

1. Introduction to Linux

1.1. History

In order to understand what Linux is we need to go back in time, about 30 years back, when computers would occupy as much space as a house, cost just as much, and run just as slow.

It started all in 1969 with the development of UNIX. It was originally developed at Bell Labs laboratories as a private research project by a small group of people, just after their lab withdrew from the MULTICS project. Ken Thompson and Dennis Ritchie (the inventors of the C language) needed to develop an operating system to meet their goals of simplicity and code re-use; they ended up with UNICS, which was later renamed to UNIX.

UNIX was far more expensive than many existing operating systems, and many users were frustrated by their inability to buy this system for their own use, its source code was vigorously guarded and protected. Back then, Andrew S. Tannenbaum, a Dutch professor, wrote a small operating system to teach his student how a real operating system works; he called it MINIX, and for the first time, developers were able to look at the source code of a UNIX-like operating system.

In 1991, a Finnish student at the University of Helsinki, a self-taught hacker, decided to start working on a free alternative to MINIX. MINIX was cheaper than most UNIX'es, but it still cost money and required licensing. To Linus, MINIX was just an academic tool rather than a professional operating system.

At that time, the GNU project was gaining momentum. Led by Richard Stallman, the GNU project's ultimate goal was to make the software Free. Yes, *Free* not *free*! Free as in *Freedom*. GNU wanted to create quality software and allow people to redistribute it, modify it, fiddle around with it, do just about anything they want except stripping the original developers from their credit. GNU had successfully created *gcc*, a free GNU C compiler (known now as the GNU Compiler Collection), a shell called *bash*, in addition to alternatives to almost all UNIX tools. GNU's software repository was growing, and it was stable enough to create a free operating system. Now, if only they had a kernel ready...

You see, the kernel is the essence of an operating system, it's the part that talks to your hardware, manages the memory, reads and writes to the hard drive, produces sound and video; without a kernel, there would be no operating system, only a whole bunch of tools. GNU were working on a free kernel called HURD, but it needed years to come out. However, in August 1991, Linus uploaded his kernel to the university website and ported GNU utilities to make

them run on his kernel, eventually, this became the GNU/Linux operating system.

1.2. What is Linux

Linux is a free operating system kernel that implements a set of standard specifications known as POSIX. It's very similar to UNIX in addition to some extensions from *System V* and *BSD* operating systems. Licensed under the GNU General Public License (a.k.a. the GPL), Linux has penetrated largely into the corporate world, transforming it from a pet project to a solid foundation for many businesses.

The open nature of Linux allowed many vendors to compile up various software packages and gather them in a distributable format, making it easier to install and run by the average user. GNU/Linux (sometimes referred to as just Linux) comes in many "flavors", it can be a full-fledged operating system that is installed on your PC, such as SuSE's distribution, or a LiveCD version which doesn't require any installation, such as Knoppix, sometimes there's a whole company behind a distribution, like Fedora, and sometimes it's just a huge effort of many individuals, like Debian and Gentoo.

Don't be intimidated by the amount of available custom distributions, most of the times your needs make the decision for you.

1.3. This Book

The book is an introduction to GNU/Linux¹, we'll go down into installation and configuration details, how the system boots and starts services, how the file system is organized, networking, and a few other topics. This, by no means, is an extensive Linux references, it isn't a guide to any specific distribution either, I'll try to keep this book as distribution-neutral as possible, however, you might note that Fedora and Debian (which greatly differ) are ones of the most popular distributions available.

I'd like to welcome you to the wonderful world of Linux.

2. Booting Linux

When you first boot Linux you'll be greeted with a login screen asking for a username and a password. Linux is a multiuser operating system, much like Microsoft's Windows 2000 and XP, so in order to use the system you need to *login*. Linux has to basic modes for running a system, a text mode (sometimes

¹ There's a debate about whether to use Linux or GNU/Linux for a name. The Free Software Foundation obviously prefers the latter since they're responsible for most of the system, however this tends to confuse some users. This book refers to the system as Linux.

referred to as *console mode*), which looks like DOS and expects commands; and a graphical mode, which displays a desktop environment.

2.1. Graphical Mode

Most recent distributions use the graphical mode by default, it looks prettier, and does a better job at not scaring new users away. The graphical mode consumes more system resources, and is intended for use on workstations, since servers need every last bit of resources.

If you're familiar with Microsoft Windows then you won't find the Linux desktop strikingly different, most tasks can be completed using a simple point-n'-click method without having to remember Linux commands.

You might find a selection of available *sessions*, which are different desktop environments that you can login to, most commonly KDE and GNOME which we'll discuss later on.

2.2. Text Mode

When the whole screen is black and is showing (usually) white characters, you know you're in text mode. You'll typically find some information about your machine, the name and version of your distribution, and a prompt waiting for you to login:

```
Fedora Core release 3 (Heidelberg)
Kernel 2.6.9-1.667 on an x86_64

localhost login:
```

The information above tells us that we're about to login to a Fedora Core distribution that's using a 2.6.9 kernel version on an Athlon64 processor, and that the computer's name is *localhost*. Now let's login.

Type in your username and hit *Enter*, now you can write your password. Note that when you start writing the password, nothing will come out on your screen, not even an asterisk. Don't worry about that, this is just a security measurement, Linux is still accepting your password.

After the system has validated your username and password, you'll find yourself in a *shell prompt* where Linux is awaiting your commands to tell it what to do¹.

1 How a shell prompt looks depends on the logged in user. Throughout this book, I'll use the "\$"

You can logout by entering the command `logout`, or shutdown the system by entering `poweroff`.

2.3. Virtual Consoles

Remember that tip about Linux being a multi-user operating system? It allows you to login to the same computer as different users, even if you have one mouse and keyboard. You can use virtual consoles to login simultaneously to more than one account and perform activities in parallel.

Let's say you have a long task to do, like installing a big package or downloading a Linux distribution, you can go to the second console, by pressing `alt+F2`, login, and start your task. You'd still have your first session waiting for you.

Virtual consoles are numbered 1-6, and you can access them by pressing `alt+Fn` to bring up the *n*th screen. The graphical user interface runs on the 7th console. Note that in order to go from a graphical console to a shell console, you need to press `ctrl+alt+Fn` rather than `alt+Fn`. You can also cycle through different consoles by pressing `alt+left-arrow` and `alt+right-arrow`.

Note that you don't have to login as a different user on each console, Linux allows to use the same username for multiple logins.

3. The UNIX Way

Before we go on any further, I think you should know one *very* important difference between Linux (and UNIX'es in general) and many other operating systems.

Those who understand UNIX know that it's not just an operating system, it's a way of doing things, it's a "toolbox" designed to do things in a pretty efficient way. The UNIX philosophy tells us to write programs that do one thing, and only one thing, but do it well. Linux follows the same philosophy.

The first thing you notice about Linux is that there are tons of commands that most people used to take for granted in other operating systems (say, DOS?). That's not the case with Linux. For every little task there's usually a little tool to do it, you want to list files in a directory? There's `ls`. You want to turn off or restart your computer, there's `shutdown`. You want to find a list of files that contain a certain text and sort them by last-modified date and automatically copy a compressed version of them to a remote server, using nothing but a shell... well, that's where the magic lies.

The default Linux shell, namely `bash`, plays the main role in connecting different

to signify a regular user's shell, and a `"#"` to signify a root shell.

bits and pieces, it provides us with three key concepts, *redirections*, *pipes* and *filters*.

3.1. Redirections

Standard input (STDIN) and standard output (STDOUT) are the places on which a program takes input and writes the output, these can be different depending on how you view it, but in most cases, STDIN is your keyboard and STDOUT is your terminal screen. Of course, we're not limited to only those two, with *input redirection*, a program can send input or output to a different place, a file for instance. Redirection of STDIN and STDOUT is done by using the "<" and ">" symbols respectively.

Let's take *cat* for example. *cat* is a program that does one thing, it sends whatever it gets to its STDIN to STDOUT, meaning if you just run *cat* in your terminal screen, then all *cat* does is repeat what you write. "Now how can *that* be useful?", I hear you wonder. Well, with output redirection, we can tell *cat* to save what we're writing to a file, technically it means that *cat* would be *redirecting* its STDOUT to a file rather than the terminal screen. Try this:

```
$ cat > my_text_file
```

Now write something, anything, just remember to press Ctrl+D whenever you're done. Ctrl+D is a special character in Linux that signifies the end of a file (a.k.a EOF), whenever *cat* reads a EOF character, it exists, and in our case, it also saves the data to *my_text_file*. So let's make sure *cat* did save our text:

```
$ cat < my_text_file
```

When you run this command, you should see the same text you've written before. The "<" symbol redirects the contents of *my_text_file* to *cat*, which like we said before, sends its STDIN (the file) to STDOUT (the screen).

3.2. Pipes

Pipes are the key to connecting multiple programs to create a chain of processing. The pipe "|" causes a program to send its standard output to the next programs standard input, think of it as a junction, some glue that sticks together multiple filters.

Pipes let us build pretty complex programs just by stringing together a few filters.

3.3. Filters

To put it simply, a filter is a program that takes text on its standard input, processes it, and sends the processed text to standard output.

Many programs can be enhanced by connecting their output to other programs' input, like if you have a long list of files and only want to see a screenful at a time, you can pass *ls*'s output to *less*'s input:

```
$ ls -l | less
```

4. Commanding Linux

The CLI (Command-line Interface) isn't always as intuitive as the GUI¹ (Graphical User Interface), there are hundreds of commands to remember, each of them has too many options to remember. When in doubt, you always need a *man* to ask.

man, short for manual, is a program that displays the manual page for a command. So you forgot how to sort files using *ls*? No problem:

```
$ man ls
```

You'll get a short description of *ls*, followed by a *SYNOPSIS* section which quickly describes *ls* options, a longer *DESCRIPTION* section, and a detailed section explaining every available option.

4.1. Documentation

Documentation in Linux is usually scattered over *man* pages, *info* pages, FAQs and HOWTOs and elsewhere. Of course, *man* is the most common place to find information.

man pages are divided into 9 sections:

1. Commands
2. System services and error numbers
3. User-level library calls

¹ Fun Fact: How to find out if a person is a UNIX guru? Ask how to pronounce GUI. He'll always pronounce as "Gooney" rather than Gee, You, Eye.

4. Special files, hardware devices, network protocols
5. File formats and conversions
6. Games and demos
7. Miscellaneous information, macro packages and conventions
8. System maintenance and operation commands
9. Kernel routines

Many man pages can exist with the same name but live in different sections, take *passwd* for example. *passwd* is a command to change a user's password, its manual page exists in section 1. However, *passwd* stores password information about users in a file also named *passwd*, which also has a manual page, only it exists in section 5. By default, *man* looks for the first manual page that matches *passwd* and displays it. So if you need to take a look at the *passwd* file's document, you need to specify a section number:

```
man 5 passwd
```

There's another useful program that displays information about a given command, but instead of filling the screen with information, it only displays the short description found in the command's manual page:

```
whatis passwd
```

When you're done viewing a manual page, just press *q* to exit and get back to the shell.

4.2. Command Switches

Most Linux commands come with a set of switches that tell the command how to do its job. Switches can be short or long, and in some cases take extra arguments. Take *ls* for example. By default, *ls* only lists non-hidden files without any extra information about each file, just a list of file names. To get a long listing, we need to pass an option to *ls*.

```
$ ls -l
```

which displays something similar to:

```
drwxr-xr-x  2 root root  3696 Jun 11 04:17 bin
drwxr-xr-x  3 root root   256 May 20 13:51 boot
drwxr-xr-x 20 root root 14540 Jun 14 12:08 dev
drwxr-xr-x 73 root root  4920 Jun 14 12:10 etc
drwxr-xr-x  7 root root   208 Jun  6 02:23 home
drwxr-xr-x 10 root root  4432 Jun 11 04:17 lib
drwxr-xr-x  3 root root    80 Jun 14 12:08 media
drwxr-xr-x 11 root root   288 Jun 11 21:24 mnt
drwxr-xr-x  7 root root   240 May 31 00:54 opt
dr-xr-xr-x 107 root root    0 Jun 14 2005 proc
drwx----- 26 root root  1408 Jun 14 12:08 root
drwxr-xr-x  2 root root  4080 Jun 11 04:17 sbin
drwxr-xr-x 10 root root    0 Jun 14 2005 sys
drwxrwxrwt 25 root root  1128 Jun 14 12:14 tmp
drwxr-xr-x 15 root root   552 Jun  9 16:42 usr
drwxr-xr-x 14 root root   384 May 31 00:51 var
```

compare this to *ls*'s regular display of the same list of files:

```
bin boot dev etc home lib media mnt opt proc root sbin sys
tmp  usr  var
```

We'll get to explaining the long listing's details later on, but for now we need to figure out how to include hidden files in directory listings:

```
$ ls -a
.  ..  .bash_logout  .bash_profile  .bashrc  .gdbinit
```

The *-a* switch tells *ls* to list *all* files no matter whether they're hidden or not. In Linux, hidden files are those that begin with a dot.

Of course, we can combine switches, like when we need a long listing that

includes all files:

```
$ ls -a -l
```

or better yet

```
$ ls -al
```

If you come from a DOS background you might find this a bit different. Linux allows the combination of shorthand command switches. So you want to get a long listing of files, including hidden ones, sorted by file size, which we want in megabytes rather than bytes, in reverse order? No problem:

```
$ ls -lahSr
```

and it really doesn't matter how you order the switches, all of the following do the exact same thing:

```
$ ls -alhrS  
$ ls -SrhlA  
$ ls -rhlaS
```

compare this to DOS-style commands:

```
$ ls -a -l -h -r -S
```

or even the long-hand form:

```
$ ls -l --all --human-readable --reverse --sort size
```

Generally speaking, shorthand switches are prefixed with a single dash, long-hand switches are prefixed with two dashes. Some switches, like `--sort` can take an extra argument (e.g. `size`) in order to know how to operate.

Most commands in Linux have a *-h*, a *--help* switch, or both. This asks the command to display its usage information¹, that is, what switches are available and what syntax do they follow. Of course, this rule doesn't apply to all commands, but it's a convention used among most of them.

4.3. *--help*

There's another convention that Linux commands adhere to when display help, it looks the same as the *SYNOPSIS* section in a *man* page. For instance:

```
Usage: shutdown [-akrhPHfFnc] [-t sec] time [warning message]
```

Sections between [brackets] are optional, you don't need to pass them if you don't to. The *[-akrhPHfFnc]* section shows all available command switches. However, *time* is mandatory, this means that you can't run *shutdown* without telling it what *time* to shutdown.

4.4. *logout* and *shutdown*

These two programs are literally your way out. Whenever you're done working with your current console, you should always run *logout*. In most cases, *exit* is a synonym of *logout*, except *logout* doesn't work if you're running a terminal in GUI mode².

shutdown is the command you'd use to turn off or restart Linux. Once you give it a time to shutdown, it notifies all logged users, blocks further logins and starts the shutdown process:

```
shutdown -r now Houston, we have a problem!
```

The *-r* switch tells *shutdown* to reboot, *-h* tells it to halt, and *-k* tells *shutdown* to simply pretend it's going to shutdown and send the warning message to all users.

