
cp: target 'd' is not a directory
```

4. What happens when you use `mv` to move a directory into itself?

   You will get an error message telling you `mv` cannot do that.

```
$ mv a a
mv: cannot move 'a' to a subdirectory of itself, 'a/a'
```

5. How would you delete all files in your current directory that start with `old`?

   You would use the glob `old*` with `rm`:

```
$ rm old*
```

6. Which of the following files would `log_[a-z]_201?_*_01.txt` match?

`log_3_2017_Jan_01.txt`	
`log_+_2017_Feb_01.txt`	
`log_b_2007_Mar_01.txt`	
`log_f_201A_Wednesday_01.txt`	X

```
$ ls log_[a-z]_201?_*_01.txt
log_f_201A_Wednesday_01.txt
```

   `log_[a-z]` matches `log_` followed by any lower case letter, so both `log_f_201A_Wednesday_01.txt` and `log_b_2007_Mar_01.txt` match. `_201?` matches any single character, so only `log_f_201A_Wednesday_01.txt` matches. Finally `*_01.txt` matches anything that ends with `_01.txt`, so our remaining option matches.

7. Create a few globs to match the following list of file names:

```
doc100
doc200
```

```
doc301
doc401
```

There are several solutions. Here are some of them:

```
doc*
doc[1-4]*
doc?0?
doc[1-4]0?
```

Answers to Explorational Exercises

1. Use the `cp` man page to find out how to make a copy of a file and have the permissions and modification time match the original.

   You would use the `-p` option. From the man page:

   ```
   $ man cp
   -p     same as --preserve=mode,ownership,timestamps
   --preserve[=ATTR_LIST]
                preserve the specified attributes (default: mode,ownership,time-
                stamps), if  possible  additional  attributes:  context,  links,
                xattr, all
   ```

2. What does the `rmdir -p` option do? Experiment with it and explain how it differs from `rm -r`.

   It causes `rmdir` to behave similarly to `mkdir -p`. If passed a tree of empty directories, it will remove all of them.

   ```
   $ find
   .
   ./a
   ./a/b
   ./a/b/c
   $ rmdir -p a/b/c
   $ ls
   ```

3. DO NOT ACTUALLY EXECUTE THIS COMMAND: What do you think `rm -ri /*` will do? (HONESTLY, DO NOT ATTEMPT TO DO THIS!)

   It will remove all files and directories writable by your user account. This includes any network file systems.

4. Other than using `-i`, is it possible to prevent `mv` from overwriting destination files?

   Yes, the `-n` or `--no-clobber` option prevents `mv` from overwriting files.

   ```
   $ cat a
   a
   $ cat b
   b
   ```

```
$ mv -n a b
$ cat b
b
```

5. Explain `cp -u`.

   The `-u` option causes `cp` to only copy a file if the destination is missing or is older than the source file.

```
$ ls -l
total 24K
drwxr-xr-x 123 emma student  12K Feb  2 05:34 ..
drwxr-xr-x   2 emma student 4.0K Feb  2 06:56 .
-rw-r--r--   1 emma student    2 Feb  2 06:56 a
-rw-r--r--   1 emma student    2 Feb  2 07:00 b
$ cat a
a
$ cat b
b
$ cp -u a b
$ cat b
b
$ cp -u a c
$ ls -l
total 12
-rw-r--r-- 1 emma student 2 Feb  2 06:56 a
-rw-r--r-- 1 emma student 2 Feb  2 07:00 b
-rw-r--r-- 1 emma student 2 Feb  2 07:00 c
```

Linux Professional Institute

Topic 3: The Power of the Command Line

3.1 Archiving Files on the Command Line

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 3.1

Weight

2

Key knowledge areas

- Files, directories
- Archives, compression

Partial list of the used files, terms and utilities

- `tar`
- Common `tar` options
- `gzip`, `bzip2`, `xz`
- `zip`, `unzip`

3.1 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	3 The Power of the Command Line
Objective:	3.1 Archiving Files on the Command Line
Lesson:	1 of 1

Introduction

Compression is used to reduce the amount of space a specific set of data consumes. Compression is commonly used for reducing the amount of space that is needed to store a file. Another common use is to reduce the amount of data sent over a network connection.

Compression works by replacing repetitive patterns in data. Suppose you have a novel. Some words are extremely common but have multiple characters, such as the word “the”. You could reduce the size of the novel significantly if you were to replace these common multi character words and patterns with single character replacements. E.g., replace “the” with a greek letter that is not used elsewhere in the text. Data compression algorithms are similar to this but more complex.

Compression comes in two varieties, *lossless* and *lossy*. Things compressed with a lossless algorithm can be decompressed back into their original form. Data compressed with a lossy algorithm cannot be recovered. Lossy algorithms are often used for images, video, and audio where the quality loss is imperceptible to humans, irrelevant to the context, or the loss is worth the saved space or network throughput.

Archiving tools are used to bundle up files and directories into a single file. Some common uses are backups, bundling software source code, and data retention.

Archive and compression are commonly used together. Some archiving tools even compress their contents by default. Others can optionally compress their contents. A few archive tools must be used in conjunction with stand-alone compression tools if you wish to compress the contents.

The most common tool for archiving files on Linux systems is `tar`. Most Linux distributions ship with the GNU version of `tar`, so it is the one that will be covered in this lesson. `tar` on its own only manages the archiving of files but does not compress them.

There are lots of compression tools available on Linux. Some common lossless ones are `bzip2`, `gzip`, and `xz`. You will find all three on most systems. You may encounter an old or very minimal system where `xz` or `bzip` is not installed. If you become a regular Linux user, you will likely encounter files compressed with all three of these. All three of them use different algorithms, so a file compressed with one tool can't be decompressed by another. Compression tools have a trade off. If you want a high compression ratio, it will take longer to compress and decompress the file. This is because higher compression requires more work finding more complex patterns. All of these tools compress data but can not create archives containing multiple files.

Stand-alone compression tools aren't typically available on Windows systems. Windows archiving and compression tools are usually bundled together. Keep this in mind if you have Linux and Windows systems that need to share files.

Linux systems also have tools for handling `.zip` files commonly used on Windows system. They are called `zip` and `unzip`. These tools are not installed by default on all systems, so if you need to use them you may have to install them. Fortunately, they are typically found in distributions' package repositories.

Compression Tools

How much disk space is saved by compressing files depends on a few factors. The nature of the data you are compressing, the algorithm used to compress the data, and the compression level. Not all algorithms support different compression levels.

Let's start with setting up some test files to compress:

```
$ mkdir ~/linux_essentials-3.1
$ cd ~/linux_essentials-3.1
$ mkdir compression archiving
$ cd compression
```

```
$ cat /etc/* > bigfile 2> /dev/null
```

Now we create three copies of this file:

```
$ cp bigfile bigfile2
$ cp bigfile bigfile3
$ cp bigfile bigfile4
$ ls -lh
total 2.8M
-rw-r--r-- 1 emma emma 712K Jun 23 08:08 bigfile
-rw-r--r-- 1 emma emma 712K Jun 23 08:08 bigfile2
-rw-r--r-- 1 emma emma 712K Jun 23 08:08 bigfile3
-rw-r--r-- 1 emma emma 712K Jun 23 08:08 bigfile4
```

Now we are going to compress the files with each aforementioned compression tool:

```
$ bzip2 bigfile2
$ gzip bigfile3
$ xz bigfile4
$ ls -lh
total 1.2M
-rw-r--r-- 1 emma emma 712K Jun 23 08:08 bigfile
-rw-r--r-- 1 emma emma 170K Jun 23 08:08 bigfile2.bz2
-rw-r--r-- 1 emma emma 179K Jun 23 08:08 bigfile3.gz
-rw-r--r-- 1 emma emma 144K Jun 23 08:08 bigfile4.xz
```

Compare the sizes of the compressed files to the uncompressed file named `bigfile`. Also notice how the compression tools added extensions to the file names and removed the uncompressed files.

Use `bunzip2`, `gunzip`, or `unxz` to decompress the files:

```
$ bunzip2 bigfile2.bz2
$ gunzip bigfile3.gz
$ unxz bigfile4.xz
$ ls -lh
total 2.8M
-rw-r--r-- 1 emma emma 712K Jun 23 08:20 bigfile
-rw-r--r-- 1 emma emma 712K Jun 23 08:20 bigfile2
-rw-r--r-- 1 emma emma 712K Jun 23 08:20 bigfile3
```

```
-rw-r--r-- 1 emma emma 712K Jun 23 08:20 bigfile4
```

Notice again that now the compressed file is deleted once it is decompressed.

Some compression tools support different compression levels. A higher compression level usually requires more memory and CPU cycles, but results in a smaller compressed file. The opposite is true for a lower level. Below is a demonstration with `xz` and `gzip`:

```
$ cp bigfile bigfile-gz1
$ cp bigfile bigfile-gz9
$ gzip -1 bigfile-gz1
$ gzip -9 bigfile-gz9
$ cp bigfile bigfile-xz1
$ cp bigfile bigfile-xz9
$ xz -1 bigfile bigfile-xz1
$ xz -9 bigfile bigfile-xz9
$ ls -lh bigfile bigfile-* *
total 3.5M
-rw-r--r-- 1 emma emma 712K Jun 23 08:08 bigfile
-rw-r--r-- 1 emma emma 205K Jun 23 13:14 bigfile-gz1.gz
-rw-r--r-- 1 emma emma 178K Jun 23 13:14 bigfile-gz9.gz
-rw-r--r-- 1 emma emma 156K Jun 23 08:08 bigfile-xz1.xz
-rw-r--r-- 1 emma emma 143K Jun 23 08:08 bigfile-xz9.xz
```

It is not necessary to decompress a file every time you use it. Compression tools typically come with special versions of common tools used to read text files. For example, `gzip` has a version of `cat`, `grep`, `diff`, `less`, `more`, and a few others. For `gzip`, the tools are prefixed with a `z`, while the prefix `bz` exists for `bzip2` and `xz` exists for `xz`. Below is an example of using `zcat` to read display a file compressed with `gzip`:

```
$ cp /etc/hosts ./
$ gzip hosts
$ zcat hosts.gz
127.0.0.1   localhost

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Archiving Tools

The `tar` program is probably the most widely used archiving tool on Linux systems. In case you are wondering why it is named how it is, it as an abbreviation for “tape archive”. Files created with `tar` are often called *tar balls*. It is very common for applications distributed as source code to be in tar balls.

The GNU version of `tar` that Linux distributions ship with has a lot of options. This lesson is going to cover the most commonly used subset.

Let’s start off by creating an archive of the files used for compression:

```
$ cd ~/linux_essentials-3.1
$ tar cf archiving/3.1.tar compression
```

The `c` option instructs `tar` to create a new archive file and the `f` option is the name of the file to create. The argument immediately following the options is always going to be the name of the file to work on. The rest of the arguments are the paths to any files or directories you wish to add to, list, or extract from the file. In the example, we are adding the directory `compression` and all of its contents to the archive.

To view the contents of a tar ball, use the `t` option of `tar`:

```
$ tar -tf 3.1.tar
compression/
compression/bigfile-xz1.xz
compression/bigfile-gz9.gz
compression/hosts.gz
compression/bigfile2
compression/bigfile
compression/bigfile-gz1.gz
compression/bigfile-xz9.xz
compression/bigfile3
compression/bigfile4
```

Notice how the options are preceded with `-`. Unlike most programs, with `tar`, the `-` isn’t required when specifying options, although it doesn’t cause any harm if it is used.

NOTE You can use the `-v` option to let `tar` output the names of files it operates on when creating or extracting an archive.

Now let's extract the file:

```
$ cd ~/linux_essentials-3.1/archiving
$ ls
3.1.tar
$ tar xf 3.1.tar
$ ls
3.1.tar  compression
```

Suppose you only need one file out of the archive. If this is the case, you can specify it after the archive's file name. You can specify multiple files if necessary:

```
$ cd ~/linux_essentials-3.1/archiving
$ rm -rf compression
$ ls
3.1.tar
$ tar xvf 3.1.tar compression/hosts.gz
compression/
compression/bigfile-xz1.xz
compression/bigfile-gz9.gz
compression/hosts.gz
compression/bigfile2
compression/bigfile
compression/bigfile-gz1.gz
compression/bigfile-xz9.xz
compression/bigfile3
compression/bigfile4
$ ls
3.1.tar  compression
$ ls compression
hosts.gz
```

With the exception of absolute paths (paths beginning with `/`), `tar` files preserve the entire path to files when they are created. Since the file `3.1.tar` was created with a single directory, that directory will be created relative to your current working directory when extracted. Another example should clarify this:

```
$ cd ~/linux_essentials-3.1/archiving
$ rm -rf compression
$ cd ../compression
$ tar cf ../tar/3.1-nodir.tar *
```

```
$ cd ../archiving
$ mkdir untar
$ cd untar
$ tar -xf ../3.1-nodir.tar
$ ls
bigfile   bigfile3  bigfile-gz1.gz  bigfile-xz1.xz  hosts.gz
bigfile2  bigfile4  bigfile-gz9.gz  bigfile-xz9.xz
```

TIP If you wish to use the absolute path in a `tar` file, you must use the `P` option. Be aware that this may overwrite important files and might cause errors on your system.

The `tar` program can also manage compression and decompression of archives on the fly. `tar` does so by calling one of the compression tools discussed earlier in this section. It is as simple as adding the option appropriate to the compression algorithm. The most commonly used ones are `j`, `J`, and `z` for `bzip2`, `xz`, and `gzip`, respectively. Below are examples using the aforementioned algorithms:

```
$ cd ~/linux_essentials-3.1/compression
$ ls
bigfile   bigfile3  bigfile-gz1.gz  bigfile-xz1.xz  hosts.gz
bigfile2  bigfile4  bigfile-gz9.gz  bigfile-xz9.xz
$ tar -czf gzip.tar.gz bigfile bigfile2 bigfile3
$ tar -cjf bzip2.tar.bz2 bigfile bigfile2 bigfile3
$ tar -cJf xz.tar.xz bigfile bigfile2 bigfile3
$ ls -l | grep tar
-rw-r--r-- 1 emma emma  450202 Jun 27 05:56 bzip2.tar.bz2
-rw-r--r-- 1 emma emma  548656 Jun 27 05:55 gzip.tar.gz
-rw-r--r-- 1 emma emma  147068 Jun 27 05:56 xz.tar.xz
```

Notice how in the example the `.tar` files have different sizes. This shows that they were successfully compressed. If you create compressed `.tar` archives, you should always add a second file extension denoting the algorithm you used. They are `.xz`, `.bz`, and `.gz` for `xz`, `bzip2`, and `gzip`, respectively. Sometimes shortened extensions such as `.tgz` are used.

It is possible to add files to already existing uncompressed tar archives. Use the `u` option to do this. If you attempt to add to a compressed archive, you will get an error.

```
$ cd ~/linux_essentials-3.1/compression
$ ls
bigfile   bigfile3  bigfile-gz1.gz  bigfile-xz1.xz  bzip2.tar.bz2  hosts.gz
bigfile2  bigfile4  bigfile-gz9.gz  bigfile-xz9.xz  gzip.tar.gz    xz.tar.xz
```

```
$ tar cf plain.tar bigfile bigfile2 bigfile3
$ tar tf plain.tar
bigfile
bigfile2
bigfile3
$ tar uf plain.tar bigfile4
$ tar tf plain.tar
bigfile
bigfile2
bigfile3
bigfile4
$ tar uzf gzip.tar.gz bigfile4
tar: Cannot update compressed archives
Try 'tar --help' or 'tar --usage' for more information.
```

Managing ZIP files

Windows machines often don't have applications to handle tar balls or many of the compression tools commonly found on Linux systems. If you need to interact with Windows systems, you can use ZIP files. A ZIP file is an archive file similar to a compressed `tar` file.

The `zip` and `unzip` programs can be used to work with ZIP files on Linux systems. The example below should be all you need to get started using them. First we create a set of files:

```
$ cd ~/linux_essentials-3.1
$ mkdir zip
$ cd zip/
$ mkdir dir
$ touch dir/file1 dir/file2
```

Now we use `zip` to pack these files into a ZIP file:

```
$ zip -r zipfile.zip dir
  adding: dir/ (stored 0%)
  adding: dir/file1 (stored 0%)
  adding: dir/file2 (stored 0%)
$ rm -rf dir
```

Finally, we unpack the ZIP file again:

```
$ ls
zipfile.zip
$ unzip zipfile.zip
Archive:  zipfile.zip
   creating: dir/
 extracting: dir/file1
 extracting: dir/file2
$ find
.
./zipfile.zip
./dir
./dir/file1
./dir/file2
```

When adding directories to ZIP files, the `-r` option causes `zip` to include a directory's contents. Without it, you would have an empty directory in the ZIP file.

Guided Exercises

1. According to the extensions, which of the following tools were used to create these files?

Filename	`tar`	`gzip`	`bzip2`	`xz`
`archive.tar`				
`archive.tgz`				
`archive.tar.xz`				

2. According to the extensions, which of these files are archives and which are compressed?

Filename	**Archive**	**Compressed**
`file.tar`		
`file.tar.bz2`		
`file.zip`		
`file.xz`		

3. How would you add a file to a `gzip` compressed `tar` file?

4. Which `tar` option instructs `tar` to include the leading `/` in absolute paths?

5. Does `zip` support different compression levels?

Explorational Exercises

1. When extracting files, does `tar` support globs in the file list?

2. How can you make sure a decompressed file is identical to the file before it was compressed?

3. What happens if you try to extract a file from a `tar` archive that already exists on your filesystem?

4. How would you extract the file `archive.tgz` without using the `tar z` option?

Summary

Linux systems have several compression and archiving tools available. This lesson covered the most common ones. The most common archiving tool is `tar`. If interacting with Windows systems is necessary, `zip` and `unzip` can create and extract ZIP files.

The `tar` command has a few options that are worth memorizing. They are `x` for extract, `c` for create, `t` for view contents, and `u` to add or replace files. The `v` option lists the files which are processed by `tar` while creating or extracting an archive.

The typical Linux distribution's repository has many compression tools. The most common are `gzip`, `bzip2`, and `xz`. Compression algorithms often support different levels that allow you to optimize for speed or file size. Files can be decompressed with `gunzip`, `bunzip2`, and `unxz`.

Compression tools commonly have programs that behave like common text file tools, with the difference being they work on compressed files. A few of them are `zcat`, `bzcat`, and `xzcat`. Compression tools typically ship with programs with the functionality of `grep`, `more`, `less`, `diff`, and `cmp`.

Commands used in the exercises:

`bunzip2`

Decompress a `bzip2` compressed file.

`bzcat`

Output the contents of a `bzip` compressed file.

`bzip2`

Compress files using the `bzip2` algorithm and format.

`gunzip`

Decompress a `gzip` compressed file.

`gzip`

Compress files using the `gzip` algorithm and format.

`tar`

Create, update, list and extract `tar` archives.

`unxz`

Decompress a `xz` compressed file.

`unzip`

Decompress and extract content from a ZIP file.

`xz` Compress files using the `xz` algorithm and format.

`zcat`

Output the contents of a `gzip` compressed file.

`zip`

Create and compress ZIP archives.

Answers to Guided Exercises

1. According to the extensions, which of the following tools were used to create these files?

Filename	`tar`	`gzip`	`bzip2`	`xz`
`archive.tar`	X			
`archive.tgz`	X	X		
`archive.tar.xz`	X			X

2. According to the extensions, which of these files are archives and which are compressed?

Filename	**Archive**	**Compressed**
`file.tar`	X	
`file.tar.bz2`	X	X
`file.zip`	X	X
`file.xz`		X

3. How would you add a file to a `gzip` compressed `tar` file?

   You would decompress the file with `gunzip`, add the file with `tar uf`, and then compress it with `gzip`

4. Which `tar` option instructs `tar` to include the leading `/` in absolute paths?

   The `-P` option. From the man page:

```
-P, --absolute-names
       Don't strip leading slashes from file names when creating  archives
```

5. Does `zip` support different compression levels?

   Yes. You would use `-#`, replacing `#` with a number from 0-9. From the man page:

```
-#
(-0, -1, -2, -3, -4, -5, -6, -7, -8, -9)
       Regulate the speed of compression using the specified  digit  #,
       where  -0  indicates  no compression (store all files), -1 indi-
       cates the fastest compression speed (less  compression)  and  -9
```

```
       indicates  the  slowest  compression speed (optimal compression,
       ignores the suffix list). The default compression level is -6.

       Though still being worked, the intention is  this  setting  will
       control  compression  speed  for  all compression methods.  Cur-
       rently only deflation is controlled.
```

Answers to Explorational Exercises

1. When extracting files, does `tar` support globs in the file list?

   Yes, you would use the `--wildcards` option. `--wildcards` must be placed right after the `tar` file when using the no dash style of options. For example:

   ```
   $ tar xf tarfile.tar --wildcards dir/file*
   $ tar --wildcards -xf tarfile.tar dir/file*
   ```

2. How can you make sure a decompressed file is identical to the file before it was compressed?

   You don't need to do anything with the tools covered in this lesson. All three of them include checksums in their file format that is verified when they are decompressed.

3. What happens if you try to extract a file from a `tar` archive that already exists on your filesystem?

   The file on your filesystem is overwritten with the version that is in the `tar` file.

4. How would you extract the file `archive.tgz` without using the `tar` `z` option?

   You would decompress it with `gunzip` first.

   ```
   $ gunzip archive.tgz
   $ tar xf archive.tar
   ```

3.2 Searching and Extracting Data from Files

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 3.2

Weight

3

Key Knowledge Areas

- Command line pipes
- I/O redirection
- Basic Regular Expressions using `.`, `[ ]`, `*`, and `?`

Partial list of the used files, terms and utilities

- `grep`
- `less`
- `cat`, `head`, `tail`
- `sort`
- `cut`
- `wc`

3.2 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	3 The Power of the Command Line
Objective:	3.2 Searching and Extracting Data from Files
Lesson:	1 of 2

Introduction

In this lab we will be focusing on redirecting or transmitting information from one source to another with the help of specific tools. The Linux command line redirects the information through specific standard channels. The standard input (*stdin* or channel 0) of the command is considered to be the keyboard and the standard output (*stdout* or channel 1) is considered the screen. There is also another channel that is meant to redirect error output (*stderr* or channel 2) of a command or a program's error messages. The input and/or output can be redirected.

When running a command, sometimes we want to transmit certain information to the command or redirect the output to a specific file. Each of these functionalities will be discussed in the next two sections.

I/O Redirection

I/O redirection enables the user to redirect information from or to a command by using a text file. As described earlier, the standard input, output and error output can be redirected, and the information can be taken from text files.

Redirecting Standard Output

To redirect standard output to a file, instead of the screen, we need to use the `>` operator followed by the name of the file. If the file doesn't exist, a new one will be created, otherwise, the information will overwrite the existing file.

In order to see the contents of the file that we just created, we can use the `cat` command. By default, this command displays the contents of a file on the screen. Consult the manual page to find out more about its functionalities.

The example below demonstrates the functionality of the operator. In the first instance, a new file is created containing the text "Hello World!":

```
$ echo "Hello World!" > text
$ cat text
Hello World!
```

In the second invocation, the same file is overwritten with the new text:

```
$ echo "Hello!" > text
$ cat text
Hello!
```

If we want to add new information at the end of the file, we need to use the `>>` operator. This operator also creates a new file if it cannot find an existing one.

The first example shows the addition of the text. As it can be seen, the new text was added on the following line:

```
$ echo "Hello to you too!" >> text
$ cat text
Hello!
Hello to you too!
```

The second example demonstrates that a new file will be created:

```
$ echo "Hello to you too!" >> text2
$ cat text2
Hello to you too!
```

Redirecting Standard Error

In order to redirect just the error messages, a user will need to employ the `2>` operator followed by the name of the file in which the errors will be written. If the file doesn't exist, a new one will be created, otherwise the file will be overwritten.

As explained, the channel for redirecting the standard error is *channel 2*. When redirecting the standard error, the channel must be specified, contrary to the other standard output where *channel 1* is set by default. For example, the following command searches for a file or directory named `games` and only writes the error into the `text-error` file, while displaying the standard output on the screen:

```
$ find /usr games 2> text-error
/usr
/usr/share
/usr/share/misc
---------Omitted output----------
/usr/lib/libmagic.so.1.0.0
/usr/lib/libdns.so.81
/usr/games
$ cat text-error
find: `games': No such file or directory
```

NOTE For more information about the `find` command, consult its man page.

For example, the following command will run without errors, therefore no information will be written in the file `text-error`:

```
$ sort /etc/passwd 2> text-error
$ cat text-error
```

As well as the standard output, the standard error can also be appended to a file with the `2>>` operator. This will add the new error at the end of the file. If the file doesn't exist, a new one will be created. The first example shows the addition of the new information into the file, whereas the second example shows that the command creates a new file where an existing one can't be found with the same name:

```
$ sort /etc 2>> text-error
$ cat text-error
sort: read failed: /etc: Is a directory
```

```
$ sort /etc/shadow 2>> text-error2
$ cat text-error2
sort: open failed: /etc/shadow: Permission denied
```

Using this type of redirection, only the error messages will be redirected to the file, the normal output will be written on the screen or go through standard output or *stdout*.

There is one particular file that technically is a *bit bucket* (a file that accepts input and doesn't do anything with it): `/dev/null`. You can redirect any irrelevant information that you might not want displayed or redirected into an important file, as shown in the example below:

```
$ sort /etc 2> /dev/null
```

Redirecting Standard Input

This type of redirection is used to input data to a command, from a specified file instead of a keyboard. In this case the `<` operator is used as shown in the example:

```
$ cat < text
Hello!
Hello to you too!
```

Redirecting standard input is usually used with commands that don't accept file arguments. The `tr` command is one of them. This command can be used to translate file contents by modifying the characters in a file in specific ways, like deleting any particular character from a file, the example below shows the deletion of the character `l`:

```
$ tr -d "l" < text
Heo!
Heo to you too!
```

For more information, consult the man page of `tr`.

Here Documents

Unlike the output redirections, the `<<` operator acts in a different way compared to the other operators. This input stream is also called *here document*. *Here document* represents the block of code or text which can be redirected to the command or the interactive program. Different types

of scripting languages, like `bash`, `sh` and `csh` are able to take input directly from the command line, without using any text files.

As can be seen in the example below, the operator is used to input data into the command, while the word after doesn't specify the file name. The word is interpreted as the delimiter of the input and it will not be taken in consideration as content, therefore `cat` will not display it:

```
$ cat << hello
> hey
> ola
> hello
hey
ola
```

Consult the man page of the `cat` command to find more information.

Combinations

The first combination that we will explore combines the redirection of the standard output and standard error output to the same file. The `&>` and `&>>` operators are used, `&` representing the combination of *channel 1* and *channel 2*. The first operator will overwrite the existing contents of the file and the second one will append or add the new information at the end of the file. Both operators will enable the creation of the new file if it doesn't exist, just like in the previous sections:

```
$ find /usr admin &> newfile
$ cat newfile
/usr
/usr/share
/usr/share/misc
---------Omitted output----------
/usr/lib/libmagic.so.1.0.0
/usr/lib/libdns.so.81
/usr/games
find: `admin': No such file or directory
$ find /etc/calendar &>> newfile
$ cat newfile
/usr
/usr/share
/usr/share/misc
---------Omitted output----------
/usr/lib/libmagic.so.1.0.0
```

```
/usr/lib/libdns.so.81
/usr/games
find: `admin': No such file or directory
/etc/calendar
/etc/calendar/default
```

Let's take a look at an example using the `cut` command:

```
$ cut -f 3 -d "/" newfile
$ cat newfile

share
share
share
---------Omitted output----------
lib
games
find: `admin': No such file or directory
calendar
calendar
find: `admin': No such file or directory
```

The `cut` command cuts specified fields from the input file by using the `-f` option, the 3rd field in our case. In order for the command to find the field, a delimiter needs to be specified as well with the `-d` option. In our case the delimiter will be the `/` character.

To find more about the `cut` command, consult its man page.

Command Line Pipes

Redirection is mostly used to store the result of a command, to be processed by a different command. This type of intermediate process can become very tedious and complicated if you want the data to go through multiple processes. In order to avoid this, you can link the command directly via *pipes*. In other words, the first command's output automatically becomes the second command's input. This connection is made by using the `|` (vertical bar) operator:

```
$ cat /etc/passwd | less
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
:
```

In the example above, the `less` command after the pipe operator modifies the way that the file is displayed. The `less` command displays the text file allowing the user to scroll up and down a line at the time. `less` is also used by default to display the man pages, as discussed in the previous lessons.

It is possible to use multiple pipes at the same time. The intermediate commands that receive input then change it and produce output are called *filters*. Let's take the `ls -l` command and try to count the number of words from the first 10 lines of the output. In order to do this, we will have to use the `head` command that by default displays the first 10 lines of a file and then count the words using the `wc` command:

```
$ ls -l | head | wc -w
10
```

As mentioned earlier, by default, `head` only displays the first 10 lines of the text file specified. This behaviour can be modified by using specific options. Check the command's man page to find more.

There is another command that displays the end of a file: `tail`. By default, this command selects the last 10 lines and displays them, but as `head` the number can also be modified. Check `tail`'s man page for more details.

NOTE The `-f` option can show the last lines of a file while it's being updated. This feature can become very useful when monitoring a file like `syslog` for ongoing activity.

The `wc` (word count) command counts by default the lines, words and bytes of a file. As shown in the exercise, the `-w` option causes the command to only count the words within the selected lines. The most common options that you can use with this command are: `-l`, which specifies to the command to only count the lines, and `-c`, which is used to count only the bytes. More variations and options of the command, as well as more information about `wc` can be found within the command's man page.

Guided Exercises

1. List the contents of your current directory, including the ownership and permissions, and redirect the output to a file called `contents.txt` within your home directory.

2. Sort the contents of the `contents.txt` file from your current directory and append it to the end of a new file named `contents-sorted.txt`.

3. Display the last 10 lines of the `/etc/passwd` file and redirect it to a new file in your user's `Documents` directory.

4. Count the number of words within the `contents.txt` file and append the output to the end of a file `field2.txt` in your home directory. You will need to use both input and output redirection.

5. Display the first 5 lines of the `/etc/passwd` file and sort the output reverse alphabetically.

6. Using the previously created `contents.txt` file, count the number of characters of the last 9 lines.

7. Count the number of files called `test` within the `/usr/share` directory and its subdirectories. Note: each line output from the `find` command represents a file.

Explorational Exercises

1. Select the second field of the `contents.txt` file and redirect the standard output and error output to another file called `field1.txt`.

2. Using the input redirection operator and the `tr` command, delete the dashes (`-`) from the `contents.txt` file.

3. What is the biggest advantage of only redirecting errors to a file?

4. Replace all recurrent spaces within the alphabetically sorted `contents.txt` file with a single space.

5. In one command line, eliminate the recurrent spaces (as done in the previous exercise), select the ninth field and sort it reverse alphabetically and non-case sensitive. How many pipes did you have to use?

Summary

In this lab you learned:

- Types of redirection
- How to use the redirection operators
- How to use pipes to filter command output

Commands used in this lesson:

`cut`
Removes sections from each line of a file.

`cat`
Displays or concatenates files.

`find`
Searches for files in a directory hierarchy.

`less`
Displays a file, allowing the user to scroll one line at the time.

`more`
Displays a file, a page at the time.

`head`
Displays the first 10 lines of a file.

`tail`
Displays the last 10 lines of a file.

`sort`
Sorts files.

`wc`
Counts by default the lines, words or bytes of a file.

Answers to Guided Exercises

1. List the contents of your current directory, including the ownership and permissions, and redirect the output to a file called `contents.txt` within your home directory.

```
$ ls -l > contents.txt
```

2. Sort the contents of the `contents.txt` file from your current directory and append it to the end of a new file named `contents-sorted.txt`.

```
$ sort contents.txt >> contents-sorted.txt
```

3. Display the last 10 lines of the `/etc/passwd` file and redirect it to a new file in the your user's `Documents` directory.

```
$ tail /etc/passwd > Documents/newfile
```

4. Count the number of words within the `contents.txt` file and append the output to the end of a file `field2.txt` in your home directory. You will need to use both input and output redirection.

```
$ wc -w < contents.txt >> field2.txt
```

5. Display the first 5 lines of the `/etc/passwd` file and sort the output reverse alphabetically.

```
$ head -n 5 /etc/passwd | sort -r
```

6. Using the previously created `contents.txt` file, count the number of characters of the last 9 lines.

```
$ tail -n 9 contents.txt | wc -c
531
```

7. Count the number of files called `test` within the `/usr/share` directory and its subdirectories. Note: each line output from the `find` command represents a file.

```
$ find /usr/share -name test | wc -l
125
```

Answers to Explorational Exercises

1. Select the second field of the `contents.txt` file and redirect the standard output and error output to another file called `field1.txt`.

```
$ cut -f 2 -d " " contents.txt &> field1.txt
```

2. Using the input redirection operand and the `tr` command, delete the dashes (`-`) from the `contents.txt` file.

```
$ tr -d "-" < contents.txt
```

3. What is the biggest advantage of only redirecting errors to a file?

   Only redirecting errors to a file can help with keeping a log file that is monitored frequently.

4. Replace all recurrent spaces within the alphabetically sorted `contents.txt` file with a single space.

```
$ sort contents.txt | tr -s " "
```

5. In one command line, eliminate the recurrent spaces (as done in the previous exercise), select the ninth field and sort it reverse alphabetically and non-case sensitive. How many pipes did you have to use?

```
$ cat contents.txt | tr -s " " | cut -f 9 -d " " | sort -fr
```

   The exercise uses 3 pipes, one for each filter.

3.2 Lesson 2

Certificate:	Linux Essentials
Version:	1.6
Topic:	3 The Power of the Command Line
Objective:	3.2 Searching and Extracting Data from Files
Lesson:	2 of 2

Introduction

In this lesson, we are going to look at tools that are used to manipulate text. These tools are frequently used by system administrators or programs to automatically monitor or identify specific recurring information.