As a side note, if you've scheduled a shutdown and decided to cancel it, *shutdown -c* will do the trick.

1 If the list of switches produced by *--help* is too long, you can always pipe it to *less*:
ls --help|less. *Ironic, isn't it?*

2 Ctrl+D will also cause a logout, mainly because the shell is also a script interpreter. But we'll get to that later.

5. Linux File Hierarchy

So far so good, by now you should know enough basics to get deeper into Linux. There are many more commands to discuss, but in order to dig in, we need to go over a few more concepts of Linux.

5.1. Everything is a File

It's said that on UNIX, therefore Linux, everything is a file, and if it's not a file then it's a process. This statement is true because there are all sorts of files, files that are more than just files. It's a generalization to keep things simple. Linux makes no difference between a file a directory, since a directory is just a file that contains names of other files. Hardware is also represented as files, *device* files, you can have a a device file that represents a hard drive, a partition on a hard drive, a CD-ROM, a sound card, etc. All these are considered to be files by the operating system and are organized into a hierarchy of directories, a tree if you would, all starting with the *root* directory `"/`.

5.2. Types of Files

Most files in Linux are just *regular* files, they can be text files, executable files, compressed files and so on.

However, there are also other *special* types of files:

- Directories: Files that contain names of other files.
- Links: Files that point to other files making them available in multiple parts of the system. It's a bit similar to a *shortcut* in Windows.
- Sockets: Files that represent network connections. They act like pipes and provide means for communication between processes.
- Named pipe: Similar to sockets except they're network-less.
- Devices: Files that represent hardware devices.

If you ask `ls` for a long listing, it'll use the first character to tell the file type

```
$ ls -l
drwx----- 2 rami users 176 Jun  6 02:25 Desktop
drwx----- 2 rami mail  48 May 22 02:57 Maildir
```

```
-rw-r--r--  1 rami users  47 Jun 17 05:09 profile
```

Symbol	Meaning
-	Regular file
d	Directory
l	Link
c	Character device
b	Block device
p	Named pipe
s	Socket

5.3. Directory Layout

The root directory “/” contains several subdirectories, most commonly:

- **/bin:** Common binaries, programs shared by the system administrator and the users.
- **/boot:** Startup files necessary to boot Linux, this directory contains a Linux kernel, boot-loader files and configuration.
- **/dev:** Contains *device* files that represent system hardware, such as the hard drive, the mouse, etc.
- **/etc:** System-wide configuration files, this is similar in a way to Control Panel in Windows.
- **/home:** Home directories for all system users except root
- **/lib:** Library files, these are shared among many programs
- **/mnt:** Mounnt point for external file systems, such as USB drives, digital cameras, etc.
- **/opt:** Extra and third-party software. You might not find this directory in all distributions
- **/proc:** A read-only virtual file system that contains information about system resources, kernel status, running processes, etc.
- **/root:** Home directory for user root

- **/sbin:** System administration programs and binaries. These aren't usually used by users.
- **/tmp:** Temporary storage space that's usually cleaned up upon reboot.
- **/usr:** This is analogous to Windows' Program Files, it also contains programs but they're usually not critical for the system.
- **/var:** Variable data. This directory contains system logs, mail queues, printer spools, Web server's site, etc.
- **/lost+found:** Existence of this directory depends on the file system, it may and may not exist. It contains files that Linux was able to save from the last failure.

Directory layout varies between Linux distributions, some use a */media* directory to mount removable media (e.g. CD-ROM, USB drive, floppy) and keep */mnt* strictly as a temporary mounting point, others omit the */opt* directory, some have a different layout of */etc*, etc.

Generally speaking, Linux distributions more or less follow a standard named FHS (File Hierarchy Standard). It defines the directory layout, what goes where, and purposes of all these directories.

One of the things that make Linux flexible is that directories don't have to be *just* directories, they can be *mount points*. Mount points are how Linux makes resources available. For instance, if you'd like to read the CD-ROM you just plugged in, you need to *mount* it to a *mount point*, which in most cases */mnt/cdrom* or */media/cdrom*. This directory doesn't literally "contain" the CD-ROM's files but it shows what files are there, on the mounted CD-ROM.

So why is this useful? Well, this way you don't have to worry about where the data is located, you can have multiple data sources and still pretend as if all this data is locally available. You could mount a Windows share to a directory and use your favorite Linux programs to search, edit, copy and move files. You could mount a remote server's hard drive using NFS¹, or mount a floppy disk, a partition, you can even allocate a portion of your free RAM and mount it as if it was just another available device.

5.4. Navigating Directories

There are a few Linux commands that allow us to navigate through this tree-structure. The *current working directory* is where you're located at the moment:

1 NFS: Network File System, similar to Microsoft Windows Networks.

```
$ pwd
/home/rami
```

So if you want to go to */etc* and change some configuration, *cd* does the job:

```
$ cd /etc
$ pwd
/etc
```

The */etc* argument that we just passed to *cd* is known as *path*. In Linux, paths are either relative or absolute.

5.4.1. Relative and Absolute Paths

An absolute path is a path that begins with a forward slash "/", meaning it starts with the root directory and navigate lower into the hierarchy. So */etc* is a subdirectory of /. However, if you tell *cd* to go to *etc* or *./etc* instead of */etc*, it'll look for a directory called *etc* in the current working directory, so it's *relative* to the current working directory:

```
$ pwd
/home/rami
cd Desktop
pwd
/home/rami/Desktop
```

The dot "." signifies the current directory. If you want to navigate up into the structure, you can use the double dots "..", they stand for the *parent* directory. So if we want to go to John's home directory from */home/rami/Desktop* we could either use an absolute path:

```
$ cd /home/john
```

or a relative path:

```
$ cd ../../john
```

which means “go up two directories, and then into a subdirectory called *john*”.

5.4.2. cd Tips

You can always get back to the last directory you used without having to remember the full path by passing a hyphen to *cd*. This as tells *cd* to go back to where it came from and print the current working directory:

```
$ pwd
/home/rami/Desktop/Documents/
$ cd /home/john
$ pwd
/home/john
$ cd -
/home/rami/Desktop/Documents
```

To go to your home directory no matter where you are, try:

```
$ pwd
/etc
$ cd
$ pwd
/home/rami
```

Calling *cd* without any arguments takes you to the home directory of the currently logged in user. In Linux, a tilde “~” also stands for home directory, so if you want to view a file in your home directory you can use:

```
$ cat ~/work.txt
```

5.4.3. Popping Directories

Linux has an alternative to *cd*, namely *pushd*. This little program does exactly what *cd* does with one exception, it remembers where you were previously using *popd*:

```
$ pwd
/home/rami
$ pushd /etc
/etc ~
$ pushd /usr/share/
/usr/share /etc ~
$ popd
/etc ~
```

Note that *cd*, *pushd* and *popd* aren't separate programs, they're features that most Linux shells provide.

5.4.4. Bash Auto-completion

By the way, you don't have to write the complete directory name when you're typing a path, *bash* takes care of that for you, you simply have to hit *Tab*. Let's say you want to navigate to */usr/share/doc*, then you could try:

```
$ cd /u<TAB>sh<TAB>do<TAB>
```

which automatically translates to

```
$ cd /usr/share/doc
```

bash appends a slash at the end of the name if it's a directory, not a file.

Sometimes, you might hear a beep, this is *bash*'s way of telling you that it found more than one alternative, hitting *Tab* again will make *bash* list all available alternatives:

```
$ cd /b<TAB><TAB>
bin/  boot/
```

5.4.5. Wild Cards

Wild cards a way to match multiple files. Let's say you want to list all files that start with an *f*, you could use a wild card:

```
$ ls f*
```

There are three types of wild cards:

- *****: This matches an variable length of characters.
- **?**: This matches a single character
- **[a-z]**: A range, this can match anything within a character range, like [a-h], [1-4], etc. It can also take another form where you specify a list of characters to match rather than a range: [1-4] is exactly the same as [1234].

5.4.6. Making a Mess

There's a number of tools that let us create and delete files and directories, copy and move them around, and finally, view our changes.

5.4.6.1. Creating Directories

To keep things organized, we use directories¹ to store files in different locations. This is done with *mkdir*:

```
$ mkdir archives
$ ls
total 0
drwxr-xr-x  2 rami users 48 Jun 20 01:59 archives
```

¹ Also called *folders* in Windows

You can tell `mkdir` to create multiple directories instead of calling it multiple times for each new directory:

```
$ cd archives
$ mkdir 2003 2004 2005
$ ls
2003 2004 2005
```

Let's say you want to store your January business reports for 2005 in a directory tree, you could create a sub-directory for each of these:

```
$ mkdir 2005/January
$ mkdir 2005/January/Business
$ mkdir 2005/January/Business/Reports
```

or you can tell `mkdir` to do that for

```
$ mkdir -p 2005/January/Business/Reports
```

There are a couple of "gotcha's" with `mkdir`, in fact, these gotcha's apply to all commands in Linux. If you want to create a directory with a space in its name, you'll need to use an *escape character*, a backslash, it's a character that tells `mkdir` not to interpret the space as a separator, thus creating multiple directories; or you can surround the directory name with quotes:

```
$ mkdir Business\ Archives
$ mkdir "Other Stuff"
$ ls
total 0
drwxr-xr-x  2 rami users 48 Jun 20 02:21 Business Archives
drwxr-xr-x  2 rami users 48 Jun 20 02:21 Other Stuff
```

The other gotcha is when you want to create a directory with a name that starts with a dash, you'll need to use an *option terminator*, a double-dash, so that `mkdir`

doesn't interpret the directory name as a switch:

```
$ mkdir -- -archives
$ ls
-archives
```

These rules apply to almost all other Linux commands, when creating or deleting files, when searching for them, etc.

5.4.6.2. Moving Files and Directories

So now you created a directory structure to store your files, it's time you move them into those directories, the same of course applies to directories:

```
$ mv report.doc 2005/reports
$ mv /home/archives /home/rami/archives
```

You can also move multiple files or directories, the last argument you pass to *mv* is the destination you're moving to:

```
$ mv 2003 2004 2005 /home/archives
```

5.4.6.3. Copying Files

Copying files is done with the *cp* command. A useful option of *cp* is to copy files recursively, which causes *cp* to copy a directory and everything it contains, including other subdirectories:

```
$ cp -R /home/archives/ /home/rami/archives
```

5.4.6.4. Removing Files

Use *rm* and *rmdir* to remove files and directories on Linux. Generally speaking,

```
$ rm /home/archives/report.doc
```

Note that when you want to remove a directory, it has to be empty, it must contain no files at all, then you can use *rmdir*.

rm can remove directories too, without requiring you to empty anything beforehand, just tell it to be recursive:

```
$ rm -r /home/archives/2005
```

Removing files in Linux can be a bit risky, there is no trash can or a recycle bin, so when you remove a file it's really gone, there's no way to get it back unless you've properly backed up your data. To protect the innocent, *mv*, *cp* and *rm* have an *interactive* mode that's activated with the *-i* switch, this will cause them to ask before they overwrite or remove an

```
$ rm -ri *  
rm: remove directory `Business Archives'? y  
rm: remove directory `Other Stuff'? Y
```

5.4.7. Finding Files

The simplest way of finding files is by using *ls* with a combination of wild cards. Obviously, this isn't very effective when it comes to searching since you can't use it to search in subdirectories; well, not easily:

```
$ ls /home/archives/*/reports*/
```

bash can understand such expressions, the command above tells *ls* to show us all files residing under reports for all archived years.

5.4.7.1. *find* and *locate*

find is a utility that not only can search for files in subdirectories, but it can also find files based on specified criteria, like files that are older than a certain date or time, files that belong to a person, or files that don't, etc.

find uses sequential search, that is, it starts looking in a directory you specify and moves on to its subdirectories comparing what it finds with your criteria, until it reaches the end.

```
$ find /etc -name ``*open*``
```

The command above asks *find* to search in */etc* for all the files that contain the word *open* in their name. You can also search by size, let's say you want files that bigger than 1MB:

```
$ find . -size +1000k
```

One of the many features of *find* is its capability to execute a command for each file it finds. Let's say you want to remove all temporary files in your home directory, you could tell *find* to execute *rm* for each **.tmp* file:

```
$ find ~ -name ``*.tmp`` -exec rm { } \;
```

find has it's limitations, it's a 25 year-old command that's been around since the oldest UNIX'es. In 1999, *locate* was developed, it's similar to *find* in that it finds files, however, it doesn't use a sequential search, except it's much faster. *locate* searches an index database that's updated only once a day, where it stores a list of the system files. The limitation of course being that *locate* doesn't always find all the files you might be looking for, if you created a file after the last database update, *locate* won't be able to find it.

To update the database manually, use the *updatedb* command.

```
$ updatedb
$ locate network
/etc/runlevels/nonnetwork
/etc/runlevels/nonnetwork/.keep
/etc/runlevels/nonnetwork/local
/etc/pcmcia/network
/etc/networks
```

Modern Linux distributions these days use a security-enhanced version of *locate*, namely *slocate*, which only displays the list of files that a user is allowed to see, like the files in */root*, which aren't publicly accessible normally. In most cases, *locate* is just a symbolic link to *slocate*.

5.5. Directory Layout, Really

For most of us, it's enough to imagine that the file system is a tree of files and directories, it's a simplification that allows us to do most common tasks, like organization, searching, and other administrative tasks. Problem is, the computer has no idea about trees or tree-like structures.

In reality, all files (and directories for the matter) are ordered in a top-down structure, one after the other, and each file is assigned an *inode*, a unique serial number that identifies a file in a Linux file system. Each *inode* is associated with a list of information about the file, like its location on the hard drive, to which directory it belongs, and where to find its actual data.

Each partition has its own *inodes*, so files with the same *inode* can exist on different partitions.

When a new file is created, it gets a free *inode* number which will be assigned with the following information:

- Owner and group owner of the file
- Type: file, directory, link, etc.
- Permissions: Who gets to do what with the file
- Date and time of creation and last modification
- Number of links to this files
- File size
- Its actual location on the hard drive

The only information that Linux doesn't store in an *inode* is the file name and its directory, it stores *inodes* in a different location and compares file names to *inode* numbers in order to determine the file hierarchy.

You can see a file's *inode* by passing the *-i* switch to *ls*:

```
$ ls -li /home/rami
719663 Desktop 1039360 Maildir 543922 profile
```

5.5.1. Symbolic Links

Remember how I told you that files can have the same *inode* only if they're on different partitions? Well, that's not entirely true. You can create files that have the same *inode* but are located in different directories.

For example, let's say you have a file in `/home/work.txt` and want to share it with the rest of us system users; you could create a *link* to this file in each of our home directories. A *link* to a file is usually another file with same *inode* number, technically speaking, it's the exact same file, only it has a different name, an alias if you will.

Linux stores file names associated with each *inode* in a special table, this table is what lets Linux know how many links are there for a certain file, and a file isn't *really* deleted until all its links are deleted¹.

These links are called *hard* links. There is no obvious equivalent for them in Windows, since they're not exactly shortcuts. To create a link to a file, you can use `ln`:

```
$ ln /home/work.txt /home/hardlink.txt
$ ls -li
total 120
1918328 -rw-r--r--  2 rami users 59556 Jun 17 23:24 hardlink.txt
1918328 -rw-r--r--  2 rami users 59556 Jun 17 23:24 work.txt
```

Note how both files have the same *inode* number, the same size, owner and date.

There's also another type of links, namely *soft* links. These links work essentially like shortcuts in Windows. They are independent files, with their own *inode*, but they point to other files in the system. These links can be broken if the file they point to is deleted, since they're merely *pointing* to that file:

```
$ ln -s /home/work.txt /home/softlink.txt
$ ls -li /home
total 60
543877 lrwxrwxrwx  1 root root      8 Jun 17 23:29 softlink.txt ->
```

¹ Actually, files aren't deleted physically. When you delete a file, Linux only marks the space it used to occupy as free space and allows programs to write to it. This explains why deleting a large file is much faster than creating it.

```
work.txt
1918328 -rw-r--r-- 1 rami users 59556 Jun 17 23:24 work.txt
```

In this case, *softlink.txt* is a different file, it's not *work.txt*, but merely a pointer to it, it also has a different *inode* number.

5.5.2. Partitioning

Partitioning is a method to divide the bulk of hard drive free space into several sections each holding a portion of data. This usually achieves higher data security in the sense that if one partition get corrupt, this doesn't mean that everything's lost, you'll still be able to retrieve other partition's data.

If you're coming from a Windows background you'll know why partitioning is a good idea. Most people install Windows on one partition (namely C:), and reserve the rest of the hard disk for data on another partition (usually D:). This way, if you need to reinstall Windows¹, you don't have to hunt for your data on one partition, since it resides on another partition.

In Linux, partitions might not be as obvious as in Windows, they're not exactly separated into "drives". They *are* separate, but separate *devices* that once you mount, they indistinguishable from other directories.

Like I told you before, in Linux *everything* is a subdirectory of "/", even partitions, and in most cases, using partitions to separate the operating system from data from configuration is a good idea.

5.5.2.1. Partition Naming Schemes

Linux orders IDE² hard drives by their installation, how they're connected to the motherboard. IDE can connect up to 4 devices on a primary and secondary channels. Each channel connects a *master* and a *slave* device. Generally, operating systems only boot from a *master* hard drive, but this isn't a requirement for Linux³.

IDE devices are named using the following scheme:

- All IDE devices are available under `/dev/hdX`, where X is a, b, c or d.
- a, b, c and d stand for:

1 Which you might need to do a lot :)

2 Also known as ATA devices.

3 To be exact, it isn't requirement for GRUB, the boot loader.

- Primary Master: `/dev/hda`
- Primary Slave: `/dev/hdb`
- Secondary Master: `/dev/hdc`
- Secondary Slave: `/dev/hdd`
- Partition numbers are appended to device names, so the first partition on secondary master would be: `/dev/hdc1`

Device names don't change no matter what hardware you have available. This means that if you have a hard drive (as a primary master) and a CD-ROM (usually a secondary slave) installed on separate IDE cables you'll have `/dev/hda` and `/dev/hdd`, not `/dev/hda` and `/dev/hdb`. Once you plug in an additional hard drive on the primary cable, you'll get a `/dev/hdb`.

Linux can also use SCSI¹ hard drives to install. In SCSI setups there is no primary and secondary, a master or a slave, everyone is equal, the only thing that matters is the order of devices. Each SCSI device takes a unique serial ID, the SCSI *host controller* then uses this ID to determine devices' order. Linux names these devices as `/dev/sdx`, where *x* is a letter from a-z.

So if you want to install Linux on the first partition of the third SCSI hard drive, you'd use `/dev/sdc1` for your installation.

Let's not forget about SATA² support. When SATA first emerged, Linux supported it using a library called *libata*³, which made Linux treat SATA hard drives as IDE hard drives naming them `/dev/hdx`, where *x* is a letter starting from e. So the first SATA hard drive on your system would be `/dev/hde`. But you might probably only find this on older versions of Linux.

Because of their higher transfer speed, Linux treats SCSI hard drives different, and since SATA is much faster than IDE, Linux developers decided that it should be treated the SCSI is. So if you have a newer Linux version and want to install it on a SATA hard drive, you'll find SATA devices named `/dev/sdx`, the same way Linux names SCSI devices.

5.5.2.2. Partition and File System Types

Linux has two major partition types, *data* and *swap* partitions. Data partitions hold normal data, like the operating system files, your working data, etc. Swap

1 Small Computer System Interface. SCSI hard drives are mostly used in server environments for their high speed.

2 Serial ATA. Many see it as a hybrid between IDE and SCSI. It supports higher transfer speeds than IDE and still is relatively inexpensive.

3 Which is, as of today, is deprecated.

partitions act like a memory extension; if you run out of RAM, Linux uses a swap partition instead. Windows uses a similar mechanism, only it uses a *pagefile* (sometimes referred to as Virtual Memory) instead of a partition.

Partitions, of course, have to have a file system, otherwise they're just useless. Partitions only tell the operating system that a portion of the hard drive is allocated, but it doesn't go any further. They do however have significant numbers to tell the system what *type* of partitions are they. Linux data partitions have the number 83 assigned, swap partitions have 82, FreeBSD partitions have a5 associated.

Linux supports many file systems, including some very exotic ones (like Amiga FFS and OS/2 HPFS), but only few are used in production environments, the most popular being the *third extended* file system, a.k.a. *ext3*, but it's by no means the only choice¹:

- **ext2**: The *second extended* file system is known for it's stability. It's very slow compared to other *journaled* file systems, and it's being replaced by *ext3*.
- **ext3**: The *third extended* file system is based on *ext2* but it adds journaling capabilities hardening its resistance to data corruption.
- **ReiserFS**: Developed by Namesys, it's known to be one of the fastest file systems around. It's very stable and most efficient when you have a system with a large number of small files (e.g. a mail server or a proxy server).
- **JFS**: It was first used on AIX, an operating system developed by IBM, it was then ported to Linux. Like ReiserFS and *ext3*, it's a journaling file system.
- **XFS**: Developed by SGI for their Irix UNIX implementation, it's known for it's robustness and efficiency when working with smaller numbers very large files (e.g. a database server).
- **FAT32**: Developed by Microsoft for use with Windows 98² to overcome serious limitations of its older FAT12 and FAT16 file systems. It's not a very common choice for Linux installations, however, if you'd like to maintain interoperability between Windows and Linux installations.

Linux also supports reading and writing to NTFS, the file system used in Windows NT, 2000 and XP, so you don't have to worry about not being able to get your data.

All of the above-listed file systems, except for *ext2* and *FAT32*, support journaling; but let me first tell you a little about journaling:

¹ For a more detailed file system comparison, see this Wikipedia article:
http://en.wikipedia.org/wiki/Comparison_of_file_systems

² It was also used in Microsoft Windows 95 OSR2.

5.5.2.3. Journaling File Systems

File systems in general are very large data structure, they tell the system how the files are ordered, where to find them and most importantly, how to delete them.

Updating file systems tend to be large operations that take many writes to many places, manipulating a files meta data (what's associated with a file's inode) and the data itself; this might cause problems in case of crashes or power failures. For instance, deleting a file in Linux involves two operations:

1. Remove its directory entry, that is, telling the system that a file doesn't exist anymore.
2. Mark the associated inode as free space

If a power failure occurs before step 2, there will be an *orphand* inode, a reserved space that isn't associated with any file, causing a storage leakage. On the other hand, if the system would preform step 2 before step 1 and a failure occurs, the space will be marked as free (while it's not) and possibly overwritten.

Linux recover from such failures by walking over the file system structures (using a utility called *fsck*) to detect inconsistencies, which can be slow given the growing sizes of hard drives.

A journaling file system works around this problem by keeping a *journal*, some information about what it's going to do, so before the operating system wants to do anything with a file's meta data, it first logs it to a journal and *then* preforms whatever operations needed. Now in case the system crashes while it's manipulating a file meta data (like marking it as free space), it has enough information in its journal to "replay" the operation that it was going to do but couldn't.

Checking a journaling file system's consistency is merely checking its journal consistency which can take a few seconds for **hundreds** of gigabytes compared to several minutes for non-journaling file systems.

Journaling file systems can be a bit slower than non-journaling ones, since they have to write more information, data about data about data! But this is insignificant compared to their stability and error-recovery capabilities. They're a compromise between performance and security.

5.5.2.4. Primary and Extended Partitions

Before we go on further, we need to get an overview of partitioning on IBM-compatible PCs (i.e. x86 architecture). There are two types of partitions on the PCs, primary and extended, this is different from Linux partition types, it's the way the BIOS sees hard drives rather than how Linux sees them. Usually,

operating systems can only boot from a primary partition that is marked as *active*, but again, to Linux that doesn't matter.

PC architecture limits us to only four primary partitions, or three primary and one extended partition. The extended partition isn't a space allocation in the real sense, it acts like a container for *logical* partitions. The reason for this additional complication is that the original design's four partition limit was too restrictive. Extended partitions allow up to 15 logical partitions, but probably nobody uses this much partitions, it would be a nightmare to manage.

Linux allows assigns the number 5 to any extended partition, so if you want to mount the first logical partition, you'd use `/dev/hda6` for example.

5.5.2.5. Partitioning Hard Drives

So back to partitioning. Most Linux distributions come with a utility to create, delete and manipulate partitions called *fdisk*. Let's run *fdisk* on *sda* to prepare it for a Linux installation (note that you should run it as *root*):

```
# fdisk /dev/sda

The number of cylinders for this disk is set to 30401.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
 1) software that runs at boot time (e.g., old versions of LILO)
 2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help):
```

Press **p** to display how the hard drive is currently partitioned:

```
Command (m for help): p

Disk /dev/sda: 250.0 GB, 250059350016 bytes
255 heads, 63 sectors/track, 30401 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
```

```
Device Boot      Start          End      Blocks      Id  System
/dev/sda1                1          5099      40957686    7  HPFS/NTFS
/dev/sda2    *          5100          8747      29302560   83  Linux
/dev/sda3                8748          8991       1959930    82  Linux swap /
Solaris
/dev/sda4                8992         30401     171975825   83  Linux

Command (m for help):
```

On my hard drive, I have four primary partitions and no extended ones, an NTFS partition for Windows XP, a partition for the root directory "/", a swap partition, and a partition for data storage mounted on */home*.

The asterisk "*" marks the active partition, otherwise known as the *bootable* partition.

Let's delete all partitions and create a more suitable scheme for a Linux-only installation. Press **d** and type the partition number to delete, repeat until all partitions are removed:

```
Command (m for help): d
Partition number (1-4): 1
```

When you're done, press **p** just to make sure

```
Command (m for help): p

Disk /dev/sda: 250.0 GB, 250059350016 bytes
255 heads, 63 sectors/track, 30401 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Device Boot      Start          End      Blocks      Id  System
```

Now we can create a few partitions, but before we go on, we need to plan our scheme. Don't worry, until we ask *fdisk* to actually write anything, all we've done is just *mark* things to be done.

5.5.2.6. How Many, How Big?

The number and size of partitions depends greatly on the environment. For instance, if you have many users that have different configurations, you might want to place */home* on a separate partition. If you're going to run a mail server then you'll most probably place */var* on a separate partition. The size of each partition also depends on the nature of files you're going to store, */usr* for instance is mostly static, it can be large, but once you finished installing your programs, */usr* doesn't grow much. In server environments, you could have */tmp* on a separate partition, thus forcing a physical limit so */tmp* won't take up valuable free space. For additional security, you could have */boot* on a small partition by itself and not mount it when the system boots, this protects it from accidental deletion.

To keep things simple, we're going to use a 20GB partition for */*, a 1GB swap partition, and the rest of the hard disk for */home*. Note that swap partitions are usually double the amount of your RAM, but this isn't a rule, some Linux systems don't use a swap partition at all when they have enough RAM.

5.5.2.7. Creating Partitions

Press **n** in *fdisk* to create a new partition, you'll be asked whether you're creating a primary or an extended partition. *fdisk* is going to ask you to specify the first cylinder in the partition, just press *Enter*:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-30401, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-30401, default 30401):
+20000M
```

Use the same steps to create 1GB swap partition, only choose partition number 2. Remember, Linux swap partitions have a different type number, so we need to change it to 82:

```
Command (m for help): t
Partition number (1-4): 2
Hex code (type L to list codes): 82
Changed system type of partition 2 to 82 (Linux swap / Solaris)
```

Now let's create the last */home* partition. You don't need to specify its size, just press enter on the last cylinder prompt and *fdisk* will use the rest of what's left.

We need to do one final step, just to keep things in shape, we're going to mark the first partition as *active*:

```
Command (m for help): a
Partition number (1-4): 1
```

Let's take a look at how the partition table looks like:

```
Command (m for help): p

Disk /dev/sda: 250.0 GB, 250059350016 bytes
255 heads, 63 sectors/track, 30401 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1  *           1         2433    19543041   83  Linux
/dev/sda2                2434         2556     987997+   82  Linux swap /
Solaris
/dev/sda3                2557        30401   223664962+  83  Linux
```

Now all you need to do is press **w** and the changes are final, *fdisk* writes the new partition table to the hard drive:

```
Command (m for help): w
```

5.5.2.8. Creating File Systems

So now you have 3 partitions almost ready to have Linux installed on them, but as far as Linux is concerned, these partitions are only marked sections on the hard drive, it doesn't know how to write the files to them yet. We need to *mkfs* each partition; in Windows terms that would be "formatting".

We're going to use ReiserFS for */*, it's faster than ext3 and it should be a good choice to boot the system from. However, we're going to use ext3 on */home* for a couple of reasons, first is because ext3 is slightly more stable than ReiserFS, and because there are utilities for Windows that are capable of reading from ext3 partitions, so in case you need any files from your */home* you can do that even if you're on Windows.