Searching within Files with `grep`

The first tool that we will discuss in this lesson is the `grep` command. `grep` is the abbreviation of the phrase “global regular expression print” and its main functionality is to search within files for the specified pattern. The command outputs the line containing the specified pattern highlighted in red.

```
$ grep bash /etc/passwd
root:x:0:0:root:/root:/bin/bash
user:x:1001:1001:User,,,,:/home/user:/bin/bash
```

`grep`, as most commands, can also be tweaked by using options. Here are the most common ones:

`-i`
: the search is case insensitive

`-r`
: the search is recursive (it searches into all files within the given directory and its subdirectories)

`-c`
: the search counts the number of matches

`-v`
: invert the match, to print lines that do not match the search term

`-E`
: turns on extended regular expressions (needed by some of the more advanced meta-characters like `|` , `+` and `?`)

`grep` has many other useful options. Consult the man page to find out more about it.

Regular Expressions

The second tool is very powerful. It is used to describe bits of text within files, also called *regular expressions*. Regular expressions are extremely useful in extracting data from text files by constructing patterns. They are commonly used within scripts or when programming with high level languages, such as Perl or Python.

When working with regular expressions, it is very important to keep in mind that *every character counts* and the pattern is written with the purpose of matching a specific sequence of characters, known as a string. Most patterns use the normal ASCII symbols, such as letters, digits, punctuation or other symbols, but it can also use Unicode characters in order to match any other type of text.

The following list explains the regular expressions meta-characters that are used to form the patterns.

`.`
: Match any single character (except newline)

`[abcABC]`
: Match any one character within the brackets

`[^abcABC]`

Match any one character except the ones in the brackets

`[a-z]`

Match any character in the range

`[^a-z]`

Match any character except the ones in the range

`sun|moon`

Find either of the listed strings

`^`

Start of a line

`$`

End of a line

All functionalities of the regular expressions can be implemented through `grep` as well. You can see that in the example above, the word is not surrounded by double quotes. To prevent the shell from interpreting the meta-character itself, it is recommended that the more complex pattern be between double quotes (" "). For the purpose of practice, we will be using double quotes when implementing regular expressions. The other quotation marks keep their normal functionality, as discussed in previous lessons.

The following examples emphasize the functionality of the regular expressions. We will need data within the file, therefore the next set of commands just appends different strings to the `text.txt` file.

```
$ echo "aaabbb1" > text.txt
$ echo "abab2" >> text.txt
$ echo "noone2" >> text.txt
$ echo "class1" >> text.txt
$ echo "alien2" >> text.txt
$ cat text.txt
aaabbb1
abab2
noone2
class1
alien2
```

The first example is a combination of searching through the file without and with regular expressions. In order to fully understand regular expressions, it is very important to show the difference. The first command searches for the exact string, anywhere in the line, whereas the second command searches for sets of characters that contain any of the characters between the brackets. Therefore, the results of the commands are different.

```
$ grep "ab" text.txt
aaabbb1
abab2
$ grep "[ab]" text.txt
aaabbb1
abab2
class1
alien2
```

The second set of examples shows the application of the beginning and the end of the line meta-character. It is very important to specify the need to put the 2 characters at the right place in the expression. When specifying the beginning of the line, the meta-character needs to be before the expression, whereas, when specifying the end of the line, the meta-character needs to be after the expression.

```
$ grep "^a" text.txt
aaabbb1
abab2
alien2
$ grep "2$" text.txt
abab2
noone2
alien2
```

On top of the previous explained meta-characters, regular expressions also have meta-characters that enable multiplication of the previously specified pattern:

`*`

Zero or more of the preceding pattern

`+`

One or more of the preceding pattern

`?`

Zero or one of the preceding pattern

For the multiplier meta-characters, the command below searches for a string that contains `ab`, a single character and one or more of the characters previously found. The result shows that `grep` found the `aaabbb1` string, matching the `abbb` part as well as `abab2`. Since the `+` character is an *extended* regular expression character, we need to pass the `-E` option to the `grep` command.

```
$ grep -E "ab.+" text.txt
aaabbb1
abab2
```

Most of the meta-characters are self-explanatory, but they can become tricky when used for the first time. The previous examples represent a small part of the regular expressions' functionality. Try all meta-characters from the above table to understand more on how they work.

Guided Exercises

Using `grep` and the `/usr/share/hunspell/en_US.dic` file, find the lines that match the following criteria:

1. All lines containing the word `cat` anywhere on the line.

2. All lines that do not contain any of the following characters: `sawgtfixk`.

3. All lines that start with any 3 letters and the word `dig`.

4. All lines that end with at least one `e`.

5. All lines that contain one of the following words: `org` , `kay` or `tuna`.

6. Number of lines that start with one or no `c` followed by the string `ati`.

Explorational Exercises

1. Find the regular expression that matches the words in the “Include” line and doesn’t match the ones in the “Exclude” line:
   - Include: `pot`, `spot`, `apot`

     Exclude: `potic`, `spots`, `potatoe`

   - Include: `arp99`, `apple`, `zipper`

     Exclude: `zoo`, `arive`, `attack`

   - Include: `arcane`, `capper`, `zoology`

     Exclude: `air`, `coper`, `zoloc`

   - Include: `0th/pt`, `3th/tc`, `9th/pt`

     Exclude: `0/nm`, `3/nm`, `9/nm`

   - Include: `Hawaii`, `Dario`, `Ramiro`

     Exclude: `hawaii`, `Ian`, `Alice`

2. What other useful command is commonly used to search within the files? What additional functionalities does it have?
3. Thinking back at the previous lesson, use one of the examples and try to look for a specific pattern within the output of the command, with the help of `grep`.

Summary

In this lab you learned:

- Regular expressions meta-characters
- How to create patterns with regular expressions
- How to search within the files

Commands used in the exercises:

`grep`
: Searches for characters or strings within a file

Answers to Guided Exercises

Using `grep` and the `/usr/share/hunspell/en_US.dic` file, find the lines that match the following criteria:

1. All lines containing the word `cat` anywhere on the line.

```
$ grep "cat" /usr/share/hunspell/en_US.dic
Alcatraz/M
Decatur/M
Hecate/M
...
```

2. All lines that do not contain any of the following characters: `sawgtfixk`.

```
$ grep -v "[sawgtfixk]" /usr/share/hunspell/en_US.dic
49269
0/nm
1/n1
2/nm
2nd/p
3/nm
3rd/p
4/nm
5/nm
6/nm
7/nm
8/nm
...
```

3. All lines that start with any 3 letters and the word `dig`.

```
$ grep "^...dig" /usr/share/hunspell/en_US.dic
cardigan/SM
condign
predigest/GDS
...
```

4. All lines that end with at least one `e`.

```
$ grep -E "e+$" /usr/share/hunspell/en_US.dic
Anglicize
Anglophobe
Anthropocene
...
```

5. All lines that contain one of the following words: `org` , `kay` or `tuna`.

```
$ grep -E "org|kay|tuna" /usr/share/hunspell/en_US.dic
Borg/SM
George/MS
Tokay/M
fortunate/UY
...
```

6. Number of lines that start with one or no `c` followed by the string `ati`.

```
$ grep -cE "^c?ati" /usr/share/hunspell/en_US.dic
3
```

Answers to Explorational Exercises

1. Find the regular expression that matches the words in the “Include” line and doesn’t match the ones in the “Exclude” line:
   - Include: `pot`, `spot`, `apot`

     Exclude: `potic`, `spots`, `potatoe`

     Answer: `pot$`
   - Include: `arp99`, `apple`, `zipper`

     Exclude: `zoo`, `arive`, `attack`

     Answer: `p+`
   - Include: `arcane`, `capper`, `zoology`

     Exclude: `air`, `coper`, `zoloc`

     Answer: `arc|cap|zoo`
   - Include: `0th/pt`, `3th/tc`, `9th/pt`

     Exclude: `0/nm`, `3/nm`, `9/nm`

     Answer: `[0-9]th.+`
   - Include: `Hawaii`, `Dario`, `Ramiro`

     Exclude: `hawaii`, `Ian`, `Alice`

     Answer: `^[A-Z]a.*i+`
2. What other useful command is commonly used to search within the files? What additional functionalities does it have?

   The `sed` command. The command can find and replace characters or sets of characters within a file.
3. Thinking back at the previous lesson, use one of the examples and try to look for a specific pattern within the output of the command, with the help of `grep`.

   I took one of the answers from the Explorational Exercises and looked for the line that has

read, write and execute as the group permissions. Your answer might be different, depending on the command that you chose and the pattern that you created.

```
$ cat contents.txt | tr -s " " | grep "^....rwx"
```

This exercise is to show you that `grep` can also receive input from different commands and it can help in filtering generated information.

3.3 Turning Commands into a Script

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 3.3

Weight

4

Key knowledge areas

- Basic shell scripting
- Awareness of common text editors (vi and nano)

Partial list of the used files, terms and utilities

- `#!` (shebang)
- `/bin/bash`
- Variables
- Arguments
- `for` loops
- `echo`
- Exit status

3.3 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	3 The Power of the Command Line
Objective:	3.3 Turning Commands into a Script
Lesson:	1 of 2

Introduction

We have been learning to execute commands from the shell thus far, but we can also enter commands into a file, and then set that file to be executable. When the file is executed, those commands are run one after the other. These executable files are called *scripts*, and they are an absolutely crucial tool for any Linux system administrator. Essentially, we can consider Bash to be a programming language as well as a shell.

Printing Output

Let's begin by demonstrating a command that you may have seen in previous lessons: `echo` will print an argument to standard output.

```
$ echo "Hello World!"
Hello World!
```

Now, we will use file redirection to send this command to a new file called `new_script`.

```
$ echo 'echo "Hello World!"' > new_script
$ cat new_script
echo "Hello World!"
```

The file `new_script` now contains the same command as before.

Making a Script Executable

Let's demonstrate some of the steps required to make this file execute the way we expect it to. A user's first thought might be to simply type the name of the script, the way they might type in the name of any other command:

```
$ new_script
/bin/bash: new_script: command not found
```

We can safely assume that `new_script` exists in our current location, but notice that the error message isn't telling us that the *file* doesn't exist, it is telling us that the *command* doesn't exist. It would be useful to discuss how Linux handles commands and executables.

Commands and `PATH`

When we type the `ls` command into the shell, for example, we are executing a file called `ls` that exists in our filesystem. You can prove this by using `which`:

```
$ which ls
/bin/ls
```

It would quickly become tiresome to type in the absolute path of `ls` every time we wish to look at the contents of a directory, so Bash has an *environment variable* which contains all the directories where we might find the commands we wish to run. You can view the contents of this variable by using `echo`.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/sn
ap/bin
```

Each of these locations is where the shell expects to find a command, delimited with colons (`:`). You will notice that `/bin` is present, but it is safe to assume that our current location is not. The

shell will search for `new_script` in each of these directories, but it will not find it and therefore will throw the error we saw above.

There are three solutions to this issue: we can move `new_script` into one of the `PATH` directories, we can add our current directory to `PATH`, or we can change the way we attempt to call the script. The latter solution is easiest, it simply requires us to specify the *current location* when calling the script using dot slash (`./`).

```
$ ./new_script
/bin/bash: ./new_script: Permission denied
```

The error message has changed, which indicates that we have made some progress.

Execute Permissions

The first investigation a user should do in this case is to use `ls -l` to look at the file:

```
$ ls -l new_script
-rw-rw-r-- 1 user user 20 Apr 30 12:12 new_script
```

We can see that the permissions for this file are set to `664` by default. We have not set this file to have *execute permissions* yet.

```
$ chmod +x new_script
$ ls -l new_script
-rwxrwxr-x 1 user user 20 Apr 30 12:12 new_script
```

This command has given execute permissions to *all* users. Be aware that this might be a security risk, but for now this is an acceptable level of permission.

```
$ ./new_script
Hello World!
```

We are now able to execute our script.

Defining the Interpreter

As we have demonstrated, we were able to simply enter text into a file, set it as an executable, and

run it. `new_script` is functionally still a normal text file, but we managed to have it be interpreted by Bash. But what if it is written in Perl, or Python?

It is very good practice to specify the type of interpreter we want to use in the first line of a script. This line is called a *bang line* or more commonly a *shebang*. It indicates to the system how we want this file to be executed. Since we are learning Bash, we will be using the absolute path to our Bash executable, once again using `which`:

```
$ which bash
/bin/bash
```

Our shebang starts with a hash sign and exclamation mark, followed by the absolute path above. Let's open `new_script` in a text editor and insert the shebang. Let's also take the opportunity to insert a *comment* into our script. Comments are ignored by the interpreter. They are written for the benefit of other users wishing to understand your script.

```
#!/bin/bash

# This is our first comment. It is also good practice to document all scripts.

echo "Hello World!"
```

We will make one additional change to the filename as well: we will save this file as `new_script.sh`. The file suffix ".sh" does not change the execution of the file in any way. It is a convention that bash scripts be labelled with `.sh` or `.bash` in order to identify them more easily, the same way that Python scripts are usually identified with the suffix `.py`.

Common Text Editors

Linux users often have to work in an environment where graphical text editors are not available. It is therefore highly recommended to develop at least some familiarity with editing text files from the command line. Two of the most common text editors are `vi` and `nano`.

vi

`vi` is a venerable text editor and is installed by default on almost every Linux system in existence. `vi` spawned a clone called *vi IMproved* or `vim` which adds some functionality but maintains the interface of `vi`. While working with `vi` is daunting for a new user, the editor is popular and well-loved by users who learn its many features.

The most important difference between `vi` and applications such as Notepad is that `vi` has three different modes. On startup, the keys `H`, `J`, `K` and `L` are used to navigate, not to type. In this *navigation mode*, you can press `I` to enter *insert mode*. At this point, you may type normally. To exit *insert mode*, you press `Esc` to return to *navigation mode*. From *navigation mode*, you can press `:` to enter *command mode*. From this mode, you can save, delete, quit or change options.

While `vi` has a learning curve, the different modes can in time allow a savvy user to become more efficient than with other editors.

nano

`nano` is a newer tool, built to be simple and easier to use than `vi`. `nano` does not have different modes. Instead, a user on startup can begin typing, and uses `Ctrl` to access the tools printed at the bottom of the screen.

```
                        [ Welcome to nano.  For basic help, type Ctrl+G. ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line M-E Redo
```

Text editors are a matter of personal preference, and the editor that you choose to use will have no bearing on this lesson. But becoming familiar and comfortable with one or more text editors will pay off in the future.

Variables

Variables are an important part of any programming language, and Bash is no different. When you start a new session from the terminal, the shell already sets some variables for you. The `PATH` variable is an example of this. We call these *environment variables*, because they usually define characteristics of our shell environment. You can modify and add environment variables, but for now let's focus on setting variables inside our script.

We will modify our script to look like this:

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=Carol

echo "Hello $username!"
```

In this case, we have created a *variable* called `username` and we have assigned it the *value* of `Carol`. Please note that there are no spaces between the variable name, the equals sign, or the assigned value.

In the next line, we have used the `echo` command with the variable, but there is a dollar sign (`$`) in front of the variable name. This is important, since it indicates to the shell that we wish to treat `username` as a variable, and not just a normal word. By entering `$username` in our command, we indicate that we want to perform a *substitution*, replacing the *name* of a variable with the *value* assigned to that variable.

Executing the new script, we get this output:

```
$ ./new_script.sh
Hello Carol!
```

- Variable names must contain only alphanumeric characters or underscores, and are case sensitive. `Username` and `username` will be treated as separate variables.
- Variable substitution may also have the format `${username}`, with the addition of the `{ }`. This is also acceptable.
- Variables in Bash have an *implicit type*, and are considered strings. This means that performing math functions in Bash is more complicated than it would be in other programming languages such as C/C++:

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=Carol
x=2
y=4
z=$x+$y
echo "Hello $username!"
echo "$x + $y"
echo "$z"
```

```
$ ./new_script.sh
Hello Carol!
2 + 4
2+4
```

Using Quotes with Variables

Let's make the following change to the value of our variable `username`:

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=Carol Smith

echo "Hello $username!"
```

Running this script will give us an error:

```
$ ./new_script.sh
./new_script.sh: line 5: Smith: command not found
Hello !
```

Keep in mind that Bash is an interpreter, and as such it *interprets* our script line-by-line. In this case, it correctly interprets `username=Carol` to be setting a variable `username` with the value `Carol`. But it then interprets the space as indicating the end of that assignment, and `Smith` as being the name of a command. In order to have the space and the name `Smith` be included as the new value of our variable, we will put double quotes (`"`) around the name.

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username="Carol Smith"

echo "Hello $username!"
```

```
$ ./new_script.sh
Hello Carol Smith!
```

One important thing to note in Bash is that double quotes and single quotes (`'`) behave very differently. Double quotes are considered "weak", because they allow the interpreter to perform substitution inside the quotes. Single quotes are considered "strong", because they prevent any substitution from occurring. Consider the following example:

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username="Carol Smith"

echo "Hello $username!"
echo 'Hello $username!'
```

```
$ ./new_script.sh
Hello Carol Smith!
Hello $username!
```

In the second `echo` command, the interpreter has been prevented from substituting `$username` with `Carol Smith`, and so the output is taken literally.

Arguments

You are already familiar with using arguments in the Linux core utilities. For example, `rm testfile` contains both the executable `rm` and one argument `testfile`. Arguments can be passed to the script upon execution, and will modify how the script behaves. They are easily implemented.

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=$1

echo "Hello $username!"
```

Instead of assigning a value to `username` directly inside the script, we are assigning it the value of a new variable `$1`. This refers to the value of the *first argument*.

```
$ ./new_script.sh Carol
Hello Carol!
```

The first nine arguments are handled in this way. There are ways to handle more than nine arguments, but that is outside the scope of this lesson. We will demonstrate an example using just

two arguments:

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username1=$1
username2=$2
echo "Hello $username1 and $username2!"
```

```
$ ./new_script.sh Carol Dave
Hello Carol and Dave!
```

There is an important consideration when using arguments: In the example above, there are two arguments `Carol` and `Dave`, assigned to `$1` and `$2` respectively. If the second argument is missing, for example, the shell will not throw an error. The value of `$2` will simply be *null*, or nothing at all.

```
$ ./new_script.sh Carol
Hello Carol and !
```

In our case, it would be a good idea to introduce some logic to our script so that different *conditions* will affect the *output* that we wish to print. We will start by introducing another helpful variable and then move on to creating *if statements*.

Returning the Number of Arguments

While variables such as `$1` and `$2` contain the value of positional arguments, another variable `$#` contains the *number of arguments*.

```
#!/bin/bash

# This is our first comment. It is also good practice to comment all scripts.

username=$1

echo "Hello $username!"
echo "Number of arguments: $#."
```

```
$ ./new_script.sh Carol Dave
Hello Carol!
Number of arguments: 2.
```

Conditional Logic

The use of conditional logic in programming is a vast topic, and won't be covered deeply in this lesson. We will focus on the *syntax* of conditionals in Bash, which differs from most other programming languages.

Let's begin by reviewing what we hope to achieve. We have a simple script which should be able to print a greeting to a single user. If there is anything other than one user, we should print an error message.

- The *condition* we are testing is the number of users, which is contained in the variable `$#`. We would like to know if the value of `$#` is `1`.
- If the condition is *true*, the *action* we will take is to greet the user.
- If the condition is *false*, we will print an error message.

Now that the logic is clear, we will focus on the *syntax* required to implement this logic.

```
#!/bin/bash

# A simple script to greet a single user.

if [ $# -eq 1 ]
then
  username=$1

  echo "Hello $username!"
else
  echo "Please enter only one argument."
fi
echo "Number of arguments: $#."
```

The conditional logic is contained between `if` and `fi`. The condition to test is located between square brackets `[ ]`, and the action to take should the condition be true is indicated after `then`. Note the spaces between the square brackets and the logic contained. Omitting this space will cause errors.

This script will output either our greeting, *or* the error message. But it will always print the `Number of arguments` line.

```
$ ./new_script.sh
Please enter only one argument.
Number of arguments: 0.
$ ./new_script.sh Carol
Hello Carol!
Number of arguments: 1.
```

Take note of the `if` statement. We have used `-eq` to do a *numerical comparison*. In this case, we are testing that the value of `$#` is *equal* to one. The other comparisons we can perform are:

`-ne`
: Not equal to

`-gt`
: Greater than

`-ge`
: Greater than or equal to

`-lt`
: Less than

`-le`
: Less than or equal to

Guided Exercises

1. The user types the following to their shell:

```
$ PATH=~/scripts
$ ls
Command 'ls' is available in '/bin/ls'
The command could not be located because '/bin' is not included in the PATH environment
variable.
ls: command not found
```

   - What has the user done?

   - What command will combine the current value of `PATH` with the new directory `~/scripts`?

2. Consider the following script. Notice that it is using `elif` to check for a second condition:

```
>  /!bin/bash

> fruit1 = Apples
> fruit2 = Oranges

  if [ $1 -lt $# ]
  then
    echo "This is like comparing $fruit1 and $fruit2!"
> elif [$1 -gt $2 ]
  then
>   echo '$fruit1 win!'
  else
>   echo "Fruit2 win!"
> done
```

   - The lines marked with a `>` contain errors. Fix the errors.

3. What will the output be in the following situations?

```
$ ./guided1.sh 3 0
```

```
$ ./guided1.sh 2 4
```

```
$ ./guided1.sh 0 1
```

Explorational Exercises

1. Write a simple script that will check if exactly two arguments are passed. If so, print the arguments in reverse order. Consider this example (note: your code may look different than this, but should lead to the same output):

```
if [ $1 == $number ]
then
  echo "True!"
fi
```

2. This code is correct, but it is not a number comparison. Use an internet search to discover how this code is different from using `-eq`.

3. There is an environment variable that will print the current directory. Use `env` to discover the name of this variable.

4. Using what you have learned in questions 2 and 3, write a short script that accepts an argument. If an argument is passed, check if that argument matches the name of the current directory. If so, print `yes`. Otherwise, print `no`.

Summary

In this section, you learned:

- How to create and execute simple scripts
- How to use a shebang to specify an interpreter
- How to set and use variables inside scripts
- How to handle arguments in scripts
- How to construct `if` statements
- How to compare numbers using numerical operators

Commands used in the exercises:

`echo`
: Print a string to standard output.

`env`
: Prints all environment variables to standard output.

`which`
: Prints the absolute path of a command.

`chmod`
: Changes permissions of a file.

Special variables used in the exercises:

`$1, $2, … $9`
: Contain positional arguments passed to the script.

`$#`
: Contains the number of arguments passed to the script.

`$PATH`
: Contains the directories that have executables used by the system.

Operators used in the exercises:

`-ne`

Not equal to

`-gt`

Greater than

`-ge`

Greater than or equal to

`-lt`

Less than

`-le`

Less than or equal to

Answers to Guided Exercises

1. The user types the following into their shell:

```
$ PATH=~/scripts
$ ls
Command 'ls' is available in '/bin/ls'
The command could not be located because '/bin' is not included in the PATH environment
variable.
ls: command not found
```

   - What has the user done?

     The user has overwritten the contents of PATH with the directory `~/scripts`. The `ls` command can no longer be found, since it isn't contained in PATH. Note that this change only affects the current session, logging out and back in with revert the change.

   - What command will combine the current value of `PATH` with the new directory `~/scripts`?

     `PATH=$PATH:~/scripts`

2. Consider the following script. Notice that it is using `elif` to check for a second condition:

```
>  /!bin/bash

> fruit1 = Apples
> fruit2 = Oranges

  if [ $1 -lt $# ]
  then
    echo "This is like comparing $fruit1 and $fruit2!"
> elif [$1 -gt $2 ]
  then
>   echo '$fruit1 win!'
  else
>   echo "Fruit2 win!"
> done
```

   - The lines marked with a `>` contain errors. Fix the errors.

```
#!/bin/bash
```

```
fruit1=Apples
fruit2=Oranges

if [ $1 -lt $# ]
then
  echo "This is like comparing $fruit1 and $fruit2!"
elif [ $1 -gt $2 ]
then
  echo "$fruit1 win!"
else
  echo "$fruit2 win!"
fi
```

3. What will the output be in the following situations?

```
$ ./guided1.sh 3 0
```

`Apples win!`

```
$ ./guided1.sh 2 4
```

`Oranges win!`

```
$ ./guided1.sh 0 1
```

`This is like comparing Apples and Oranges!`

Answers to Explorational Exercises

1. Write a simple script that will check if exactly two arguments are passed. If so, print the arguments in reverse order. Consider this example (note: your code may look different than this, but should lead to the same output):

```
if [ $1 == $number ]
then
  echo "True!"
fi
```

```
#!/bin/bash

if [ $# -ne 2 ]
then
  echo "Error"
else
  echo "$2 $1"
fi
```

2. This code is correct, but it is not a number comparison. Use an internet search to discover how this code is different from using `-eq`.

   Using `==` will compare *strings*. That is, if the characters of both variables match up exactly, then the condition is true.

`abc == abc`	*true*
`abc == ABC`	*false*
`1 == 1`	*true*
`1+1 == 2`	*false*

   String comparisons lead to unexpected behavior if you are testing for numbers.

3. There is an environment variable that will print the current directory. Use `env` to discover the name of this variable.

   `PWD`

4. Using what you have learned in questions 2 and 3, write a short script that accepts an

argument. If an argument is passed, check if that argument matches the name of the current directory. If so, print `yes`. Otherwise, print `no`.

```
#!/bin/bash

if [ "$1" == "$PWD" ]
then
  echo "yes"
else
  echo "no"
fi
```

3.3 Lesson 2

Certificate:	Linux Essentials
Version:	1.6
Topic:	3 The Power of the Command Line
Objective:	3.3 Turning Commands into a Script
Lesson:	2 of 2

Introduction

In the last section, we used this simple example to demonstrate Bash scripting:

```
#!/bin/bash

# A simple script to greet a single user.

if [ $# -eq 1 ]
then
  username=$1

  echo "Hello $username!"
else
  echo "Please enter only one argument."
fi
echo "Number of arguments: $#."
```

- All scripts should begin with a *shebang*, which defines the path to the interpreter.

- All scripts should include comments to describe their use.
- This particular script works with an *argument*, which is passed to the script when it is called.
- This script contains an *if statement*, which tests the conditions of a built-in variable `$#`. This variable is set to the number of arguments.
- If the number of arguments passed to the script equals 1, then the value of the first argument is passed to a new variable called `username` and the script echoes a greeting to the user. Otherwise, an error message is displayed.
- Finally, the script echoes the number of arguments. This is useful for debugging.

This is a useful example to begin explaining some of the other features of Bash scripting.

Exit Codes

You will notice that our script has two possible states: either it prints `"Hello <user>!"` or it prints an error message. This is quite normal for many of our core utilities. Consider `cat`, which you are no doubt becoming very familiar with.

Let's compare a successful use of `cat` with a situation where it fails. A reminder that our example above is a script called `new_script.sh`.

```
$ cat -n new_script.sh

     1  #!/bin/bash
     2
     3  # A simple script to greet a single user.
     4
     5  if [ $# -eq 1 ]
     6  then
     7    username=$1
     8
     9    echo "Hello $username!"
    10  else
    11    echo "Please enter only one argument."
    12  fi
    13  echo "Number of arguments: $#."
```

This command succeeds, and you will notice that the `-n` flag has also printed line numbers. These are very helpful when debugging scripts, but please note that they *are not* part of the script.

Now we are going to check the value of a new built-in variable `$?`. For now, just notice the output:

```
$ echo $?
0
```

Now let's consider a situation where `cat` will fail. First we will see an error message, and then check the value of `$?`.

```
$ cat -n dummyfile.sh
cat: dummyfile.sh: No such file or directory
$ echo $?
1
```

The explanation for this behaviour is this: any execution of the `cat` utility will return an *exit code*. An exit code will tell us if the command succeeded, or experienced an error. An exit code of *zero* indicates that the command completed successfully. This is true for almost every Linux command that you work with. Any other exit code will indicate an error of some kind. The exit code of the *last command to run* will be stored in the variable `$?`.

Exit codes are usually not seen by human users, but they are very useful when writing scripts. Consider a script where we may be copying files to a remote network drive. There are many ways that the copy task may have failed: for example our local machine might not be connected to the network, or the remote drive might be full. By checking the exit code of our copy utility, we can alert the user to problems when running the script.

It is very good practice to implement exit codes, so we will do this now. We have two paths in our script, a success and a failure. Let's use zero to indicate success, and one to indicate failure.

```
 1  #!/bin/bash
 2
 3  # A simple script to greet a single user.
 4
 5  if [ $# -eq 1 ]
 6  then
 7    username=$1
 8
 9    echo "Hello $username!"
10    exit 0
11  else
12    echo "Please enter only one argument."
13    exit 1
14  fi
```

```
    15  echo "Number of arguments: $#."
```

```
$ ./new_script.sh Carol
Hello Carol!
$ echo $?
0
```

Notice that the `echo` command on line 15 was ignored entirely. Using `exit` will end the script immediately, so this line is never encountered.

Handling Many Arguments

So far our script can only handle a single username at a time. Any number of arguments besides one will cause an error. Let's explore how we can make this script more versatile.

A user's first instinct might be to use more positional variables such as `$2`, `$3` and so on. Unfortunately, we can't anticipate the number of arguments that a user might choose to use. To solve this issue, it will be helpful to introduce more built-in variables.

We will modify the logic of our script. Having zero arguments should cause an error, but any other number of arguments should be successful. This new script will be called `friendly2.sh`.

```
 1  #!/bin/bash
 2
 3  # a friendly script to greet users
 4
 5  if [ $# -eq 0 ]
 6  then
 7    echo "Please enter at least one user to greet."
 8    exit 1
 9  else
10    echo "Hello $@!"
11    exit 0
12  fi
```

```
$ ./friendly2.sh Carol Dave Henry
Hello Carol Dave Henry!
```

There are two built-in variables which contain all arguments passed to the script: `$@` and `$*`. For

the most part, both behave the same. Bash will *parse* the arguments, and separate each argument when it encounters a space between them. In effect, the contents of `$@` look like this:

`0`	`1`	`2`
`Carol`	`Dave`	`Henry`

If you are familiar with other programming languages, you might recognize this type of variable as an *array*. Arrays in Bash can be created simply by putting space between elements like the variable `FILES` in script `arraytest` below:

```
FILES="/usr/sbin/accept /usr/sbin/pwck/ usr/sbin/chroot"
```

It contains a list of many items. So far this isn't very helpful, because we have not yet introduced any way of handling these items individually.

For Loops

Let's refer to the `arraytest` example shown before. If you recall, in this example we are specifying an array of our own called `FILES`. What we need is a way to "unpack" this variable and access each individual value, one after the other. To do this, we will use a structure called a *for loop*, which is present in all programming languages. There are two variables that we will refer to: one is the range, and the other is for the individual value that we are currently working on. This is the script in its entirety:

```
#!/bin/bash

FILES="/usr/sbin/accept /usr/sbin/pwck/ usr/sbin/chroot"

for file in $FILES
do
  ls -lh $file
done
```

```
$ ./arraytest
lrwxrwxrwx 1 root root 10 Apr 24 11:02 /usr/sbin/accept -> cupsaccept
-rwxr-xr-x 1 root root 54K Mar 22 14:32 /usr/sbin/pwck
-rwxr-xr-x 1 root root 43K Jan 14 07:17 /usr/sbin/chroot
```

If you refer again to the `friendly2.sh` example above, you can see that we are working with a

range of values contained within a single variable `$@`. For clarity's sake, we will call the latter variable `username`. Our script now looks like this:

```
 1  #!/bin/bash
 2
 3  # a friendly script to greet users
 4
 5  if [ $# -eq 0 ]
 6  then
 7    echo "Please enter at least one user to greet."
 8    exit 1
 9  else
10    for username in $@
11    do
12      echo "Hello $username!"
13    done
14    exit 0
15  fi
```

Remember that the variable that you define here can be named whatever you wish, and that all the lines inside `do… done` will be executing once for each element of the array. Let's observe the output from our script:

```
$ ./friendly2.sh Carol Dave Henry
Hello Carol!
Hello Dave!
Hello Henry!
```

Now let's assume that we want to make our output seem a little more human. We want our greeting to be on one line.

```
 1  #!/bin/bash
 2
 3  # a friendly script to greet users
 4
 5  if [ $# -eq 0 ]
 6  then
 7    echo "Please enter at least one user to greet."
 8    exit 1
 9  else
10    echo -n "Hello $1"
```

```
11    shift
12    for username in $@
13    do
14      echo -n ", and $username"
15    done
16    echo "!"
17    exit 0
18  fi
```

A couple of notes:

- Using `-n` with `echo` will *suppress the newline* after printing. This means that all echoes will print to the same line, and the newline will be printed only after the `!`` on line 16.
- The `shift` command will remove the first element of our array, so that this:

`0`	`1`	`2`
`Carol`	`Dave`	`Henry`

Becomes this:

`0`	`1`
`Dave`	`Henry`

Let's observe the output:

```
$ ./friendly2.sh Carol
Hello Carol!
$ ./friendly2.sh Carol Dave Henry
Hello Carol, and Dave, and Henry!
```

Using Regular Expressions to Perform Error Checking

It's possible that we want to verify all arguments that the user is entering. For example, perhaps we want to ensure that all names passed to `friendly2.sh` contain *only letters*, and any special characters or numbers will cause an error. To perform this error checking, we will use `grep`.

Recall that we can use regular expressions with `grep`.

```
$ echo Animal | grep "^[A-Za-z]*$"
```

```
Animal
$ echo $?
0
```

```
$ echo 4n1ml | grep "^[A-Za-z]*$"
$ echo $?
1
```

The `^` and the `$` indicate the beginning and end of the line respectively. The `[A-Za-z]` indicates a range of letters, upper or lower case. The `*` is a *quantifier*, and modifies our range of letters so that we are matching zero to many letters. In summary, our `grep` will succeed if the input is *only* letters, and fails otherwise.

The next thing to note is that `grep` is returning exit codes based on whether there was a match or not. A positive match returns `0`, and a no match returns a `1`. We can use this to test our arguments inside our script.

```
 1  #!/bin/bash
 2
 3  # a friendly script to greet users
 4
 5  if [ $# -eq 0 ]
 6  then
 7    echo "Please enter at least one user to greet."
 8    exit 1
 9  else
10    for username in $@
11    do
12      echo $username | grep "^[A-Za-z]*$" > /dev/null
13      if [ $? -eq 1 ]
14      then
15        echo "ERROR: Names must only contains letters."
16        exit 2
17      else
18        echo "Hello $username!"
19      fi
20    done
21    exit 0
22  fi
```

On line 12, we are redirecting standard output to `/dev/null`, which is a simple way to suppress it.