```
# mkreiserfs /dev/sda1
# mkswap /dev/sda2
# mke2fs -j /dev/sda3
```

Note the *-j* option passed to *mke2fs*, it tells *mke2fs* to create a journal, like I said before, ext3 is essentially ext2 with a journal. All these utilities are parts of the *mkfs* set of tools, you can always use *mkfs* instead of *mk** and specify a file system type:

```
# mkfs -t reiserfs /dev/sda1
# mkfs -t swap /dev/sda2
# mkfs -t ext3 /dev/sda3
```

Both snippets do exactly the same, prepare partitions to store data.

5.5.2.9. Mounting

Accessing data stored on a partition can only be done after *mounting* it. Linux doesn't assign explicit names to partitions (e.g. C:, D:, etc.), a partition can be mounted to any directory. In order to mount the newly-created partitions, use *mount*:

```
# mount /dev/sda3 /home
```

This will cause Linux to make the third partition available via */home*. Accessing the CD-ROM, for instance, is done using the same way, you *mount* the CD-ROM device */dev/cdrom* to a directory */mnt/cdrom* and then pretend it's a regular directory just like any other, except you can't write to it.

Generally speaking, device files (those residing in */dev*) are accessed using *raw* data, 1's and 0's. When you create a new file system, Linux writes a bunch of 1's and 0's at the beginning of the device that represents the hard drive (*/dev/sda*) to identify the available file system. *mount* understands how these 1's and 0's are ordered and what they mean.

Unmounting file systems is done with *umount*.

```
umount /dev/cdrom
```

5.5.2.10. File System Table

Also known as *fstab*. */etc/fstab* is the file that that determines what's mounted on startup, it contains a few entries with mounting options:

```
$ cat /etc/fstab
/dev/sda2      /              reiserfs      noatime        0 0
/dev/sda4      /home         reiserfs      defaults       0 0
/dev/sda3      none          swap          sw             0 0
# NOTE: This is critical for proper booting
none          /proc         proc          defaults       0 0
```

The fields in order are:

- The device to be mounted
- The mount point to use when mounting that devices
- The file system type
- Mount options

- Dump number which determines what file systems need to be dumped when backing up.
- Pass number which determines the order in which the file systems are *fsck*'ed

5.5.2.11. Fixing File Systems

If Linux is cut down of power and the computer is suddenly turned off without properly unmounting file systems some data can be misplaced. You can fix that using *fsck*:

```
fsck /dev/hda1
```

You can also tell *shutdown* to *fsck* all disks on next reboot:

```
shutdown -r -F now
```

5.6. File Security

Linux follows the good ol' UNIX security model, which is a quite robust one. On Linux, every file is owned by a user and a group, and Linux forces three sets of permissions, each for a certain category of users:

- User Permissions: File permissions for the owner of a file.
- Group Permissions: File permissions for the group in which the file exists.
- Other Permissions¹: Permissions for who isn't the owner of the file, nor exists in that group.

For each category of users, Linux grants or denies any of these permissions:

- Read: Allows (or denies) users to read a file (by using *cat* for instance), or read files that exists in a directory
- Write: Allows users to write to a file (or delete it). It also includes creating files in directories.
- Execute: Allows users to run a program, you won't be able to run *ls* if it doesn't have execute permissions. In the case of directories, execute permissions allows listing files.

1 Also known as: World Permissions

When you ask `ls` for a long listing, it also displays the file permissions. Remember the set of letters next to each file's name? Well, this is how `ls` displays permissions:

```
$ ls -l /home/rami
total 1
drwx----- 12 rami users 1088 Jun 20 22:13 Desktop
$ ls -l /bin/ls
-rwxr-xr-x 1 root root 83168 May 30 01:06 /bin/ls
```

The first character, as we have already seen, signifies the file type, the rest are the permissions. Take *Desktop* for example, it's a directory that belongs to the user *rami* from group *users*, and it has the following permissions:

- User: *rw*x, that is, the owner *rami* is allowed to read, write and execute.
- Group: ---, that is, nobody in the *users* group can do anything with *Desktop*.
- Others: ---, same as group permissions, nothing is allowed.

Another example is */bin/ls*, the file belongs to *root* from the *root* group, and it has the following permissions

- User: *rw*x, *root* is allowed to read, write and execute *ls*.
- Group: *r*-x, the *root* group is allowed to read the file (which is necessary in order to execute it) and execute it, but nobody except the user *root* can overwrite it or delete it.
- World: *r*-x, the same permissions as the group permissions apply to everyone else.

5.6.1. Permission Codes

Linux uses a numbering scheme for its permissions where numbers add up to form the wanted permissions:

- - **or 0** for No permissions
- **x or 1** for execute permissions
- **w or 2** for write permissions

- **r or 4** for read permissions

Linux also represents user categories with letters:

- **u** for user, the file owner
- **g** for group
- **o** for others, world permissions

Adding up permission numbers forms the permissions we're looking for, so if you want to assign **rw** permissions to a file, you'll use the number **6**, which is 4+2, or r+w.

5.6.2. Changing Permissions

Sometimes you might want to change the permissions of a file, either for administrative purposes, or maybe just for sharing a file with the rest. *chmod* does exactly this, better yet, it can use either the numeric or the alphanumeric scheme for permissions.

Let's say we want to share the *Desktop* with the rest of system users, everybody in the group *users*. We can add read and execute permissions so everybody else in the group can have access to the files but isn't allowed to change them, still keeping others away:

```
$ chmod 750 /home/rami/Desktop
```

The problem is, in order to do a simple change (like adding group permissions) without modifying all permissions you'd need to calculate the correct numeric permissions, this gets tiring. You could use the alphanumeric permissions instead:

```
$ chmod g+rw /home/rami/Desktop
```

+ and **-** tell *chmod* to add or remove a certain permission from a file. Permission groups can be separated by a comma. So let's revert the permissions on *Desktop*:

```
$ chmod u+rx,g-rwx /home/rami/Desktop
```

6. User and Groups

Linux, like we said, is a multi-user operating system. This means that not only Linux allows different users to be logged in at a time, but it also sets permissions based on them.

There comes a time where you, the system administrator, would want to add new users and groups to be able to secure your system properly. But before we go on, let me tell you a little about how Linux stores user and group information:

6.1. */etc/passwd*

Linux stores information about user in a plain text file. No, don't worry, the password is always encrypted, only it's plain-text encryption, a bunch of meaningless characters.

Let's take a look at one line of */etc/passwd*:

```
$ cat /etc/passwd
root:$1$kP6upGFq$00z8onPdGLE68a1/:0:0:root:/root:/bin/bash
nobody:*:65534:65534:nobody:/:/bin/false
rami:$1$kP6upGFq$00z8onPdGLE68a1/:1000:100:~/home/rami:/bin/bash
```

Each line in *passwd* is a list of information about each user, the information is separated by a colon, this information in order is:

- A user's username
- His or her password
- The user ID, or the *UID*
- His or her group ID, or the *GID* (i.e. to which group the username belongs)
- Miscellaneous information: Historically, this field should contain the user's phone number, address, etc.
- The user's home directory
- The users default shell, that is, which program to run once the user is logged in.

The root user is always the user with the user with 0 UID and 0 GID, so you can

name the root user Administrator if you want to.

The other user, *nobody*, although it might sound a bit offensive, *nobody* is user who has almost no permissions except on certain directories. Processes also act as users, but we'll get to that later when we talk about processes and permissions.

There are a few problems with */etc/passwd* currently, most new Linux distributions use a new format called *shadow passwords* for additional security.

6.2. Shadow Passwords

By default, all users can read the */etc/passwd* file, in order to determine who else is available on the system, or figuring out what's the user's default login shell. The problem is that if users can read the password file then so can they read other users' passwords, and although these passwords are encrypted, a malicious user can apply a brute-force attack¹ and eventually decrypt the password.

The solution to this problem is by *shadowing* passwords, storing them in an alternate location, namely */etc/shadow* and the only person who's allowed to see the password list is *root*. With this new scheme, the old password file will look like this:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
nobody:*:65534:65534:nobody:/:/bin/false
rami:x:1000:100:~/home/rami:/bin/bash
```

The password field is replaced with an **x** instead of the encrypted password. Note that the user *nobody* has an asterisk for its password, that is, it's just not allowed to login.

Checking the shadow file permissions will tell you that only root can read the shadow file:

```
$ ls -l /etc/shadow
-r----- 1 root root 1315 Jun 21 00:07 /etc/shadow
```

¹ Brute-forcing a password means trying all possible password combinations until the password is found.

6.3. Changing Passwords

It is said the a chain is only as strong as its weakest link. In all operating systems, the weakest link is the user, or more specifically, the user's password. A strict password policy is essential for security, passwords should be at least 8 characters long, and are a mix of numbers and capital and small letters.

If you've discovered a user with a weak password, ask him to run *passwd* to change it. Linux forces certain policies on passwords in order to make the system even more secure, a user's password can never be too short or based on a dictionary word. When you're changing you're password, *passwd* doesn't show anything on the screen, but don't worry, keep on typing, *passwd* is reading your new password

```
$ passwd
Changing password for rami
(current) UNIX password:
New UNIX password: 123
BAD PASSWORD: it's WAY too short
New UNIX password: word
BAD PASSWORD: it is too short
New UNIX password: w0nderland
BAD PASSWORD: it is based on a dictionary word
passwd: Authentication token manipulation error
```

None of these passwords are acceptable to Linux. Of course, if you're root, you can do whatever you want, you can even mess up your system and not be questioned. Let's try that again

```
$ passwd
Changing password for rami
(current) UNIX password:
New UNIX password: AllThingsLinux
Retype new UNIX password: AllThingsLinux
passwd: password updated successfully
```

Now that's more like it. As root, you can also change other user's passwords, just

tell *passwd* the username:

```
$ passwd rami
```

passwd also changes group passwords using the *-g* switch:

```
$ passwd -g users
```

6.4. Switching Current Group

There's a reason for why groups have passwords. At any time, a user can switch his current group to another one, provided he knows the group's password. It's a mechanism to prevent having to *chmod* files every now and then:

```
$ newgrp developers
Password: LogMeIn
$ touch file.txt
$ ls -l file.txt
-rw-r--r--  1 rami developers 0 Jun 21 00:54 file.txt
```

The user *rami*, who exists in the *users* group is now effectively running as user *rami* from group *developers*. This can be useful when certain files or directories only have permissions set for one group but not the other, instead of *chmod*'ing the group's files to allow others to read those files, you can assign the group a password and let others know it.

6.5. /etc/group

The other file you should know about is */etc/group*. Linux stores group information in this file, it's quite simpler than */etc/passwd*:

```
$ cat /etc/group
root:x:0:root
users:x:100:games
portage:x:250:portage,rami
```

```
nobody:x:65534:
```

The *group* also separates group information by a colon, it stores the group name, the group password (or a *shadow* password), the GID, and which users are part of this group.

Every user has a default group which is specified in his entry in */etc/passwd*, but a user can belong to other groups too, in which case, the additional groups are specified in */etc/group*.

6.6. User Private Group

Older Linux distributions used to put all users in the same group, usually *users*. However, to allow more flexibility, new distributions use a scheme called *UPG*, or User Private Group, in which each user assigned his or her own group, a group that contains only one user, hence "private group".

As a side note, you assign a user or a group any UID and GID you want, but most distributions use a UID/GID number larger than 500 to create regular user. The IDs under 500 are used for services.

6.7. Logging in as Someone Else

As well as being able to change your effective group ID, you can also change the effective user ID without logging in and out, you can run as an *su*¹, a substitute user:

```
$ su frank
Password: icanbefrank
$ who am i
frank      pts/1          Jun 21 00:42
```

If you call *su* without passing a username, it'll default to *root*. Like in in case, *root* can do whatever he wants, even login as another user without being asked for a password.

To execute a single command, *su* seems like an overkill, in this case you can *sudo*, a command that let's you execute other commands as someone else:

1 *su*: substitute user. Many like to think of it as *super* user since it defaults to *root*.

```
$ sudo -u frank ls /home/frank/
```

6.8. Changing Ownership

When a file or a directory is owned by the wrong user or group, this error can be fixed with *chown*, for *change owner*. Of course Linux will check before executing *chown* for sufficient permissions:

```
$ chown frank:users report.doc
```

If you need to change ownership for a directory and all the files inside it, you can tell *chown* to run recursively:

```
$ chown -R frank:users /home/archives
```

6.9. Special Modes

Linux has some other useful access modes:

- **Sticky Bit:** Historically, the sticky bit was used to keep file in memory. If you set the sticky bit for a directory, any file you access is kept in memory, literally it "sticks" in the swap memory. Memory these is getting cheaper, so the sticky bit isn't used for this purpose any more. However, it's used for a different purpose. If you set the sticky bit for a directory then only the owner of the file and the owner of the directory can remove files. This is commonly used in directories like */tmp*, where all users have write access, yet still they shouldn't be deleting each other's files.
- **SetUID:** If only root has access to */etc/shadow*, then how can users change their passwords when *passwd* runs with their permissions. The answer is SUID. Setting the SUID bit on */bin/passwd* and let *root* own that file. Next time when *passwd* is execute, it won't be running as the logged in user, it will run as the *owner* of the program, in this case, *root*.
- **SetGID:** The SGID bit does the same as SUID, it causes programs to run as another group, the owner group of the file.

Actually, permissions aren't a set of three-digit numbers, but four. The first digit sets the sticky bit, SUID and SGID, and the other three set the desired

permissions. Of course, you can use an alphanumeric or a numeric scheme in order to do that:

- Sticky bit: represented by a **t** or **1**
- SUID: represented by **s** or **4**
- SGID: represented by **s** or **2**

For example, you can change SUID for `/bin/passwd` and disallow changing passwords:

```
# chmod u-s /bin/passwd
```

Let's also set the sticky bit for `/tmp`:

```
# chmod 1777 /tmp
```

6.10. Adding and Removing Users and Groups

To add a new user to your Linux system, use `useradd`. You need to remember to change the user's password, otherwise he won't be able to login:

```
# useradd john
# passwd john
New UNIX Password: johnpass
Retype new UNIX Password: johnpass
passwd: password updated successfully
```

If you change your mind and decide that *john* doesn't deserve logging in, you can always remove him:

```
# userdel john
```

`useradd` only adds a user to the system, that is, create an entry in `/etc/passwd`, a user still needs a home directory to store his files. You can tell `useradd` to do just

that:

```
# useradd -m john
# ls -al /home/john
total 36
drwxr-xr-x  2 john users   160 Jun 21 02:25 .
drwxr-xr-x  8 root  root   232 Jun 21 02:25 ..
-rw-r--r--  1 john users   240 Jun 21 02:25 .bash_logout
-rw-r--r--  1 john users   232 Jun 21 02:25 .bash_profile
-rw-r--r--  1 john users   812 Jun 21 02:25 .bashrc
```

So if we just added *john*, how come he has files in his home directory? Well, *useradd* uses a *skeleton* when it creates users' home directories, like a template for new home directories, you can find it in */etc/skel*. These files are per-user *bash* configuration files, but you can do more than that, like add welcome message to */etc/skel* in a *welcome.txt* file.

To specify a group that the new user belongs to, use the *-g* switch:

```
# useradd -m -g users tom
```

The commands to add and remove groups are similar to users:

```
# groupadd designers
# groupdel developers
```

You can modify a group's name using *groupmod*:

```
# groupmod -n designerz designers
```

6.11. Who's There?

Sometimes you need to know who's currently logged in to your system, for this purpose, use *who*:

```
$ who
root      vc/1          Jun 21 02:37
rami      vc/2          Jun 20 21:15
```

Currently, there are two users logged in, *root* on the first virtual console, and *rami* on the second. You can also use *who* to determine who are you logged in as, just use any two arguments:

```
$ who am i
rami      pts/1          Jun 21 00:42
$ who mom likes
rami      pts/1          Jun 21 00:42
```

7. Linux Processes

Like I said before, Linux is a multi-tasking multi-user operating system, it can run multiple programs without complaining. Running a command usually (but not necessary) initiates a single process, some commands on the other hand start a series of processes, like Mozilla, the browser.

Furthermore, when you run a command, its process use the same permissions as the currently logged in user, sometimes referred to as the *effective* user.

7.1. Interactive Processes

The processes you start from the terminal are known as interactive processes, they aren't normally started with the system, but rather run by a logged in user. These processes occupy the terminal until they're done.

Actually, the interactive processes that do occupy the terminal are processes running in the *foreground*. Alternatively, you can run a process in the *background*, letting it do its whatever it needs to do while you're still using the terminal.

All the commands we used so far run in the foreground, only the time taken to

run them was too short to notice. Other commands like *less* occupy the whole terminal, and expect user input, in this case, the terminal is only useful for entering commands that the foreground program can understand.

On the other hand, when a process is run in the background, you won't be prevented from doing other things on the terminal, you just let background processes run peaceful and quite.

The Linux shell, *bash*, offers a mechanism to control these processes called *job control* which allows switching between multiple running commands. Of course, only one job can run in the foreground, but there's virtually no limit to how many jobs or processes run in the background.

Take *updatedb* for example. It's a command that takes a while to finish re-indexing files, but while it's doing that, you shouldn't be waiting for it to finish, you should start it in the background, and continue doing your regular job. Starting a command in the background, a background job, is done by appending an ampersand to the command:

```
# updatedb &
[1] 12824
# jobs
[1]+  Running                  updatedb &
```

The number you see between brackets after you start a background job is the job's ID. Each job you start, either a foreground or a background job, is assigned a new ID, starting with 1. The *process* that the job manages is also assigned an ID, a PID if you will. PID's are global, any process you start has a unique PID throughout the whole system, while job IDs are specific to each terminal.

If you forget to start a command in the background, you can always send it there by pressing **Ctrl+Z**. Note that this *stops* the current job and then sends it to the background, it won't continue running until you tell it to. Once you send a process to the background, you should tell it to continue working using the *bg* command. Of course, you could always bring a process to the foreground using the *fg* command.

```
# updatedb
Ctrl+Z
[1]+  Stopped                  updatedb
# bg 1
```

```
[1]+ updatedb &
# jobs
[1]+  Running                  updatedb &
# fg 1
```

7.2. Scheduled Processes

Also known as automatic or batch processes. These processes run without being connected to the terminal, rather they're queued in a spooler and processed on a first-come first-served basis using one of two criteria:

- At a certain time or date using the *at* command or a *cron* daemon.
- At low-load times using the *batch* command. These processes are run when the system load is lower than 0.8. Certain commands and programs don't need to be run urgently, might take a considerably long time to finish, or might cause a big hit on the system performance. The system administrator would rather run them at low-load times.

The *cron* daemon is a program that runs process periodically. Every minute, *crond* will read */etc/crontab* and will execute all the scripts in */etc/cron.**. These directories tell *cron* when to run the scripts:

```
$ ls -d /etc/cron.*
/etc/cron.d /etc/cron.daily /etc/cron.hourly /etc/cron.monthly
/etc/cron.weekly
```

Let's take a look at */etc/crontab*:

```
$ cat /etc/crontab
# Global variables
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/
```

```
# check scripts in cron.hourly, cron.daily, cron.weekly and
cron.monthly
0 * * * * root rm -f /var/spool/cron/lastrun/cron.hourly
1 3 * * * root rm -f /var/spool/cron/lastrun/cron.daily
15 4 * * 6 root rm -f /var/spool/cron/lastrun/cron.weekly
30 5 1 * * root rm -f /var/spool/cron/lastrun/cron.monthly
```

The last four lines are the scripts that run every now and then, the fields in order are:

- Minutes: 0-59
- Hours: 0-23
- Day of month: 1-31
- Month: 1-12
- Day of week: 0-6
- User to run the command as
- The script or command to run

There's also *atd*, a simpler scheduling daemon that allows you to schedule one task at a time, it doesn't require much configuration. You can tell at what commands to run by specifying a time to run:

```
at now
warning: commands will be executed using /bin/sh
at>touch /tmp/atd
Ctrl+D
```

The *now* part can be any other different time, like *3am + 2days*, *midnight*, *10:14 Apr 12*, etc.

To see a list of pending jobs use *atq*, *atrm* to remove a job.

7.3. Daemons

In a Windows world, these are known as services, processes that run continuously. In most cases, these are server processes, web servers or mail servers, but they don't have to be. *cron* is a program that runs as a daemon but it's not a server, it's a scheduler. Most of these processes are run when the system boots, but again, they don't have to be.

7.4. Process Attributes

Whenever you start a process it is assigned a few information:

- PID: A unique ID used to refer to the process.
- PPID: The ID of the parent process, the parent PID, that's the process which started the next process.
- "Niceness": A concept similar to *priority*, it's a number that determines how friendly a process is to other processes. A process priority is calculated based on its *nice* number and it's recent CPU usage. Generally speaking, a higher nice number means a process uses less system resources and vice versa.
- TTY: A terminal number, the terminal to which the process is connected
- RUID and EUID: That is the *real* UID and the *effective* UID. Usually they're the same, but they don't have to be. The RUID is the owner of an executable while the EUID is the UID of who run the process. For instance:

```
$ ls -l /bin/bash
-rwxr-xr-x 1 root root 694012 May 18 05:13 /bin/bash
$ bash
$ ps -af
UID          PID  PPID  C  STIME TTY          TIME CMD
rami         21438 21355  0  22:37 pts/1        00:00:00 bash
rami         21440 21438  0  22:37 pts/1        00:00:00 ps -af
```

In this example, *bash* is owned by *root* which is its RUID. However, when its run, it is executed with *rami's* permissions, its EUID.

- RGID and EGID: The concept applies to the *real* GID and *effective* GID. That is the owner group and the group that runs a command.

7.5. Process Information

If you want to know what processes are currently running, use the `ps` command. With no options passed, `ps` displays the processes running the current terminal:

```
$ ps
  PID TTY          TIME CMD
 21521 pts/1        00:00:00 bash
 21524 pts/1        00:00:00 ps
```

Of course, this amount of information is not nearly enough for a system administrator, not even the user himself. If you want to know all the processes that a certain user started, pipe `ps`'s output to `grep`:

```
$ ps -ef | grep rami
rami      21521 11411  0 22:46 pts/1    00:00:00 /bin/bash
rami      21583 21521  0 22:50 pts/1    00:00:00 ps -ef
rami      21584 21521  0 22:50 pts/1    00:00:00 grep rami
```

We can also tell `ps` to display all processes with user oriented information using a wide format. Let's see how many `bash` processes are running currently:

```
$ ps auxw | grep bash
rami      11423  0.0  0.1  2748  1384 pts/2    Ss   17:41   0:00
/bin/bash
rami      21475  0.0  0.1  2748  1384 pts/3    Ss   22:42   0:00
/bin/bash
root      21481  0.0  0.1  2492  1424 pts/3    S    22:42   0:00 bash
rami      21521  0.0  0.1  2748  1388 pts/1    Ss   22:46   0:00
/bin/bash
rami      21624  0.0  0.0  1700   452 pts/1    R+   22:53   0:00 grep
bash
```

The problem with `ps` is that it only gives process information in snapshot of time,

that's not particularly useful with short-life processes. In this case, use the `top` command to display processes information in real-time:

```
$ top
Tasks:  85 total,   2 running,  83 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.9% us,  1.4% sy,   0.0% ni, 91.6% id,   3.0% wa,   0.1% hi,
0.0% si
Mem:   903344k total,  883216k used,   20128k free,  133048k
buffers
Swap: 1959920k total,   2816k used, 1957104k free,  341048k
cached

   PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
11231 root        15   0   336m  77m 6640 S   4.0   8.8   3:53.19 X
21508 rami        15   0   125m  31m  12m S   4.0   3.6   0:34.81
kxineplayer
11375 rami        15   0 29048  7348 5104 S   2.0   0.8   1:42.75 artsd
   1 root        16   0   1588   484  424 S   0.0   0.1   0:00.29 init
   2 root        34  19     0     0     0 S   0.0   0.0   0:00.01
ksoftirqd/0
   3 root        10  -5     0     0     0 S   0.0   0.0   0:00.12 events/0
   4 root        10  -5     0     0     0 S   0.0   0.0   0:00.00 khelper
   9 root        10  -5     0     0     0 S   0.0   0.0   0:00.00 kthread
  20 root        20  -5     0     0     0 S   0.0   0.0   0:00.00 kacpid
  18 root         8 -10     0     0     0 S   0.0   0.0   0:00.39 vesafb
```

`top` displays a plethora of information, such as how many processes are running, sleeping or stopped, the amount of CPU usage, memory usage, virtual memory usage, etc.

`top` gives a more detailed view than `ps`, in fact it uses `ps` to update it's list of information, it also uses other programs like `uptime` which displays the first line in `top`'s view:

```
$ uptime
```

```
23:11:29 up 5:32, 6 users, load average: 0.39, 0.25, 0.25
```

Viewing process relationship is done using *ps*tree, a command that displays a tree of processes with information about who started what, who are parents to which processes:

```
$ pstree -A
init--+-events/0
    |-iiimd
    |-init---sh---pstree
    |-khelper
    |-khpsbpkt
    |-khubd
    |-2*[kjournald]
    |-knodemgrd_0
    |-krfcommd
    |-kseriod
    |-ksoftirqd/0
    |-kswapd0
    |-kthread--+-aio/0
        |       |-ata/0
        |       |-kacpid
        |       |-kauditd
        |       |-kblockd/0
        |       |-kmirrord/0
        |       `--2*[pdflush]
    |-rpc.idmapd
    |-scsi_eh_0
    |-scsi_eh_1
    |-shpchpd_event
    |-sshd---bash
    |-udev---2*[udev]
```

7.6. Signals

Processes communicate with each using *signals*. Using a signal, you interrupt a process, ask it to exit, or kill it. The *kill* command is used to send these various signals to processes, *kill -l* shows a list of known signals:

```
$ kill -l
 1) SIGHUP          2) SIGINT          3) SIGQUIT        4) SIGILL
 5) SIGTRAP        6) SIGABRT        7) SIGBUS         8) SIGFPE
 9) SIGKILL       10) SIGUSR1       11) SIGSEGV       12) SIGUSR2
13) SIGPIPE       14) SIGALRM       15) SIGTERM       17) SIGCHLD
18) SIGCONT       19) SIGSTOP       20) SIGTSTP       21) SIGTTIN
22) SIGTTOU       23) SIGURG        24) SIGXCPU       25) SIGXFSZ
26) SIGVTALRM     27) SIGPROF       28) SIGWINCH      29) SIGIO
30) SIGPWR        31) SIGSYS        34) SIGRTMIN      35) SIGRTMIN+1
36) SIGRTMIN+2   37) SIGRTMIN+3   38) SIGRTMIN+4   39) SIGRTMIN+5
40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8   43) SIGRTMIN+9
44) SIGRTMIN+10  45) SIGRTMIN+11  46) SIGRTMIN+12  47) SIGRTMIN+13
48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14  51) SIGRTMAX-13
52) SIGRTMAX-12  53) SIGRTMAX-11  54) SIGRTMAX-10  55) SIGRTMAX-9
56) SIGRTMAX-8   57) SIGRTMAX-7   58) SIGRTMAX-6   59) SIGRTMAX-5
60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2   63) SIGRTMAX-1
64) SIGRTMAX
```

Signals are of most interest to Linux developers, you wouldn't want to know all the signals in detail, but the most commonly known signals are:

- **SIGINT**: Interrupt a process. Most processes exit when they receive this signal, but they can ignore it.
- **SIGTERM**: Terminate a process. Tell the process to terminate properly, and do its "housekeeping".

- **SIGHUP:** HUP is short for Hang UP. Most processes simply reread their configuration their configuration files.
- **SIGKILL:** This signal cannot be ignored, the system kills the process no matter what its doing.

```
# updatedb &
[1] 22004
# pgrep updatedb
22004
# kill -SIGTERM 22004
# jobs
[1]+  Terminated                  updatedb
```

7.7. Zombies

As spooky as it might sound, zombie processes are processes that are already dead!

You see, when a parent process starts a child process it assigns a PID to it and it reserves that PID until the child exists properly.

Whenever you send a SIGTERM to a child process it starts doing its regular "housekeeping", that is, cleaning up the memory it reserved, deleting temporary files, etc. It also sends a signal to its parent telling it that it's about to exit. Parents wait for their children to exit, but they still count them as did, since they got a signal about it. In this case, a PID is reserved, while the child process is considered dead; it becomes a *zombie*.

All processes are children, and I do mean *all*, no exceptions. However, they don't always appear so. The parent of all processes in Linux is *init*.

8. Booting Linux

Of of the most powerful features of Linux is its open method for booting up the system. Linux's booting process determines how to start the operating system, how to start services and when, the order of starting services, etc.

This open method of booting can also be a lot of help when you need to find out

the source of certain problems, so understanding the boot process is essential.

In reality, everything related to booting in Linux is stored in a text file, except for the kernel. But let's talk a little about how PCs start first.

8.1. The Boot Process

When you press the power button, you actually initialize a very small program, a mini operating system if you will, called the BIOS (Basic Input/Output System). The BIOS is stored on a read-only memory called ROM¹, so it's always available to the user, this mini OS is the basic means to communication with the hardware.

The BIOS first tests the system, then the peripherals, and then looks at what drives are available. It also reads its configuration to determine your choice about the boot device. Most new BIOSes are able to boot from hard drives, CD-ROMS, floppy disks, USB flash drives and even network.

When the BIOS is done testing, it looks for available hard drives, finds out the hard drive it's going to boot from, then reads it's MBR (Master Boot Record) and loads its content into the main memory. After that, the BIOS has nothing to do with how the system boots.

The MBR contains a boot loader, in Linux's case, it contains either GRUB (GRand Unified Bootloader) or LILO (Linux LOader). Older distributions used to use LILO to boot Linux, but most recent ones have switched to GRUB which I'm going to discuss.

GRUB has certain advantages over LILO, mostly its flexibility. GRUB also has a minimalistic shell which allows you to modify its booting options even before it loads any operating system. GRUB can be used to boot many operating systems, including Windows, FreeBSD, and others.

To be honest, the MBR doesn't contain the boot loader, it's just too big to fit, so MBR only contains a *part* of the bootloader; GRUB calls it a stage.