We don't want to see any output from the `grep` command, we only want to test its exit code, which happens on line 13. Notice also that we are using an exit code of `2` to indicate an invalid argument. It is generally good practice to use different exit codes to indicate different errors; in this way, a savvy user can use these exit codes to troubleshoot.

```
$ ./friendly2.sh Carol Dave Henry
Hello Carol!
Hello Dave!
Hello Henry!
$ ./friendly2.sh 42 Carol Dave Henry
ERROR: Names must only contains letters.
$ echo $?
2
```

Guided Exercises

1. Read the contents of `script1.sh` below:

```
#!/bin/bash

if [ $# -lt 1 ]
then
  echo "This script requires at least 1 argument."
  exit 1
fi

echo $1 | grep "^[A-Z]*$" > /dev/null
if [ $? -ne 0 ]
then
  echo "no cake for you!"
  exit 2
fi

echo "here's your cake!"
exit 0
```

   What is the output of these commands?

   - `./script1.sh`
   - `echo $?`
   - `./script1.sh cake`
   - `echo $?`
   - `./script1.sh CAKE`
   - `echo $?`

2. Read the contents of file `script2.sh`:

```
for filename in $1/*.txt
do
   cp $filename $filename.bak
done
```

   Describe the purpose of this script as you understand it.

Explorational Exercises

1. Create a script that will take any number of arguments from the user, and print only those arguments which are numbers greater than 10.

Summary

In this section, you learned:

- What exit codes are, what they mean, and how to implement them
- How to check the exit code of a command
- What `for` loops are, and how to use them with arrays
- How to use `grep`, regular expressions and exit codes to check user input in scripts.

Commands used in the exercises:

`shift`
: This will remove the first element of an array.

Special Variables:

`$?`
: Contains the exit code of the last command executed.

`$@`, **`$*`**
: Contain all arguments passed to the script, as an array.

Answers to Guided Exercises

1. Read the contents of `script1.sh` below:

```
#!/bin/bash

if [ $# -lt 1 ]
then
  echo "This script requires at least 1 argument."
  exit 1
fi

echo $1 | grep "^[A-Z]*$" > /dev/null
if [ $? -ne 0 ]
then
  echo "no cake for you!"
  exit 2
fi

echo "here's your cake!"
exit 0
```

   What is the output of these commands?

   - Command: `./script1.sh`

     Output: `This script requires at least 1 argument.`

   - Command: `echo $?`

     Output: `1`

   - Command: `./script1.sh cake`

     Output: `no cake for you!`

   - Command: `echo $?`

     Output: `2`

   - Command: `./script1.sh CAKE`

     Output: `here's your cake!`

- Command: `echo $?`

  Output: `0`

2. Read the contents of file `script2.sh`:

```
for filename in $1/*.txt
do
   cp $filename $filename.bak
done
```

Describe the purpose of this script as you understand it.

This script will make backup copies of all files ending with `.txt` in a subdirectory defined in the first argument.

Answers to Explorational Exercises

1. Create a script that will take any number of arguments from the user, and print only those arguments that are numbers greater than 10.

```
#!/bin/bash

for i in $@
do
  echo $i | grep "^[0-9]*$" > /dev/null
  if [ $? -eq 0 ]
  then
    if [ $i -gt 10 ]
    then
      echo -n "$i "
    fi
  fi
done
echo ""
```

Linux Professional Institute

Topic 4: The Linux Operating System

4.1 Choosing an Operating System

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 4.1

Weight

1

Key knowledge areas

- Differences between Windows, OS X and Linux
- Distribution life cycle management

Partial list of the used files, terms and utilities

- GUI versus command line, desktop configuration
- Maintenance cycles, beta and stable

4.1 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	4 The Linux Operating System
Objective:	4.1 Choosing an Operating System
Lesson:	1 of 1

Introduction

No matter if you are using your computer system at home, at the university or within an enterprise, there still has to be a decision made on the operating system that you will use. This decision may be made by you, especially if it is your computer, but you may also be responsible for making the choice across systems in your business. As always, being well-informed about the choices available will aid you in making a responsible decision. In this lesson we aim to help keep you informed of the operating system choices that you may be considering.

What is an Operating System

One of the first things that we must be sure of before we begin our journey in choosing an operating system is to understand what we mean by the term. The operating system lies at the heart of your computer and allows applications to run within and on top of it. Additionally, the operating system will contain drivers to access the computer's hardware such as disks and partitions, screens, keyboards, network cards and so on. We often abbreviate the operating system to simply the *OS*. Today there are many operating systems available for both business computer usage as well as for those looking for something in the home. If we want to simplify the selection available to us, we can group selections as follows:

- Linux-based Operating Systems
  - Enterprise Linux
  - Consumer Linux
- UNIX
- macOS
- Windows-based Operation Systems
  - Windows Servers
  - Windows Desktops

Choosing a Linux Distribution

The Linux Kernel and Linux Distributions

When talking about Linux distributions the operating system is Linux. Linux is the *kernel* and at the core of every Linux distribution. The software of the Linux kernel is maintained by a group of individuals, lead by Linus Torvalds. Torvalds is employed by an industry consortium called The Linux Foundation to work on the Linux kernel.

NOTE The Linux kernel was first developed by Linus Torvalds, a student from Finland, back in 1991. In 1992, the first Kernel release under the GNU General Public License version 2 (GPLv2) was version 0.12.

Linux Kernel

As we have mentioned, all Linux distributions run the same operating system, Linux.

Linux Distribution

When people talk about Red Hat Linux, or Ubuntu Linux they are referring to the *Linux distribution*. The Linux distribution will ship with a Linux kernel and an environment that makes the kernel useful in a way that we can interact with it. At a minimum we would need a command line shell such as Bash and a set of basic commands allowing us to access and manage the system. Often, of course, the Linux distribution will have a full Desktop Environment such as Gnome or KDE.

Even though each Linux distribution runs the Linux operating system, distributions can and do vary on the version of the operating system that is used. By this, we mean, *the version of the Linux Kernel that is in use* when the distribution boots.

TIP If you have access to a Linux command line at the moment, you can easily check the

version of the Linux kernel that you are running by reading the *kernel release*:

```
$ uname -r
4.15.0-1019-aws
```

Types of Linux Distributions

It may seem an obvious choice to always run the latest version of the Linux kernel but it is not quite as simple as that. We can vaguely categorize Linux distributions into three sets:

- Enterprise Grade Linux Distributions
  - Red Hat Enterprise Linux
  - CentOS
  - SUSE Linux Enterprise Server
  - Debian GNU/Linux
  - Ubuntu LTS
- Consumer Grade Linux Distributions
  - Fedora
  - Ubuntu non-LTS
  - openSUSE
- Experimental and Hacker Linux Distributions
  - Arch
  - Gentoo

This, of course, is a just a very small subset of possible distributions but the importance is the difference between *enterprise*, *consumer* and *experimental* distributions and why each exists.

Enterprise Grade Linux

Distributions such as CentOS (*Community Enterprise OS*) are designed to be deployed within large organizations using enterprise hardware. The needs of the enterprise are very different from the needs of the small business, hobbyist or home user. In order to ensure the availability of their services, enterprise users have higher requirements regarding the stability of their hard- and software. Therefore, enterprise Linux distributions tend to include older releases of the kernel and other software, which are known to work reliably. Often the distributions port important updates like security fixes back to these stable versions. In return, enterprise Linux

distributions might lack support for the most recent consumer hardware and provide older versions of software packages. However, like consumer Linux distributions, enterprises tend to also choose mature hardware components and build their services on stable software versions.

Consumer Grade Linux

Distributions such as Ubuntu are more targeted for small business or home and hobbyist users. As such, they are also likely to be using the latest hardware found on consumer grade systems. These systems will need the latest drivers to make the most of the new hardware but the maturity of both the hardware and the drivers is unlikely to meet the needs of larger enterprises. For the consumer market, however, the latest kernel is exactly what is needed with the most modern drivers even if they are little under tested. The newer Linux kernels will have the latest drivers to support the very latest hardware that are likely to be in use. Especially with the development we see with Linux in the gaming market it is tremendously important that the latest drivers are available to these users.

NOTE

Some distributions like Ubuntu provide both consumer grade versions which contain recent software and receive updates for a rather small period of time, as well as so-called Long Term Support versions, LTS for short, which are more suited for enterprise environments.

Experimental and Hacker Linux Distributions

Distributions such as Arch Linux or Gentoo Linux live on the cutting edge of technology. They contain the most recent versions of software, even if these versions still contain bugs and untested features. In return, these distributions tend to use a rolling release model which allows them to deliver updates at any time. These distributions are used by advanced users who want to always receive the most recent software and are aware that functionality can break at any time and are able to repair their systems in such cases.

In short, when considering Linux as your operating system, if you are using enterprise grade hardware on your servers or desktops then you can make use of either enterprise grade or consumer grade Linux distributions. If you are using consumer grade hardware and need to make the most of the latest hardware innovations then you are likely to need a similar Linux distribution to match the needs of the hardware.

Some Linux distributions are related to each other. Ubuntu, for example, is based on Debian Linux and uses the same packaging system, DPKG. Fedora, as another example, is a testbed for RedHat Enterprise Linux, where potential features of future RHEL versions can be explored ahead of their availability in the enterprise distribution.

As well as the distributions we have mentioned here there are many other Linux distributions. One advantage that comes with Linux being open source software is that many people can

develop what they think Linux should look like. As such we have many hundreds of distributions. To view more Linux distributions you may choose to visit The Distro Watch Web Site, the maintainers of the website list the top 100 downloads of Linux distributions, allowing you to compare and see what is currently popular.

Linux Support Lifecycle

As you might expect, enterprise Linux distributions have a longer support life than consumer or community editions of Linux. For example Red Hat Enterprise Linux has support for 10 years. Red Hat Enterprise Linux 8 was launched in May 2019, while software updates and support are available until May 2029.

Consumer editions often will only have community support via forums. Software updates are often available for 3 releases. If we take Ubuntu as an example, at the time of writing 19.04 is the latest available having updates through the release of 19.10 and stopping in January 2020. Ubuntu also supply editions with long term support, known as *LTS* editions, which have 5 years of support from the original release. The current LTS version is 18.04 which will have software updates until 2023. These LTS versions make Ubuntu a possible option for the enterprise with commercial support available from Canonical (the company behind the Ubuntu brand) or independent consulting firms.

NOTE The Ubuntu distributions use date-based version numbers in the format YY.MM: For example, version 19.04 was released April 2019.

Linux as Your Desktop

Using Linux as your desktop system may be more challenging in an enterprise where desktop support focusses on commercial operating system offerings. However it is not only the support that may prove challenging. An enterprise customer may also have made large investments into software solutions that tie them into specific desktop operating systems. Having said this, there are many examples of Linux desktops being integrated into large organizations with companies like Amazon even having their own Linux distribution Amazon Linux 2. This is used on their AWS cloud platform but also internally for both servers and desktops.

Using Linux in a smaller business or at home becomes a lot easier and can be a rewarding experience, removing the need for licensing and opening your eyes to the wealth of free and open source software that is available for Linux. You will also find that there are many different desktop environments available. The most common being Gnome and KDE, however many others exists. The decision comes down to personal preference.

Using Linux on Servers

Using Linux as your server operating system is common practice in the enterprise sector. Servers are maintained by engineers who specialize in Linux. So even with thousands of users, the users can remain ignorant of the servers that they are connecting to. The server operating system is not important to them and, in general, client applications will not differ between Linux and other operating systems in the backend. It is also true that as more applications are virtualized or containerized within local and remote clouds, the operating system is masked even more and the embedded operating system is likely to be Linux.

Linux in the Cloud

Another opportunity to become familiar with Linux is to deploy Linux within one of the many public clouds available. Creating an account with one of the many others cloud providers will allow you to quickly deploy many different Linux distributions quickly and easily.

Non Linux Operating Systems

Yes, incredible as it seems, there are operating systems that are not based on the Linux kernel. Of course, over the years there have been many and some have fallen by the wayside but there are still other choices that are available to you. Either at home or in the office.

Unix

Before we had Linux as an operating system there was Unix. Unix used to be sold along with hardware and still today several commercial Unixes such as AIX and HP-UX are available on the market. While Linux was highly inspired by Unix (and the lack of its availability for certain hardware), the family of BSD operating systems is directly based on Unix. Today, FreeBSD, NetBSD and OpenBSD, along with some other related BSD systems, are available as free software.

Unix was heavily used in the enterprise but we have seen a decline in the fortunes of Unix with the growth of Linux. As Linux has grown and the enterprise support offerings have also grown, we have seen Unix slowly start to vanish. Solaris, originally from Sun before moving to Oracle, has recently disappeared. This was one of the larger Unix Operating Systems used by telecommunication companies, heralded as *Telco Grade Unix*.

Unix Operating Systems include:

- AIX
- FreeBSD, NetBSD, OpenBSD

- HP-UX
- Irix
- Solaris

macOS

macOS (previously OS X) from Apple dates back to 2001. Based very much on BSD Unix, and making use of the Bash command line shell, it is a friendly system to use if you are used to using Unix or Linux operating systems. If you're using macOS you can open the terminal application to access the command line. Running the same `uname` command again we can check the reported operating system:

```
$ uname -s
Darwin
```

NOTE We make use of the option `-s` in this case to return the OS name. We previously used `-r` to return the kernel version number.

Microsoft Windows

We can still say that the majority of desktops and laptops out there will be Windows based. The operating system has been truly successful and has dominated the desktop market for years. Although it is proprietary software and is not free, often the operating system license is included when you buy the hardware so it becomes the easy choice to make. There is, of course, wide support for Windows throughout hardware and software vendors as well of course many open source applications are also available for Windows. The future for Windows does not seem as bright as it has been. With fewer desktops and laptops being sold now the focus is on the tablet and phone market. This has been dominated by Apple and Android and it is hard for Microsoft to gain ground.

As a server platform Microsoft does now allow its customers to choose between a GUI (*Graphical User Interface*) and command line only version. The separation of the GUI and the command line is an important one. Most of the time the GUI of older Microsoft Servers will be loaded but no-one will use it. Consider an Active Directory Domain Controller... users use it all the time to authenticate to the domain, but it is managed remotely from administrators' desktops and not the server.

Guided Exercises

1. Which project makes up the common component of all Linux distributions?

CentOS	
Red Hat	
Ubuntu	
Linux Kernel	
CoreOS	

2. Which operating system is reported in use for macOS from Apple?

OS X	
OSX	
Darwin	
MacOS	

3. How does a Linux distribution differ from the Linux kernel?

The kernel is part of a distribution, the distribution as applications to surround the kernel to make it useful	
The kernel is the Linux distribution	
All distributions that use the same kernel are the same	

4. Which of the following is a desktop environment in Linux?

Mint	
Elementary	
Zorin	
Gnome	

5. Which component of an operating system allows access to hardware?

Drivers	
Shells	
Service	
Application	

Explorational Exercises

1. Retrieve the current Kernel release of your Linux system if you have access to the command line.

2. Using your preferred search engine locate and identify public cloud providers available to you. These could include AWS, Google Cloud, Rackspace and many more. Choose one and see which operating systems are available to deploy.

Summary

In this section you have learned how to differentiate between different operating systems commonly available. We discussed:

- Linux Based Operating Systems
- UNIX
- macOS
- Windows Based Operation Systems

Within the Linux category we could further break the selection down into distributions with long term support and those with a shorter support cycle. LTS versions being more suited to the Enterprise and shorter term support being targeted toward home and hobby users.

- Enterprise Grade Linux Distributions
  - Red Hat Enterprise Linux
  - CentOS
  - SUSE Linux Enterprise Server
  - Debian GNU/Linux
  - Ubuntu LTS
- Consumer Grade Linux Distributions
  - Fedora
  - Ubuntu non-LTS
  - openSUSE
- Experimental and Hacker Linux Distributions
  - Arch
  - Gentoo

Answers to Guided Exercises

1. Which project makes up the common component of all Linux distributions?

CentOS	
Red Hat	
Ubuntu	
Linux Kernel	X
CoreOS	

2. Which operating system is reported in use for OS X from Apple?

OS X	
OSX	
Darwin	X
MacOS	

3. How does a Linux distribution differ from the Linux kernel?

The kernel is part of a distribution, the distribution as applications to surround the kernel to make it useful	X
The kernel is the Linux distribution	
All distributions that use the same kernel are the same	

4. Which of the following is a desktop environment in Linux?

Mint	
Elementary	
Zorin	
Gnome	X

5. Which component of an operating system allows access to hardware?

Drivers	X
Shells	
Service	
Application	

Answers to Explorational Exercises

1. Retrieve the current kernel release of your Linux system if you have access to the command line.

   ```
   $ uname -r
   4.15.0-47-generic
   ```

2. Using your preferred search engine locate and identify public cloud providers available to you. These could include AWS, Google Cloud, Rackspace and many more. Choose one and see which operating systems are available to deploy.

   AWS, as an example, allows you to deploy many Linux distributions such as Debian, Red Hat, SUSE or Ubuntu as well as Windows.

4.2 Understanding Computer Hardware

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 4.2

Weight

2

Key Knowledge Areas

- Hardware

Partial list of the used files, terms and utilities

- Motherboards, processors, power supplies, optical drives, peripherals
- Hard drives, solid state disks and partitions, `/dev/sd*`
- Drivers

4.2 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	4 The Linux Operating System
Objective:	4.2 Understanding Computer Hardware
Lesson:	1 of 1

Introduction

Without hardware, software is nothing more than another form of literature. Hardware processes the commands described by the software and provides mechanisms for storage, input, and output. Even the cloud is ultimately backed by hardware.

As an operating system, one of Linux' responsibilities is providing software with interfaces to access a system's hardware. Most configuration specifics are beyond the scope of this lesson. However, users are often concerned with the performance, capacity, and other factors of system hardware since they impact the ability of a system to adequately support specific applications. This lesson discusses hardware as separate physical items using standard connectors and interfaces. The standards are relatively static. But the form factor, performance, and capacity characteristics of hardware are constantly evolving. Regardless of how changes may blur physical distinctions the conceptual aspects of hardware described in this lesson still apply.

NOTE

At various points within this lesson command line examples are used to demonstrate ways to access information about hardware. Most examples are from a Raspberry Pi B+ but should apply to most systems. Understanding these commands is not required to understand this material.

Power Supplies

All of the active components in a computer system require electricity to operate. Unfortunately, most sources of electricity are not appropriate. Computer system hardware requires specific voltages with relatively tight tolerances. Which is not what is available from your local wall outlet.

Power supplies normalize available sources of power. Standard voltage requirements allow manufacturers to create hardware components that can be used in systems anywhere in the world. Desktop power supplies tend to use the electricity from wall outlets as a source. Server power supplies tend to be more critical so they can often connect to multiple sources to assure that they continue operating if one source fails.

Consuming power generates heat. Excessive heat can cause system components to operate slowly or even fail. Most systems have some form of fan to move air for more efficient cooling. Components such as processors often generate heat that air flow alone cannot dissipate. These hot components attach special fins known as heat sinks to help dissipate the heat they generate. Heat sinks often have their own small local fan to ensure adequate air flow.

Motherboard

All of a system's hardware needs to interconnect. A motherboard normalizes that interconnection using standardized connectors and form factors. It also provides support for the configuration and electrical needs of those connectors.

There are a large number of motherboard configurations. They support different processors and memory systems. They have different combinations of standardized connectors. And they adapt to the different sizes of the packaging that contains them. Except perhaps for the ability to connect specific external devices, motherboard configuration is effectively transparent to users. Administrators are mostly exposed to motherboard configuration when there is a need to identify specific devices.

When power is first applied there is motherboard specific hardware that must be configured and initialized before the system can operate. Motherboards use programming stored in nonvolatile memory known as firmware to deal with motherboard specific hardware. The original form of motherboard firmware was known as BIOS (*Basic Input/Output System*). Beyond basic configuration settings BIOS was mostly responsible for identifying, loading, and transferring operation to an operating system such as Linux. As hardware evolved, firmware expanded to support larger disks, diagnostics, graphical interfaces, networking, and other advanced capabilities independent of any loaded operating system. Early attempts to advance firmware beyond basic BIOS was often specific to a motherboard manufacturer. Intel defined a standard for advanced firmware known as EFI (*Extensible Firmware Interface*). Intel contributed EFI to a

standards organization to create UEFI (*Unified Extensible Firmware Interface*). Today, most motherboards use UEFI. BIOS and EFI are almost never seen on recent systems. Regardless, most people still refer to motherboard firmware as BIOS.

There are very few firmware settings of interest to general users so only individuals responsible for system hardware configuration typically need to deal with firmware and its settings. One of the few commonly changed options is enabling virtualization extensions of modern CPUs.

Memory

System memory holds the data and program code of currently running applications. When they talk about computer memory most people are referring to this system memory. Another common term used for system memory is the acronym RAM (*Random Access Memory*) or some variation of that acronym. Sometimes references to the physical packaging of the system memory such as DIMM, SIMM or DDR are also used.

Physically, system memory is usually packaged on individual circuit board modules that plug into the motherboard. Individual memory modules currently range in size from 2 GB to 64 GB. For most general purpose applications 4 GB is the minimum system memory people should consider. For individual workstations 16 GB is typically more than sufficient. However even 16 GB might be limiting for users running gaming, video, or high-end audio applications. Servers often require 128 GB or even 256 GB of memory to efficiently support user loads.

For the most part Linux allows users to treat system memory as a black box. An application is started and Linux takes care of allocating the system memory required. Linux releases the memory for use by other applications when an application completes. But what if an application requires more than the available system memory? In this case, Linux moves idle applications from system memory into a special disk area known as swap space. Linux moves idle applications from the disk swap space back to system memory when they need to run.

Systems without dedicated video hardware often use a portion of the system memory (often 1 GB) to act as video display storage. This reduces the effective system memory. Dedicated video hardware typically has its own separate memory that is not available as system memory.

There are several ways to obtain information about system memory. As a user, the total amount of memory available and in use are typically the values of interest. One source of information would be to run the command `free` along with the parameter `-m` to use megabytes in the output:

```
$ free -m
               total        used        free      shared  buff/cache   available
Mem:             748          37          51          14         660         645
```

```
Swap:            99           0          99
```

The first line specifies the total memory available to the system (`total`), the memory in use (`used`) and the free memory (`free`). The second line displays this information for the swap space. Memory indicated as `shared` and `buff/cache` is currently used for other system functions, although the amount indicated in `available` could be used for application.

Processors

The word "processor" implies that something is being processed. In computers most of that processing is dealing with electrical signals. Typically those signals are treated as having one of the binary values 1 or 0.

When people talk about computers they often use the word processor interchangeably with the acronym CPU (*Central Processing Unit*). Which is not technically correct. Every general purpose computer has a CPU that processes the binary commands specified by software. So it is understandable that people interchange processor and CPU. However, in addition to a CPU, modern computers often include other task specific processors. Perhaps the most recognizable additional processor is a GPU (*Graphical Processing Unit*). Thus, while a CPU is a processor, not all processors are CPUs.

For most people CPU architecture is a reference to the instructions that the processor supports. Although Intel and AMD make processors supporting the same instructions it is meaningful to differentiate by vendor because of vendor specific packaging, performance, and power consumption differences. Software distributions commonly use these designations to specify the minimum set of instructions they require to operate:

i386

References the 32-bit instruction set associated with the Intel 80386.

x86

Typically references the 32-bit instruction sets associated with successors to the 80386 such as 80486, 80586, and Pentium.

x64 / x86-64

References processors that support both the 32-bit and 64-bit instructions of the x86 family.

AMD

A reference to x86 support by AMD processors.

AMD64

A reference to x64 support by AMD processors.

ARM

References a *Reduced Instruction Set Computer* (RISC) CPU that is not based on the x86 instruction set. Commonly used by embedded, mobile, tablet, and battery operated devices. A version of Linux for ARM is used by the Raspberry Pi.

The file `/proc/cpuinfo` contains detailed information about a system's processor. Unfortunately the details are not friendly to general users. A more general result can be obtained with the command `lscpu`. Output from a Raspberry Pi B+:

```
$ lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
Model:                 4
Model name:            ARMv7 Processor rev 4 (v7l)
CPU max MHz:           1400.0000
CPU min MHz:           600.0000
BogoMIPS:              38.40
Flags:                 half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32
lpae evtstrm crc32
```

To most people the myriad of vendors, processor families, and specification factors represent a bewildering array of choices. Regardless, there are several factors associated with CPUs and processors that even general users and administrators often need to consider when they need to specify operational environments:

Bit size

For CPUs this number relates to both the native size of data it manipulates and the amount of memory it can access. Most modern systems are either 32-bit or 64-bit. If an application needs access to more than 4 gigabytes of memory then it must run on a 64-bit system since 4 gigabytes is the maximum address that can be represented using 32 bits. And, while 32-bit applications can typically run on 64-bit systems, 64-bit applications cannot run on 32-bit systems.

Clock speed

Often expressed in megahertz (MHz) or gigahertz (GHz). This relates to how fast a processor processes instructions. But processor speed is just one of the factors impacting system response times, wait times, and throughput. Even an active multi-tasking user rarely keeps a CPU of a common desktop PC active more than 2 or 3 percent of the time. Regardless, if you frequently use computationally intensive applications involving activities such as encryption or video rendering then CPU speed may have a significant impact on throughput and wait time.

Cache

CPUs require a constant stream of both instructions and data in order to operate. The cost and power consumption of a multi-gigabyte system memory that could be accessed at CPU clock speeds would be prohibitive. CPU-speed cache memory is integrated onto the CPU chip to provide a high-speed buffer between CPUs and system memory. Cache is separated into multiple layers, commonly referenced as L1, L2, L3 and even L4. In the case of cache, more is often better.

Cores

Core refers to an individual CPU. In addition to core representing a physical CPU, *Hyper-Threading Technology* (HTT) allows a single physical CPU to concurrently process multiple instructions thus virtually acting as multiple physical CPUs. Most typically, multiple physical cores are packaged as a single physical processor chip. However, there are motherboards that support multiple physical processor chips. In theory having more cores to process tasks would always seem to yield better system throughput. Unfortunately, desktop applications often only keep CPUs busy 2 or 3 percent of the time, so adding more mostly idle CPUs is likely to result in minimal improvement to throughput. More cores are best suited to running applications that are written to have multiple independent threads of operation such as video frame rendering, web page rendering, or multi-user virtual machine environments.

Storage

Storage devices provide a method for retaining programs and data. *Hard Disk Drives* (HDDs) and *Solid State Drives* (SSDs) are the most common form of storage device for servers and desktops. USB memory sticks and optical devices such as DVD are also used but rarely as a primary device.

As the name implies, a hard disk drive stores information on one or more rigid physical disks. The physical disks are covered with magnetic media to make storage possible. The disks are contained within a sealed package since dust, small particles, and even finger prints would interfere with the ability of the HDD to read and write the magnetic media.

SSDs are effectively more sophisticated versions of USB memory sticks with significantly larger

capacity. SSDs store information in microchips so there are no moving parts.

Although the underlying technologies for HDDs and SSDs are different, there are important factors that can be compared. HDD capacity is based on scaling physical components while SSD capacity depends on the number of microchips. Per gigabyte, SSDs cost between 3 and 10 times what an HDD costs. To read or write, an HDD must wait for a location on a disk to rotate to a known location while SSDs are random access. SSD access speeds are typically 3 to 5 times faster than HDD devices. Since they have no moving parts SSDs consume less power and are more reliable than HDDs.

Storage capacity is constantly increasing for HDDs and SSDs. Today, 5 terabyte HDDs and 1 terabyte SSDs are commonly available. Regardless, large storage capacity is not always better. When a storage device fails the information it contained is no longer available. And of course, backup takes longer when there is more information to back up. For applications which read and write a lot of data, latency and performance may be more important than capacity.

Modern systems use SCSI (*Small Computer System Interface*) or SATA (*Serial AT Attachment*) to connect with storage devices. These interfaces are typically supported by the appropriate connector on the motherboard. Initial load comes from a storage device attached to the motherboard. Firmware settings define the order in which devices are accessed for this initial loading.

Storage systems known as RAID (*Redundant Array of Independent Disks*) are a common implementation to avoid loss of information. A RAID array consists of multiple physical devices containing duplicate copies of information. If one device fails all of the information is still available. Different physical RAID configurations are referenced as 0, 1, 5, 6, and 10. Each designation has different storage size, performance characteristics and ways to store redundant data or checksums for data recovery. Beyond some administrative configuration overhead, the existence of RAID is effectively transparent to users.

Storage devices commonly read and write data as blocks of bytes. The `lsblk` command can be used to list the block devices available to a system. The following example was run on a Raspberry Pi using an SD card as a storage device. The details of the output are covered by information in the *Partitions* and *Drivers* lessons that follow:

```
$ lsblk
NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
mmcblk0     179:0    0 29.7G  0 disk
+-mmcblk0p1 179:1    0 43.9M  0 part /boot
+-mmcblk0p2 179:2    0 29.7G  0 part /
```

Partitions

A storage device is effectively a long sequence of storage locations. Partitioning is the mechanism that tells Linux if it is to see these storage locations as a single sequence or multiple independent sequences. Each partition is treated as if it is an individual device. Most of the time partitions are created when a system is first configured. If change is needed, administrative tools are available to manage device partitioning.

So why would multiple partitions be desirable? Some examples for using partitions would be managing available storage, isolating encryption overhead, or supporting multiple file systems. Partitions make it possible to have a single storage device that can boot under different operating systems.

While Linux can recognize the storage sequence of a raw device a raw device cannot be used as-is. To use a raw device it must be formatted. Formatting writes a file system to a device and prepares it for file operations. Without a file system a device cannot be used for file-related operations.

Users see partitions as if they are individual devices. This makes it easy to overlook the fact that they're still dealing with a single physical device. In particular, device to device operations that are actually partition to partition operations will not have the expected performance. A single device is one physical mechanism with one set of read/write hardware. More importantly, you can't use partitions of a single physical device as a fault tolerant design. If the device fails, all partitions fail so there would be no fault tolerance.

NOTE *Logical Volume Manager* (LVM) is a software capability that allows administrators to combine individual disks and disk partitions and treat them as if they are a single drive.

Peripherals

Servers and workstations need a combination of CPU, system memory, and storage to operate. But these fundamental components don't directly interface with the external world. Peripherals are the devices that provide systems with input, output, and access to the rest of the real world.

Most motherboards have built-in external connectors and firmware support for common legacy peripheral interfaces supporting devices such as keyboard, mouse, sound, video, and network. Recent motherboards typically have an Ethernet connector to support networks, a HDMI connector supporting basic graphical needs, and one or more USB (*Universal Serial Bus*) connectors for mostly everything else. There are several versions of USB with different speed and physical characteristics. Several versions of USB ports are common on a single motherboard.

Motherboards may also have one or more expansion slots. Expansion slots allow users to add special circuit boards known as expansion cards that support custom, legacy, and non-standard peripherals. Graphics, sound, and network interfaces are common expansion cards. Expansion cards also support RAID, and special format legacy interfaces involving serial and parallel connections.

System on a Chip (SoC) configurations achieve power, performance, space, and reliability advantages over motherboard configurations by packaging processors, system memory, SSD, and hardware to control peripherals as a single integrated circuit package. The peripherals supported by SoC configurations are limited by the components packaged. Thus, SoC configurations tend to be developed for specific uses. Phones, tablets, and other portable devices are often based on SoC technology.

Some systems incorporate peripherals. Laptops are similar to workstations but incorporate default display, keyboard, and mouse peripherals. All-In-One systems are similar to laptops but require mouse and keyboard peripherals. Motherboard or SoC based controllers are often packaged with integral peripherals appropriate to a specific use.

Drivers and Device Files

So far, this lesson has presented information about processors, memory, disks, partitioning, formatting and peripherals. But requiring general users to deal with the specific details for each of the devices in their system would make those systems unusable. Likewise, software developers would need to modify their code for every new or modified device they need to support.

The solution to this “dealing with the details” problem is provided by a device driver. Device drivers accept a standard set of requests then translate those requests into the device appropriate control activities. Device drivers are what allow you and the applications you run to read from the file `/home/carol/stuff` without worrying about whether that file is on a hard drive, solid state drive, memory stick, encrypted storage, or some other device.

Device files are found in the `/dev` directory and identify physical devices, device access, and supported drivers. By convention, in modern systems using SCSI or SATA based storage devices the specification filename starts with the prefix `sd`. The prefix is followed by a letter such as `a` or `b` indicating a physical device. After the prefix and device identifier comes a number indicating a partition within the physical device. So, `/dev/sda` would reference the entire first storage device while `/dev/sda3` would reference partition 3 in the first storage device. The device file for each type of device has a naming convention appropriate to the device. While covering all of the possible naming conventions is beyond the scope of this lesson it is important to remember that these conventions are critical to making system administration possible.

While it is beyond the scope of this lesson to cover the contents of the `/dev` directory it is informative to look at the entry for a storage device. Device files for SD cards typically use `mmcblk` as a prefix:

```
$ ls -l mmcblk*
brw-rw---- 1 root disk 179, 0 Jun 30 01:17 mmcblk0
brw-rw---- 1 root disk 179, 1 Jun 30 01:17 mmcblk0p1
brw-rw---- 1 root disk 179, 2 Jun 30 01:17 mmcblk0p2
```

The listing details for a device file are different from typical file details:

- Unlike a file or directory the first letter of the permissions field is `b`. This indicates that blocks are read from and written to the device in blocks rather than individual characters.
- The size field is two values separated by a comma rather than a single value. The first value generally indicates a particular driver within the kernel and the second value specifies a specific device handled by the driver.
- The filename uses a number for the physical device so the naming convention adapts by specifying the partition suffix as a `p` followed by a digit.

NOTE Each system device should have an entry in `/dev`. Since the contents of the `/dev` directory are created at installation there are often entries for every possible driver and device even if no physical device exists.