GRUB is known as a multi-stage boot loader, which means that it stores a part of it in the MBR (known as Stage 1) and let's it take care of loading the rest of GRUB's stages. Once it's done, it displays a list of operating systems it can boot and waits for a certain amount of time, and unless you change your mind, it boots the default system. GRUB also has another useful feature, it doesn't have to overwrite the MBR each time you change its boot options. LILO on the other hand does.

There's also another way of booting an operating system, any operating system, it's called *direct loading*. Systems like Microsoft DOS and Windows 95 use direct

¹ In newer PCs its stored on an EPROM, an Erasable Programmable Read-Only Memory, an EPROM allows "flashing" the BIOS and upgrade it.

loading to boot the system where there is no intermediate code between the bootloader and the system's kernel. Usually these system's overwrite whatever resides in the MBR.

These system's suffer from a limitation on the x86 architecture, a 1024 cylinder limitation. Older systems had to have their bootloader in the first 1024 of the hard drive, that's roughly 528MB, this affected even Windows 98 and ME. With a more advanced bootloader (like Windows's NTLDR or Linux's GRUB) a system can overcome this limitation¹.

So, GRUB reads its configuration file, figures out which kernel to load, reads it and puts into memory and let's it do the rest. The kernel detects the hardware and loads any modules it needs to support it, then...

8.2. *init*

Once the kernel is loaded, it finds `/sbin/init` and executes it.

`init` is the parent of all processes in Linux, it's responsible for running absolutely everything. `init` first reads `/etc/inittab`, then it sets paths, configures available hardware, sets the clock and starts system services.

`Inittab`, short of `init table`, describes how `init` should setup the system and boot to which run level. A run level is a configuration of processes, which can be different at various mode. There are many run levels which you can choose from, depending on what you want from the system. For instance, a single-user run level only runs the services that allow one user to connect to the system, no networking services, no other extra services. There's a multi-user run level which, in addition to starting single-user services, starts networking services, daemons, etc. There's also a graphical run-level, which runs a graphical desktop in addition to multi-user services. Let's take a look at `inittab` first and explaining it: Note that this `inittab` comes from Fedora Core 4, but the concept is the same with all other distributions.

```
$ cat /etc/inittab
# Default runlevel. The runlevels used by RHS are:
# 0 - halt (Do NOT set initdefault to this)
# 1 - Single user mode
# 2 - Multiuser, without NFS (The same as 3, if you do not have
networking)
# 3 - Full multiuser mode
```

¹ Actually, they need to overcome a few other limitations as well, but this is beyond the scope of this book.

```
# 4 - unused
# 5 - X11
# 6 - reboot (Do NOT set initdefault to this)
#
id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

# When our UPS tells us power has failed, assume we have a few minutes
# of power left.  Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powerd installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting
Down"

# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown
Cancelled"
```

```
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

# Run xdm in runlevel 5
x:5:once:/etc/X11/prefdm -nodaemon
```

Note that lines starting with a `#` are comments, you can ignore them.

8.3. Run Levels

There are 7 run levels, number 0 to 6:

- Halt (0): This is the run level where the system is turned off. When you ask Linux to switch to run level 0, it'll shutdown all the services in the current run level and turn off the computer. So you shouldn't have this as your default run level.
- Single-user (1): Usually starts only the necessary services to have a shell and do administrative tasks.
- Multi-user (2): This differs from distribution to another, but in most cases this run level supports multiple-user logins but disables networking.
- Full Multi-user (3): In addition to the above multi-user run level, it starts networking services and servers.
- Custom (4): On most distributions this run level isn't used, you're free to configure your system to start and stop the services you want in this run level.
- Graphical (5): It's the same as run level 3 in addition to a graphical desktop.
- Reboot (6): Like in run level 0, the system turns off all services, but instead of turning off the computer it restarts it.

There's another run level which *always* runs called *sysinit*, it's the run level where

the system initializes the absolute necessities. Note that these run levels are only conventional, technically Linux doesn't *require* run level 1 to be a single-user run level or run level 5 to be graphical, but almost all distributions are based on this convention. Gentoo¹, for example, has run level 1 associated with single-user, run level 2 with nonnetwork, and run levels 3, 4 and 5 associated with a default run level, no otherwise specific meanings.

The line:

```
id:5:initdefault:
```

tells *init* what the default run-level is, which in this case is 5, the graphical desktop.

The next line:

```
si::sysinit:/etc/rc.d/rc.sysinit
```

tells *init* that when it switches to *sysinit* run level it should run a script located in */etc/rc.d/rc.sysinit*.

The next lines tell *init* which script to run for each run level:

```
l0:0:wait:/etc/rc.d/rc 0
```

in Fedora's case, all run levels are controlled by a single script in */etc/rc.d/rc* which is passed an argument to let it know which run level to switch to.

8.4. System V Init

Historically, there has been two styles of run level, the traditional BSD and the System V boot styles. Most Linux distributions use System V boot style, only a few of them like Slackware use BSD boot style.

In a System V boot style, configuration files and services are located in different directories, each directory represents a run level:

```
$ ls -ld /etc/rc.d/rc*
```

1 Gentoo: A source-based distributions, famous for it's package management system.

```

ls -ld /etc/rc.d/rc*
-rwxr-xr-x 1 root root 2371 Oct 31 2004 /etc/rc.d/rc
drwxr-xr-x 2 root root 4096 Jun 19 00:32 /etc/rc.d/rc0.d
drwxr-xr-x 2 root root 4096 Jun 19 00:32 /etc/rc.d/rc1.d
drwxr-xr-x 2 root root 4096 Jun 19 00:32 /etc/rc.d/rc2.d
drwxr-xr-x 2 root root 4096 Jun 19 00:32 /etc/rc.d/rc3.d
drwxr-xr-x 2 root root 4096 Jun 19 00:32 /etc/rc.d/rc4.d
drwxr-xr-x 2 root root 4096 Jun 19 00:54 /etc/rc.d/rc5.d
drwxr-xr-x 2 root root 4096 Jun 19 00:32 /etc/rc.d/rc6.d
-rwxr-xr-x 1 root root 220 Jun 23 2003 /etc/rc.d/rc.local
-rwxr-xr-x 1 root root 17935 May 9 22:16 /etc/rc.d/rc.sysinit

```

Each of these directories contains a number of scripts, files starting with either the letter "S" or "K", that is, *Start* or *Kill*. These scripts define what gets started and what is stopped when switching to or from run levels. The extra */etc/rc.d/rc.local* directory is where you store your own startup or shutdown scripts, hence the *rc.local*.

In addition to the "S" and "K" letters, the service name is preceded with a number that defines where it falls in the booting order. Some services depend on each other and cannot be started without others before them, many network services for example depend on one service called *xinetd*.

The next few lines in *inittab* tell *init* to *spawn* 6 instances of *mingetty*, a terminal emulator for Linux:

```
1:2345:respawn:/sbin/mingetty tty1
```

it also tells *init* that this applies to all run levels, 2 through 5, and whenever *mingetty* exits, *init* would start it again; *init* is the parent of all processes (like you can see in *ptree*), remember? So it can know who of its children has exited (or killed) and bring it back to life again.

```

init--acpid
|-6*[agetty]
|-cifsoplockd

```

```
|-cron---cron---bash---run-crons---slocate---updatedb
|-dbus-daemon-1
|-dcopserver
|-events/0
|-gconfd-2
|-hald
|-syslog-ng
|-udev
|-vesafb
`-xinetd
```

8.5. GRUB Configuration

GRUB's configuration file is located in `/boot/grub/grub.conf` or `/boot/grub/menu.lst`, depending on the distribution. This file contains a list of the operating systems GRUB can boot on your computer, this includes Windows, all sorts of Linux distributions, BSDs, and many other systems. GRUB is responsible for loading the operating system's kernel and pass configuration parameters to it. When GRUB doesn't know how to load a system (for example, Windows), it *chainloads* the boot process, literally gives up all control and asks the system's bootloader to start the system.

Let's take a look at GRUB's configuration file first:

```
default 0
timeout 5
splashimage=(hd0,1)/boot/grub/splash.xpm.gz

title GNU/Linux
root (hd0,1)
kernel /boot/vmlinuz root=/dev/sda2
initrd /boot/initrd.img

title Windows XP
rootnoverify (hd0,0)
```

```
chainloader +1
```

This is a minimal configuration file, some distributions have this file a bit more complicated, but don't worry, you'll recognize the differences.

GRUB numbers hard drive in a very different way than Linux does, it doesn't have distinct names for hard drives (e.g. hda, hdd, sda) it assigns a number for each hard drive depending on the order it finds it, numbering starts with 0 and up. Note that GRUB doesn't include any other media except hard drives in the numbering, so if you have a primary master hard drive, a primary slave CD-ROM and a secondary master hard drive, GRUB will only recognize two hard drives and number them hd0 and hd1, regardless of the CD-ROM; again, this doesn't matter whether it's a SCSI hard drive, an IDE hard drive or a SATA hard drive.

In the above configuration file, you see that GRUB can boot one of two systems, namely Linux and Windows, and the *default* directive tells it which one to boot if you don't interrupt it. GRUB will wait for *timeout* seconds until it starts the default boot. The next line isn't required, it just adds a juicy splash image when GRUB displays the list of systems.

```
splashimage=(hd0,1)/boot/grub/splash.xpm.gz
```

In my setup, I have Windows on the first partition, Linux on the second, this explains the *hd(0,1)* part which tells GRUB to grab it's splash image from */boot/grub/splash.xpm.gz* from the first hard drive on the second partition, which in this case is */dev/sda2*, but it doesn't have to be.

The next few lines describe a system entry, specifically a Linux boot entry. The first is obvious, it's that title that appears in the GRUB boot menu, GNU/Linux.

GRUB's root file system shouldn't be confused with Linux's root file system, it has nothing to do with it. GRUB loads the rest of its stages from the root file system, in my case, its root is (hd0,1), in which it looks for */boot* and then it's stage files.

After defining the GRUB root file system, we need to tell it which kernel to load:

```
kernel /boot/vmlinuz root=/dev/sda2
```

This line tells GRUB to use */boot/vmlinuz* as a kernel, it also tells GRUB to pass the *root* parameter to the kernel. In a Linux world, when you update your

hardware, all you need is a new kernel, you don't have to go through the process of updating device drivers, waiting for "Found New Hardware" or restarting countlessly, just use a new kernel. Of course, you're not going to bring a new kernel every time you want a certain feature, so you can pass it certain options to let it know what to do. In the above line, the only option passed to the kernel is where to find its root, that is, where to find its / partition, in my case it's `/dev/sda2`.

If you take a look at Fedora's `grub.conf`, you'll see a difference:

```
kernel /boot/vmlinuz-2.6.11-1 ro root=/dev/sda2 rhgb quiet
```

Notice the `rhgb` option, this came from RedHat Linux (Fedora's predecessor) and stands for RedHat Graphical Boot. It's the pretty screen you see when you boot Fedora or RedHat. `rhgb` can cause a few problems with drivers like nVidia's, so if you face any problems just remove it from the kernel options.

The line that follows the kernel is option, you might and might not have it, it specifies which `inird` image to load. We'll get to it shortly.

```
initrd /boot/initrd.img
```

Now let's take a look at the next system entry. It tells GRUB how to boot Windows and it's fairly simple since there's no kernel to load, no options to remember.

```
title Windows XP
rootnoverify (hd0,0)
chainloader +1
```

When GRUB tries to boot a system it needs to load its stages from `root`, each time it does that it verifies that the root does contain stage files, but what happens when there are no stage files at all, like in the case of Windows? That's what `rootnoverify` is for, it tells GRUB not to look for stages since it's going to chainload the boot process anyway.

The next line activates the chainloader.

GRUB has a *fallback* mechanism, in which if a system doesn't boot, it'll try the next one until it finds one that does. You can active it using the *fallback* directive:

```
default 0
timeout 5
fallback 1
```

8.6. Initial Ramdisk

Or as Linux calls it, *initrd*.

The Linux kernel can be either a modular or a monolithic kernel. Most Linux distributions use a modular kernel, in which the kernel is just a small core that detects hardware and loads appropriate modules to handle that hardware, think of kernel modules as device drivers on Windows. Monolithic kernels on the other hand have everything they need built-in, there are no modules involved. Each of these types has its advantages:

Modular Kernels:

- They're flexible and mostly hardware independent.
- They dynamically load modules as they're needed instead of having them built into the kernel
- Take slightly more time to load into memory.
- Rarely need recompilation.
- Used mostly in desktop distributions.

Monolithic Kernels:

- They're usually smaller in size than modular kernels.
- Take slightly less time to load.
- Contain only the absolute necessities, no extra modules.
- Require recompilation whenever new hardware is added.
- Used mostly in server environments.

The problem with modular kernels is that they need an *initrd* image in order to

boot properly. The reason for this is to solve a loop. You see, Linux needs to load its modules from the file system, but if file system support *is* a module, how can Linux work around this? The idea is to load the absolute minimum configuration that enables Linux to mount the file systems and then load the rest of its modules.

`initrd` is a compressed file that contains a minimal root system, GRUB loads the `initrd` image after loading the kernel and runs a startup file called `/linuxrc` which is also contained in `initrd`. This file loads the appropriate Linux modules, just about enough to recognize hard drives and mount the file systems, then it is removed from memory and Linux executes `init` then continues booting.

You can mount the `initrd` image in order to view its contents, assuming the image file name is `initrd.img`, which might be different depending on your version or distributions:

```
# cp /boot/initrd.img /tmp/initrd.img.gz
# gunzip /tmp/initrd.img.gz
# mkdir /mnt/initrd
# mount /tmp/initrd.img /mnt/initrd -o loop
```

The lines above make a copy of the `initrd` image file, uncompresses it and then mounts it to `/mnt/initrd`.

9. The Linux Shell

The kernel is the heart of the system, it's what manages running processes, access to hardware, reading and writing to file systems. The next most important thing in Linux is the *shell*. There are many definitions for what a shell is, but the closest analogy would probably be, a "talking" program, a program whose only purpose in life is to do what it's told. The shell is one of the means of communication with the system.

Essentially, the shell is a programming language, there's a certain protocol that you should follow to communicate with the shell, a syntax, a language if you will. This language dictates what the shell understands, how it understands, and what it can do with it.

Many shells exist for Linux, each having its advantages and disadvantages:

- **sh**: also known as, Bourne Shell, after its inventor. It's the original UNIX shell, fairly minimalistic and it's been superseded by `bash`.

- **bash**: also known as Bourne *Again* Shell. Developed by GNU, it's currently the standard Linux shell, and it has also replaced `sh` in many other operating systems.
- **csh**: The C Shell, its syntax resembles the C programming language, one of programmer's favorites.
- **tcsh**: Turbo C shell, a superset of `csh`, it's easier to use and more user-friendly compared to `csh`.
- **ksh**: Korn Shell, one of the very famous Linux shells, it's a superset of `bsh`, only its configuration can be troublesome.
- **ash**: A `bsh`-compatible shell, it's known for its small memory footprint and mostly used in rescue disks.