Guided Exercises

1. Describe these terms:

Processor	
CPU	
GPU	

2. If you are primarily running video editing applications (a computationally intensive activity) which components and characteristics would you expect to have impact on system usability:

   CPU cores - CPU speed - Available system memory - Storage system - GPU - Video display

3. What would you expect the name of the device file in `/dev` to be for partition 3 of the third SATA drive in a system:

`sd3p3`	
`sdcp3`	
`sdc3`	

Explorational Exercises

1. Run the `lsblk` command on your system. Identify the parameters below. If a system is not immediately available, consider the `lsblk -f` listing for the Raspberry Pi system mentioned in the "Storage" section above:

```
$ lsblk -f
NAME        FSTYPE LABEL  UUID                                 MOUNTPOINT
mmcblk0
+-mmcblk0p1 vfat   boot   9304-D9FD                            /boot
+-mmcblk0p2 ext4   rootfs 29075e46-f0d4-44e2-a9e7-55ac02d6e6cc /
```

   - The type of devices and how many
   - The partition structure of each device
   - The type of file system and mount for each partition

Summary

A system is the sum of its components. Different components impact cost, performance, and usability in different ways. While there are common configurations for workstations and servers there is no single best configuration.

Answers to Guided Exercises

1. Describe these terms:

   Processor

   A general term that applies to any type of processor. Often used incorrectly as a synonym for CPU.

   CPU

   A Central Processing Unit. A processing unit providing support for general purpose computational tasks.

   GPU

   A Graphical Processing Unit. A processing unit optimized for supporting activities relating to the presentation of graphics.

2. If you are primarily running video editing applications (a computationally intensive activity) which components and characteristics would you expect to have impact on system usability:

   CPU cores

   Yes. Multiple cores support the concurrent presentation and rendering tasks required by video editing.

   CPU speed

   Yes. Video rendering requires a significant amount of computational activity.

   Available system memory

   Likely. The uncompressed video used in editing is large. General purpose systems often come with 8 gigabytes of memory. 16 or even 32 gigabytes of memory allows the system to handle more frames of uncompressed video making editing activities more efficient.

   Storage system

   Yes. Video files are large. The overhead of local SSD drives supports more efficient transfer. Slower network drives are likely to be counterproductive.

   GPU

   No. GPU primarily impacts the presentation of the rendered video.

   Video display

   No. The video display primarily impacts the presentation of the rendered video.

3. What would you expect the name of the device file in `/dev` to be for partition 3 of the third SATA drive in a system:

`sd3p3`	Not correct. Drive 3 would be `sdc` not `sd3`
`sdcp3`	Not correct. Partition 3 would be `3` not `p3`
`sdc3`	Correct

Answers to Explorational Exercises

1. Run the `lsblk` command on your system. Identify the parameters below. If a system is not immediately available, consider the `lsblk -f` listing for the Raspberry Pi system mentioned in the "Storage" section above:

```
$ lsblk -f
NAME        FSTYPE LABEL  UUID                                 MOUNTPOINT
mmcblk0
+-mmcblk0p1 vfat   boot   9304-D9FD                            /boot
+-mmcblk0p2 ext4   rootfs 29075e46-f0d4-44e2-a9e7-55ac02d6e6cc /
```

   Answers that follow are based the `lsblk -f` listing for the Raspberry Pi system above. Your answers may differ:

   The type of devices and how many

   There is one device: `mmcblk0`. You know by convention that the `mmcblk` would be an SD memory card.

   The partition structure of each device

   There are two partitions: `mmcblk0p1` and `mmcblk0p2`.

   The type of file system and mount for each partition

   Partition 1 uses the `vfat` file system. It is used to boot the system and is mounted as `/boot`. Partition 2 uses the `ext4` file system. It is used as the primary file system and is mounted as `/`.

4.3 Where Data is Stored

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 4.3

Weight

3

Key knowledge areas

- Programs and configuration
- Processes
- Memory addresses
- System messaging
- Logging

Partial list of the used files, terms and utilities

- `ps`, `top`, `free`
- `syslog`, `dmesg`
- `/etc/`, `/var/log/`
- `/boot/`, `/proc/`, `/dev/`, `/sys/`

4.3 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	4 The Linux Operating System
Objective:	4.3 Where Data is Stored
Lesson:	1 of 2

Introduction

For an operating system, everything is considered data. For Linux, everything is considered a file: programs, regular files, directories, block devices (hard disks, etc.), character devices (consoles, etc.), kernel processes, sockets, partitions, links, etc. The Linux directory structure, starting from the *root* `/`, is a collection of files containing data. The fact that everything is a file is a powerful feature of Linux as it allows for tweaking virtually every single aspect of the system.

In this lesson we will be discussing the different locations in which important data is stored as established by the Linux Filesystem Hierarchy Standard (FHS). Some of these locations are real directories which store data persistently on disks, whereas others are pseudo filesystems loaded in memory which give us access to kernel subsystem data such as running processes, use of memory, hardware configuration and so on. The data stored in these virtual directories is used by a number of commands that allow us to monitor and handle it.

Programs and their Configuration

Important data on a Linux system are — no doubt — its programs and their configuration files. The former are executable files containing sets of instructions to be run by the computer's

processor, whereas the latter are usually text documents that control the operation of a program. Executable files can be either binary files or text files. Executable text files are called scripts. Configuration data on Linux is traditionally stored in text files too, although there are various styles of representing configuration data.

Where Binary Files are Stored

Like any other file, executable files live in directories hanging ultimately from `/`. More specifically, programs are distributed across a three-tier structure: the first tier (`/`) includes programs that can be necessary in single-user mode, the second tier (`/usr`) contains most multi-user programs and the third tier (`/usr/local`) is used to store software that is not provided by the distribution and has been compiled locally.

Typical locations for programs include:

`/sbin`
: It contains essential binaries for system administration such as `parted` or `ip`.

`/bin`
: It contains essential binaries for all users such as `ls`, `mv`, or `mkdir`.

`/usr/sbin`
: It stores binaries for system administration such as `deluser`, or `groupadd`.

`/usr/bin`
: It includes most executable files — such as `free`, `pstree`, `sudo` or `man` — that can be used by all users.

`/usr/local/sbin`
: It is used to store locally installed programs for system administration that are not managed by the system's package manager.

`/usr/local/bin`
: It serves the same purpose as `/usr/local/sbin` but for regular user programs.

Recently some distributions started to replace `/bin` and `/sbin` with symbolic links to `/usr/bin` and `/usr/sbin`.

NOTE The `/opt` directory is sometimes used to store optional third-party applications.

Apart from these directories, regular users can have their own programs in either:

- `/home/$USER/bin`
- `/home/$USER/.local/bin`

TIP You can find out what directories are available for you to run binaries from by referencing the `PATH` variable with `echo $PATH`. For more information on `PATH`, review the lessons on variables and shell customization.

We can find the location of programs with the `which` command:

```
$ which git
/usr/bin/git
```

Where Configuration Files are Stored

The `/etc` Directory

In the early days of Unix there was a folder for each type of data, such as `/bin` for binaries and `/boot` for the kernel(s). However, `/etc` (meaning *et cetera*) was created as a catch-all directory to store any files that did not belong in the other categories. Most of these files were configuration files. With the passing of time more and more configuration files were added so `/etc` became the main folder for configuration files of programs. As said above, a configuration file usually is a local, plain text (as opposed to binary) file which controls the operation of a program.

In `/etc` we can find different patterns for config files names:

- Files with an *ad hoc* extension or no extension at all, for example

  `group`
  : System group database.

  `hostname`
  : Name of the host computer.

  `hosts`
  : List of IP addresses and their hostname translations.

  `passwd`
  : System user database — made up of seven fields separated by colons providing information about the user.

`profile`

System-wide configuration file for Bash.

`shadow`

Encrypted file for user passwords.

- Initialization files ending in `rc`:

`bash.bashrc`

System-wide `.bashrc` file for interactive bash shells.

`nanorc`

Sample initialization file for GNU nano (a simple text editor that normally ships with any distribution).

- Files ending in `.conf`:

`resolv.conf`

Config file for the resolver — which provide access to the Internet Domain Name System (DNS).

`sysctl.conf`

Config file to set system variables for the kernel.

- Directories with the `.d` suffix:

Some programs with a unique config file (`*.conf` or otherwise) have evolved to have a dedicated `*.d` directory which help build modular, more robust configurations. For example, to configure logrotate, you will find `logrotate.conf`, but also the `logrotate.d` directories.

This approach comes in handy in those cases where different applications need configurations for the same specific service. If, for example, a web server package contains a logrotate configuration, this configuration can now be placed in a dedicated file in the `logrotate.d` directory. This file can be updated by the webserver package without interfering with the remaining logrotate configuration. Likewise, packages can add specific tasks by placing files in the `/etc/cron.d` directory instead of modifying `/etc/crontab`.

In Debian — and Debian derivatives — such an approach has been applied to the list of reliable sources read by the package management tool `apt`: apart from the classic `/etc/apt/sources.list`, now we find the `/etc/apt/sources.list.d` directory:

```
$ ls /etc/apt/sources*
/etc/apt/sources.list
/etc/apt/sources.list.d:
```

Configuration Files in `HOME` (Dotfiles)

At user level, programs store their configurations and settings in hidden files in the user's home directory (also represented `~`). Remember, hidden files start with a dot (`.`) — hence their name: *dotfiles*.

Some of these dotfiles are Bash scripts that customize the user's shell session and are sourced as soon as the user logs into the system:

`.bash_history`
: It stores the command line history.

`.bash_logout`
: It includes commands to execute when leaving the login shell.

`.bashrc`
: Bash's initialization script for non-login shells.

`.profile`
: Bash's initialization script for login shells.

NOTE Refer to the lesson on "Command Line Basics" to learn more about Bash and its init files.

Other user-specific programs' config files get sourced when their respective programs are started: `.gitconfig`, `.emacs.d`, `.ssh`, etc.

The Linux Kernel

Before any process can run, the kernel must be loaded into a protected area of memory. After that, the process with PID `1` (more often than not `systemd` nowadays) sets off the chain of processes, that is to say, one process starts other(s) and so on. Once the processes are active, the Linux kernel is in charge of allocating resources to them (keyboard, mouse, disks, memory, network interfaces, etc).

NOTE Prior to `systemd`, `/sbin/init` was always the first process in a Linux system as part of the *System V Init* system manager. In fact, you still find `/sbin/init`

currently but linked to `/lib/systemd/systemd`.

Where Kernels are Stored: `/boot`

The kernel resides in `/boot` — together with other boot-related files. Most of these files include the kernel version number components in their names (kernel version, major revision, minor revision and patch number).

The `/boot` directory includes the following types of files, with names corresponding with the respective kernel version:

`config-4.9.0-9-amd64`

Configuration settings for the kernel such as options and modules that were compiled along with the kernel.

`initrd.img-4.9.0-9-amd64`

Initial RAM disk image that helps in the startup process by loading a temporary root filesystem into memory.

`System-map-4.9.0-9-amd64`

The `System-map` (on some systems it will be named `System.map`) file contains memory address locations for kernel symbol names. Each time a kernel is rebuilt the file's contents will change as the memory locations could be different. The kernel uses this file to lookup memory address locations for a particular kernel symbol, or vice-versa.

`vmlinuz-4.9.0-9-amd64`

The kernel proper in a self-extracting, space-saving, compressed format (hence the `z` in `vmlinuz`; `vm` stands for virtual memory and started to be used when the kernel first got support for virtual memory).

`grub`

Configuration directory for the `grub2` bootloader.

TIP

Because it is a critical feature of the operating system, more than one kernel and its associated files are kept in `/boot` in case the default one becomes faulty and we have to fall back on a previous version to — at least — be able to boot the system up and fix it.

The `/proc` Directory

The `/proc` directory is one of the so-called virtual or pseudo filesystems since its contents are not

written to disk, but loaded in memory. It is dynamically populated every time the computer boots up and constantly reflects the current state of the system. `/proc` includes information about:

- Running processes
- Kernel configuration
- System hardware

Besides all the data concerning processes that we will see in the next lesson, this directory also stores files with information about the system's hardware and the kernel's configuration settings. Some of these files include:

`/proc/cpuinfo`

It stores information about the system's CPU:

```
$ cat /proc/cpuinfo
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 158
model name  : Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
stepping    : 10
cpu MHz     : 3696.000
cache size  : 12288 KB
(...)
```

`/proc/cmdline`

It stores the strings passed to the kernel on boot:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.9.0-9-amd64 root=UUID=5216e1e4-ae0e-441f-b8f5-8061c0034c74 ro
quiet
```

`/proc/modules`

It shows the list of modules loaded into the kernel:

```
$ cat /proc/modules
nls_utf8 16384 1 - Live 0xffffffffc0644000
isofs 40960 1 - Live 0xffffffffc0635000
udf 90112 0 - Live 0xffffffffc061e000
crc_itu_t 16384 1 udf, Live 0xffffffffc04be000
```

```
fuse 98304 3 - Live 0xffffffffc0605000
vboxsf 45056 0 - Live 0xffffffffc05f9000 (O)
joydev 20480 0 - Live 0xffffffffc056e000
vboxguest 327680 5 vboxsf, Live 0xffffffffc05a8000 (O)
hid_generic 16384 0 - Live 0xffffffffc0569000
(...)
```

The `/proc/sys` Directory

This directory includes kernel configuration settings in files classified into categories per subdirectory:

```
$ ls /proc/sys
abi  debug  dev  fs  kernel  net  user  vm
```

Most of these files act like a switch and — therefore — only contain either of two possible values: `0` or `1` ("on" or "off"). For instance:

`/proc/sys/net/ipv4/ip_forward`

The value that enables or disables our machine to act as a router (be able to forward packets):

```
$ cat /proc/sys/net/ipv4/ip_forward
0
```

There are some exceptions, though:

`/proc/sys/kernel/pid_max`

The maximum PID allowed:

```
$ cat /proc/sys/kernel/pid_max
32768
```

WARNING Be extra careful when changing the kernel settings as the wrong value may result in an unstable system.

Hardware Devices

Remember, in Linux "everything is a file". This implies that hardware device information as well as the kernel's own configuration settings are all stored in special files that reside in virtual

directories.

The `/dev` Directory

The *device* directory `/dev` contains device files (or nodes) for all connected hardware devices. These device files are used as an interface between the devices and the processes using them. Each device file falls into one of two categories:

Block devices

Are those in which data is read and written in blocks which can be individually addressed. Examples include hard disks (and their partitions, like `/dev/sda1`), USB flash drives, CDs, DVDs, etc.

Character devices

Are those in which data is read and written sequentially one character at a time. Examples include keyboards, the text console (`/dev/console`), serial ports (such as `/dev/ttyS0` and so on), etc.

When listing device files, make sure you use `ls` with the `-l` switch to differentiate between the two. We can — for instance — check for hard disks and partitions:

```
# ls -l /dev/sd*
brw-rw---- 1 root disk 8, 0 may 25 17:02 /dev/sda
brw-rw---- 1 root disk 8, 1 may 25 17:02 /dev/sda1
brw-rw---- 1 root disk 8, 2 may 25 17:02 /dev/sda2
(...)
```

Or for serial terminals (TeleTYpewriter):

```
# ls -l /dev/tty*
crw-rw-rw- 1 root tty     5,  0 may 25 17:26 /dev/tty
crw--w---- 1 root tty     4,  0 may 25 17:26 /dev/tty0
crw--w---- 1 root tty     4,  1 may 25 17:26 /dev/tty1
(...)
```

Notice how the first character is `b` for block devices and `c` for character devices.

TIP

The asterisk (`*`) is a globbing character than means 0 or more characters. Hence its importance in the `ls -l /dev/sd*` and `ls -l /dev/tty*` commands above. To learn more about these special characters, refer to the lesson on globbing.

Furthermore, `/dev` includes some special files which are quite useful for different programming purposes:

`/dev/zero`
: It provides as many null characters as requested.

`/dev/null`
: Aka *bit bucket*. It discards all information sent to it.

`/dev/urandom`
: It generates pseudo-random numbers.

The `/sys` Directory

The *sys filesystem* (`sysfs`) is mounted on `/sys`. It was introduced with the arrival of kernel 2.6 and meant a great improvement on `/proc/sys`.

Processes need to interact with the devices in `/dev` and so the kernel needs a directory which contains information about these hardware devices. This directory is `/sys` and its data is orderly arranged into categories. For instance, to check on the MAC address of your network card (`enp0s3`), you would `cat` the following file:

```
$ cat /sys/class/net/enp0s3/address
08:00:27:02:b2:74
```

Memory and Memory Types

Basically, for a program to start running, it has to be loaded into memory. By and large, when we speak of memory we refer to *Random Access Memory* (RAM) and — when compared to mechanical hard disks — it has the advantage of being a lot faster. On the down side, it is volatile (i.e., once the computer shuts down, the data is gone).

Notwithstanding the aforementioned — when it comes to memory — we can differentiate two main types in a Linux system:

Physical memory
: Also known as *RAM*, it comes in the form of chips made up of integrated circuits containing millions of transistors and capacitors. These, in turn, form memory cells (the basic building block of computer memory). Each of these cells has an associated hexadecimal code — a memory address — so that it can be referenced when needed.

Swap

Also known as *swap space*, it is the portion of virtual memory that lives on the hard disk and is used when there is no more RAM available.

On the other hand, there is the concept of *virtual memory* which is an abstraction of the total amount of usable, addressing memory (RAM, but also disk space) as seen by applications.

`free` parses `/proc/meminfo` and displays the amount of free and used memory in the system in a very clear manner:

```
$ free
              total        used        free      shared  buff/cache   available
Mem:        4050960     1474960     1482260       96900     1093740     2246372
Swap:       4192252           0     4192252
```

Let us explain the different columns:

`total`

Total amount of physical and swap memory installed.

`used`

Amount of physical and swap memory currently in use.

`free`

Amount of physical and swap memory currently not in use.

`shared`

Amount of physical memory used — mostly — by `tmpfs`.

`buff/cache`

Amount of physical memory currently in use by kernel buffers and the page cache and slabs.

`available`

Estimate of how much physical memory is available for new processes.

By default `free` shows values in kibibytes, but allows for a variety of switches to display its results in different units of measurement. Some of these options include:

`-b`

Bytes.

`-m`

Mebibytes.

`-g`

Gibibytes.

`-h`

Human-readable format.

`-h` is always comfortable to read:

```
$ free -h
              total        used        free      shared  buff/cache   available
Mem:           3,9G        1,4G        1,5G         75M        1,0G        2,2G
Swap:          4,0G          0B        4,0G
```

NOTE A kibibyte (KiB) equals 1,024 bytes while a kilobytes (KB) equals 1000 bytes. The same is respectively true for mebibytes, gibibytes, etc.

Guided Exercises

1. Use the `which` command to find out the location of the following programs and complete the table:

Program	`which` **command**	**Path to Executable (output)**	**User needs** `root` **privileges?**
`swapon`			
`kill`			
`cut`			
`usermod`			
`cron`			
`ps`			

2. Where are the following files to be found?

File	`/etc`	`~`
`.bashrc`		
`bash.bashrc`		
`passwd`		
`.profile`		
`resolv.conf`		
`sysctl.conf`		

3. Explain the meaning of the number elements for kernel file `vmlinuz-4.15.0-50-generic` found in `/boot`:

Number Element	**Meaning**
`4`	
`15`	
`0`	
`50`	

4. What command would you use to list all hard drives and partitions in `/dev`?

Explorational Exercises

1. Device files for hard drives are represented based on the controllers they use — we saw `/dev/sd*` for drives using SCSI (Small Computer System Interface) and SATA (Serial Advanced Technology Attachment), but
   - How were old IDE (Integrated Drive Electronics) drives represented?
   - And modern NVMe (Non-Volatile Memory Express) drives?

2. Take a look at the file `/proc/meminfo`. Compare the contents of this file to the output of the command `free` and identify which key from `/proc/meminfo` correspond to the following fields in the output of `free`:

`free` **output**	`/proc/meminfo` **field**
`total`	
`free`	
`shared`	
`buff/cache`	
`available`	

Summary

In this lesson you have learned about the location of programs and their configuration files in a Linux system. Important facts to remember are:

- Basically, programs are to be found across a three-level directory structure: `/`, `/usr` and `/usr/local`. Each of these levels may contain `bin` and `sbin` directories.
- Configuration files are stored in `/etc` and `~`.
- Dotfiles are hidden files that start with a dot (`.`).

We have also discussed the Linux kernel. Important facts are:

- For Linux, everything is a file.
- The Linux kernel lives in `/boot` together with other boot-related files.
- For processes to start executing, the kernel has to first be loaded into a protected area of memory.
- The kernel job is that of allocating system resources to processes.
- The `/proc` virtual (or pseudo) filesystem stores important kernel and system data in a volatile way.

Likewise, we have explored hardware devices and learned the following:

- The `/dev` directory stores special files (aka nodes) for all connected hardware devices: *block devices* or *character devices*. The former transfer data in blocks; the latter, one character at a time.
- The `/dev` directory also contains other special files such as `/dev/zero`, `/dev/null` or `/dev/urandom`.
- The `/sys` directory stores information about hardware devices arranged into categories.

Finally, we touched upon memory. We learned:

- A program runs when it is loaded into memory.
- What RAM (Random Access Memory) is.
- What Swap is.
- How to display the use of memory.

Commands used in this lesson:

`cat`

Concatenate/print file content.

`free`

Display amount of free and used memory in the system.

`ls`

List directory contents.

`which`

Show location of program.

Answers to Guided Exercises

1. Use the `which` command to find out the location of the following programs and complete the table:

Program	`which` **command**	**Path to Binary (output)**	**User needs** `root` **privileges?**
`swapon`	`which swapon`	`/sbin/swapon`	Yes
`kill`	`which kill`	`/bin/kill`	No
`cut`	`which cut`	`/usr/bin/cut`	No
`usermod`	`which usermod`	`/usr/sbin/usermod`	Yes
`cron`	`which cron`	`/usr/sbin/cron`	Yes
`ps`	`which ps`	`/bin/ps`	No

2. Where are the following files to be found?

File	`/etc`	`~`
`.bashrc`		X
`bash.bashrc`	X	
`passwd`	X	
`.profile`		X
`resolv.conf`	X	
`sysctl.conf`	X	

3. Explain the meaning of the number elements for kernel file `vmlinuz-4.15.0-50-generic` found in `/boot`:

Number Element	**Meaning**
`4`	Kernel version
`15`	Major revision
`0`	Minor revision
`50`	Patch number

4. What command would you use to list all hard drives and partitions in `/dev`?

```
ls /dev/sd*
```

Answers to Explorational Exercises

1. Device files for hard drives are represented based on the controllers they use — we saw `/dev/sd*` for drives using SCSI (Small Computer System Interface) and SATA (Serial Advanced Technology Attachment), but
   - How were old IDE (Integrated Drive Electronics) drives represented?

     `/dev/hd*`

   - And modern NVMe (Non-Volatile Memory Express) drives?

     `/dev/nvme*`

2. Take a look at the file `/proc/meminfo`. Compare the contents of this file to the output of the command `free` and identify which key from `/proc/meminfo` correspond to the following fields in the output of `free`:

`free` **output**	`/proc/meminfo` **field**
`total`	`MemTotal` / `SwapTotal`
`free`	`MemFree` / `SwapFree`
`shared`	`Shmem`
`buff/cache`	`Buffers`, `Cached` and `SReclaimable`
`available`	`MemAvailable`

Linux Professional Institute

4.3 Lesson 2

Certificate:	Linux Essentials
Version:	1.6
Topic:	4 The Linux Operating System
Objective:	4.3 Where Data is Stored
Lesson:	2 of 2

Introduction

After exploring programs and their configuration files, in this lesson we will learn how commands are executed as processes. Likewise, we will be commenting on the system messaging, the use of the kernel ring buffer and how the arrival of `systemd` and its journal daemon — `journald` — changed the ways things had been done so far regarding system logging.

Processes

Every time a user issues a command, a program is run and one or more processes are generated.

Processes exist in a hierarchy. After the kernel is loaded in memory on boot, the first process is initiated which — in turn — starts other processes, which, again, can start other processes. Every process has a unique identifier (`PID`) and parent process identifier (`PPID`). These are positive integers that are assigned in sequential order.

Exploring Processes Dynamically: `top`

You can get a dynamic listing of all running processes with the `top` command:

```
$ top

top - 11:10:29 up  2:21,  1 user,  load average: 0,11, 0,20, 0,14
Tasks:  73 total,   1 running,  72 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,0 us,  0,3 sy,  0,0 ni, 99,7 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem :  1020332 total,   909492 free,    38796 used,    72044 buff/cache
KiB Swap:  1046524 total,  1046524 free,        0 used.   873264 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
   436 carol     20   0   42696   3624   3060 R  0,7  0,4   0:00.30 top
     4 root      20   0       0      0      0 S  0,3  0,0   0:00.12 kworker/0:0
   399 root      20   0   95204   6748   5780 S  0,3  0,7   0:00.22 sshd
     1 root      20   0   56872   6596   5208 S  0,0  0,6   0:01.29 systemd
     2 root      20   0       0      0      0 S  0,0  0,0   0:00.00 kthreadd
     3 root      20   0       0      0      0 S  0,0  0,0   0:00.02 ksoftirqd/0
     5 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 kworker/0:0H
     6 root      20   0       0      0      0 S  0,0  0,0   0:00.00 kworker/u2:0
     7 root      20   0       0      0      0 S  0,0  0,0   0:00.08 rcu_sched
     8 root      20   0       0      0      0 S  0,0  0,0   0:00.00 rcu_bh
     9 root      rt   0       0      0      0 S  0,0  0,0   0:00.00 migration/0
    10 root       0 -20       0      0      0 S  0,0  0,0   0:00.00 lru-add-drain
    (...)
```

As we saw above, `top` can also give us information about memory and CPU consumption of the overall system as well as for each process.

`top` allows the user some interaction.

By default, the output is sorted by the percentage of CPU time used by each process in descending order. This behavior can be modified by pressing the following keys from within `top`:

`M`

Sort by memory usage.

`N`

Sort by process ID number.

`T`

Sort by running time.

`P`

Sort by percentage of CPU usage.

To switch between descending/ascending order just press `R`.

TIP A fancier and more user-friendly version of `top` is `htop`. Another — perhaps more exhaustive — alternative is `atop`. If not already installed in your system, I encourage you to use your package manager to install them and give them a try.

A Snapshot of Processes: `ps`

Another very useful command to get information about processes is `ps`. Whereas `top` provides dynamic information, that of `ps` is static.

If invoked without options, the output of `ps` is quite discrete and relates only to the processes attached to the current shell:

```
$ ps
  PID TTY          TIME CMD
 2318 pts/0    00:00:00 bash
 2443 pts/0    00:00:00 ps
```

The displayed information has to do with the process identifier (`PID`), the terminal in which the process is run (`TTY`), the CPU time taken by the process (`TIME`) and the command which started the process (`CMD`).

A useful switch for `ps` is `-f` which shows the full-format listing:

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
carol     2318  1682  0 08:38 pts/1    00:00:00 bash
carol     2443  2318  0 08:46 pts/1    00:00:00 ps -f
```

In combination with other switches, `-f` shows the relationship between parent and child processes:

```
$ ps -uf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
carol     2318  0.0  0.1  21336  5140 pts/1    Ss   08:38   0:00 bash
carol     2492  0.0  0.0  38304  3332 pts/1    R+   08:51   0:00  \_ ps -uf
carol     1780  0.0  0.1  21440  5412 pts/0    Ss   08:28   0:00 bash
carol     2291  0.0  0.7 305352 28736 pts/0    Sl+  08:35   0:00  \_ emacs index.en.adoc
-nw
```

```
(...)
```

Likewise, `ps` can show the percentage of memory used when invoked with the `-v` switch:

```
$ ps -v
  PID TTY      STAT   TIME  MAJFL   TRS   DRS   RSS %MEM COMMAND
 1163 tty2     Ssl+   0:00      1    67 201224 5576  0.1 /usr/lib/gdm3/gdm-x-session (...)
 (...)
```

NOTE Another visually attractive command that shows the hierarchy of processes is `pstree`. It ships with all major distributions.

Process Information in the `/proc` Directory

We have already seen the `/proc` filesystem. `/proc` includes a numbered subdirectory for every running process in the system (the number is the `PID` of the process):

```
carol@debian:~# ls /proc
1    108  13   17   21   27   354  41   665  8    9
10   109  14   173  22   28   355  42   7    804  915
103  11   140  18   23   29   356  428  749  810  918
104  111  148  181  24   3    367  432  75   811
105  112  149  19   244  349  370  433  768  83
106  115  15   195  25   350  371  5    797  838
107  12   16   2    26   353  404  507  798  899
(...)
```

Thus, all the information about a particular process is included within its directory. Let us list the contents of the first process — that whose `PID` is `1` (the output has been truncated for readability):

```
# ls /proc/1/
attr        cmdline          environ  io        mem       ns
autogroup   comm             exe      limits    mountinfo numa_maps
auxv        coredump_filter  fd       loginuid  mounts    oom_adj
...
```

You can check — for instance — the process executable:

```
# cat /proc/1/cmdline; echo
```

```
/sbin/init
```

As you can see, the binary that started the hierarchy of processes was `/sbin/init`.

NOTE Commands can be concatenated with the semicolon (`;`). The point in using the `echo` command above is to provide a newline. Try and run simply `cat /proc/1/cmdline` to see the difference.

The System Load

Each process on a system can potentially consume system resources. The so-called system load tries to aggregate the overall load of the system into a single numeric indicator. You can see the current load with the command `uptime`:

```
$ uptime
 22:12:54 up 13 days, 20:26,  1 user,  load average: 2.91, 1.59, 0.39
```

The three last digits indicate the system's load average for the last minute (`2.91`), the last five minutes (`1.59`) and the last fifteen minutes (`0.39`), respectively.

Each of these numbers indicates how many processes were waiting either for CPU resources or for input/output operations to complete. This means that these processes were ready to run if they had received the respective resources.

System Logging and System Messaging

As soon as the kernel and the processes start executing and communicating with each other, a lot of information is produced. Most of it is sent to files — the so-called log files or, simply, *logs*.

Without logging, searching for an event that happened on a server would give sysadmins many a headache, hence the importance of having a standardized and centralized way of keeping track of any system events. Besides, logs are determinant and telling when it comes to troubleshooting and security as well as reliable data sources for understanding system statistics and making trend predictions.

Logging with the syslog Daemon

Traditionally, system messages have been managed by the standard logging facility — syslog — or any of its derivatives — syslog-ng or rsyslog. The logging daemon collects messages from other services and programs and stores them in log files, typically under `/var/log`. However, some

services take care of their own logs (take — for example — the Apache HTTPD web server). Likewise, the Linux kernel uses an in-memory ring buffer for storing its log messages.

Log Files in `/var/log`

Because logs are data that varies over time, they are normally found in `/var/log`.

If you explore `/var/log`, you will realize that the names of logs are — to a certain degree — quite self-explanatory. Some examples include:

`/var/log/auth.log`
: It stores information about authentication.

`/var/log/kern.log`
: It stores kernel information.

`/var/log/syslog`
: It stores system information.

`/var/log/messages`
: It stores system and application data.

NOTE The exact name and contents of log files may vary across Linux distributions.

Accessing Log Files

When exploring log files, remember to be root (if you do not have reading permissions) and use a pager such as `less`;

```
# less /var/log/messages
Jun  4 18:22:48 debian liblogging-stdlog:  [origin software="rsyslogd" swVersion="8.24.0" x-
pid="285" x-info="http://www.rsyslog.com"] rsyslogd was HUPed
Jun 29 16:57:10 debian kernel: [    0.000000] Linux version 4.9.0-8-amd64 (debian-
kernel@lists.debian.org) (gcc version 6.3.0 20170516 (Debian 6.3.0-18+deb9u1) ) #1 SMP
Debian 4.9.130-2 (2018-10-27)
Jun 29 16:57:10 debian kernel: [    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.9.0-
8-amd64 root=/dev/sda1 ro quiet
```

Alternatively, you can use `tail` with the `-f` switch to read the most recent messages of the file and dynamically show new lines as they are appended:

```
# tail -f /var/log/messages
Jul  9 18:39:37 debian kernel: [    2.350572] RAPL PMU: hw unit of domain psys 2^-0 Joules
Jul  9 18:39:37 debian kernel: [    2.512802] input: VirtualBox USB Tablet as
/devices/pci0000:00/0000:00:06.0/usb1/1-1/1-1:1.0/0003:80EE:0021.0001/input/input7
Jul  9 18:39:37 debian kernel: [    2.513861] Adding 1046524k swap on /dev/sda5.  Priority:-
1 extents:1 across:1046524k FS
Jul  9 18:39:37 debian kernel: [    2.519301] hid-generic 0003:80EE:0021.0001:
input,hidraw0: USB HID v1.10 Mouse [VirtualBox USB Tablet] on usb-0000:00:06.0-1/input0
Jul  9 18:39:37 debian kernel: [    2.623947] snd_intel8x0 0000:00:05.0: white list rate for
1028:0177 is 48000
Jul  9 18:39:37 debian kernel: [    2.914805] IPv6: ADDRCONF(NETDEV_UP): enp0s3: link is not
ready
Jul  9 18:39:39 debian kernel: [    4.937283] e1000: enp0s3 NIC Link is Up 1000 Mbps Full
Duplex, Flow Control: RX
Jul  9 18:39:39 debian kernel: [    4.938493] IPv6: ADDRCONF(NETDEV_CHANGE): enp0s3: link
becomes ready
Jul  9 18:39:40 debian kernel: [    5.315603] random: crng init done
Jul  9 18:39:40 debian kernel: [    5.315608] random: 7 urandom warning(s) missed due to
ratelimiting
```

You will find the output into the following format:

- Timestamp
- Hostname from which the message came from
- Name of program/service that generated the message
- The PID of the program that generated the message
- Description of the action that took place

Most log files are written in plain text; however, a few can contain binary data as is the case with `/var/log/wtmp` — which stores data relevant to successful logins. You can use the `file` command to determine which is the case:

```
$ file /var/log/wtmp
/var/log/wtmp: dBase III DBT, version number 0, next free block index 8
```

These files are normally read using special commands. `last` is used to interpret the data in `/var/log/wtmp`:

```
$ last
```

```
carol    tty2         :0               Thu May 30 10:53   still logged in
reboot   system boot  4.9.0-9-amd64    Thu May 30 10:52   still running
carol    tty2         :0               Thu May 30 10:47 - crash  (00:05)
reboot   system boot  4.9.0-9-amd64    Thu May 30 09:11   still running
carol    tty2         :0               Tue May 28 08:28 - 14:11  (05:42)
reboot   system boot  4.9.0-9-amd64    Tue May 28 08:27 - 14:11  (05:43)
carol    tty2         :0               Mon May 27 19:40 - 19:52  (00:11)
reboot   system boot  4.9.0-9-amd64    Mon May 27 19:38 - 19:52  (00:13)
carol    tty2         :0               Mon May 27 19:35 - down   (00:03)
reboot   system boot  4.9.0-9-amd64    Mon May 27 19:34 - 19:38  (00:04)
```

NOTE Similar to `/var/log/wtmp`, `/var/log/btmp` stores information about failed login attempts and the special command to read its contents is `lastb`.

Log Rotation

Log files can grow a lot over a few weeks or months and take up all free disk space. To tackle this, the utility `logrotate` is used. It implements log rotation or cycling which implies actions such as moving log files to a new name, archiving and/or compressing them, sometimes emailing them to the sysadmin and eventually deleting them as they grow old. The conventions used for naming these rotated log files are diverse (adding a suffix with the date, for example); however, simply adding a suffix with an integer is commonplace:

```
# ls /var/log/apache2/
access.log  error.log  error.log.1  error.log.2.gz  other_vhosts_access.log
```

Note how `error.log.2.gz` has already been compressed with gzip (hence the `.gz` suffix).

The Kernel Ring Buffer

The kernel ring buffer is a fixed-size data structure that records kernel boot messages as well as any live kernel messages. The function of this buffer — a very important one — is that of logging all the kernel messages produced on boot — when `syslog` is not yet available. The `dmesg` command prints the kernel ring buffer (which used to be also stored in `/var/log/dmesg`). Because of the extension of the ring buffer, this command is normally used in combination with the text filtering utility `grep` or a pager such as `less`. For instance, to search for boot messages:

```
$ dmesg | grep boot
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.9.0-9-amd64 root=UUID=5216e1e4-ae0e-
441f-b8f5-8061c0034c74 ro quiet
[    0.000000] smpboot: Allowing 1 CPUs, 0 hotplug CPUs
```

```
[    0.000000] Kernel command line: BOOT_IMAGE=/boot/vmlinuz-4.9.0-9-amd64
root=UUID=5216e1e4-ae0e-441f-b8f5-8061c0034c74 ro quiet
[    0.144986] AppArmor: AppArmor disabled by boot time parameter
(...)
```

NOTE As the kernel ring buffer grows with new messages over time, the oldest ones will fade away.

The System Journal: systemd-journald

As of 2015, systemd replaced SysV Init as a *de facto* system and service manager in most major Linux distributions. As a consequence, the journal daemon — journald — has become the standard logging component, superseding syslog in most aspects. The data is no longer stored in plain text but in binary form. Thus, the `journalctl` utility is necessary to read the logs. On top of that, journald is syslog compatible and can be integrated with syslog.

`journalctl` is the utility that is used to read and query systemd's journal database. If invoked without options, it prints the entire journal:

```
# journalctl
-- Logs begin at Tue 2019-06-04 17:49:40 CEST, end at Tue 2019-06-04 18:13:10 CEST. --
jun 04 17:49:40 debian systemd-journald[339]: Runtime journal (/run/log/journal/) is 8.0M,
max 159.6M, 151.6M free.
jun 04 17:49:40 debian kernel: microcode: microcode updated early to revision 0xcc, date =
2019-04-01
Jun 04 17:49:40 debian kernel: Linux version 4.9.0-8-amd64 (debian-kernel@lists.debian.org)
(gcc version 6.3.0 20170516 (Debian 6.3.0-18+deb9u1) )
Jun 04 17:49:40 debian kernel: Command line: BOOT_IMAGE=/boot/vmlinuz-4.9.0-8-amd64
root=/dev/sda1 ro quiet
(...)
```

However, if invoked with the `-k` or `--dmesg` switches, it will be equivalent to using the `dmesg` command:

```
# journalctl -k
[    0.000000] Linux version 4.9.0-9-amd64 (debian-kernel@lists.debian.org) (gcc version
6.3.0 20170516 (Debian 6.3.0-18+deb9u1) ) #1 SMP Debian 4.9.168-1+deb9u2 (2019-05-13)
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.9.0-9-amd64 root=UUID=5216e1e4-ae0e-
441f-b8f5-8061c0034c74 ro quiet
(...)
```

Other interesting options for `journalctl` include:

`-b`, **`--boot`**

It shows boot information.

`-u`

It shows messages about a specified unit. Roughly, a unit can be defined as any resource handled by systemd. For example `journalctl -u apache2.service` is used to read messages about the `apache2` web server.

`-f`

It shows most recent journal messages and keeps printing new entries as they are appended to the journal — much like `tail -f`.

Guided Exercises

1. Have a look at the following listing of `top` and answer the following questions:

```
carol@debian:~$ top

top - 13:39:16 up 31 min,  1 user,  load average: 0.12, 0.15, 0.10
Tasks:  73 total,   2 running,  71 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.1 us,  0.4 sy,  0.0 ni, 98.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1020332 total,   698700 free,   170664 used,   150968 buff/cache
KiB Swap:  1046524 total,  1046524 free,        0 used.   710956 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  605 nobody    20   0 1137620 132424  34256 S  6.3 13.0   1:47.24 ntopng
  444 www-data  20   0  364780   4132   2572 S  0.3  0.4   0:00.44 apache2
  734 root      20   0   95212   7004   6036 S  0.3  0.7   0:00.36 sshd
  887 carol     20   0   46608   3680   3104 R  0.3  0.4   0:00.03 top
    1 root      20   0   56988   6688   5240 S  0.0  0.7   0:00.42 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:00.09 ksoftirqd/0
    4 root      20   0       0      0      0 S  0.0  0.0   0:00.87 kworker/0:0
(...)
```

   - Which processes have been started by the user `carol`?

   - What virtual directory of `/proc` should you visit to search for data regarding the `top` command?

   - What process was run first? How can you tell?

   - Complete the table specifying in what area of `top` output the following information is found:

Information about ...	Summary Area	Task Area
Memory		
Swap		
PID		

Information about ...	Summary Area	Task Area
CPU time		
Commands		

2. What command is used to read the following binary logs?
   - `/var/log/wtmp`
   - `/var/log/btmp`
   - `/run/log/journal/2a7d9730cd3142f4b15e20d6be631836/system.journal`
3. In combination with `grep`, what commands would you use to find out the following information about your Linux system?
   - When the system was last rebooted (`wtmp`)
   - Which hard disks are installed (`kern.log`)
   - When the last login occurred (`auth.log`)
4. What two commands would you use to have the kernel ring buffer displayed?
5. Indicate where the following log messages belong:
   - `Jul 10 13:37:39 debian dbus[303]: [system] Successfully activated service 'org.freedesktop.nm_dispatcher'`

`/var/log/auth.log`	
`/var/log/kern.log`	
`/var/log/syslog`	
`/var/log/messages`	

- `Jul 10 11:23:58 debian kernel: [ 1.923349] usbhid: USB HID core driver`

`/var/log/auth.log`	
`/var/log/kern.log`	
`/var/log/syslog`	
`/var/log/messages`	

- `Jul 10 14:02:53 debian sudo: pam_unix(sudo:session): session opened for user root by carol(uid=0)`

`/var/log/auth.log`	
`/var/log/kern.log`	
`/var/log/syslog`	
`/var/log/messages`	

- `Jul 10 11:23:58 debian NetworkManager[322]: <info> [1562750638.8672] NetworkManager (version 1.6.2) is starting…`

`/var/log/auth.log`	
`/var/log/kern.log`	
`/var/log/syslog`	
`/var/log/messages`	

6. Have `journalctl` query information about the following units?

Unit	**Command**
`ssh`	
`networking`	
`rsyslog`	
`cron`	

Explorational Exercises

1. Reconsider the `top` output from the guided exercises and answer the following questions:
   - What two steps would you follow to kill the *apache* web server?
   - In the summary area, how could you display the information about physical memory and swap using progress bars?
   - Now, sort the processes by memory usage:
   - Now that you have memory information displayed in progress bars and processes sorted by memory usage, save these configurations so that you get them as default next time you use `top`:
   - What file stores `top's` configuration settings? Where does it live? How can you check for its existence?
2. Learn about the command `exec` in Bash. Try to demonstrate its functionality by starting a Bash session, finding the Bash process with `ps`, then run `exec /bin/sh` and search for the process with the same PID again.
3. Follow these steps to explore kernel events and udev's dynamic management of devices:
   - Hotplug a USB drive into your computer. Run `dmesg` and pay attention to the last lines. What is the most recent line?
   - Bearing in mind the output from the previous command, run `ls /dev/sd*` and make sure your USB drive appears in the listing. What is the output?
   - Now remove the USB drive and run `dmesg` again. How does the most recent line read?
   - Run `ls /dev/sd*` again and make sure your device disappeared from the listing. What is the output?

Summary

In the context of data storage, the following topics have been discussed in this lesson: process management and system logging and messaging.

Regarding process management, we have learned the following:

- Programs generate processes and processes exist in a hierarchy.
- Every process has a unique identifier (`PID`) and a parent process identifier (`PPID`).
- `top` is a very useful command to dynamically and interactively explore the running processes of the system.
- `ps` can be used to obtain a snapshot of the current running processes in the system.
- The `/proc` directory includes directories for every running process in the system named after their PIDs.
- The concept of system load average — which is very useful to check on CPU utilization/overloading.

Concerning system logging, we must remember that:

- A log is a file where system events are recorded. Logs are invaluable when it comes to troubleshooting.
- Logging has traditionally been handled by special services such as syslog, syslog-ng or rsyslog. Nevertheless, some programs use their own logging daemons.
- Because logs are variable data, they are kept in `/var` and — sometimes — their names can give you a clue about their content (`kern.log`, `auth.log`, etc.)
- Most logs are written in plain text and can be read with any text editor as long as you have the right permissions. However, a few of them are binary and must be read using special commands.
- To avoid problems with disk space, *log rotation* is carried out by the logrotate utility.
- As for the kernel, it uses a circular data structure — the ring buffer — where boot messages are kept (old messages fade away over time).
- The system and service manager systemd replaced System V init in virtually all distros with journald becoming the standard logging service.
- To read systemd's journal, the `journalctl` utility is needed.

Commands used in this lesson:

`cat`

Concatenate/print file content.

`dmesg`

Print the kernel ring buffer.

`echo`

Display a line of text or a newline.

`file`

Determine file type.

`grep`

Print lines matching a pattern.

`last`

Show a listing of last logged in users.

`less`

Display contents of file one page at a time.

`ls`

List directory contents.

`journalctl`

Query the `systemd` journal.

`tail`

Display the last lines of a file.

Answers to Guided Exercises

1. Have a look the following listing of `top` and answer the following questions:

```
carol@debian:~$ top

top - 13:39:16 up 31 min,  1 user,  load average: 0.12, 0.15, 0.10
Tasks:  73 total,   2 running,  71 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.1 us,  0.4 sy,  0.0 ni, 98.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1020332 total,   698700 free,   170664 used,   150968 buff/cache
KiB Swap:  1046524 total,  1046524 free,        0 used.   710956 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  605 nobody    20   0 1137620 132424  34256 S  6.3 13.0   1:47.24 ntopng
  444 www-data  20   0  364780   4132   2572 S  0.3  0.4   0:00.44 apache2
  734 root      20   0   95212   7004   6036 S  0.3  0.7   0:00.36 sshd
  887 carol     20   0   46608   3680   3104 R  0.3  0.4   0:00.03 top
    1 root      20   0   56988   6688   5240 S  0.0  0.7   0:00.42 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:00.09 ksoftirqd/0
    4 root      20   0       0      0      0 S  0.0  0.0   0:00.87 kworker/0:0
(...)
```

   - Which processes have been started by the user `carol`?

     Answer: Only one: `top`.

   - What virtual directory of `/proc` should you visit to search for data regarding the `top` command?

     Answer: `/proc/887`

   - What process was run first? How can you tell?

     Answer: `systemd`. Because it is the one with `PID` #1.

   - Complete the table specifying in what area of `top` output the following information is found:

Information about ...	**Summary Area**	**Task Area**
Memory	Yes	Yes
Swap	Yes	No

Information about ...	Summary Area	Task Area
`PID`	No	Yes
CPU time	Yes	Yes
Commands	No	Yes

2. What command is used to read the following binary logs?
   - `/var/log/wtmp`

     Answer: `last`
   - `/var/log/btmp`

     Answer: `lastb`
   - `/run/log/journal/2a7d9730cd3142f4b15e20d6be631836/system.journal`

     Answer: `journalctl`
3. In combination with `grep`, what commands would you use to find out the following information about your Linux system?
   - When the system was last rebooted (`wtmp`)

     Answer: `last`
   - Which hard disk are installed (`kern.log`)

     Answer: `less /var/log/kern.log`
   - When the last login occurred (`auth.log`)

     Answer: `less /var/log/auth.log`
4. What two commands would you use to have the kernel ring buffer displayed?

   `dmesg` and `journalctl -k` (also `journalctl --dmesg`).
5. Indicate where the following log messages belong:
   - `Jul 10 13:37:39 debian dbus[303]: [system] Successfully activated service 'org.freedesktop.nm_dispatcher'`

`/var/log/auth.log`	

`/var/log/kern.log`	
`/var/log/syslog`	X
`/var/log/messages`	

- `Jul 10 11:23:58 debian kernel: [ 1.923349] usbhid: USB HID core driver`

`/var/log/auth.log`	
`/var/log/kern.log`	X
`/var/log/syslog`	
`/var/log/messages`	X

`Jul 10 14:02:53 debian sudo: pam_unix(sudo:session): session opened for user root by carol(uid=0)`

`/var/log/auth.log`	X
`/var/log/kern.log`	
`/var/log/syslog`	
`/var/log/messages`	

- `Jul 10 11:23:58 debian NetworkManager[322]: <info> [1562750638.8672] NetworkManager (version 1.6.2) is starting…`

`/var/log/auth.log`	
`/var/log/kern.log`	
`/var/log/syslog`	
`/var/log/messages`	X

6. Have `journalctl` query information about the following units:

Unit	**Command**
`ssh`	`journalctl -u ssh.service`
`networking`	`journalctl -u networking.service`
`rsyslog`	`journalctl -u rsyslog.service`
`cron`	`journalctl -u cron.service`

Answers to Explorational Exercises

1. Reconsider the `top` output from the guided exercises and answer the following questions:
   - What two steps would you follow to kill the *apache* web server?

     First, press `k`; then provide a `kill` value.

   - In the summary area, how could you display the information about physical memory and swap using progress bars?

     By pressing `m` once or twice.

   - Now, sort the processes by memory usage:

     `M`

   - Now that you have memory information displayed in progress bars and processes sorted by memory usage, save these configurations so that you get them as default next time you use `top`:

     `W`

   - What file stores `top's` configuration settings? Where does it live? How can you check for its existence?

     The file is `~/.config/procps/toprc` and lives in the user's home directory (`~`). Since it is a hidden file (it resides in a directory whose name starts with a dot), we can check for its existence with `ls -a` (list all files). This file can be generated by pressing `Shift` + `W` while in `top`.

2. Learn about the command `exec` in Bash. Try to demonstrate its functionality by starting a Bash session, finding the Bash process with `ps`, then run `exec /bin/sh` and search for the process with the same PID again.

   `exec` replaces a process with another command. In the following example we can see that the Bash process is replaced by `/bin/sh` (instead of `/bin/sh` becoming a child process):

```
$ echo $$
19877
$ ps auxf | grep 19877 | head -1
carol  19877  0.0  0.0   7448  3984 pts/25   Ss   21:17   0:00  \_ bash
$ exec /bin/sh
```

```
sh-5.0$ ps auxf | grep 19877 | head -1
carol   19877  0.0  0.0   7448  3896 pts/25   Ss   21:17   0:00  \_ /bin/sh
```

3. Follow these steps to explore kernel events and udev's dynamic management of devices:
   - Hotplug a USB drive into your computer. Run `dmesg` and pay attention to the last lines. What is the most recent line?

     You should get something along the lines of `[ 1967.700468] sd 6:0:0:0: [sdb] Attached SCSI removable disk`.

   - Bearing in mind the output from the previous command, run `ls /dev/sd*` and make sure your USB drive appears in the listing. What is the output?

     Depending on the number of devices connected to your system, you should get something like `/dev/sda /dev/sda1 /dev/sdb /dev/sdb1 /dev/sdb2`. In our case, we find our USB drive (`/dev/sdb`) and its two partitions (`/dev/sdb1` and `/dev/sdb2`).

   - Now remove the USB drive and run `dmesg` again. How does the most recent line read?

     You should get something along the lines of `[ 2458.881695] usb 1-9: USB disconnect, device number 6`.

   - Run `ls /dev/sd*` again and make sure your device disappeared from the listing. What is the output?

     In our case, `/dev/sda /dev/sda1`.

4.4 Your Computer on the Network

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 4.4

Weight

2

Key knowledge areas

- Internet, network, routers
- Querying DNS client configuration
- Querying network configuration

Partial list of the used files, terms and utilities

- `route`, `ip route show`
- `ifconfig`, `ip addr show`
- `netstat`, `ss`
- `/etc/resolv.conf`, `/etc/hosts`
- `IPv4`, `IPv6`
- `ping`
- `host`

4.4 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	4 The Linux Operating System
Objective:	4.4 Your Computer on the Network
Lesson:	1 of 1

Introduction

In today's world, computing devices of any kind exchange information through networks. At the very heart of the concept of computer networks are the physical connections between a device and its peer(s). These connections are called *links*, and they are the most basic connection between two different devices. Links can be established through various media, such as copper cables, optical fibres, radio waves or lasers.

Each link is connected with an interface of a device. Each device can have multiple interfaces and thus be connected to multiple links. Through these links computers can form a network; a small community of devices that can directly connect to each other. There are numerous examples of such networks in the world. To be able to communicate beyond the scope of a link layer network, devices use routers. Think of link layer networks as islands connected by routers which connect the islands — just like bridges which information has to travel to reach a device which is part of a remote island.

This model leads to several different layers of networking:

Link Layer

Handles the communication between directly connected devices.

Network Layer

Handles routing outside of individual networks and the unique addressing of devices beyond a single link layer network.

Application Layer

Enables individual programs to connect to each other.

When first invented, computer networks used the same methods of communication as telephones in that they were circuit switched. This means that a dedicated and direct link had to be formed between two nodes for them to be able to communicate. This method worked well, however, it required all the space on a given link for only two hosts to communicate.

Eventually computer networks moved over to something called *packet switching*. In this method the data is grouped up with a header, which contains information about where the information is coming from and where it's going to. The actual content information is contained in this frame and sent over the link to the recipient indicated in the frame's header. This allows for multiple devices to share a single link and communicate almost simultaneously.

Link Layer Networking

The job of any packet is to carry information from the source to its destination through a link connecting both devices. These devices need a way to identify themselves to each other. This is the purpose of a *link layer address*. In an ethernet network, *Media Access Control Addresses* (MAC) are used to identify individual devices. A MAC address consists of 48 bits. They are not necessarily globally unique and cannot be used to address peers outside of the current link. Thus these addresses can not be used to route packets to another links. The recipient of a packet checks whether the destination address matches its own link layer and, if it does, processes the packet. Otherwise the packet is dropped. The exception to this rule is *broadcast packets* (a packet sent to everyone in a given local network) which are always accepted.

The command `ip link show` displays a list of all the available network interfaces and their link layer addresses as well as some other information about the maximum packet size:

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT
```

```
group default qlen 1000
    link/ether 00:0c:29:33:3b:25 brd ff:ff:ff:ff:ff:ff
```

The above output shows that the device has two interfaces, `lo` and `ens33`. `lo` is the *loopback device* and has the MAC address `00:00:00:00:00:00` while `ens33` is an ethernet interface and has the MAC address `00:0c:29:33:3b:25`.

IPv4 Networking

To visit websites such as Google or Twitter, to check emails or to allow businesses to connect to each other, packets need to be able to roam from one link layer network to another. Often, these networks are connected just indirectly, with several intermediate link layer networks which packets have to cross to reach their actual destination.

The link layer addresses of a network interface cannot be used outside that specific link layer network. Since this address has no significance to devices in other link layer networks, a form of globally unique addresses are needed in order to implement routing. This addressing scheme, along with the overall concept of routing, is implemented by the *Internet Protocol* (IP).

NOTE

A *protocol* is a set of procedures of doing something so that all parties following the protocol are compatible to each other. A protocol can be seen as the definition of a standard. In computing, the Internet Protocol is a standard agreed upon by everyone so that different devices produced by different manufacturers can all communicate with each other.

IPv4 Addresses

IP addresses, like MAC addresses, are a way to indicate where a data packet comes from and where it is going to. IPv4 was the original protocol. IPv4 addresses are 32 bits wide giving a theoretical maximum number of 4,294,967,296 addresses. However, the number of those addresses useable by devices is much smaller as some ranges are reserved for special use cases such as broadcast addresses (which are used to reach all participants of a specific network), multicast addresses (similar to broadcast addresses, but a device must tune in like a radio) or those reserved for private use.

In their human readable format IPv4 addresses are denoted as four digits separated by a dot. Each digit can range from `0` to `255`. For example, take the following IP address:

```
192.168.0.1
```

Technically, each of these digits represents eight individual bits. Thus this address could also be written like this:

```
11000000.10101000.00000000.00000001
```

In practice the decimal notation as seen above is used. However, the bitwise representation is still important to understand subnetting.

IPv4 Subnets

In order to support routing, IP addresses can be split into two parts: the network and host portions. The network portion identifies the network that the device is on and is used to route packets to that network. The host portion is used to specifically identify a given device on a network and to hand the packet over to its specific recipient once it has reached its link layer network.

IP addresses can be broken into subnet and host parts at any point. The so-called *subnet mask* defines where this split happens. Let's reconsider the binary representation of the IP address from the former example:

```
11000000.10101000.00000000.00000001
```

Now for this IP address, the subnet mask sets each bit which belongs to the network part to `1` and each bit that belongs to the host part to `0`:

```
11111111.11111111.11111111.00000000
```

In practice the netmask is written in the decimal notation:

```
255.255.255.0
```

This means that this network ranges from `192.168.0.0` to `192.168.0.255`. Note that the first three numbers, which have all bits set in the net mask, stay unchanged in the IP addresses.

Finally, there is an alternative notation for the subnet mask, which is called *Classless Inter-Domain Routing* (CIDR). This notation just indicates how many bits are set in the subnet mask and adds this number to the IP address. In the example, 24 out of 32 bits are set to `1` in the subnet mask. This can be expressed in CIDR notation as `192.168.0.1/24`

Private IPv4 Addreses

As mentioned before, certain sections of the IPv4 address space are reserved for special use cases. One of these use cases are private address assignments. The following subnets are reserved for private addressing:

- `10.0.0.0/8`
- `172.16.0.0/12`
- `192.168.0.0/16`

Addresses out of these subnets can be used by anyone. However, these subnets can not be routed on the public internet as they are potentially used by numerous networks at the same time.

Today, most networks use these internal addresses. They allow internal communication without the need of any external address assignment. Most internet connections today just come with a single external IPv4 address. Routers map all the internal addresses to that single external IP address when forwarding packets to the internet. This is called *Network Address Translation* (NAT). The special case of NAT where a router maps internal addresses to a single external IP address is sometimes call *masquerading*. This allows any device on the inside network to establish new connections with any global IP address on the internet.

NOTE With masquerading, the internal devices can not be referenced from the internet since they do not have a globally valid address. However, this is not a security feature. Even when using masquerading, a firewall is still needed.

IPv4 Address Configuration

There are two main ways to configure IPv4 addresses on a computer. Either by assigning addresses manually or by using the *Dynamic Host Configuration Protocol* (DHCP) for automatic configuration.

When using DHCP, a central server controls which addresses are handed out to which devices. The server can also supply devices with other information about the network such as the IP addresses of DNS servers, the IP address of the default router or, in the case of more complicated setups, to start an operating system from the network. DHCP is enabled by default on many systems, therefore you will likely already have an IP address when you are connected to a network.

IP addresses can also be manually added to an interface using the command `ip addr add`. Here, we add the address `192.168.0.5` to the interface `ens33`. The network uses the netmask `255.255.255.0` which equals `/24` in CIDR notation:

```
$ sudo ip addr add 192.168.0.5/255.255.255.0 dev ens33
```

Now we can verify that the address was added using the `ip addr show` command:

```
$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
25: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 00:0c:29:33:3b:25 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.5/24 192.168.0.255 scope global ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::010c:29ff:fe33:3b25/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

The above output shows both the `lo` interface and the `ens33` interface with its address assigned with the command above.

To verify the reachability of a device, the `ping` command can be used. It sends a special type of message called an *echo request* in which the sender asks the recipient for a response. If the connection between the two devices can be successfully established, the recipient will send back an echo reply, thus verifying the connection:

```
$ ping -c 3 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=2.16 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=64 time=1.85 ms
64 bytes from 192.168.0.1: icmp_seq=3 ttl=64 time=3.41 ms

--- 192.168.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 5ms
rtt min/avg/max/mdev = 1.849/2.473/3.410/0.674 ms
```

The `-c 3` parameter makes `ping` stop after sending three echo requests. Otherwise, `ping` continues to run forever and has to be stopped by pressing `Ctrl` + `C`.

IPv4 Routing

Routing is the process in which a packet gets from the source network to the destination network. Each device maintains a routing table which contains information about which IP network can be directly reached through the device's attachment to link layer networks and which IP network can be reached by passing packets on to a router. Finally, a *default route* defines a router which receives all packets which did not match any other route.

When establishing a connection, the device looks up the target's IP address in its routing table. If an entry matches the address, the packet is either sent to the respective link layer network or passed on to the router indicated in the routing table.

Routers themselves maintain routing tables, too. When receiving a packet, a router also looks up the destination address in its own routing table and sends the packet on to the next router. This is repeated until the packet arrives at the router on the destination network. Each router involved in this journey is called a *hop*. This last router finds a direct connected link for the target address in its routing table and sends the packets to its target interface.

Most home networks only have one way out, the singular router that came from the *internet service provider* (ISP). In this case a device just forwards all packets that aren't for the internal network directly to the home router which will then send it to the provider's router for further forwarding. This is an example of the default route.

The command `ip route show` lists the current IPv4 routing table:

```
$ ip route show
127.0.0.0/8 via 127.0.0.1 dev lo0
192.168.0.0/24 dev ens33 scope link
```

To add a default route, all that's needed is the internal address of the router that's going to be the default gateway. If, for example, the router has the address `192.168.0.1`, then the following command sets it up as a default route:

```
$ sudo ip route add default via 192.168.0.1
```

To verify, run `ip route show` again:

```
$ ip route show
default via 192.168.0.1 dev ens33
127.0.0.0/8 via 127.0.0.1 dev lo0
```

```
192.168.0.0/24 dev ens33 scope link
```

IPv6 Networking

IPv6 was designed to deal with the shortcomings of IPv4, mainly the lack of addresses as more and more devices were being brought online. However, IPv6 also includes other features such as new protocols for automatic network configuration. Instead of 32 bits per address IPv6 uses 128. This allows for approximately 2^{128} addresses. However, like IPv4, the number of globally unique usable addresses is a lot smaller due to sections of the allocation being reserved for other uses. This large number of addresses means there are more than enough public addresses for every device currently connected to the internet and for many more to come, thus reducing the need for masquerading and its issues such as the delay during translation and the impossibility to directly connect to masqueraded devices.

IPv6 Addresses

Written down, the addresses use 8 groups of 4 hexadecimal digits each separated by a colon:

```
2001:0db8:0000:abcd:0000:0000:0000:7334
```

NOTE Hexadecimal digits range from `0` to `f`, so each digit can contain one of 16 different values.

To make it easier leading zeros from each group can be removed when written down however each group must contain at least one digit:

```
2001:db8:0:abcd:0:0:0:7334
```

Where multiple groups containing only zeros follow directly after each other they may be entirely replaced by '::':

```
2001:db8:0:abcd::7334
```

However, this can only happen once in each address.

IPv6 Prefix

The first 64 bits of an IPv6 address are known as the *routing prefix*. The prefix is used by routers to

determine which network a device belongs to and therefore which path the data needs to be sent on. Subnetting always happens within the prefix. ISPs usually hand out /48 or /58 prefixes to their customers, leaving them 16 or 8 bits for their internal subnetting.

There are three major prefix types in IPv6:

Global Unique Address

Wherein the prefix is assigned from the blocks reserved for global addresses. These addresses ware valid in the entire internet.

Unique Local Address

May not be routed in the internet. They may, however, be routed internally within an organization. These addresses are used within a network to ensure the devices still have an address even when there is no internet connection. They are the equivalent of the private address ranges from IPv4. The first 8 bits are always `fc` or `fd`, followed by 40 randomly generated bits.

Link Local Address

Are only valid on a particular link. Every IPv6 capable network interface has one such address, starting with `fe80`. These addresses are used internally by IPv6 to request additional addresses using automatic configuration and to find other computers on the network using the Neighbor Discovery protocol.

IPv6 Interface Identifier

While the prefix determines in which network a device resides, the interface identifier is used to enumerate the devices within a network. The last 64 bits in an IPv6 address form the interface identifier, just like the last bits of an IPv4 address.

When an IPv6 address is assigned manually, the interface identifier is set as part of the address. When using automatic address configuration, the interface identifier is either chosen randomly or derived from the device's link layer address. This makes a variation of the link layer address appear within the IPv6 address.

IPv6 Address Configuration

Like IPv4, IPv6 address can be either assigned manually or automatically. However, IPv6 has two different types of automated configuration, DHCPv6 and *Stateless Address Autoconfiguration* (SLAAC).

SLAAC is the easier of the two automated methods and built right into the IPv6 standard. The

messages use the new *Neighbor Discovery Protocol* which allows devices to find each other and request information regarding a network. This information is sent by routers and can contain IPv6 prefixes which the devices may use by combining them with an interface identifier of their choice, as long as the resulting address it not yet in use. The devices do not provide feedback to the router about the actual addresses they have created.

DHCPv6, on the other hand, is the updated DHCP made to work with the changes of IPv6. It allows for finer control over the information handed out to clients, like allowing for the same address to be handed out to the same client every time, and sending out more options to the client than SLAAC. With DHCPv6, clients need to get the explicit consent of a DHCP server in order to use an address.

The method to manually assign an IPv6 address to an interface is the same as with IPv4:

```
$ sudo ip addr add 2001:db8:0:abcd:0:0:0:7334/64 dev ens33
```

To verify the assignment has worked the same `ip addr show` command is used:

```
$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
25: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 00:0c:29:33:3b:25 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.5/24 192.168.0.255 scope global ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::010c:29ff:fe33:3b25/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
    inet6 2001:db8:0:abcd::7334/64 scope global
       valid_lft forever preferred_lft forever
```

Here we also see the link-local address `fe80::010c:29ff:fe33:3b25/64`.

Like IPv4, the `ping` command can also be used to confirm the reachability of devices through IPv6:

```
$ ping 2001:db8:0:abcd::1
```

```
PING 2001:db8:0:abcd::1(2001:db8:0:abcd::1) 56 data bytes
64 bytes from 2001:db8:0:abcd::1: icmp_seq=1 ttl=64 time=0.030 ms
64 bytes from 2001:db8:0:abcd::1: icmp_seq=2 ttl=64 time=0.040 ms
64 bytes from 2001:db8:0:abcd::1: icmp_seq=3 ttl=64 time=0.072 ms

--- 2001:db8:0:abcd::1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 43ms
rtt min/avg/max/mdev = 0.030/0.047/0.072/0.018 ms
```

NOTE On some Linux systems, `ping` does not support IPv6. These systems provide a dedicated `ping6` command instead.

To verify the link-local address again use `ping` again. But since all interfaces use the `fe80::/64` prefix, the correct interface has to be specified along with the address:

```
$ ping6 -c 1 fe80::010c:29ff:fe33:3b25%ens33
PING fe80::010c:29ff:fe33:3b25(fe80::010c:29ff:fe33:3b25) 56 data bytes
64 bytes from fe80::010c:29ff:fe33:3b25%ens33: icmp_seq=1 ttl=64 time=0.049 ms

--- fe80::010c:29ff:fe33:3b25 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.049/0.049/0.049/0.000 ms
```

DNS

IP addresses are difficult to remember and don't exactly have a high coolness factor if you're trying to market a service or product. This is where the *Domain Name System* comes into play. In its simplest form DNS is a distributed phone book that maps friendly rememberable domain names such as `example.com` to IP addresses. When, for example, a user navigates to a website, they enter the DNS hostname as part of the URL. The web browser then sends the DNS name to whichever DNS resolver has been configured. That DNS resolver will in turn find out the address that correlates to the domain. The resolver then replies with that address and the web browser tries to reach the web server at that IP address.

The resolvers that Linux uses to look up DNS data are configured in the `/etc/resolv.conf` configuration file:

```
$ cat /etc/resolv.conf
search lpi
nameserver 192.168.0.1
```

When the resolver performs a name lookup, it first checks the `/etc/hosts` file to see if it contains an address for the requested name. If it does, it returns that address and does not contact the DNS. This allows network administrators to provide name resolution without having to go through the effort of configuration a complete DNS server. Each line in that file contains one IP address followed by one or more names:

```
127.0.0.1          localhost.localdomain   localhost
::1                localhost.localdomain   localhost
192.168.0.10       server
2001:db8:0:abcd::f server
```

To perform a lookup in the DNS, use the command `host`:

```
$ host learning.lpi.org
learning.lpi.org has address 208.94.166.198
```

More detailed information can be retrieved using the command `dig`:

```
$ dig learning.lpi.org
; <<>> DiG 9.14.3 <<>> learning.lpi.org
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 21525
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: 2ac55879b1adef30a93013705d3306d2128571347df8eadf (bad)
;; QUESTION SECTION:
;learning.lpi.org.      IN  A

;; ANSWER SECTION:
learning.lpi.org.   550 IN  A   208.94.166.198

;; Query time: 3 msec
;; SERVER: 192.168.0.1#53(192.168.0.1)
;; WHEN: Sat Jul 20 14:20:21 EST 2019
;; MSG SIZE  rcvd: 89
```

Here we also see the name of the DNS record types, in this case `A` for IPv4.

Sockets

A *socket* is a communication endpoint for two programs talking to each other. If the socket is connected over a network, the programs can run on different devices, such as a web browser running on a user's laptop and a web server running in a company's data center.

There are three main types of sockets:

Unix Sockets

Which connect processes running on the same device.

UDP (User Datagram Protocol) Sockets

Which connect applications using a protocol which is fast but not resilient.