There are tons of other shells for Linux, but the most famous of them would be `bash` for its easiness of use and advanced features.

9.1. Environment Variables

When you run a command in `bash`, it doesn't automatically know how to run it, it looks for it in a list of directories and runs the first one it finds. In reality, `bash` searches a variable called `PATH`, which is a list of directory names separated by a colon. To see what the variable contains, use `echo`:

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

So whenever you type a command, `bash` looks in `/sbin`, `/bin`, `/usr/sbin`, and `/usr/bin` in order, if a command exists in more than of these directories, `bash` runs the first it finds. You can see look in the same directories for a command using `whereis`:

```
$ whereis ls
ls: /bin/ls /usr/bin/ls /usr/share/man/man1/ls.1.gz
/usr/share/man/man1p/ls.1p.gz
```

Of course, when you run `ls` you don't run both copies, you run one of them, the first `bash` finds:

```
$ which ls
/bin/ls
```

If you need to edit the PATH, you can use *export*. Let's say you have a few scripts in your home directory which you'd like to be able to run without specifying the full path:

```
$ cleantmp
bash: cleantmp: command not found
$ export PATH=$PATH:/home/jake/bin/
$ echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin:/home/jake/bin/
$ cleantmp
Cleaning....
```

You can view all your environment variables (that can be a lot) using the *env* command:

```
$ env
MANPATH=/usr/local/share/man:/usr/share/man:/usr/share/binutils-
data/i686-pc-linux-gnu/2.15.92.0.2/man:/usr/share/gcc-data/i686-pc-
linux-gnu/3.4.4/man::/opt/blackdown-jdk-1.4.2.02/man:/usr/qt/3/doc/man
HOSTNAME=tux
TERM=xterm
SHELL=/bin/bash
USER=rami
PWD=/home/rami
EDITOR=/bin/nano
HOME=/home/rami
OLDPWD=/home/rami
```

These are just a few variables, you might see many more, only these are the most common ones.

Remember *pushd* and *popd*, well these two commands use two environment variables in order to work, namely `$PWD` and `$OLDPWD`, which contain your current working directory, and where you were before.

Changes to environment variables aren't permanent, you need to tell bash to export a variable every time it runs

9.2. Bash Configuration

Global bash configuration is located in `/etc/bash/`, you'll find at least two files *bashrc* and *bash_logout*. *bashrc* is read when *bash* first runs, hence *rc* (runtime configuration), *bash_logout* is read when *bash* exits.

In most distributions *bashrc* sets the terminal prompt, check the window size before bash runs and set bash's width accordingly, some of them do different things depending on the terminal. You can edit this file to add your path to it, but you really shouldn't, this file belongs to the system. You should edit your local configuration file.

bash searches for a hidden configuration file in the user's home directory, if it finds a `~/.bashrc` it reads it and uses it the same it uses `/etc/bash/bashrc`. You should edit this file to add anything you want to run once you login.

bash also stores another file in your home directory `~/.bash_history` which contains a list of the commands you run previously.

You can change your terminal prompt in this file, just change the variable `PS1` to something else:

```
$ echo PS1
$ export PS1=[\u@\h \W]\$
[root@plethora /etc/bash/]$
```

9.3. Aliases

Sometimes you want to have shortcuts to common commands, like *pgrep* is a shortcut for passing *ps*'s output to *grep*. One solution is to create scripts with names you chose, the other is to use *aliases*.

Not only can you use aliases to create shortcuts for other commands, you can even redefine how certain commands work. Take a look at the default aliases of Fedora:

```
$ alias
alias cp='cp -i'
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias mv='mv -i'
alias rm='rm -i'
```

Fedora, for example, redefines how *cp*, *mv* and *rm* work, it makes them work in interactive mode by default, preventing accidental overwrites. It also defines a few *ls* aliases like *ll* instead of *ls -l*.

In case you don't like the default aliases, you can redefine them, or even define new ones:

```
$ ls
Desktop Work Stuff
$ alias ls="ls -l"
$ ls
total 24
drwxr-xr-x  2 rami users 4096 Jun 25 00:18 Desktop
drwxr-xr-x  2 rami users 4096 Jun 25 00:18 Stuff
drwxr-xr-x  2 rami users 4096 Jun 25 00:18 Work
```

Removing aliases is done with *unalias*:

```
$ unalias ls
$ ls
Desktop Work Stuff
```

9.4. Useful Features

bash's history can very useful feature for repetitive commands, using the Up and Down arrows to navigate through the history, but you can use a few shortcuts to get your way around:

- **!*x*** executes the latest command in the history list starting with an '*x*'
- **!*2*** runs command number 2 from the **history** output
- **!*-2*** runs the command before last
- **!!** runs the last command

You should also know about a few key bindings that *bash* uses to navigate:

- **Ctrl+P** Previous line (same as Up arrow)
- **Ctrl+n** Next line (same as Down arrow)
- **Ctrl+b** Go back one character (same as Left arrow)
- **Ctrl+f** Go forward one character (same as Right arrow)
- **Ctrl+a** Go to the beginning of the line (same as End button)
- **Ctrl+e** Go back to the end of the line (same as Home button)

9.5. Scripting

Remember the good ol' days of DOS batch files? In Linux there's a similar concept except it's not just as primitive, they're called *shell scripts*.

Shell scripts are essential text files with a list of commands, let's say you need to run five or six commands in order to back your server, you can open a text file, type the commands you want to run, save the file and then use it just as you would with any other command; don't forget to set executable permissions on.

Like we discussed before, *bash* isn't just a shell, it's really a programming language, and a very sophisticated one too.

9.5.1. #!

Since Linux has many shells, it needs to figure out which one to use in order to interpret a shell script. Whenever you create a shell script, you should start it with a shebang, a `#!`, and follow it by the full path to the shell you intend to use:

```
#!/bin/bash
```

Many shell scripts, however, use `/bin/sh` as the default shell, and they simply assume that `bash` is the default shell, since many distributions have `/bin/sh` linked to `/bin/bash`.

This also works for scripts on other programming languages, like Perl, Ruby and Python:

```
#!/usr/bin/perl -w
#!/usr/bin/python
#!/bin/env ruby
```

9.5.2. Quotes

Quotes have a special meaning in `bash`, like we used to enclose directories with spaces in their names inside double-quotes, we can enclose other commands in *backticks* in order to use their output as a string:

```
$ pidof bash
12483
$ kill 12483; # is the same as
$ kill `pidof bash`
```

9.5.3. Loops

Sometimes you need to apply the same commands for every file a directory, or kill every certain process, in this case you should use loops:

```
for i in `ls`
```

```
do
cp $i /backup/$i.bak
chown backup:users /backup/$i.bak
done
```

The code above tells *bash* to read *ls*'s output and assign the variable *\$i* which you use inside the loop.

Sometimes you need a loop that just won't end, in this case use *while* or *until*.

```
while `sleep 2`
do
ping -n1 yahoo.com
done
```

This loop causes *bash* to ping yahoo.com every two seconds, it'll keep doing that until *sleep* exits with an error, which of course it won't.

9.5.4. Conditions

If you want to allow your backup script to only be run by a certain user, you can add an *if* check to make sure:

```
if [ $USER="root" ]; then
# continue backup....
fi
```

You can also use an *if* check to see if certain values are less than, greater than or equal to other values:

```
if [ $year -lt 1901 -o $year -gt 2099 ]; then
echo "Invalid year"
fi
```

-lt stands for less than, *-gt* stands for greater than and *-o* stands for OR.

9.6. Sourcing

If you have a bash script without executable permissions, you can always *source* it in order for it to run, either by using *sh* or by using a dot:

```
$ sh script.sh
$ . script.sh
```

10. Text Processing

Linux is filled with commands to manipulate files, this comes from its historic backgrounds where UNIX used plain text as a standard format for communication and needed easy methods of filtering and text manipulation.

The simplest of all commands is *cat*. As you already know, *cat* is a tool that repeats whatever it gets on its standard input to the standard output. It can read file:

```
$ cat /etc/passwd
```

It can also number the lines as it sends them, which is also what *nl* does:

```
$ cat /etc/resolv.conf -n
 1 nameserver 82.137.200.83
 2 nameserver 192.168.2.14
 3 nameserver 192.168.2.9
 4 search localhost.localdomain
```

The opposite of *cat* is *tac* (but of course), it does exactly the same as *cat* only it reads backwards, from the last line on.

10.1. tail and head

These two commands are mostly used when viewing log files, especially long ones, when *cat* would take a while to view the whole file.

tail and *head* are very similar, *tail* views the last 10 lines of a file, *head* views the

first 10.

Of course, they're not limited to this, you can always tell any of the two to show more or less lines:

```
$ tail -n 20 /var/log/messages
ReiserFS: sda2: Using r5 hash to sort names
VFS: Mounted root (reiserfs filesystem) readonly.
Freeing unused kernel memory: 208k freed
etc...
```

You can tell *tail* to watch a file and keep you updated, so you can see errors (if any) as they appear:

```
$ tail -f /var/log/messages
```

10.2. Cutting

cut extracts a range of characters or fields from a given input. Fields are pieces of text separated by a delimiter, like */etc/passwd* entries where each piece of information about a user is a field.

You can tell *cut* to display a range using the *-c* option:

```
$ echo abcdefghijk | cut -c3-8
cdefgh
```

You can also cut more than one range:

```
$ echo abcdefghijk | cut -c2,4-6,8-
bdefhijk
```

The last part of the option tells *cut* to extract the range from the 8th character till the end of the line.

The same concept applies to fields instead of characters when you use a

delimiter, use the *-f* option instead of *-c*, and *-d* to tell *cut* what delimiter to use:

```
$ cut -d: -f1,2,3 /etc/passwd | head -n 5
root:x:0
bin:x:1
daemon:x:2
adm:x:3
lp:x:4
```

10.3. A Few More...

To replace TABs with spaces in a text file, use the *expand* command. You can also reverse the effect with *unexpand*.

If you need to join a few files together, use the *paste* command:

```
$ paste textfile1 textfile2 etc...
```

When you're done writing to a file using *cat* you should press Ctrl+D to append the EOF character. Alternatively you can use a marker instead that when *cat* reaches, it ends the file:

```
$ cat > /tmp/text << END
line 1
line 2
END
```

10.4. Regular Expressions

Some text manipulation tools in Linux require you to know about regular expressions.

Regular expressions are a way to describe patterns, matching patterns, think of them as fancy wild cards that aren't limited to *** and *?*. They're used to search for text, match and replace using various Linux tools.

Characters	Search Match
x (or any character)	Strings containing an 'x'
\<KEY	Words beginning with 'KEY'
WORD\>	Words ending with 'WORD'
^	Beginning of a line
\$	End of a line
[Range]	Range of ASCII characters enclosed
[^c]	Not the character 'c'
\[Interpret character '[' literally
"cat*"	Strings containing 'ca' or 'cat' plus anything
"."	Match any single character

10.5. *grep*

grep is a program that searches text for patterns, it can search from text coming from standard input, a file, or a list of files.

```
$ grep title /boot/grub/grub.conf
title GNU/Linux (2.6.11-r8)
title Windows XP
```

You use an alternating pattern to search for text OR another text, note that the *-i* switch makes *grep* use case-insensitive search. *egrep* is an advanced version of *grep*, you can use it the same way you'd use *grep* but it supports more advanced regular expressions:

```
$ egrep -i "linux|windows" /boot/grub/grub.conf
title GNU/Linux (2.6.11-r8)
title Windows XP
```

You can use *grep* to exclude patterns rather than include them, use the *-v* switch, that is, search for anything that does **not** match the pattern. Let's view

grub.conf excluding blank lines:

```
$ grep -v "^$" /boot/grub/grub.conf
default 1
timeout 5
splashimage=(hd0,1)/boot/grub/splash.xpm.gz
title GNU/Linux (2.6.11-r8)
root (hd0,1)
kernel /boot/kernel-2.6.11-gentoo-r8 root=/dev/sda2
initrd /boot/splash
title Windows XP
rootnoverify (hd0,0)
chainloader +1
```

10.6. vi

One of the most famous editors around. *vi* was used in the oldest UNIX'es and it continued its way to Linux, although most current Linux distributions use another editor based on *vi* called *vim*, *VI iMproved*, but they're very similar so you don't have to worry.

As opposed to many text editors, *vi* has modes, a command mode where you can navigate a text file, and an insert mode, where you actually write the text. *vi* starts in command mode by default, press **i** to go to insert mode and write some text, **Esc** key brings you back to command mode.

10.6.1. Navigating

Since *vi* doesn't use the mouse it uses shortcuts to navigate the text, you can use these shortcuts in command mode:

- **b** and **e**: go to the **beginning** or the **end** of the word
- **(** and **)**: go the beginning or the end of the current sentence
- **{** and **}**: go the beginning or the end of the current paragraph.

These commands are run by just pressing the letters on the keyboard, other commands (like write and quit) can only be entered after you press the colon.

10.6.2. Inserting Text

You can go to insert mode by using one of the following commands:

- **a**: Append text with cursor on the last letter of the line
- **A**: Append text with cursor after last letter at the end of the line
- **i**: Insert text at the current position
- **o**: Insert text on a new line below
- **O**: Insert text on a new line above
- **s**: Delete the current letter and insert
- **S**: Delete the current line and insert

10.6.3. Deleting Text

Deleting text is done by either pressing the **Del** or **Backspace** button in insert mode, or using shortcuts in command mode.

Generally, deleting shortcuts are preceded with a **d** followed by what you want to delete, so if you want to delete a word you'd use **dw**, if it's a line then use **dd**, you can delete to the end of the line by pressing **d\$**, or delete until the end of the paragraph by using **d}**.

10.6.4. Copying and Pasting

In *vi* copying is called *yanking*. Yanking shortcuts are entered in command mode and cannot be used in insert mode, these shortcuts are similar to delete shortcuts in the sense that they too are followed by what you want to yank.

Yanking a word for instance is done with **yw**, if it's a line then use **yy**, you can also yank more than one line by using **3yy** where **3** is the number of lines you want to yank.

Use **p** to paste the last yanked text.

10.6.5. Searching

Activate the search mode by pressing **/** in command mode and enter the text you're searching for then pressing enter. If you want to search for the same text again it's enough to press **/** then Enter, *vi* will look for the last text you searched for.

You can also search and replace text with:

```
:% s/SEARCH/REPLACE
```

10.6.6. Saving and Quitting

Once you're done working with the text file you should save it, you could do that by using the write command, press the colon and enter **:w**.

Quitting is done with the **:q** command, but *vi* warns you if you've done any editing, in case you're sure you want to quit without saving use the **:q!** command.

You can also **:qw** to quit and save in one go rather than **:w** then **:q**.