TCP (Transmission Control Protocol) Sockets

Which are more reliable than UDP sockets and, for example, confirm the receipt of data.

Unix sockets can only connect applications running on the same device. TCP and UDP sockets however can connect over a network. TCP allows for a stream of data that always arrives in the exact order it was sent. UDP is more fire and forget; the packet is sent but its delivery at the other end is not guaranteed. UDP does however lack the overhead of TCP, making it perfect for low latency applications such as online video games.

TCP and UDP both use ports to address multiple sockets on the same IP address. While the source port for a connection is usually random, target ports are standardized for a specific service. HTTP is, for example, usually hosted at port 80, HTTPS is run on port 443. SSH, a protocol to securely log into a remote Linux system, listens on port 22.

The `ss` command allows an administrator to investigate all of the sockets on a Linux computer. It shows everything from the source address, destination address and type. One of its best features is the use of filters so a user can monitor the sockets in whatever connection state they would like. `ss` can be run with a set of options as well as a filter expression to limit the information shown.

When executed without any options, the command shows a list of all established sockets. Using the `-p` option includes information on the process using each socket. The `-s` option shows a summary of sockets. There are many more options available for this tool but the last set of major ones are `-4` and `-6` for narrowing down the IP protocol to either IPv4 or IPv6 respectively, `-t` and `-u` allow the administrator to select TCP or UDP sockets and `-l` to show only socket which listen for new connections.

The following command, for example, lists all TCP sockets currently in use:

```
$ ss -t
State       Recv-Q  Send-Q    Local Address:Port      Peer Address:Port
ESTAB       0       0           192.168.0.5:49412      192.168.0.1:https
ESTAB       0       0           192.168.0.5:37616      192.168.0.1:https
ESTAB       0       0           192.168.0.5:40114      192.168.0.1:https
ESTAB       0       0           192.168.0.5:54948      192.168.0.1:imap
...
```

Guided Exercises

1. A network engineer is asked to assign two IP addresses to the `ens33` interface of a host, one IPv4 address (`192.168.10.10/24`) and one IPv6 address (`2001:0:0:abcd:0:8a2e:0370:7334/64`). What commands must they enter to achieve this?

2. Which addresses from the list below are private?

`192.168.10.1`	
`120.56.78.35`	
`172.16.57.47`	
`10.100.49.162`	
`200.120.42.6`	

3. What entry would you add into the hosts file to assign `192.168.0.15` to `example.com`?

4. What effect would the following command have?

```
sudo ip -6 route add default via 2001:db8:0:abcd::1
```

Explorational Exercises

1. Name the DNS record type used to serve the following requests:
   - Textual data
   - Reverse IP address lookup
   - A domain that has no address of its own and relies on another domain for this information
   - Mail Server
2. Linux has a feature called bridging, what does it do and how is it useful?
3. What option needs to be supplied to the `ss` command in order to view all established UDP sockets?
4. Which command shows a summary of all sockets running on a Linux device?
5. The following output is generated by the command from the previous exercise. How many TCP and UDP sockets are active?

```
Total: 978 (kernel 0)
TCP:   4 (estab 0, closed 0, orphaned 0, synrecv 0, timewait 0/0), ports 0

Transport Total     IP        IPv6
*     0         -         -
RAW   1         0         1
UDP   7         5         2
TCP   4         3         1
INET      12        8         4
FRAG      0         0         0
```

Summary

This topic covers networking your Linux computer. First we learned about the various levels of networking:

- The link layer which connects devices directly.
- The networking layer which provides routing between networks and a global address space.
- The application layer where applications connect to each other.

We have seen how IPv4 and IPv6 are used to address individual computers, and how TCP and UDP enumerate sockets used by applications to connect to each other. We also learned how DNS is used to resolve names to IP addresses.

Commands used in the exercises:

`dig`
: Query DNS information and provide verbose information about the DNS queries and responses.

`host`
: Query DNS information and provide condensed output.

`ip`
: Configure networking on Linux, including network interfaces, addresses and routing.

`ping`
: Test the connectivity to a remote device.

`ss`
: Show information regarding sockets.

Answers to Guided Exercises

1. A network engineer is asked to assign two IP addresses to the `ens33` interface of a host, one IPv4 address (`192.168.10.10/24`) and one IPv6 address (`2001:0:0:abcd:0:8a2e:0370:7334/64`). What commands must they enter to achieve this?

```
sudo ip addr add 192.168.10.10/24 dev ens33
sudo ip addr add 2001:0:0:abcd:0:8a2e:0370:7334/64 dev ens33
```

2. Which addresses from the list below are private?

`192.168.10.1`	X
`120.56.78.35`	
`172.16.57.47`	X
`10.100.49.162`	X
`200.120.42.6`	

3. What entry would you add into the hosts file to assign `192.168.0.15` to `example.com`?

```
192.168.0.15  example.com
```

4. What effect would the following command have?

```
sudo ip -6 route add default via 2001:db8:0:abcd::1
```

   It would add a default route into the table that sends all IPv6 traffic to the router with an internal address of `2001:db8:0:abcd::1`.

Answers to Explorational Exercises

1. Name the DNS record type used to serve the following requests:
   - Textual data

     TXT
   - Reverse IP address lookup

     PTR
   - A domain that has no address of its own and relies on another domain for this information

     CNAME
   - Mail Server

     MX
2. Linux has a feature called bridging, what does it do and how is it useful?

   A bridge connects multiple networking interfaces. All interfaces connected to a bridge can communicate as if they were connected to the same link layer network: All devices use IP addresses from the same subnet and do not require a router in order to connect to each other.
3. What option needs to be supplied to the `ss` command in order to view all established UDP sockets?

   The `-u` option shows all established UDP sockets.
4. Which command shows a summary of all sockets running on a Linux device?

   The `ss -s` command shows a summary of all sockets
5. The following output is generated by the command from the previous exercise. How many TCP and UDP sockets are active?

```
Total: 978 (kernel 0)
TCP:   4 (estab 0, closed 0, orphaned 0, synrecv 0, timewait 0/0), ports 0

Transport Total     IP        IPv6
*     0         -         -
RAW   1         0         1
```

```
UDP   7         5         2
TCP   4         3         1
INET      12        8         4
FRAG      0         0         0
```

11 TCP and UDP sockets are active.

Linux Professional Institute

Topic 5: Security and File Permissions

5.1 Basic Security and Identifying User Types

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 5.1

Weight

2

Key knowledge areas

- Root and standard users
- System users

Partial list of the used files, terms and utilities

- `/etc/passwd`, `/etc/shadow`, `/etc/group`
- `id`, `last`, `who`, `w`
- `sudo`, `su`

5.1 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	5 Security and File Permissions
Objective:	5.1 Basic Security and Identifying User Types
Lesson:	1 of 1

Introduction

This lesson will focus on the basic terminology of the accounts, access controls and security of local Linux systems, the command line interface (CLI) tools in a Linux system for basic security access controls and the basic files to support user and group accounts, including those used for elementary privilege escalation.

Basic security in Linux systems is modeled after Unix access controls that, despite being nearly fifty years-old, are quite effective in comparison to some popular consumer operating systems of much newer lineage. Even some other, popular, Unix-based operating systems tend to "take liberties" that are focused on "ease-of-access," while Linux does not.

Modern Linux desktop environments and interfaces simplify the creation and management of users and often automate the assignment of access controls when a user logs in — e.g., to the display, audio and other services — requiring virtually no manual system administrator intervention. However, it is important to understand the basic concepts of an underlying Linux operating system.

Accounts

Security involves many concepts, one of the most common being the general concept of access controls. Before one can tackle file access controls such as ownership and permissions, one must understand the basic concepts of Linux user *accounts*, which are broken out into several types.

Every user on a Linux system has an associated account which besides login information (like username and password) also defines how, and where, the user can interact with the system. Privileges and access controls define the "boundaries" within which each user can operate.

Identifiers (UIDs/GIDs)

The *User* and *Group Identifiers* (UIDs/GIDs) are the basic, enumerated references to accounts. Early implementations were limited 16-bit (values 0 to 65535) integers, but 21st century systems support 64-bit UIDs and GIDs. Users and groups are enumerated independently, so the same ID can stand for both a user and group.

Every user has not only an UID, but also a *primary GID*. The primary GID for a user can be unique to that user alone, and may end up not being used by any other users. However, this group could also be a group shared by numerous users. In addition to these primary groups, each user can be member of other groups, too.

By default on Linux systems, every user is assigned to a group with the same name as their username, and same GID as their UID. E.g., create a new user named `newuser` and, by default, its default group is `newuser` as well.

The Superuser Account

On Linux the superuser account is `root`, which always has UID 0. The superuser is sometimes called the *system administrator*, and has unlimited access and control over the system, including other users.

The default group for the superuser has the GID `0` and is also named `root`. The home directory for the superuser is a dedicated, top level directory, `/root`, only accessible by the `root` user himself.

Standard User Accounts

All accounts other than `root` are technically regular user accounts, but on a Linux system the colloquial term *user account* often means a "regular" (unprivileged) user account. They typically have the following properties, with select exceptions:

- UIDs starting at 1000 (4 digits), although some legacy systems may start at 500.

- A defined home directory, usually a subdirectory of `/home`, depending on the site-local configuration.
- A defined login shell. In Linux the default shell is usually the *Bourne Again Shell* (`/bin/bash`), though others may be available.

If a user account does not have a valid shell in their attributes, the user will not be able to open an interactive shell. Usually `/sbin/nologin` is used as an invalid shell. This may be purposeful, if the user will only be authenticated for services other than console or SSH access, e.g., Secure FTP (`sftp`) access only.

NOTE To avoid confusion, the term *user account* will only apply to standard or regular user accounts going forward. E.g., *system account* will be used to explain a Linux user account that is of the system user account type.

System Accounts

System accounts are typically pre-created at system installation time. These are for facilities, programs and services that will not run as the superuser. In an ideal world, these would all be operating system facilities.

The system accounts vary, but their attributes include:

- UIDs are usually under 100 (2-digit) or 500-1000 (3-digit).
- Either *no* dedicated home directory or a directory that is usually not under `/home`.
- No valid login shell (typically `/sbin/nologin`), with rare exceptions.

Most system accounts on Linux will never login, and do not need a defined shell in their attributes. Many processes owned and executed by system accounts are forked into their own environment by the system management, running with the specified, system account. These accounts usually have limited or, more often than not, *no* privileges.

NOTE From the standpoint of the LPI Linux Essentials, system accounts are UIDs <1000, with 2 or 3 digit UIDs (and GIDs).

In general, the system accounts should *not* have a valid login shell. The opposite would be a security issue in most cases.

Service Accounts

Service accounts are typically created when services are installed and configured. Similar to system accounts, these are for facilities, programs and services that will not run as the superuser.

In much documentation, system and service accounts are similar, and interchanged often. This includes the location of home directories typically being outside of `/home`, if defined at all (service accounts are often more likely to have one, compared to system accounts), and no valid login shell. Although there is no strict definition, the primary difference between system and service accounts breaks down to UID/GID.

System account

UID/GID <100 (2-digit) or <500-1000 (3-digit)

Service account

UID/GID >1000 (4+ digit), but not a "standard" or "regular" user account, an account for services, with an UID/GID >1000 (4+ digits)

Some Linux distributions still have pre-reserved service accounts under UID <100, and those could be considered a system account as well, even though they are not created at system installation. E.g., on Fedora-based (including Red Hat) Linux distributions, the user for the Apache Web server has UID (and GID) 48, clearly a system account, despite having a home directory (usually at `/usr/share/httpd` or `/var/www/html/`).

NOTE From the standpoint of the LPI Linux Essentials, system accounts are UIDs <1000, and regular user accounts are UIDs >1000. Since the regular user accounts are >1000, these UIDs can also include service accounts.

Login Shells and Home Directories

Some accounts have a login shell, while others do not for security purposes as they do not require interactive access. The default login shell on most Linux distributions is the *Bourne Again Shell*, or `bash`, but other shells may be available, like the C Shell (`csh`), Korn shell (`ksh`) or Z shell (`zsh`), to name a few.

A user can change their login shell using the `chsh` command. By default the command runs in interactive mode, and displays a prompt asking which shell should be used. The answer should be the full path to the shell binary, like below:

```
$ chsh

Changing the login shell for emma
Enter the new value, or press ENTER for the default
    Login Shell [/bin/bash]: /usr/bin/zsh
```

You can also run the command in non-interactive mode, with the `-s` parameter followed by the

path to the binary, like so:

```
$ chsh -s /usr/bin/zsh
```

Most accounts have a defined home directory. On Linux, this is usually the only location where that user account has guaranteed write access, with some exceptions (e.g., temporary file system areas). However, some accounts are purposely setup to not have any write access to even their own home directory, for security purposes.

Getting Information About Your Users

Listing basic user information is a common, everyday practice on a Linux system. In some cases, users will need to switch users and raise privilege to complete privileged tasks.

Even users have the ability to list attributes and access from the command line, using the commands below. Basic information under limited context is not a privileged operation.

Listing the current information of a user at the command line is as simple as a two letter command, `id`. The output will vary based on your login ID:

```
$ id
uid=1024(emma) gid=1024(emma) 1024(emma),20(games),groups=10240(netusers),20480(netadmin)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

In the preceding listing, the user (`emma`) has identifiers which breakdown as follows:

- `1024` = User ID (UID), followed by the username (common name aka login name) in parenthesis.
- `1024` = the *primary* Group ID (GID), followed by the groupname (common name) in parenthesis.
- A list of additional GIDs (groupnames) the user also belongs to.

Listing the last time users have logged into the system is done with the command `last`:

```
$ last
emma     pts/3        ::1              Fri Jun 14 04:28   still logged in
reboot   system boot  5.0.17-300.fc30. Fri Jun 14 04:03   still running
reboot   system boot  5.0.17-300.fc30. Wed Jun  5 14:32 - 15:19  (00:46)
reboot   system boot  5.0.17-300.fc30. Sat May 25 18:27 - 19:11  (00:43)
reboot   system boot  5.0.16-100.fc28. Sat May 25 16:44 - 17:06  (00:21)
reboot   system boot  5.0.9-100.fc28.x Sun May 12 14:32 - 14:46  (00:14)
```

```
root     tty2                         Fri May 10 21:55 - 21:55  (00:00)
    ...
```

The information listed in columns may vary, but some notable entries in the preceding listing are:

- A user (`emma`) logged in via the network (pseudo TTY `pts/3`) and is still logged in.
- The time of current boot is listed, along with the kernel. In the example above, about 25 minutes before the user logged in.
- The superuser (`root`) logged in via a virtual console (TTY `tty2`), briefly, in mid-May.

A variant of the `last` command is the `lastb` command, which lists all the last bad login attempts.

The commands `who` and `w` list only the active logins on the system:

```
$ who
emma  pts/3        2019-06-14 04:28 (::1)

$ w
 05:43:41 up  1:40,  1 user,  load average: 0.25, 0.53, 0.51
USER     TTY        LOGIN@   IDLE   JCPU   PCPU WHAT
emma  pts/3     04:28    1:14m  0.04s  0.04s -bash
```

Both commands list some of the same information. For example, one user (`emma`) is logged in with a pseudo TTY device (`pts/3`) and the time of login is 04:28.

The `w` command lists more information, including the following:

- The current time and how long the system has been up
- How many users are connected
- The *load averages* for the past 1, 5 and 15 minutes

And the additional information for each active user session.

- Select, total CPU utilization times (`IDLE`, `JCPU` and `PCPU`)
- The current process (`-bash`). The total CPU utilization time of that process is the last item (`PCPU`).

Both commands have further options to list various, additional information.

Switching Users and Escalating Privilege

In an ideal world, users would never need to escalate privilege to complete their tasks. The system would always "just work" and everything would be configured for various access.

Luckily for us, Linux — out-of-the-box — works like this for most users who are not system administrators, despite always following the *least privilege* security model.

However, there are commands that allow for privilege escalation when needed. Two of the most important ones are `su` and `sudo`.

On most Linux systems today, the `su` command is only used for escalating privileges to root, which is the default user if a username is not specified after the command name. While it can be used to switch to another user, it is not good practice: users should login from another system, over the network, or physical console or terminal on the system.

```
emma ~$ su -
Password:
root ~#
```

After entering the superuser (`root`) password, the user has a superuser shell (notice the `#` at the end of the command prompt) and is, for all intents and purposes, the superuser (`root`).

Sharing passwords is a very bad security practice, so there should be very few, if any, need for the `su` command in a modern Linux system.

The dollar symbol (`$`) should terminate the command line prompt for a non-privileged user shell, while the pound symbol (`#`) should terminate the command line prompt for the superuser (`root`) shell prompt. It is highly recommend that final character of any prompt *never* be changed from this "universally understood" standard, since this nomenclature is used in learning materials, including these.

WARNING

Never switch to the superuser (`root`) without passing the login shell (`-`) parameter. Unless explicitly instructed otherwise by the OS or software vendor when `su` is required, always execute `su -` with extremely limited exceptions. User environments may cause undesirable configuration changes and issues when used in full privilege mode as superuser.

What is the biggest issue with using `su` to switch to the superuser (`root`)? If a regular user's session has been compromised, the superuser (`root`) password could be captured. That's where "Switch User Do" (or "Superuser Do") comes in:

```
$ cat /sys/devices/virtual/dmi/id/board_serial
cat: /sys/devices/virtual/dmi/id/board_serial: Permission denied

$ sudo cat /sys/devices/virtual/dmi/id/board_serial
[sudo] password for emma:
/6789ABC/
```

In the preceding listing, the user is attempting to look up the serial number of their system board. However, the permission is denied, as that information is marked privileged.

However, using `sudo`, the user enters their own password to authenticate who they are. If they have been authorized in the `sudoers` configuration to run that command with privilege, with the options allowed, it will work.

TIP By default, the first authorized `sudo` command will authenticate subsequent `sudo` commands for a (very short) period of time. This is configurable by the system administrator.

Access Control Files

Nearly all operating systems have a set of places used to store access controls. In Linux these are typically text files located under the `/etc` directory, which is where system configuration files should be stored. By default, this directory is readable by every user on the system, but writable only by `root`.

The main files related to user accounts, attributes and access control are:

`/etc/passwd`
: This file stores basic information about the users on the system, including UID and GID, home directory, shell, etc. Despite the name, no passwords are stored here.

`/etc/group`
: This file stores basic information about all user groups on the system, like group name and GID and members.

`/etc/shadow`
: This is where user passwords are stored. They are hashed, for security.

`/etc/gshadow`
: This file stores more detailed information about groups, including a hashed password which lets users temporarily become a member of the group, a list of users who can become a

member of the group at and time and a list of group administrators.

WARNING These files are not designed to and should never be edited directly. This lesson only covers the information stored in these files, and not editing these files.

By default, every user can enter `/etc` and read the files `/etc/passwd` and `/etc/group`. And also by default no user, except `root`, may read the files `/etc/shadow` or `/etc/gshadow`.

There are also files involved with basic privilege escalation on Linux systems, like on the commands `su` and `sudo`. By default, these are only accessible by the `root` user.

`/etc/sudoers`

This file controls who can use the `sudo` command, and how.

`/etc/sudoers.d`

This directory may contain files that supplement the settings on the `sudoers` file.

From the standpoint of the LPI Linux Essentials exam, just know the path and filename of the default sudo configuration file, `/etc/sudoers`. Its configuration is beyond the scope of these materials.

WARNING Even though `/etc/sudoers` is a text file, it should never be edited directly. If any changes to its contents are needed, they should be made using the `visudo` utility.

/etc/passwd

The file `/etc/passwd` is commonly referred to as the “password file”. Each line contains multiple fields always delimited by a colon (`:`). Despite the name, the actual one-way password hash is nowadays not stored in this file.

The typical syntax for a line on this file is:

```
USERNAME:PASSWORD:UID:GID:GECOS:HOMEDIR:SHELL
```

Where:

`USERNAME`

The username aka login (name), like `root`, `nobody`, `emma`.

`PASSWORD`

Legacy location of the password hash. Almost always `x`, indicating that the password is stored in the file `/etc/shadow`.

`UID`

User ID (UID), like `0`, `99`, `1024`.

`GID`

Default Group ID (GID), like `0`, `99`, `1024`.

`GECOS`

A CSV list of user information including name, location, phone number. For example: `Emma Smith,42 Douglas St,555.555.5555`

`HOMEDIR`

Path to the user's home directory, like `/root`, `/home/emma`, etc.

`SHELL`

The default shell for this user, like `/bin/bash`, `/sbin/nologin`, `/bin/ksh`, etc.

For example, the following line describes the user `emma`:

```
emma:x:1000:1000:Emma Smith,42 Douglas St,555.555.5555:/home/emma:/bin/bash
```

Understanding the GECOS Field

The GECOS field contains three (3) or more fields, delimited by a comma (`,`), aka a list of *Comma Separated Values* (CSV). Although there is no enforced standard, the fields are usually in the following order:

```
NAME,LOCATION,CONTACT
```

Where:

`NAME`

is the user's "Full Name", or the "Software Name" in the case of a service account.

`LOCATION`

is usually the user's physical location within a building, room number or the contact

department or person in the case of a service account.

`CONTACT`

lists contact information such as home or work telephone number.

Additional fields may include additional contact information, such as a home number or e-mail address. To change the information on the GECOS field, use the `chfn` command and answer the questions, like below. If no username is provided after the command name, you will change information for the current user:

```
$ chfn

Changing the user information for emma
Enter the new value, or press ENTER for the default
    Full Name: Emma Smith
    Room Number []: 42
    Work Phone []: 555.555.5555
    Home Phone []: 555.555.6666
```

/etc/group

The `/etc/group` file contains fields always delimited by a colon (`:`), storing basic information about the groups on the system. It is sometimes called the "group file". The syntax for each line is:

```
NAME:PASSWORD:GID:MEMBERS
```

Where:

`NAME`

is the group name, like `root`, `users`, `emma`, etc.

`PASSWORD`

legacy location of an optional group password hash. Almost always `x`, indicating that the password (if defined) is stored in the file `/etc/gshadow`.

`GID`

Group ID (GID), like `0`, `99`, `1024`.

`MEMBERS`

a comma-separated list of usernames which are members of the group, like `jsmith,emma`.

The example below shows a line containing information about the `students` group:

```
students:x:1023:jsmith,emma
```

The user does not need to be listed in the members field when the group is the primary group for a user. If a user is listed, then it is redundant — i.e., there is no change in functionality, listed or not.

NOTE Use of passwords for groups are beyond the scope of this section, however if defined the password hash is stored in the file `/etc/gshadow`. This is also beyond the scope of this section.

/etc/shadow

The following table lists the attributes stored in the file `/etc/shadow`, commonly referred to as the "shadow file". The file contains fields always delimited by a colon (`:`). Although the file has many fields, most are beyond the scope of this lesson, other than the first two.

The basic syntax for a line on this file is:

```
USERNAME:PASSWORD:LASTCHANGE:MINAGE:MAXAGE:WARN:INACTIVE:EXPDATE
```

Where:

`USERNAME`
: The username (same as `/etc/passwd`), like `root`, `nobody`, `emma`.

`PASSWORD`
: A one-way hash of the password, including preceding salt. For example: `!!`, `!$1$01234567$ABC...`, `$6$012345789ABCDEF$012....`

`LASTCHANGE`
: Date of the last password change in days since the "epoch", such as `17909`.

`MINAGE`
: Minimum password age in days.

`MAXAGE`
: Maximum password age in days.

`WARN`

Warning period before password expiration, in days.

`INACTIVE`

Maximum password age past expiration, in days.

`EXPDATE`

Date of password expiration, in days since the “epoch”.

In the example below you can see a sample entry from the `/etc/shadow` file. Note that some values, like `INACTIVE` and `EXPDATE` are undefined.

```
emma:$6$nP532JDDogQYZF8I$bjFNh9eT1xpb9/n6pmjlIwgu7hGjH/eytSdttbmVv0MlyTMFgBIXESFNUmTo9EGxxH1
OT1HGQzR0so4n1npbE0:18064:0:99999:7:::
```

The “epoch” of a POSIX system is midnight (0000), Universal Coordinate Time (UTC), on Thursday, January 1st, 1970. Most POSIX dates and times are in either seconds since “epoch”, or in the case of the file `/etc/shadow`, days since “epoch”.

NOTE The shadow file is designed to be only readable by the superuser, and select, core system authentication services that check the one-way password hash at login or other authentication-time.

Although different authentication solutions exist, the elementary method of password storage is the one-way *hash function*. This is done so the password is never stored in clear-text on a system, as the hashing function is not reversible. You can turn a password into a hash, but (ideally) it is not possible to turn a hash back into a password.

At most, a brute force method is required to hash all combinations of a password, until one matches. To mitigate the issue of a password hash being cracked on one system, Linux systems use a random “salt” on each password hash for a user. So the hash for a user password on one Linux system will usually not be the same as on another Linux system, even if the password is the same.

In the file `/etc/shadow`, the password may take several forms. These forms typically include the following:

`!!`

This means a “disabled” account (no authentication possible), with no password hash stored.

`!$1$01234567$ABC…`

A “disabled” account (due to the initial exclamation mark), with a prior hash function, hash salt

and hash string stored.

`$1$0123456789ABC$012…`

An “enabled” account, with a hash function, hash salt and hash string stored.

The hash function, hash salt and hash string are preceded and delimited by a dollar symbol (`$`). The hash salt length must be between eight and sixteen (8-16) characters. Examples of the three most common are as follows:

`$1$01234567$ABC…`

A hash function of MD5 (`1`), with an example hash length of eight.

`$5$01234567ABCD$012…`

A hash function of SHA256 (`5`), with an example hash length of twelve.

`$6$01234567ABCD$012…`

A hash function of SHA512 (`6`), with an example hash length of twelve.

NOTE

The MD5 hash function is considered cryptographically insecure with today’s (2010s and later) level of ASIC and even general computing SIMD performance. E.g., the US Federal Information Processing Standards (FIPS) does not allow MD5 to be used for any cryptographic functions, only very limited aspects of validation, but not the integrity of digital signatures or similar purposes.

From the standpoint of the LPI Linux Essentials objectives and exam, just understand the password hash for a local user is only stored in the file `/etc/shadow` which only select, authentication services can read, or the superuser can modify via other commands.

Guided Exercises

1. Consider the following output of the `id` command:

```
$ id emma
uid=1000(emma) gid=1000(emma)
groups=1000(emma),4(adm),5(tty),10(uucp),20(dialout),27(sudo),46(plugdev)
```

   In which files are the following attributes stored?

UID and GID	
Groups	

   - Additionally, in which file is the user password stored?

2. Which of the following types of cryptography is used by default to store passwords locally on a Linux system?
   - Asymmetric
   - One-way Hash
   - Symmetric
   - ROT13

3. If an account has a User ID (UID) enumerated under 1000, what type of account is this?

4. How can you get a list of the active logins in your system, and a count of them as well?

5. Using the `grep` command, we got the result below with information about the user `emma`.

```
$ grep emma /etc/passwd
emma:x:1000:1000:Emma Smith,42 Douglas St,555.555.5555,:/home/emma:/bin/ksh
```

   Fill in the blanks of the chart with the appropriate information using the output of the previous command.

Username	
Password	
UID	
Primary GID	
GECOS	
Home Directory	
Shell	

Explorational Exercises

1. Compare the results of `last` to `w` and `who`. What details are missing from each of the commands compared to one another?

2. Try issuing the commands `who` and `w -his`.
   - What information has been removed from the output of the `w` command with the "no header" (`-h`) and "short" (`-s`) options?
   - What information has been added in the output the `w` command with the "ip address" (`-i`) option?

3. Which file is the file that stores a user account's one-way password hash?

4. Which file contains the list of groups a user account is a member of? What logic could be used to compile a list of a groups a user account is a member of?

5. One or more (1+) of the following files are not readable by regular, unprivileged users, by default. Which ones?
   - `/etc/group`
   - `/etc/passwd`
   - `/etc/shadow`
   - `/etc/sudoers`

6. How would you change the current user's login shell to the Korn Shell (`/usr/bin/ksh`) in non-interactive mode?

7. Why is the home directory of the `root` user not placed within `/home` directory?

Summary

In this lesson we have discovered the Linux user and group databases. We have learned the most important properties of users and groups, including their names and their numeric IDs. We have also investigated how password hashing works on Linux and how users are assigned to groups.

All of this information is stored in the following four files, which provide the most basic, local security access controls on a Linux system:

`/etc/passwd`
: All system-local user account POSIX attributes, other than password hash, readable by all.

`/etc/group`
: All system-local group account POSIX attributes, readable by all.

`/etc/shadow`
: All system-local user password hashes (and expiration information), unreadable by any (only select processes).

`/etc/sudoers`
: All system-local privilege escalation information/allowance by the `sudo` command.

The following commands were discussed in this lesson:

`id`
: List real (or effective) user and group IDs.

`last`
: List users who logged in last.

`who`
: List users who are currently logged in.

`w`
: Similar to `who` but with additional context.

`su`
: Switch to another user with a login shell, or run commands as that user, by passing that user's password.

sudo

Switch User (or Superuser) Do, if entitled, the current user enters their own password (if required) to raise privilege.

chsh

Change a user's shell.

chfn

Change the user's information on the GECOS field.

Answers to Guided Exercises

1. Consider the following output of the `id` command:

```
$ id emma
uid=1000(emma) gid=1000(emma)
groups=1000(emma),4(adm),5(tty),10(uucp),20(dialout),27(sudo),46(plugdev)
```

   In which files are the following attributes stored?

UID and GID	`/etc/passwd`
Groups	`/etc/group`

   - Additionally, in which file is the user password stored?

     The hashed user password is stored in `/etc/shadow`.

2. Which of the following types of cryptography is used by default to store passwords locally on a Linux system?

   By default, a one-way hash is used to store passwords.

3. If an account has a User ID (UID) enumerated under 1000, what type of account is this?

   Accounts with a UID lower than 1000 generally are system accounts.

4. How can you get a list of the active logins in your system, and a count of them as well?

   Use the `w` command. Besides a list of all active logins, it will also show information like how many users are logged in, along the system load and uptime.

5. Using the `grep` command, we got the result below with information about the user `emma`.

```
$ grep emma /etc/passwd
emma:x:1000:1000:Emma Smith,42 Douglas St,555.555.5555,:/home/emma:/bin/ksh
```

   Fill in the blanks of the chart with the appropriate information using the output of the previous command.

Username	`emma`

Password	`x` - should always be `x` for a valid, active user login
UID	`1000`
Primary GID	`1000`
GECOS	`Emma Smith,42 Douglas St,555.555.5555`
Home Directory	`/home/emma`
Shell	`/bin/ksh`

Answers to Explorational Exercises

1. Compare the results of `last` to `w` and `who`. What details are missing from each of the commands compared to one another?

   The `w` and `who` tools only list current users logged into the system, whereas `last` also lists users that have disconnected. The `w` command lists system utilization, while `who` does not.

2. Try issuing the commands `who` and `w -his`.
   - What information has been removed from the output of the `w` command with the "no header" (`-h`) and "short" (`-s`) options?

     The header is not printed, which is useful for parsing, and the login time and select CPU information is not listed, respectively.

   - What information has been added in the output the `w` command with the "ip address" (`-i`) option?

     This prints the IP address, instead of attempting DNS resolution, printing the hostname. This option to `w` better matches the default output of the `last` command.

3. Which file is the file that stores a user account's one-way password hash?

   The file `/etc/shadow` stores a user account's one-way password hash, since it is not readable by a regular, unprivileged user account, unlike file `/etc/passwd`.

4. Which file contains the list of groups a user account is a member of? What logic could be used to compile a list of a groups a user account is a member of?

   The file `/etc/group` has a CSV list of usernames in the last field, "members", of any line for a group.

   Any line in the file `/etc/group` where the user is listed in the final field, "members", would mean the user is a member of that group — assuming it is correctly formatted (CSV delimited). Additionally, the user's primary group membership in the `/etc/passwd` file will also have a matching entry in the `/etc/group` file for both the group name and GID.

5. One or more (1+) of the following files are not readable by regular, unprivileged users, by default. Which ones?
   - `/etc/group`
   - `/etc/passwd`

- `/etc/shadow`
- `/etc/sudoers`

  The files `/etc/shadow` and `/etc/sudoers` are not readable by default, except by select services or the superuser. These answers will be customized, based on the systems and usernames used in the laboratory.

6. How would you change the current user's login shell to the Korn Shell (`/usr/bin/ksh`) in non-interactive mode?