11. Networking in Linux

Linux supports various networking technologies and protocols such as Ethernet (i.e. LAN), PPP, SLIP, Token Ring and many others, the most common being Ethernet. Linux also supports many network protocols like TCP/IP, IPX/SPX, NetBIOS and others. I'm only going to discuss TCP/IP configuration since it's the most common and used one.

11.1. IP Addresses

Internet Protocol Addresses are composed of four bytes. The convention is to write addresses in what is called 'dotted decimal notation'. In this form each byte is converted to a decimal number (0-255) dropping any leading zero's unless the number is zero and written with each byte separated by a '.' character. By convention each interface of a host or router has an IP address. It is legal for the same IP address to be used on each interface of a single machine in some circumstances but usually each interface will have its own address.

Internet Protocol Networks are contiguous sequences of IP addresses. All addresses within a network have a number of digits within the address in common. The portion of the address that is common amongst all addresses within the network is called the 'network portion' of the address. The remaining digits are called the 'host portion'. The number of bits that are shared by all addresses within a network is called the netmask and it is role of the netmask to determine which addresses belong to the network it is applied to and which don't. For example, consider the following:

```
-----
```

```

Host Address      192.168.110.23
Network Mask     255.255.255.0
Network Portion  192.168.110.
Host portion          .23
-----
Network Address  192.168.110.0
Broadcast Address 192.168.110.255
-----

```

11.2. Network Interfaces

Each network interface in Linux is configured to have an IP, in addition to that you could configure the same interface with additional IP addresses called *virtual* addresses.

The mother of all network configuration commands is *ifconfig*, it exists on every Linux distribution, but different distributions use different start-up scripts, and different start-up scripts read configuration files from different places; interfaces are named eth0, eth1, etc. to point that they're *ethernet* interfaces:

- Fedora and RedHat: /etc/sysconfig/network-scripts/ifcfg-eth0
- Debian: /etc/init.d/network
- Gentoo: /etc/conf.d/net

IP address assignment can be fixed or dynamic. When it's fixed, it's the system administrator's role to maintain the list of available IP addresses and make sure they don't clash with each other. In other cases a DHCP server can be used to dynamically assign IP addresses for workstations.

Assigning an IP to an interface can be done using the *ifconfig* command:

```

$ ifconfig eth0 192.168.1.25 netmask 255.255.255.0
$ ifconfig eth0 up

```

The same command is used to display the current network configuration:

```

$ ifconfig

```

```
eth0      Link encap:Ethernet  HWaddr 00:30:1B:B4:AE:30
          inet addr:172.25.1.232  Bcast:172.25.1.255
Mask:255.255.255.0
          inet6 addr: fe80::230:1bff:feb4:ae30/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:188877 errors:0 dropped:0 overruns:0 frame:0
          TX packets:99006 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:280748456 (267.7 Mb)  TX bytes:6607233 (6.3 Mb)
          Interrupt:10 Base address:0x6000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:62 errors:0 dropped:0 overruns:0 frame:0
          TX packets:62 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3100 (3.0 Kb)  TX bytes:3100 (3.0 Kb)
```

Of course, you don't want to do that manually every time you boot Linux, you should use your distribution's configuration files. Let's take a look at Fedora's:

```
$ cat /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE=eth0
BOOTPROTO=none
HWADDR=00:50:BF:B3:67:1B
IPADDR=172.25.1.233
NETMASK=255.255.255.0
ONBOOT=yes
TYPE=Ethernet
GATEWAY=172.25.1.254
```

There are a few directives that define what networks this interface connects to. The IPADDR and NETMASK defines a static IP address for this network, ONBOOT tells Linux to actually start this interface when it boots rather than waiting for manual start-up. The default gateway is usually the router's IP and it's supposed to be your way out to the internet.

11.3. Name servers

In order to connect to the Internet, you need to know what your name server is. A nameserver (i.e. DNS server) is what tells you IP addresses of sites you want to connect to. Linux cannot use a name to connect to different computers over the network, it needs to find out its IP address, more specifically, *resolve* it's name to an IP address; name servers take care of this. Whenever you connect to a site, say google.com, Linux first contacts the name server, asks it for google.com's IP address and then connects to it.

Name server configuration is located in almost all distributions in */etc/resolv.conf*:

```
nameserver 82.137.200.83
search localdomain
```

You don't have to use a name server in every case though, you could use a static file in which you store a list of computer names and their IP addresses:

```
$ cat /etc/hosts
127.0.0.1      localhost
172.25.1.232  rami
```

But in order to resolve an IP address, Linux needs to know who to ask first, the DNS server or the hosts file, this is determined in *hosts.conf*.

```
$ cat /etc/hosts
order hosts, bind
```

11.4. Network Tools

There are a few tools that can help you with managing your network using Linux, *ping* is one tool that lets you know if a workstation is up or not:

```
$ ping 172.25.1.232
PING 172.25.1.232 (172.25.1.232) 56(84) bytes of data.
64 bytes from 172.25.1.232: icmp_seq=1 ttl=64 time=0.042 ms
64 bytes from 172.25.1.232: icmp_seq=2 ttl=64 time=0.038 ms
64 bytes from 172.25.1.232: icmp_seq=3 ttl=64 time=0.032 ms
64 bytes from 172.25.1.232: icmp_seq=4 ttl=64 time=0.042 ms
--- 172.25.1.232 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2998ms
rtt min/avg/max/mdev = 0.032/0.038/0.042/0.007 ms
```

as opposed to:

```
$ ping 172.25.1.231
PING 172.25.1.231 (172.25.1.231) 56(84) bytes of data.
From 172.25.1.232 icmp_seq=2 Destination Host Unreachable
From 172.25.1.232 icmp_seq=3 Destination Host Unreachable
From 172.25.1.232 icmp_seq=4 Destination Host Unreachable
From 172.25.1.232 icmp_seq=6 Destination Host Unreachable
--- 172.25.1.231 ping statistics ---
8 packets transmitted, 0 received, +6 errors, 100% packet loss, time
6999ms
, pipe 3
```

netstat is a querying utility that can give you a tremendous amount of information, for instance, if you want to see the current connections to and from your machine:

```
$ netstat -n
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address      State
tcp    0      0 192.168.1.10:139  192.168.1.153:1992  ESTABLISHED
tcp    0      0 192.168.1.10:22   192.168.1.138:1114  ESTABLISHED
```

```
tcp 0 0 192.168.1.10:80 192.168.1.71:18858 TIME_WAIT
```

To see some information on your routing table, *netstat* is again the utility that displays that:

```
$ netstat -r
Kernel IP routing table
Destination Gateway Genmask Flags MSS Window irtt Iface
172.25.1.0 * 255.255.255.0 U 0 0 0 eth0
loopback localhost 255.0.0.0 UG 0 0 0 lo
default 172.25.1.254 0.0.0.0 UG 0 0 0 eth0
```

12. Gooney!

Also known as the GUI, the Graphical User Interface.

The designers of UNIX aimed for (sometimes extreme) flexibility, so they designed UNIX as separate components built on top of each other, think building blocks. Back at those days, there way no user interface, partially because there were no graphic cards to handle many colors, and partially because computers weren't as wide-spread so they didn't have to be easy to use.

When the time of graphics came, there had to be a way of integrating graphics into the good ol' UNIX without too much hassle, and still keeping it as flexible as possible; so came the X server.

As absurd as it might sound, UNIX graphics is implemented using a client/server model. In the late 1980's, developers at MIT created the X Window System which was the foundation of graphics in UNIX.

The idea behind X was to separate who's responsible for the display from who's responsible for *asking* for a display. This roughly maps to a client/server¹ model in which the X window server listens for connections and is able to generate graphics, and an X window client connects to the server and tells it what to do.

Like in many applications in UNIX, X isn't the only window system, Sun's workstations came with Sunview and OpenWindows window managed, DEC developed a version of X called DECWindows; all of them were compatible with X, so you could run X clients under other X-compatible servers.

1 X has the client/server concept flipped out! But we'll get to that later.

Originally, X was developed to run three programs as a demonstration, *xclock*, *xterm* and *xload*. Later on, the idea of a *window manager* was added. X doesn't know anything about windows or how to manage them, there's just no such thing in X, if you run an application under X you won't see the maximize/minimize or close button, you won't be able to resize it or move it around, X is only a server that draws graphics on your screen.

Window managers come to help. A window manager is an application that runs on top of an X server and controls how applications are displayed, they wrap around them and add borders and a few buttons, they also control which windows are raised or lowered. You're probably used to raising a window by left-clicking on it, maximizing it by double-clicking its title bar. In reality, these aren't the system's choices, nor they are X's choices, they're window managers' choices. Some window managers give you the ability to change that, they can be configured to raise windows by double-middle-clicking on the lower left corner; not that you'd want to do that.

On top of a window manager comes a DE, a Desktop Environment. The most famous of the two being KDE¹ and GNOME². The role of desktop environments is to integrate a suite of applications and generally make it easy to use a computer. When you start a graphical interface in Linux you'll see a taskbar, a few applications that share the same *look and feel*, an application menu, icons on the desktop, a desktop, and a few other extras, that's what a DE is.

Desktop environments share the same look and feel, they do that by utilizing programming libraries (called *widget toolkits*) who are in turn responsible for calling X and telling it to draw. These libraries contain code for drawing buttons, menus, trees, icons, and almost everything else, they can also be used to develop applications for one desktop or the other. KDE uses the Qt widget toolkit, whereas GNOME uses the GTK widget toolkit.

Desktop environments might utilize session managers (also known as display managers). A session, to put it simply, is the collection of programs that you're running. If you want to run programs on start-up (when you're not using a DE) just put their names in *~/.xsession*.

After you login to your terminal session and if you're not in run level 5, you can start your X window server by running *startx*. *startx* is a script that reads X configuration, reads *.xinitrc* in your home directory and then it reads *.xsession* and executes all the programs you specified in it. Since many Linux gurus are simply lazy, they decided that managing *.xsession* is too much work so they came up with XDM, the X Display Manager. XDM is a graphical program that runs X and displays a login prompt, similar in a way to how you login to Windows XP, except not as good-looking. Desktop environments come with their own managers, KDM the KDE Display Manager, and GDM, GNOME's Display Manager. These display

1 KDE: K Desktop Environment.

2 GNOME: GNU Object Model Environment

managers remember what programs were you running when you logged out of your session and they'll rerun them when you login back again.

12.1. Configuration

X has a lot of configuration files but you don't have to worry, the most important files are just a few.

Linux distributions used to use a version of the X server called Xfree86, but due to licensing issues this started to change and distributions are (or already have) switching to X.org. The configuration file's syntax is still the same, but the file names might change. Xfree86's configuration file is located in `/etc/X11/XF86Config`, X.org's is in `/etc/X11/xorg.conf`.

The configuration file is split into sections, each describing a piece of hardware like the keyboard, the mouse, the display card, the monitor, etc.

```
$ cat /etc/X11/xorg.conf
Section "ServerLayout"
    Identifier      "X.org Configured"
    Screen          0  "Screen0"  0  0
    InputDevice     "Mouse0"  "CorePointer"
    InputDevice     "Keyboard0" "CoreKeyboard"
EndSection

Section "Files"
    RgbPath          "/usr/lib/X11/rgb"
    ModulePath       "/usr/lib/modules"
    FontPath         "/usr/share/fonts/misc/"
    FontPath         "/usr/share/fonts/TTF/"
    FontPath         "/usr/share/fonts/Type1/"
    FontPath         "/usr/share/fonts/CID/"
    FontPath         "/usr/share/fonts/75dpi/"
    FontPath         "/usr/share/fonts/100dpi/"
    FontPath         "/usr/share/fonts/TTF"
    FontPath         "/usr/share/fonts/corefonts"
    FontPath         "/usr/share/fonts/sharefonts"
```

```
    FontPath      "/usr/share/fonts/freefont"
    FontPath      "/usr/share/fonts/artwiz"
    FontPath      "/usr/share/fonts/ttf-bitstram-vera"
    FontPath      "/usr/share/fonts/unifont"
EndSection

Section "Module"
    Load  "glx"
    Load  "record"
    Load  "extmod"
    Load  "dbe"
    Load  "dri"
    Load  "xtrap"
    Load  "freetype"
    Load  "type1"
EndSection

Section "InputDevice"
    Identifier  "Keyboard0"
    Driver      "kbd"
EndSection

Section "InputDevice"
    Identifier  "Mouse0"
    Driver      "mouse"
    Option      "Protocol" "IMPS/2"
    Option      "Device"  "/dev/psaux"
    Option      "ZAxisMapping" "4 5"
EndSection

Section "Monitor"
    Identifier  "Monitor0"
```

```

        VendorName  "Monitor Vendor"
        ModelName   "Monitor Model"
EndSection

Section "Device"
    Option          "NoLogo"                "True"
    Identifier      "Card0"
    Driver          "nvidia"
    VendorName     "nVidia Corporation"
    BoardName      "Unknown Board"
    BusID          "PCI:2:0:0"
EndSection

Section "Screen"
    Identifier      "Screen0"
    Device          "Card0"
    Monitor         "Monitor0"
    DefaultDepth   24
    SubSection     "Display"
        Viewport   0 0
        Depth      1
    EndSubSection
    SubSection     "Display"
        Viewport   0 0
        Depth      4
    EndSubSection
    SubSection     "Display"
        Viewport   0 0
        Depth      8
    EndSubSection
    SubSection     "Display"
        Viewport   0 0

```

```

        Depth      15
    EndSubSection
    SubSection "Display"
        Viewport   0 0
        Depth      16
    EndSubSection
    SubSection "Display"
        Viewport   0 0
        Modes      "1280x1024"
        Depth      24
    EndSubSection
EndSection

```

The ServerLayout section describes what devices X server should use when it starts, the Files section is mostly used to tell X where to look for font's. Note that X doesn't understand TrueType fonts by default, it needs an extra module called Freetype, the Modules section takes care of specifying what modules to load. Different modules do different things, for instance, a module called Xinerama let's X use dual-screen setups, the DRI module sometimes speeds up the display by allowing direct rendering.

There are two InputDevice sections, one describes the keyboard, the other describes the mouse. You might want to check the mouse device if X isn't working properly, in some distributions the mouse devices is located in */dev/input/mice* rather than */dev/psaux* or */dev/mouse*.

The next Monitor and Device sections describes the monitor and display card, it tells X to use the *nvidia* display driver and calls a driver option called NoLogo which omits displaying the nVidia logo when X starts up.

The next section is the most important one, the Screen section. The Screen section defines the resolutions that X can work in, it tells it how many colors to use and at what screen width and height.

The problem with this file is that it takes a lot of work to manually configure it, you should use automatic configuration instead, it doesn't *always* work, but you should give it a try:

```
# X -configure
```

```
Your xorg.conf file is /root/xorg.conf.new
```

```
To test the server, run 'X -config /root/xorg.conf.new'
```

Copyright © Rami Kayyali

www.ramikayyali.com