```
$ chsh -s /usr/bin/ksh
```

7. Why is the home directory of the `root` user not placed within `/home` directory?

   Because the `root` account is required to troubleshoot and fix errors, which might include file system issues related to the `/home` directory. In such cases, `root` should be fully functional even when the `/home` file system is not available yet.

5.2 Creating Users and Groups

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 5.2

Weight

2

Key knowledge areas

- User and group commands
- User IDs

Partial list of the used files, terms and utilities

- `/etc/passwd`, `/etc/shadow`, `/etc/group`, `/etc/skel/`
- `useradd`, `groupadd`
- `passwd`

5.2 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	5 Security and File Permissions
Objective:	5.2 Creating Users and Groups
Lesson:	1 of 1

Introduction

Managing users and groups on a Linux machine is one of the key aspects of system administration. In fact, Linux is a multi-user operating system in which multiple users can use the same machine at the same time.

Information about users and groups is stored in four files within the `/etc/` directory tree:

`/etc/passwd`
: a file of seven colon-delimited fields containing basic information about users

`/etc/group`
: a file of four colon-delimited fields containing basic information about groups

`/etc/shadow`
: a file of nine colon-delimited fields containing encrypted user passwords

`/etc/gshadow`
: a file of four colon-delimited fields file containing encrypted group passwords

All of these files are updated by a suite of command-line tools for user and group management, which we'll discuss later in this lesson. They can also be managed by graphical applications, specific to each Linux distribution, which provide simpler and more immediate management interfaces.

WARNING Even though the files are plain text, do not edit them directly. Always use the tools provided with your distribution for this purpose.

The File `/etc/passwd`

`/etc/passwd` is a world-readable file that contains a list of users, each on a separate line:

```
frank:x:1001:1001::/home/frank:/bin/bash
```

Each line consists of seven colon-delimited fields:

Username

The name used when the user logs into the system.

Password

The encrypted password (or an `x` if shadow passwords are used).

User ID (UID)

The ID number assigned to the user in the system.

Group ID (GID)

The primary group number of the user in the system.

GECOS

An optional comment field, which is used to add extra information about the user (such as the full name). The field can contain multiple comma-separated entries.

Home directory

The absolute path of the user's home directory.

Shell

The absolute path of the program that is automatically launched when the user logs into the system (usually an interactive shell such as `/bin/bash`).

The File `/etc/group`

`/etc/group` is a world-readable file that contains a list of groups, each on a separate line:

```
developer:x:1002:
```

Each line consists of four colon-delimited fields:

Group Name

The name of the group.

Group Password

The encrypted password of the group (or an `x` if shadow passwords are used).

Group ID (GID)

The ID number assigned to the group in the system.

Member list

A comma-delimited list of users belonging to the group, except those for whom this is the primary group.

The File `/etc/shadow`

`/etc/shadow` is a file readable only by root and users with root privileges and contains the encrypted passwords of the users, each on a separate line:

```
frank:$6$i9gjM4Md4MuelZCd$7jJa8Cd2bbADFH4dwtfvTvJLOYCCCBf/.jYbK1IMYx7Wh4fErXcc2xQVU2N1gb97yI
YaiqH.jjJammzof2Jfr/:18029:0:99999:7:::
```

Each line consists of nine colon-delimited fields:

Username

The name used when user logs into the system.

Encrypted password

The encrypted password of the user (if the value is `!`, the account is locked).

Date of last password change

The date of the last password change, as number of days since 01/01/1970. A value of `0` means

that the user must change the password at the next access.

Minimum password age

The minimum number of days, after a password change, which must pass before the user will be allowed to change the password again.

Maximum password age

The maximum number of days that must pass before a password change is required.

Password warning period

The number of days, before the password expires, during which the user is warned that the password must be changed.

Password inactivity period

The number of days after a password expires during which the user should update the password. After this period, if the user does not change the password, the account will be disabled.

Account expiration date

The date, as number of days since 01/01/1970, in which the user account will be disabled. An empty field means that the user account will never expire.

A reserved field

A field that is reserved for future use.

The File `/etc/gshadow`

`/etc/gshadow` is a file readable only by root and by users with root privileges that contains encrypted passwords for groups, each on a separate line:

```
developer:$6$7QUIhUX1WdO6$H7kOYgsboLkDseFHpk04lwAtweSUQHipoxIgo83QNDxYtYwgmZTCU0qSCuCkErmyR2
63rvHiLctZVDR7Ya9Ai1::
```

Each line consists of four colon-delimited fields:

Group name

The name of the group.

Encrypted password

The encrypted password for the group (it is used when a user, who is not a member of the

group, wants to join the group using the `newgrp` command — if the password starts with `!`, no one is allowed to access the group with `newgrp`).

Group administrators

A comma-delimited list of the administrators of the group (they can change the password of the group and can add or remove group members with the `gpasswd` command).

Group members

A comma-delimited list of the members of the group.

Now that we've seen where user and group information is stored, let's talk about the most important command-line tools to update these files.

Adding and Deleting User Accounts

In Linux, you add a new user account with the `useradd` command, and you delete a user account with the `userdel` command.

If you want to create a new user account named `frank` with a default setting, you can run the following:

```
# useradd frank
```

After creating the new user, you can set a password using `passwd`:

```
# passwd frank
Changing password for user frank.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

Both of these commands require root authority. When you run the `useradd` command, the user and group information stored in password and group databases are updated for the newly created user account and, if specified, the home directory of the new user is created as well as a group with the same name as the user account.

TIP

Remember that you can always use the `grep` utility to filter the password and group databases, displaying only the entry that refers to a specific user or group. For the above example you can use

> `cat /etc/passwd | grep frank`
>
> or
>
> `grep frank /etc/passwd`
>
> to see basic information on the newly created `frank` account.

The most important options which apply to the `useradd` command are:

`-c`
Create a new user account with custom comments (for example full name).

`-d`
Create a new user account with a custom home directory.

`-e`
Create a new user account by setting a specific date on which it will be disabled.

`-f`
Create a new user account by setting the number of days after the password expires during which the user should update the password.

`-g`
Create a new user account with a specific GID

`-G`
Create a new user account by adding it to multiple secondary groups.

`-m`
Create a new user account with its home directory.

`-M`
Create a new user account without its home directory.

`-s`
Create a new user account with a specific login shell.

`-u`
Create a new user account with a specific UID.

Once the new user account is created, you can use the `id` and `groups` commands to find out its

UID, GID and the groups to which it belongs.

```
# id frank
uid=1000(frank) gid=1000(frank) groups=1000(frank)
# groups frank
frank : frank
```

TIP

Remember to check and eventually edit the `/etc/login.defs` file, which defines the configuration parameters that control the creation of users and groups. For example, you can set the range of UIDs and GIDs that can be assigned to new user and group accounts, specify that you don't need to use the `-m` option to create the new user's home directory and if the system should automatically create a new group for each new user.

If you want to delete a user account, you can use the `userdel` command. In particular, this command updates the information stored in the account databases, deleting all entries referring to the specified user. The `-r` option also removes the user's home directory and all of its contents, along with the user's mail spool. Other files, located elsewhere, must be searched for and deleted manually.

```
# userdel -r frank
```

As before, you need root authority to delete user accounts.

The Skeleton Directory

When you add a new user account, even creating its home directory, the newly created home directory is populated with files and folders that are copied from the skeleton directory (by default `/etc/skel`). The idea behind this is simple: a system administrator wants to add new users having the same files and directories in their home. Therefore, if you want to customize the files and folders that are created automatically in the home directory of the new user accounts, you must add these new files and folders to the skeleton directory.

TIP

Note that the profile files that are usually found in the skeleton directory are hidden files. Therefore, if you want to list all the files and folders in the skeleton directory, which will be copied to the home dir of the newly created users, you must use the `ls -Al` command.

Adding and Deleting Groups

As for group management, you can add or delete groups using the `groupadd` and `groupdel` commands.

If you want to create a new group named `developer`, you can run the following command as root:

```
# groupadd -g 1090 developer
```

The `-g` option of this command creates a group with a specific GID.

If you want to delete the `developer` group, you can run the following:

```
# groupdel developer
```

WARNING

Remember that when you add a new user account, the primary group and the secondary groups to which it belongs must exist before launching the `useradd` command. Also, you cannot delete a group if it is the primary group of a user account.

The `passwd` Command

This command is primarily used to change a user's password. Any user can change their password, but only root can change any user's password.

Depending on the `passwd` option used, you can control specific aspects of password aging:

`-d`

Delete the password of a user account (thus setting an empty password, making it a passwordless account).

`-e`

Force the user account to change the password.

`-l`

Lock the user account (the encrypted password is prefixed with an exclamation mark).

`-u`

Unlock the user account (it removes the exclamation mark).

`-S`

Output information about the password status for a specific account.

These options are available only for root. To see the full list of options, refer to the man pages.

Guided Exercises

1. For each of the following entries, indicate the file to which it refers:
   - `developer:x:1010:frank,grace,dave`
   - `root:x:0:0:root:/root:/bin/bash`
   - `henry:$1$.AbCdEfGh123456789A1b2C3d4.:18015:20:90:5:30::`
   - `henry:x:1000:1000:User Henry:/home/henry:/bin/bash`
   - `staff:!:dave:carol,emma`

2. Observe the following output to answer the next seven questions:

```
# cat /etc/passwd | tail -3
dave:x:1050:1050:User Dave:/home/dave:/bin/bash
carol:x:1051:1015:User Carol:/home/carol:/bin/sh
henry:x:1052:1005:User Henry:/home/henry:/bin/tcsh
# cat /etc/group | tail -3
web_admin:x:1005:frank,emma
web_developer:x:1010:grace,kevin,christian
dave:x:1050:
# cat /etc/shadow | tail -3
dave:$6$AbCdEfGh123456789A1b2C3D4e5F6G7h8i9:0:20:90:7:30::
carol:$6$q1w2e3r4t5y6u7i8AbcDeFgHiLmNoPqRsTu:18015:0:60:7:::
henry:!$6$123456789aBcDeFgHa1B2c3d4E5f6g7H8I9:18015:0:20:5:::
# cat /etc/gshadow | tail -3
web_admin:!:frank:frank,emma
web_developer:!:kevin:grace,kevin,christian
dave:!::
```

   - What is the User ID (UID) and Group ID (GID) of `carol`?
   - What shell is set for `dave` and `henry`?

- What is the name of the primary group of `henry`?
- What are the members of the `web_developer` group? Which of these are group administrators?
- Which user cannot log into the system?
- Which user should change the password the next time they log into the system?
- How many days must pass before a password change is required for `carol`?

Explorational Exercises

1. Working as root, run the `useradd -m dave` command to add a new user account. What operations does this command perform? Assume that `CREATE_HOME` and `USERGROUPS_ENAB` in `/etc/login.defs` are set to yes.

2. Now that you have created the `dave` account, can this user login to the system?

3. Identify the User ID (UID) and Group ID (GID) of `dave` and all members of the `dave` group.

4. Create the `sys_admin`, `web_admin` and `db_admin` groups and identify their Group IDs (GIDs).

5. Add a new user account named `carol` with UID 1035 and set `sys_admin` as its primary group and `web_admin` and `db_admin` as its secondary groups.

6. Delete the `dave` and `carol` user accounts and the `sys_admin`, `web_admin` and `db_admin` groups that you have previously created.

7. Run the `ls -l /etc/passwd /etc/group /etc/shadow /etc/gshadow` command and describe the output that it gives you in terms of file permissions. Which of these four files are shadowed for security reasons? Assume your system uses shadow passwords.

8. Run the `ls -l /usr/bin/passwd` command. Which special bit is set and what is its meaning?

Summary

In this lesson, you learned:

- The fundamentals of user and group management in Linux
- Manage user and group information stored in password and group databases
- Maintain the skeleton directory
- Add and remove user accounts
- Add and remove group accounts
- Change the password of user accounts

The following commands were discussed in this lesson:

`useradd`
: Create a new user account.

`groupadd`
: Create a new group account.

`userdel`
: Delete a user account.

`groupdel`
: Delete a group account.

`passwd`
: Change the password of user accounts and control all aspects of password aging.

Answers to Guided Exercises

1. For each of the following entries, indicate the file to which it refers:
   - `developer:x:1010:frank,grace,dave`

     `/etc/group`
   - `root:x:0:0:root:/root:/bin/bash`

     `/etc/passwd`
   - `henry:$1$.AbCdEfGh123456789A1b2C3d4.:18015:20:90:5:30::`

     `/etc/shadow`
   - `henry:x:1000:1000:User Henry:/home/henry:/bin/bash`

     `/etc/passwd`
   - `staff:!:dave:carol,emma`

     `/etc/gshadow`
2. Observe the following output to answer the next seven questions:

```
# cat /etc/passwd | tail -3
dave:x:1050:1050:User Dave:/home/dave:/bin/bash
carol:x:1051:1015:User Carol:/home/carol:/bin/sh
henry:x:1052:1005:User Henry:/home/henry:/bin/tcsh
# cat /etc/group | tail -3
web_admin:x:1005:frank,emma
web_developer:x:1010:grace,kevin,christian
dave:x:1050:
# cat /etc/shadow | tail -3
dave:$6$AbCdEfGh123456789A1b2C3D4e5F6G7h8i9:0:20:90:7:30::
carol:$6$q1w2e3r4t5y6u7i8AbcDeFgHiLmNoPqRsTu:18015:0:60:7:::
henry:!$6$123456789aBcDeFgHa1B2c3d4E5f6g7H8I9:18015:0:20:5:::
# cat /etc/gshadow | tail -3
web_admin:!:frank:frank,emma
web_developer:!:kevin:grace,kevin,christian
dave:!::
```

   - What is the User ID (UID) and Group ID (GID) of `carol`?

The UID is 1051 and the GID is 1015 (the third and fourth fields in `/etc/passwd`).

- What shell is set for `dave` and `henry`?

  `dave` uses /bin/bash and `henry` uses /bin/tcsh (the seventh field in `/etc/passwd`).

- What is the name of the primary group of `henry`?

  The group name is `web_admin` (the first field in `/etc/group`).

- What are the members of the `web_developer` group? Which of these are group administrators?

  The members are `grace`, `kevin` and `christian` (the fourth field in `/etc/group`), but only `kevin` is the administrator of the group (the third field in `/etc/gshadow`).

- Which user cannot log into the system?

  The `henry` user account is locked (it has an exclamation mark in front of the password hashes in `/etc/shadow`).

- Which user should change the password the next time they log into the system?

  If the third field (Date of Last Password Change) in `/etc/shadow` is 0, the user should change their password the next time they log into the system. Therefore, `dave` must change their password.

- How many days must pass before a password change is required for `carol`?

  60 days (the fifth field in `/etc/shadow`).

Answers to Explorational Exercises

1. Working as root, run the `useradd -m dave` command to add a new user account. What operations does this command perform? Assume that `CREATE_HOME` and `USERGROUPS_ENAB` in `/etc/login.defs` are set to yes.

   The command adds a new user, named `dave`, to the list of users in the system. The home directory of `dave` is created (by default `/home/dave`) and the files and directories contained in the skeleton directory are copied to the home directory. Finally, new group is created with the same name as the user account.

2. Now that you have created the `dave` account, can this user login to the system?

   No, because the `dave` account is locked (see the exclamation mark in `/etc/shadow`).

   ```
   # cat /etc/shadow | grep dave
   dave:!:18015:0:99999:7:::
   ```

   If you set a password for `dave`, the account will be unlocked. You can do this using the `passwd` command.

   ```
   # passwd dave
   Changing password for user dave.
   New UNIX password:
   Retype new UNIX password:
   passwd: all authentication tokens updated successfully.
   ```

3. Identify the User ID (UID) and Group ID (GID) of `dave` and all members of the `dave` group.

   ```
   # cat /etc/passwd | grep dave
   dave:x:1015:1019::/home/dave:/bin/sh
   # cat /etc/group | grep 1019
   dave:x:1019:
   ```

   The UID and GID of `dave` are 1015 and 1019 respectively (the third and fourth fields in `/etc/passwd`) and the `dave` group has no members (the fourth field in `/etc/group` is empty).

4. Create the `sys_admin`, `web_admin` and `db_admin` groups and identify their Group IDs (GIDs).

   ```
   # groupadd sys_admin
   ```

```
# groupadd web_admin
# groupadd db_admin
# cat /etc/group | grep admin
sys_admin:x:1020:
web_admin:x:1021:
db_admin:x:1022:
```

The GIDs for the `sys_admin`, `web_admin` and `db_admin` groups are 1020, 1021 and 1022 respectively.

5. Add a new user account named `carol` with UID 1035 and set `sys_admin` as its primary group and `web_admin` and `db_admin` as its secondary groups.

```
# useradd -u 1035 -g 1020 -G web_admin,db_admin carol
# id carol
uid=1035(carol) gid=1020(sys_admin) groups=1020(sys_admin),1021(web_admin),1022(db_admin)
```

6. Delete the `dave` and `carol` user accounts and the `sys_admin`, `web_admin` and `db_admin` groups that you have previously created.

```
# userdel -r dave
# userdel -r carol
# groupdel sys_admin
# groupdel web_admin
# groupdel db_admin
```

7. Run the `ls -l /etc/passwd /etc/group /etc/shadow /etc/gshadow` command and describe the output that it gives you in terms of file permissions. Which of these four files are shadowed for security reasons? Assume your system uses shadow passwords.

```
# ls -l /etc/passwd /etc/group /etc/shadow /etc/gshadow
-rw-r--r-- 1 root root    853 mag  1 08:00 /etc/group
-rw-r----- 1 root shadow 1203 mag  1 08:00 /etc/gshadow
-rw-r--r-- 1 root root   1354 mag  1 08:00 /etc/passwd
-rw-r----- 1 root shadow 1563 mag  1 08:00 /etc/shadow
```

The `/etc/passwd` and `/etc/group` files are world readable and are shadowed for security reasons. When shadow passwords are used, you can see an `x` in the second field of these files because the encrypted passwords for users and groups are stored in `/etc/shadow` and `/etc/gshadow`, which are readable only by root and, in some systems, also by members

belonging to the `shadow` group.

8. Run the `ls -l /usr/bin/passwd` command. Which special bit is set and what is its meaning?

```
# ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 42096 mag 17  2015 /usr/bin/passwd
```

The `passwd` command has the SUID bit set (the fourth character of this line), which means that the command is executed with the privileges of the file's owner (thus root). This is how ordinary users can change their password.

5.3 Managing File Permissions and Ownership

Reference to LPI objectives

Linux Essentials version 1.6, Exam 010, Objective 5.3

Weight

2

Key knowledge areas

- File and directory permissions and ownership

Partial list of the used files, terms and utilities

- `ls -l`, `ls -a`
- `chmod`, `chown`

Linux Professional Institute

5.3 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	5 Security and File Permissions
Objective:	5.3 Managing File Permissions and Ownership
Lesson:	1 of 1

Introduction

Being a multi-user system, Linux needs some way to track who owns each file and whether or not a user is allowed to perform actions on that file. This is to ensure the privacy of users who might want to keep the content of their files confidential, as well as to ensure collaboration by making certain files accessible to multiple users.

This is done through a three-level permissions system: every file on disk is owned by a user and a user group and has three sets of permissions: one for its owner, one the group who owns the file and one for everyone else. In this lesson, you will learn how to query the permissions for a file and how to manipulate them.

Querying Information about Files and Directories

The command `ls` is used to get a listing of the contents of any directory. In this basic form, all you get are the filenames:

```
$ ls
```

```
Another_Directory  picture.jpg  text.txt
```

But there is much more information available for each file, including type, size, ownership and more. To see this information you must ask `ls` for a "long form" listing, using the `-l` parameter:

```
$ ls -l
total 536
drwxrwxr-x 2 carol carol   4096 Dec 10 15:57 Another_Directory
-rw------- 1 carol carol 539663 Dec 10 10:43 picture.jpg
-rw-rw-r-- 1 carol carol   1881 Dec 10 15:57 text.txt
```

Each column on the output above has a meaning:

- The *first* column on the listing shows the file type and permissions.

  For example, on `drwxrwxr-x`:

  - The first character, `d`, indicates the file type.
  - The next three characters, `rwx`, indicate the permissions for the owner of the file, also referred to as *user* or `u`.
  - The next three characters, `rwx`, indicate the permissions of the *group* owning the file, also referred to as `g`.
  - The last three characters, `r-x`, indicate the permissions for anyone else, also known as *others* or `o`.
- The *second* column indicates the number of hard links pointing to that file. For a directory, this means the number of subdirectories, plus a link to itself (`.`) and the parent directory (`..`).
- The *third* and *fourth* columns show ownership information: respectively the user and group that own the file.
- The *fifth* column shows the filesize, in bytes.
- The *sixth* column shows the precise date and time, or *timestamp*, when the file was last modified.
- The *seventh* and last column shows the file name.

If you wish to see the filesizes in "human readable" format, add the `-h` parameter to `ls`. Files less than one kilobyte in size will have the size shown in bytes. Files with more than one kilobyte and less than one megabyte will have a `K` added after the size, indicating the size is in kilobytes. The same follows for filesizes in the megabyte (`M`) and gigabyte (`G`) ranges:

```
$ ls -lh
total 1,2G
drwxrwxr-x 2 carol carol 4,0K Dec 10 17:59 Another_Directory
----r--r-- 1 carol carol    0 Dec 11 10:55 foo.bar
-rw-rw-r-- 1 carol carol 1,2G Dec 20 18:22 HugeFile.zip
-rw------- 1 carol carol 528K Dec 10 10:43 picture.jpg
---xr-xr-x 1 carol carol   33 Dec 11 10:36 test.sh
-rwxr--r-- 1 carol carol 1,9K Dec 20 18:13 text.txt
-rw-rw-r-- 1 carol carol 2,6M Dec 11 22:14 Zipped.zip
```

To only show information on a specific set of files, add the names of these files to `ls`:

```
$ ls -lh HugeFile.zip test.sh
total 1,2G
-rw-rw-r-- 1 carol carol 1,2G Dec 20 18:22 HugeFile.zip
---xr-xr-x 1 carol carol   33 Dec 11 10:36 test.sh
```

What about Directories?

If you try to query information about a directory using `ls -l`, it will show you a listing of the directory contents instead:

```
$ ls -l Another_Directory/
total 0
-rw-r--r-- 1 carol carol 0 Dec 10 17:59 another_file.txt
```

To avoid this and query information about the directory itself, add the `-d` parameter to `ls`:

```
$ ls -l -d Another_Directory/
drwxrwxr-x 2 carol carol 4096 Dec 10 17:59 Another_Directory/
```

Seeing Hidden Files

The directory listing we have retrieved using `ls -l` before is incomplete:

```
$ ls -l
total 544
drwxrwxr-x 2 carol carol   4096 Dec 10 17:59 Another_Directory
```

```
-rw------- 1 carol carol 539663 Dec 10 10:43 picture.jpg
-rw-rw-r-- 1 carol carol   1881 Dec 10 15:57 text.txt
```

There are three other files in that directory, but they are hidden. On Linux, files whose name starts with a period (`.`) are automatically hidden. To see them we need to add the `-a` parameter to `ls`:

```
$ ls -l -a
total 544
drwxrwxr-x 3 carol carol   4096 Dec 10 16:01 .
drwxrwxr-x 4 carol carol   4096 Dec 10 15:56 ..
drwxrwxr-x 2 carol carol   4096 Dec 10 17:59 Another_Directory
-rw------- 1 carol carol 539663 Dec 10 10:43 picture.jpg
-rw-rw-r-- 1 carol carol   1881 Dec 10 15:57 text.txt
-rw-r--r-- 1 carol carol      0 Dec 10 16:01 .thisIsHidden
```

The file `.thisIsHidden` is simply hidden because its name starts with `.`.

The directories `.` and `..` however are special. `.` is a pointer to the current directory, while `..` is a pointer to the parent directory (the directory which contains the current directory). In Linux, every directory contains at least these two special directories.

TIP You can combine multiple parameters for `ls` (and many other Linux commands). `ls -l -a` can, for example, be written as `ls -la`.

Understanding Filetypes

We have already mentioned that the first letter in each output of `ls -l` describes the type of the file. The three most common file types are:

`-` (normal file)

A file can contain data of any kind. Files can be modified, moved, copied and deleted.

`d` (directory)

A directory contains other files or directories and helps to organize the file system. Technically, directories are a special kind of file.

`l` (soft link)

This “file” is a pointer to another file or directory elsewhere in the filesystem.

In addition to those, there are three other file types that you should at least know about, but are

out of scope for this lesson:

`b` (block device)

This file stands for a virtual or physical device, usually disks or other kinds of storage devices. For example, the first hard disk in the system might be represented by `/dev/sda`.

`c` (character device)

This file stands for a virtual or physical device. Terminals (like the main terminal on `/dev/ttyS0`) and serial ports are common examples of character devices.

`s` (socket)

Sockets serve as "conduits" passing information between two programs.

WARNING Do not alter any of the permissions on block devices, character devices or sockets, unless you know what you're doing. This may prevent your system from working!

Understanding Permissions

In the output of `ls -l` the file permissions are shown right after the filetype, as three groups of three characters each, in the order `r`, `w` and `x`. Here is what they mean. Keep in mind that a dash `-` represents the lack of a particular permission.

Permissions on Files

`r`

Stands for *read* and has an octal value of `4` (don't worry, we will discuss octals shortly). This means permission to open a file and read its contents.

`w`

Stands for *write* and has an octal value of `2`. This means permission to edit or delete a file.

`x`

Stands for *execute* and has an octal value of `1`. This means that the file can be run as an executable or script.

So, for example, a file with permissions `rw-` can be read and written to, but cannot be executed.

Permissions on Directories

`r`

Stands for *read* and has an octal value of `4`. This means permission to read the directory's contents, like filenames. But it *does not* imply permission to read the files themselves.

`w`

Stands for *write* and has an octal value of `2`. This means permission to create or delete files in a directory, or change their names, permissions and owners. If a user has the write permission on a directory, the user can change permissions of any file in the directory, even if the user has no permissions on the file or if the file is owned by another user.

`x`

Stands for *execute* and has an octal value of `1`. This means permission to enter a directory, but not to list its files (for that, the `r` permission is needed).

The last bit about directories may sound a bit confusing. Let's imagine, for example, that you have a directory named `Another_Directory`, with the following permissions:

```
$ ls -ld Another_Directory/
d--xr-xr-x 2 carol carol 4,0K Dec 20 18:46 Another_Directory
```

Also imagine that inside this directory you have a shell script called `hello.sh` with the following permissions:

```
-rwxr-xr-x 1 carol carol 33 Dec 20 18:46 hello.sh
```

If you are the user `carol` and try to list the contents of `Another_Directory`, you will get an error message, as your user lacks read permission for that directory:

```
$ ls -l Another_Directory/
ls: cannot open directory 'Another_Directory/': Permission denied
```

However, the user `carol` *does have* execute permissions, which means that she can enter the directory. Therefore, the user `carol` can access files inside the directory, as long as she has the correct permissions *for the respective file*. In this example, the user has full permissions for the script `hello.sh`, so she *can* run the script, even if she *can't* read the contents of the directory containing it. All that is needed is the complete filename.

```
$ sh Another_Directory/hello.sh
```

```
Hello LPI World!
```

As we said before, permissions are specified in sequence: first for the owner of the file, then for the owning group, and then for other users. Whenever someone tries to perform an action on the file, the permissions are checked in the same fashion. First the system checks if the current user owns the file, and if this is true it applies the first set of permissions only. Otherwise, it checks if the current user belongs to the group owning the file. In that case, it applies the second set of permissions only. In any other case, the system will apply the third set of permissions. This means that if the current user is the owner of the file, only the owner permissions are effective, even if the group or other permissions are more permissive than the owner's permissions.

Modifying File Permissions

The command `chmod` is used to modify the permissions for a file, and takes at least two parameters: the first one describes which permissions to change, and the second one points to the file or directory where the change will be made. However, the permissions to change can be described in two different ways, or "modes".

The first one, called *symbolic mode* offers fine grained control, allowing you to add or revoke a single permission without modifying others on the set. The other mode, called *numeric mode*, is easier to remember and quicker to use if you wish to set all permission values at once.

Both modes will lead to the same end result. So, for example, the commands:

```
$ chmod ug+rw-x,o-rwx text.txt
```

and

```
$ chmod 660 text.txt
```

will produce exactly the same output, a file with the permissions set:

```
-rw-rw---- 1 carol carol  765 Dec 20 21:25 text.txt
```

Now, let's understand how each mode works.

Symbolic Mode

When describing which permissions to change in *symbolic mode* the first character(s) indicate(s) whose permissions you will alter: the ones for the user (`u`), for the group (`g`), for others (`o`) and/or for all the three together (`a`).

Then you need to tell the command what to do: you can grant a permission (`+`), revoke a permission (`-`), or set it to a specific value (`=`).

Lastly, you specify which permission you wish to affect: read (`r`), write (`w`), or execute (`x`).

For example, imagine we have a file called `text.txt` with the following permission set:

```
$ ls -l text.txt
-rw-r--r-- 1 carol carol 765 Dec 20 21:25 text.txt
```

If you wish to grant write permissions to members of the group owning the file, you would use the `g+w` parameter. It is easier if you think about it this way: "For the group (`g`), grant (`+`) write permissions (`w`)". So, the command would be:

```
$ chmod g+w text.txt
```

Let's check the result with `ls`:

```
$ ls -l text.txt
-rw-rw-r-- 1 carol carol 765 Dec 20 21:25 text.txt
```

If you want to remove read permissions for the owner of the same file, think about it as: "For the user (`u`) revoke (`-`), read permissions (`r`)". So the parameter is `u-r`, like so:

```
$ chmod u-r text.txt
$ ls -l text.txt
--w-rw-r-- 1 carol carol 765 Dec 20 21:25 text.txt
```

What if we want to set the permissions exactly as `rw-` for everyone? Then think of it as: "For all (`a`), set exactly (`=`), read (`r`), write (`w`), and no execute (`-`)". So:

```
$ chmod a=rw- text.txt
$ ls -l text.txt
```

```
-rw-rw-rw- 1 carol carol 765 Dec 20 21:25 text.txt
```

Of course, it is possible to modify multiple permissions at the same time. In this case, separate them with a comma (`,`):

```
$ chmod u+rwx,g-x text.txt
$ ls -lh text.txt
-rwxrw-rw- 1 carol carol 765 Dec 20 21:25 text.txt
```

The example above can be read as: "For the user (`u`), grant (`+`) read, write and execute (`rwx`) permissions, for the group (`g`) revoke (`-`), execute permissions (`x`)".

When run on a directory, `chmod` modifies only the directory's permissions. `chmod` has a recursive mode, useful when you want to change the permissions for "all files inside a directory and its subdirectories". To use this, add the parameter `-R` after the command name and before the permissions to change, like so:

```
$ chmod -R u+rwx Another_Directory/
```

This command can be read as: "Recursively (`-R`), for the user (`u`), grant (`+`) read, write and execute (`rwx`) permissions".

WARNING Be careful and think twice before using the `-R` switch, as it is easy to change permissions on files and directories which you do not want to change, especially on directories with a large number of files and sub-directories.

Numeric Mode

In *numeric mode*, the permissions are specified in a different way: as a three-digit numeric value on octal notation, a base-8 numeric system.

Each permission has a corresponding value, and they are specified in the following order: first comes read (`r`), which is `4`, then write (`w`), which is `2` and last is execute (`x`), represented by `1`. If there is no permission, use the value zero (`0`). So, a permission of `rwx` would be 7 (`4+2+1`) and `r-x` would be `5` (`4+0+1`).

The first of the three digits on the permission set represents the permissions for the user (`u`), the second for the group (`g`) and the third for the other (`o`). If we wanted to set the permissions for a file to `rw-rw----`, the octal value would be `660`:

```
$ chmod 660 text.txt
$ ls -l text.txt
-rw-rw---- 1 carol carol 765 Dec 20 21:25 text.txt
```

Besides this, the syntax in *numeric mode* is the same as in *symbolic mode*, the first parameter represents the permissions you wish to change, and the second one points to the file or directory where the change will be made.

TIP If a permission value is *odd*, the file surely is executable!

Which syntax to use? The *numeric mode* is recommended if you want to change the permissions to a specific value, for example `640` (`rw- r-- ---`).

The *symbolic mode* is more useful if you want to flip just a specific value, regardless of the current permissions for the file. For example, I can add execute permissions for the user using just `chmod u+x script.sh` without regard to, or even touching, the current permissions for the group and others.

Modifying File Ownership

The command `chown` is used to modify the ownership of a file or directory. The syntax is quite simple:

```
chown username:groupname filename
```

For example, let's check a file called `text.txt`:

```
$ ls -l text.txt
-rw-rw---- 1 carol carol 1881 Dec 10 15:57 text.txt
```

The user who owns the file is `carol`, and the group is also `carol`. Now, let's modify the group owning the file to some other group, like `students`:

```
$ chown carol:students text.txt
$ ls -l text.txt
-rw-rw---- 1 carol students 1881 Dec 10 15:57 text.txt
```

Keep in mind that the user who owns a file does not need to belong to the group who owns a file. In the example above, the user `carol` does not need to be a member of the `students` group.

However, she does have to member of the group in order to transfer the file's group ownership to that group.

User or group can be omitted if you do not wish to change them. So, to change just the group owning a file you would use `chown :students text.txt`. To change just the user, the command would be `chown carol text.txt`. Alternatively, you could use the command `chgrp students text.txt` to change only the group.

Unless you are the system administrator (root), you cannot change ownership of a file owned by another user or a group that you don't belong to. If you try to do this, you will get the error message `Operation not permitted`.

Querying Groups

Before changing the ownership of a file, it might be useful to know which groups exist on the system, which users are members of a group and to which groups a user belongs. Those tasks can be accomplished with two commands, `groups` and `groupmems`.

To see which groups exist on your system, simply type `groups`:

```
$ groups
carol students cdrom sudo dip plugdev lpadmin sambashare
```

And if you want to know to which groups a user belongs, add the username as a parameter:

```
$ groups carol
carol : carol students cdrom sudo dip plugdev lpadmin sambashare
```

To do the reverse, displaying which users belong to a group, use `groupmems`. The parameter `-g` specifies the group, and `-l` will list all of its members:

```
$ sudo groupmems -g cdrom -l
carol
```

TIP `groupmems` can only be run as root, the system administrator. If you are not currently logged in as root, add `sudo` before the command.

Special Permissions

Besides the read, write and execute permissions for user, group and others, each file can have three other *special permissions* which can alter the way a directory works or how a program runs. They can be specified either in symbolic or numeric mode, and are as follows:

Sticky Bit

The sticky bit, also called the *restricted deletion flag*, has the octal value `1` and in symbolic mode is represented by a `t` within the other's permissions. This applies only to directories, and on Linux it prevents users from removing or renaming a file in a directory unless they own that file or directory.

Directories with the sticky bit set show a `t` replacing the `x` on the permissions for *others* on the output of `ls -l`:

```
$ ls -ld Sample_Directory/
drwxr-xr-t 2 carol carol 4096 Dec 20 18:46 Sample_Directory/
```

In numeric mode, the special permissions are specified using a "4-digit notation", with the first digit representing the special permission to act upon. For example, to set the sticky bit (value `1`) for the directory `Another_Directory` in numeric mode, with permissions `755`, the command would be:

```
$ chmod 1755 Another_Directory
$ ls -ld Another_Directory
drwxr-xr-t 2 carol carol 4,0K Dec 20 18:46 Another_Directory
```

Set GID

Set GID, also known as SGID or Set Group ID bit, has the octal value `2` and in symbolic mode is represented by an `s` on the *group* permissions. This can be applied to executable files or directories. On executable files, it will grant the process resulting from executing the file access to the privileges of the group who owns the file. When applied to directories, it will make every file or directory created under it inherit the group from the parent directory.

Files and directories with SGID bit show an `s` replacing the `x` on the permissions for the *group* on the output of `ls -l`:

```
$ ls -l test.sh
```

```
-rwxr-sr-x 1 carol carol 33 Dec 11 10:36 test.sh
```

To add SGID permissions to a file in symbolic mode, the command would be:

```
$ chmod g+s test.sh
$ ls -l test.sh
-rwxr-sr-x 1 carol root     33 Dec 11 10:36 test.sh
```

The following example will make you better understand the effects of SGID on a directory. Suppose we have a directory called `Sample_Directory`, owned by the user `carol` and the group `users`, with the following permission structure:

```
$ ls -ldh Sample_Directory/
drwxr-xr-x 2 carol users 4,0K Jan 18 17:06 Sample_Directory/
```

Now, let's change to this directory and, using the command `touch`, create an empty file inside it. The result would be:

```
$ cd Sample_Directory/
$ touch newfile
$ ls -lh newfile
-rw-r--r-- 1 carol carol 0 Jan 18 17:11 newfile
```

As we can see, the file is owned by the user `carol` and group `carol`. But, if the directory had the SGID permission set, the result would be different. First, let's add the SGID bit to the `Sample_Directory` and check the results:

```
$ sudo chmod g+s Sample_Directory/
$ ls -ldh Sample_Directory/
drwxr-sr-x 2 carol users 4,0K Jan 18 17:17 Sample_Directory/
```

The `s` on the group permissions indicates that the SGID bit is set. Now, let's change to this directory and, again, create an empty file with the `touch` command:

```
$ cd Sample_Directory/
$ touch emptyfile
$ ls -lh emptyfile
 -rw-r--r-- 1 carol users 0 Jan 18 17:20 emptyfile
```

As we can see, the group who owns the file is `users`. This is because the SGID bit made the file inherit the group owner of its parent directory, which is `users`.

Set UID

SUID, also known as Set User ID, has the octal value `4` and is represented by an `s` on the *user* permissions in symbolic mode. It only applies to files and its behavior is similar to the SGID bit, but the process will run with the privileges of the *user* who owns the file. Files with the SUID bit show an `s` replacing the `x` on the permissions for the user on the output of `ls -l`:

```
$ ls -ld test.sh
-rwsr-xr-x 1 carol carol 33 Dec 11 10:36 test.sh
```

You can combine multiple special permissions in one parameter by adding them together. So, to set SGID (value `2`) and SUID (value `4`) in numeric mode for the script `test.sh` with permissions `755`, you would type:

```
$ chmod 6755 test.sh
```

And the result would be:

```
$ ls -lh test.sh
-rwsr-sr-x 1 carol carol 66 Jan 18 17:29 test.sh
```

TIP

If your terminal supports color, and these days most of them do, you can quickly see if these special permissions are set by glancing at the output of `ls -l`. For the sticky bit, the directory name might be shown in a black font with blue background. The same applies for files with the SGID (yellow background) and SUID (red background) bits. Colors may be different depending on which Linux distribution and terminal settings you use.

Guided Exercises

1. Create a directory named `emptydir` using the command `mkdir emptydir`. Now, using `ls`, list the permissions for the directory `emptydir`.

2. Create an empty file named `emptyfile` with the command `touch emptyfile`. Now, using `chmod` with symbolic notation, add execute permissions for the owner of the file `emptyfile`, and remove write and execute permissions for everyone else. Do this using only one `chmod` command.

3. What will be the permissions for a file called `text.txt` after you use the command `chmod 754 text.txt`?

4. Let's assume a file named `test.sh` is a shell script with the following permissions and ownership:

```
-rwxr-sr-x 1 carol root     33 Dec 11 10:36 test.sh
```

   - What are the permissions for the owner of the file?

   - If the user `john` runs this script, under which user's privileges will it be run?

   - Using the numeric notation, which should be the syntax of chmod to "unset" the special permission granted to this file?

5. Consider this file:

```
$ ls -l /dev/sdb1
brw-rw---- 1 root disk 8, 17 Dec 21 18:51 /dev/sdb1
```

   Which kind of file is `sdb1`? And who can write to it?

6. Consider the following 4 files:

```
drwxr-xr-t 2 carol carol 4,0K Dec 20 18:46 Another_Directory
----r--r-- 1 carol carol    0 Dec 11 10:55 foo.bar
-rw-rw-r-- 1 carol carol 1,2G Dec 20 18:22 HugeFile.zip
drwxr-sr-x 2 carol users 4,0K Jan 18 17:26 Sample_Directory
```

Write down the corresponding permissions for each file and directory using numeric 4-digit notation.

`Another_Directory`

`foo.bar`

`HugeFile.zip`

`Sample_Directory`

Explorational Exercises

1. Try this on a terminal: create an empty file called `emptyfile` with the command `touch emptyfile`. Now "zero out" the permissions for the file with `chmod 000 emptyfile`. What will happen if you change the permissions for `emptyfile` by passing only *one* value for `chmod` in numeric mode, such as `chmod 4 emptyfile`? What if we use two, such as `chmod 44 emptyfile`? What can we learn about the way `chmod` reads the numerical value?

2. Can you execute a file for which you have execute, but not read permissions (`--x`)? Why or why not?

3. Consider the permissions for the temporary directory on a Linux system, `/tmp`:

   ```
   $ ls -ld /tmp
   drwxrwxrwt  19 root root  16K Dec 21 18:58 tmp
   ```

   User, group and others have full permissions. But can a regular user delete *any* files inside this directory? Why is this?

4. A file called `test.sh` has the following permissions: `-rwsr-xr-x`, meaning the SUID bit is set. Now, run the following commands:

   ```
   $ chmod u-x test.sh
   $ ls -l test.sh
   -rwSr-xr-x 1 carol carol 33 Dec 11 10:36 test.sh
   ```

   What did we do? What does the uppercase `S` mean?

5. How would you create a directory named `Box` where all the files are automatically owned by the group `users`, and can only be deleted by the user who created them?

Summary

As a multi-user system, Linux needs a way to track who owns and who can access each file. This is done through a three-level permissions system, and in this lesson we learned all about how this system works.

In this lesson you have learned how use `ls` to get information about file permissions, how to control or change who can create, delete or modify a file with `chmod`, both in *numeric* and *symbolic* notation and how to change the ownership of files with `chown` and `chgrp`.

The following commands were discussed in this lesson:

`ls`

List files, optionally including details such as permissions.

`chmod`

Change the permissions of a file or directory.

`chown`

Change the owning user and/or group of a file or directory.

`chgrp`

Change the owning group of a file or directory.

Answers to Guided Exercises

1. Create a directory named `emptydir` using the command `mkdir emptydir`. Now, using `ls`, list the permissions for the directory `emptydir`.

   Add the `-d` parameter to `ls` to see the file attributes of a directory, instead of listing its contents. Therefore the answer is:

   ```
   ls -l -d emptydir
   ```

   Bonus points if you merged the two parameters in one, as in `ls -ld emptydir`.

2. Create an empty file named `emptyfile` with the command `touch emptyfile`. Now, using `chmod` in symbolic notation, add execute permissions for the owner of the file `emptyfile`, and remove write and execute permissions for everyone else. Do this using only one `chmod` command.

   Think about it this way:

   - "For the user who owns the file (`u`) add (`+`) execute (`x`) permissions", so `u+x`.
   - "For the group (`g`) and other users (`o`), remove (`-`) write (`w`) and execute (`x`) permissions", so `go-wx`.

     To combine these two sets of permissions, we add a comma between them. So the final result is:

     ```
     chmod u+x,go-wx emptyfile
     ```

3. What will be the permissions of a file called `text.txt` after I use the command `chmod 754 text.txt`?

   Remember that in numeric notation each digit represents a set of three permissions, each one with a respective value: *read* is `4`, *write* is `2`, *execute* is `1` and no permission is `0`. We get the value for a digit by adding the corresponding values for each permission. `7` is `4+2+1`, or `rwx`, `5` is `4+0+1`, so `r-x` and `4` is just read, or `r--`. The permissions for `text.txt` would be

   ```
   rwxr-xr--
   ```

4. Let's assume a file named `test.sh` is a shell script with the following permissions and

ownership:

```
-rwxr-sr-x 1 carol root     33 Dec 11 10:36 test.sh
```

- What are the permissions for the owner of the file?

  The permissions for the owner (2nd to 4th characters in the output of `ls -l`) are `rwx`, so the answer is: “to read, to write to and to execute the file”.

- If the user `john` runs this script, under which user’s privileges will it be run?

  Pay attention to the permissions for the *group*. They are `r-s`, which means the SGID bit is set. The group who owns this file is `root`, so the script, even when started by a regular user, will be run with root privileges.

- Using the numeric notation, which should be the syntax of chmod to “unset” the special permission granted to this file?

  We can “unset” the special permissions by passing a 4th digit, `0`, to `chmod`. The current permissions are `755`, so the command should be `chmod 0755`.

5. Consider this file:

```
$ ls -l /dev/sdb1
brw-rw---- 1 root disk 8, 17 Dec 21 18:51 /dev/sdb1
```

   Which kind of file is `sdb1`? And who can write to it?

   The first character of the output from `ls -l` shows the kind of file. `b` is a *block device*, usually a disk (internal or external), connected to the machine. The owner (`root`) and any users of the group `disk` can write to it.

6. Consider the following 4 files:

```
drwxr-xr-t 2 carol carol 4,0K Dec 20 18:46 Another_Directory
----r--r-- 1 carol carol    0 Dec 11 10:55 foo.bar
-rw-rw-r-- 1 carol carol 1,2G Dec 20 18:22 HugeFile.zip
drwxr-sr-x 2 carol users 4,0K Jan 18 17:26 Sample_Directory
```

   Write down the corresponding permissions for each file and directory using 4-digit numeric notation.

The corresponding permissions, in numeric notation, are as follows:

`Another_Directory`

Answer: `1755`

`1` for the sticky bit, `755` for the regular permissions (`rwx` for the user, `r-x` for group and others).

`foo.bar`

Answer: `0044`

No special permissions (so the first digit is `0`), no permissions for the user (`---`) and just read (`r—r--`) for group and others.

`HugeFile.zip`

Answer: `0664`

No special permissions, so the first digit is `0`. `6` (`rw-`) for the user and group, `4` (`r--`) for the others.

`Sample_Directory`

Answer: `2755`

`2` for the SGID bit, `7` (`rwx`) for the user, `5` (`r-x`) for the group and others.

Answers to Explorational Exercises

1. Try this on a terminal: create an empty file called `emptyfile` with the command `touch emptyfile`. Now "zero out" the permissions for the file with `chmod 000 emptyfile`. What will happen if you change the permissions for `emptyfile` by passing only *one* value for `chmod` in numeric notation, such as `chmod 4 emptyfile`? What if we use two, such as `chmod 44 emptyfile`? What can we learn about the way `chmod` reads the numerical value?

   Remember that we "zeroed out" the permissions for `emptyfile`. So, its initial state would be:

   ```
   ---------- 1 carol carol    0 Dec 11 10:55 emptyfile
   ```

   Now, let's try the first command, `chmod 4 emptyfile`:

   ```
   $ chmod 4 emptyfile
   $ ls -l emptyfile
   -------r-- 1 carol carol 0 Dec 11 10:55 emptyfile
   ```

   The permissions for *others* were changed. And what if we try two digits, such as `chmod 44 emptyfile`?

   ```
   $ chmod 44 emptyfile
   $ ls -l emptyfile
   ----r--r-- 1 carol carol 0 Dec 11 10:55 emptyfile
   ```

   Now, the permissions for *group* and *others* were affected. From this, we can conclude that in numeric notation `chmod` reads the value "backwards", from the least significant digit (*others*) to the most significant one (*user*). If you pass one digit, you modify the permissions for *others*. With two digits you modify *group* and *others*, and with three you modify *user*, *group* and *others* and with four digits you modify *user*, *group*, *others* and the special permissions.

2. Can you execute a file for which you have execute, but not read permissions (`--x`)? Why or why not?

   At first, the answer seems obvious: If you have execute permission, the file should run. This applies to programs in binary format that are executed directly by the kernel. However, there are programs (e.g. shell scripts) that must first be read and interpreted, so in these cases the read permission (`r`) must also be set.

3. Consider the permissions for the temporary directory on a Linux system, `/tmp`:

```
$ ls -ld /tmp
drwxrwxrwt  19 root root  16K Dec 21 18:58 tmp
```

   User, group and others have full permissions. But can a regular user delete *any* files inside this directory? Why is this?

   `/tmp` is what we call a *world writeable* directory, meaning that any user can write to it. But we don't want one user modifying files created by others, so the sticky bit_ is set (as indicated by the `t` on the permissions for *others*). This means that a user can delete files in `/tmp`, but only if they created that file.

4. A file called `test.sh` has the following permissions: `-rwsr-xr-x`, meaning the SUID bit is set. Now, run the following commands:

```
$ chmod u-x test.sh
$ ls -l test.sh
-rwSr-xr-x 1 carol carol 33 Dec 11 10:36 test.sh
```

   What did we do? What does the uppercase `S` mean?

   We removed execute permissions for the user who owns the file. The `s` (or `t`) takes the place of the `x` on the output of `ls -l`, so the system needs a way to show if the user has execute permissions or not. It does this by changing the case of the special character.

   A lowercase `s` on the first group of permissions means that the user who owns the file has execute permissions and that the SUID bit is set. An uppercase `S` means that the user who owns the file lacks (`-`) execute permissions and that the SUID bit is set.

   The same can be said for SGID. A lowercase `s` on the second group of permissions means that the group who owns the file has execute permissions and that the SGID bit is set. An uppercase `S` means that the group who owns the file lacks (`-`) execute permissions and that the SGID bit is set.

   This is also true for the sticky bit, represented by the `t` in the third group of permissions. Lowercase `t` means the sticky bit is set and that others have execute permissions. Uppercase `T` means the sticky bit is set and that others do not have execute permissions.

5. How would you create a directory named `Box` where all the files are automatically owned by the group `users`, and can only be deleted by the user who created them?

This is a multi-step process. The first step is to create the directory:

```
$ mkdir Box
```

We want every file created inside this directory to be automatically assigned to the group `users`. We can do this by setting this group as the owner of the directory, and then by setting the SGID bit on it. We also need to make sure that any member of the group can write to that directory.

Since we do not care about what the other permissions are, and want to "flip" only the special bits, it makes sense to use the symbolic mode:

```
$ chown :users Box/
$ chmod g+wxs Box/
```

Note that if you current user does not belong to the group `users`, you will have to use the command `sudo` before the commands above to do the change as root.

Now for the last part, making sure that only the user who created a file is allowed to delete it. This is done by setting the sticky bit (represented by a `t`) on the directory. Remember that it is set on the permissions for others (`o`).

```
$ chmod o+t Box/
```

The permissions on the directory `Box` should appear as follows:

```
drwxrwsr-t 2 carol users  4,0K Jan 18 19:09 Box
```

Of course, you can specify SGID and the sticky bit using only one `chmod` command:

```
$ chmod g+wxs,o+t Box/
```

Bonus points if you thought of that.

5.4 Special Directories and Files

Reference to LPI objectives

Linux Essentials v1.6, Exam 010, Objective 5.4

Weight

1

Key knowledge areas

- Using temporary files and directories
- Symbolic links

Partial list of the used files, terms and utilities

- `/tmp/`, `/var/tmp/` and Sticky Bit
- `ls -d`
- `ln -s`

Linux Professional Institute

5.4 Lesson 1

Certificate:	Linux Essentials
Version:	1.6
Topic:	5 Security and File Permissions
Objective:	5.4 Special Directories and Files
Lesson:	1 of 1

Introduction

On Linux, everything is treated as a file. However, some files get a special treatment, either because of the place they are stored, such as temporary files, or the way they interact with the filesystem, such as links. In this lesson, we will learn where such files are located, how they work and how to manage them.

Temporary Files

Temporary files are files used by programs to store data that is only needed for a short time. These can be the data of running processes, crash logs, scratch files from an autosave, intermediary files used during a file conversion, cache files and so on.

Location of temporary files

Version 3.0 of the *Filesystem Hierarchy Standard* (FHS) defines standard locations for temporary files on Linux systems. Each location has a different purpose and behavior, and it is recommended that developers follow the conventions set by the FHS when writing temporary data to disk.

`/tmp`

According to the FHS, programs should not assume that files written here will be preserved between invocations of a program. The *recommendation* is that this directory be cleared (all files erased) during system boot-up, although this is *not mandatory*.

`/var/tmp`

Another location for temporary files, but this one *should not be cleared* during the system boot-up, i.e. files stored here will usually persist between reboots.

`/run`

This directory contains run-time variable data used by running processes, such as process identifier files (`.pid`). Programs that need more than one run-time file may create subdirectories here. This location *must be cleared* during system boot-up. The purpose of this directory was once served by `/var/run`, and on some systems `/var/run` may be a symbolic link to `/run`.

Note that there is nothing which prevents a program to create temporary files elsewhere on the system, but it is good practice to respect the conventions set by the FHS.

Permissions on temporary files

Having system-wide temporary directories on a multiuser system presents some challenges regarding access permissions. At first thought one may think that such directories would be "world-writable", i.e. any user could write or delete data in it. But if this were to be true, how could we prevent a user from erasing or modifying files created by another?

The solution is a special permission called the *sticky bit*, which applies both to directories and files. However, for security reasons, the Linux kernel ignores the sticky bit when it is applied to files. When this special bit is set for a directory, it prevents users from removing or renaming a file within that directory unless they own the file.

Directories with the sticky bit set show a `t` replacing the `x` on the permission for *others* in the output of `ls -l`. For example, let's check the permissions for the `/tmp` and `/var/tmp` directories:

```
$ ls -ldh /tmp/ /var/tmp/
drwxrwxrwt 25 root root 4,0K Jun  7 18:52 /tmp/
drwxrwxrwt 16 root root 4,0K Jun  7 09:15 /var/tmp/
```

As you can see by the `t` replacing the `x` on the permission for *others*, both directories have the sticky bit set.

To set the sticky bit on a directory using `chmod` in numeric mode, use the four-digit notation and `1` as the first digit. For example:

```
$ chmod 1755 temp
```

will set the sticky bit for the directory named `temp` and the permissions as `rwxr-xr-t`.

When using the symbolic mode, use the parameter `t`. So, `+t` to set the sticky bit, and `-t` to disable it. Like so:

```
$ chmod +t temp
```

Understanding Links

We have already said that on Linux everything is treated as a file. But there is a *special* kind of file, called a *link*, and there are two types of links on a Linux system:

Symbolic links

Also called *soft links*, they point to the path of another file. If you delete the file the link points to (called *target*) the link will still exist, but it "stops working", as it now points to "nothing".

Hard links

Think of a hard link as a second name for the original file. They are *not* duplicates, but instead are an additional entry in the file system pointing to the same place (inode) on the disk.

TIP

An *inode* is a data structure that stores attributes for an object (like a file or directory) on a filesystem. Among those attributes are the filename, permissions, ownership and on which blocks of the disk the data for the object is stored. Think of it as an entry on an index, hence the name, which comes from "index node".

Working with Hard Links

Creating Hard Links

The command to create a hard link on Linux is `ln`. The basic syntax is:

```
$ ln TARGET LINK_NAME
```

The `TARGET` must exist already (this is the file the link will point to), and if the target is not on the

current directory, or if you want to create the link elsewhere, you *must* specify the full path to it. For example, the command

```
$ ln target.txt /home/carol/Documents/hardlink
```

will create a file named `hardlink` on the directory `/home/carol/Documents/`, linked to the file `target.txt` on the current directory.

If you leave out the last parameter (`LINK_NAME`), a link with the same name as the target will be created in the current directory.

Managing Hard Links

Hard links are entries in the filesystem which have different names but point to the same data on disk. All such names are equal and can be used to refer to a file. If you change the contents of one of the names, the contents of all other names pointing to that file change since all these names point to the very same data. If you delete one of the names, the other names will still work.

This happens because when you "delete" a file the data is not actually erased from the disk. The system simply deletes the entry on the filesystem table pointing to the inode corresponding to the data on the disk. But if you have a second entry pointing to the same inode, you can still get to the data. Think of it as two roads converging on the same point. Even if you block or redirect one of the roads, you can still reach the destination using the other.

You can check this by using the `-i` parameter of `ls`. Consider the following contents of a directory:

```
$ ls -li
total 224
3806696 -r--r--r-- 2 carol carol 111702 Jun  7 10:13 hardlink
3806696 -r--r--r-- 2 carol carol 111702 Jun  7 10:13 target.txt
```

The number before the permissions is the inode number. See that both the file `hardlink` and the file `target.txt` have the same number (`3806696`)? This is because one is a hard link of the other.

But which one is the original and which one is the link? You can't really tell, as for all practical purposes they are the same.

Note that every hard link pointing to a file increases the *link count* of the file. This is the number right after the permissions on the output of `ls -l`. By default, every file has a link count of `1` (directories have a count of `2`), and every hard link to it increases the count by one. So, that is the reason for the link count of `2` on the files in the listing above.

In contrast to symbolic links, you can only create hard links to files, and both the link and target must reside in the same file system.

Moving and Removing Hard Links

Since hard links are treated as regular files, they can be deleted with `rm` and renamed or moved around the filesystem with `mv`. And since a hard link points to the same inode of the target, it can be moved around freely, without fear of "breaking" the link.

Symbolic links

Creating Symbolic Links

The command used to create a symbolic link is also `ln`, but with the `-s` parameter added. Like so:

```
$ ln -s target.txt /home/carol/Documents/softlink
```

This will create a file named `softlink` in the directory `/home/carol/Documents/`, pointing to the file `target.txt` in the current directory.

As with hard links, you can omit the link name to create a link with the same name as the target in the current directory.

Managing Symbolic Links

Symbolic links point to another path in the filesystem. You can create soft links to files *and* directories, even on different partitions. It is pretty easy to spot a symbolic link on the output of `ls`:

```
$ ls -lh
total 112K
-rw-r--r-- 1 carol carol 110K Jun  7 10:13 target.txt
lrwxrwxrwx 1 carol carol   12 Jun  7 10:14 softlink -> target.txt
```

In the example above, the first character on the permissions for the file `softlink` is `l`, indicating a symbolic link. Furthermore, just after the filename we see the name of the target the link points to, the file `target.txt`.

Note that on file and directory listings, soft links themselves always show the permissions `rwx` for the user, the group and others, but in practice the access permissions for them are the same as those for the target.

Moving and Removing Symbolic Links

Like hard links, symbolic links can be removed using `rm` and moved around or renamed using `mv`. However, special care should be taken when creating them, to avoid "breaking" the link if it is moved from its original location.

When creating symbolic links you should be aware that unless a path is fully specified the location of the target is interpreted as *relative* to the location of the link. This may create problems if the link, or the file it points to, is moved.

This is easier to understand with an example. Say that we have a file named `original.txt` in the current directory, and we want to create a symbolic link to it called `softlink`. We could use:

```
$ ln -s original.txt softlink
```

And apparently all would be well. Let's check with `ls`:

```
$ ls -lh
total 112K
-r--r--r-- 1 carol carol 110K Jun  7 10:13 original.txt
lrwxrwxrwx 1 carol carol   12 Jun  7 19:23 softlink -> original.txt
```

See how the link is constructed: `softlink` points to (→) `original.txt`. However, let's see what happens if we move the link to the parent directory and try to display its contents using the command `less`:

```
$ mv softlink ../
$ less ../softlink
../softlink: No such file or directory
```

Since the path to `original.txt` was not specified, the system assumes that it is in the same directory as the link. When this is no longer true, the link stops working.

The way to prevent this is to always specify the full path to the target when creating the link:

```
$ ln -s /home/carol/Documents/original.txt softlink
```

This way, no matter where you move the link it will still work, because it points to the absolute location of the target. Check with `ls`:

```
$ ls -lh
total 112K
lrwxrwxrwx 1 carol carol   40 Jun  7 19:34 softlink -> /home/carol/Documents/original.txt
```

Guided Exercises

1. Imagine a program needs to create a one-use temporary file that will never be needed again after the program is closed. What would be the correct directory in which to to create this file?

2. Which is the temporary directory that *must* be cleared during the boot process?

3. What is the parameter for `chmod` in *symbolic* mode to enable the sticky bit on a directory?

4. Imagine there is a file named `document.txt` on the directory `/home/carol/Documents`. What is the command to create a symbolic link to it named `text.txt` on the current directory?

5. Explain the difference between a hard link to a file and a copy of this file.

Explorational Exercises

1. Imagine that inside a directory you create a file called `recipes.txt`. Inside this directory, you will also create a hard link to this file, called `receitas.txt`, and a symbolic (or *soft*) link to this called `rezepte.txt`.

   ```
   $ touch recipes.txt
   $ ln recipes.txt receitas.txt
   $ ln -s receitas.txt rezepte.txt
   ```

   The contents of the directory should appear like so:

   ```
   $ ls -lhi
   total 160K
   5388833 -rw-r--r-- 4 carol carol 77K jun 17 17:25 receitas.txt
   5388833 -rw-r--r-- 4 carol carol 77K jun 17 17:25 recipes.txt
   5388837 lrwxrwxrwx 1 carol carol  12 jun 24 10:12 rezepte.txt -> receitas.txt
   ```

   Remember that, as a hard link, `receitas.txt` points to the same inode that `recipes.txt`. What would happen to the soft link `rezepte.txt` if the name `receitas.txt` is deleted? Why?

2. Imagine you have a flash drive plugged into your system, and mounted on `/media/youruser/FlashA`. You want to create in your home directory a link called `schematics.pdf`, pointing to the file `esquema.pdf` in the root directory of the flash drive. So, you type the command:

   ```
   $ ln /media/youruser/FlashA/esquema.pdf ~/schematics.pdf
   ```

   What would happen? Why?

3. Consider the following output of `ls -lah`:

   ```
   $ ls -lah
   total 3,1M
   drwxr-xr-x 2 carol carol 4,0K jun 17 17:27 .
   drwxr-xr-x 5 carol carol 4,0K jun 17 17:29 ..
   -rw-rw-r-- 1 carol carol 2,8M jun 17 15:45 compressed.zip
   -rw-r--r-- 4 carol carol  77K jun 17 17:25 document.txt
   -rw-rw-r-- 1 carol carol 216K jun 17 17:25 image.png
   ```

```
-rw-r--r-- 4 carol carol  77K jun 17 17:25 text.txt
```

- How many links point to the file `document.txt`?

- Are they soft or hard links?

- Which parameter should you pass to `ls` to see which inode each file occupies?

4. Imagine you have in your `~/Documents` directory a file named `clients.txt` containing some client names, and a directory named `somedir`. Inside this there is a *different* file *also* named `clients.txt` with different names. To replicate this structure, use the following commands.

```
$ cd ~/Documents
$ echo "John, Michael, Bob" > clients.txt
$ mkdir somedir
$ echo "Bill, Luke, Karl" > somedir/clients.txt
```

You then create a link inside `somedir` named `partners.txt` pointing to this file, with the commands:

```
$ cd somedir/
$ ln -s clients.txt partners.txt
```

So, the directory structure is:

```
Documents
|-- clients.txt
`-- somedir
    |-- clients.txt
    `-- partners.txt -> clients.txt
```

Now, you move `partners.txt` from `somedir` to `~/Documents`, and list its contents.

```
$ cd ~/Documents/
$ mv somedir/partners.txt .
$ less partners.txt
```

Will the link still work? If so, which file will have its contents listed? Why?

5. Consider the following files:

```
-rw-r--r-- 1 carol carol 19 Jun 24 11:12 clients.txt
lrwxrwxrwx 1 carol carol 11 Jun 24 11:13 partners.txt -> clients.txt
```

What are the access permissions for `partners.txt`? Why?

Summary

In this lesson, you learned:

- Where temporary files are stored.
- What is the special permission applied to them.
- What links are.
- The difference between *symbolic* and *hard* links.
- How to create links.
- How to move, rename or remove them.

The following commands were discussed in this lesson:

- `ln`
- The `-i` parameter to `ls`

Answers to Guided Exercises

1. Imagine a program needs to create a one-use temporary file that will never be needed again after the program is closed. What would be the correct directory in which to create this file?

   Since we don't care about the file after the program finishes running, the correct directory is `/tmp`.

2. Which is the temporary directory that *must* be cleared during the boot process?

   The directory is `/run` or, on some systems, `/var/run`.

3. What is the parameter for `chmod` in *symbolic* mode to enable the sticky bit on a directory?

   The symbol for the sticky bit in symbolic mode is `t`. Since we want to enable (add) this permission to the directory, the parameter should be `+t`.

4. Imagine there is a file named `document.txt` on the directory `/home/carol/Documents`. What is the command to create a symbolic link to it named `text.txt` in the current directory?

   `ln -s` is the command to create a symbolic link. Since you should specify the full path to the file you are linking to, the command is:

   ```
   $ ln -s /home/carol/Documents/document.txt text.txt
   ```

5. Explain the difference between a hard link to a file and a copy of this file.

   A hard link is just another name for a file. Even though it looks like a duplicate of the original file, for all purposes both the link and the original are the same, as they point to the same data on disk. Changes made to the contents of the link will be reflected on the original, and vice-versa. A copy is a completely independent entity, occupying a different place on disk. Changes to the copy will not be reflected on the original, and vice-versa.

Answers to Explorational Exercises

1. Imagine that inside a directory you create a file called `recipes.txt`. Inside this directory, you will also create a hard link to this file, called `receitas.txt`, and a symbolic (or *soft*) link to this called `rezepte.txt`.

   ```
   $ touch recipes.txt
   $ ln recipes.txt receitas.txt
   $ ln -s receitas.txt rezepte.txt
   ```

   The contents of the directory should be like so:

   ```
   $ ls -lhi
   total 160K
   5388833 -rw-r--r-- 4 carol carol 77K jun 17 17:25 receitas.txt
   5388833 -rw-r--r-- 4 carol carol 77K jun 17 17:25 recipes.txt
   5388837 lrwxrwxrwx 1 carol carol  12 jun 24 10:12 rezepte.txt -> receitas.txt
   ```

   Remember that, as a hard link, `receitas.txt` points to the same inode that `recipes.txt`. What would happen to the soft link `rezepte.txt` if the name `receitas.txt` is deleted? Why?

   The soft link `rezepte.txt` would stop working. This is because soft links point to names, not inodes, and the name `receitas.txt` no longer exists, even if the data is still on the disk under the name `recipes.txt`.

2. Imagine you have a flash drive plugged into your system, and mounted on `/media/youruser/FlashA`. You want to create in your home directory a link called `schematics.pdf`, pointing to the file `esquema.pdf` in the root directory of the flash drive. So, you type the command:

   ```
   $ ln /media/youruser/FlashA/esquema.pdf ~/schematics.pdf
   ```

   What would happen? Why?

   The command would fail. The error message would be `Invalid cross-device link`, and it makes the reason clear: hard links cannot point to a target in a different partition or device. The only way to create a link like this is to use a *symbolic* or *soft* link, adding the `-s` parameter to `ln`.

3. Consider the following output of `ls -lah`:

```
$ ls -lah
total 3,1M
drwxr-xr-x 2 carol carol 4,0K jun 17 17:27 .
drwxr-xr-x 5 carol carol 4,0K jun 17 17:29 ..
-rw-rw-r-- 1 carol carol 2,8M jun 17 15:45 compressed.zip
-rw-r--r-- 4 carol carol  77K jun 17 17:25 document.txt
-rw-rw-r-- 1 carol carol 216K jun 17 17:25 image.png
-rw-r--r-- 4 carol carol  77K jun 17 17:25 text.txt
```

- How many links point to the file `document.txt`?

  Every file starts with a link count of `1`. Since the link count for the file is `4`, there are three links pointing to that file.

- Are they soft or hard links?

  They are hard links, since soft links do not increase the link count of a file.

- Which parameter should you pass to `ls` to see which inode each file occupies?

  The parameter is `-i`. The inode will be shown as the first column in the output of `ls`, like below:

  ```
  $ ls -lahi
  total 3,1M
  5388773 drwxr-xr-x 2 rigues rigues 4,0K jun 17 17:27 .
  5245554 drwxr-xr-x 5 rigues rigues 4,0K jun 17 17:29 ..
  5388840 -rw-rw-r-- 1 rigues rigues 2,8M jun 17 15:45 compressed.zip
  5388833 -rw-r--r-- 4 rigues rigues  77K jun 17 17:25 document.txt
  5388837 -rw-rw-r-- 1 rigues rigues 216K jun 17 17:25 image.png
  5388833 -rw-r--r-- 4 rigues rigues  77K jun 17 17:25 text.txt
  ```

4. Imagine you have in your `~/Documents` directory a file named `clients.txt` containing some client names, and a directory named `somedir`. Inside this there is a *different* file *also* named `clients.txt` with different names. To replicate this structure, use the following commands.

```
$ cd ~/Documents
$ echo "John, Michael, Bob" > clients.txt
$ mkdir somedir
$ echo "Bill, Luke, Karl" > somedir/clients.txt
```

You then create a link inside `somedir` named `partners.txt` pointing to this file, with the commands:

```
$ cd somedir/
$ ln -s clients.txt partners.txt
```

So, the directory structure is:

```
Documents
|-- clients.txt
`-- somedir
    |-- clients.txt
    `-- partners.txt -> clients.txt
```

Now, you move `partners.txt` from `somedir` to `~/Documents`, and list its contents.

```
$ cd ~/Documents/
$ mv somedir/partners.txt .
$ less partners.txt
```

Will the link still work? If so, which file will have its contents listed? Why?

This is a "tricky" one, but the link will work, and the file listed will be the one in `~/Documents`, containing the names `John`, `Michael`, `Bob`.

Remember that since you did not specify the full path to the target `clients.txt` when creating the soft link `partners.txt`, the target location will be interpreted as being relative to the location of the link, which in this case is the current directory.

When the link was moved from `~/Documents/somedir` to `~/Documents`, it should stop working, since the target was no longer in the same directory as the link. However, it just so happens that there is a file named `clients.txt` on `~/Documents`, so the link will point to this file, instead of the original target inside `~/somedir`.

To avoid this, always specify the full path to the target when creating a symbolic link.

5. Consider the following files:

```
-rw-r--r-- 1 rigues rigues 19 Jun 24 11:12 clients.txt
lrwxrwxrwx 1 rigues rigues 11 Jun 24 11:13 partners.txt -> clients.txt
```

What are the access permissions for `partners.txt`? Why?

The access permissions for `partners.txt` are `rw-r—r--`, as links always inherit the same access permissions as the target.

Imprint

PDF generated: 2025-12-04

The LPI Learning Materials are an initiative of Linux Professional Institute (https://lpi.org). Learning Materials and their translations can be found at https://learning.lpi.org.

For questions and comments on this edition as well as on the entire project write an email to: learning@lpi.org.