

Linux

mlowman

October 2007

1 Intro

This file should introduce you to the Linux system: how it works, how it's set up, and commands that you need to know. We'll be assuming that there's already a system set up, so you can follow along. That might make this all easier to follow.

2 GUIs

You are used to a GUI (Graphical User Interface). The GUI is an integral part of Windows and OS X; in fact, neither can run without one. And for Windows in particular, almost every administrative task is meant to be configured with a GUI interface. In some cases, a GUI is the only option.

On Linux, this is not the case.

As in all UNIXes, the underlying interface is the command line. A system called X provides those graphics you see on a Linux workstation's screen, but X is *optional*. Every single task has a CLI (command line interface). And why would a server waste valuable resources running a completely unneeded graphics system? It wouldn't. For this reason, you have installed your server boxes without X. We will install X later, when we explore the GUI, but for now I'll prove to you that we don't need it.

3 The tty

This is where you enter the system. There will be a simple prompt, perhaps the kernel version, date, and/or hostname, and a blinking cursor after the

word `login:` . Here you type your username, then enter. When you enter your password, no characters will appear. This is for security, not because of a broken keyboard. Type your password, then enter. Note: A computer may take a few centiseconds to display the password prompt. Don't type until you see it, or everyone will be able to see your password. That's rare, though.

3.1 Multiple ttys

A Linux system has more than one (gasp!). To switch ttys, press `Alt+F n` where n is the number of the tty. On a default system, you probably have six ttys running `login` (the program). If you're running X, that's usually placed on `tty7` (although multiple sessions would go on multiple ttys). And other things could go on later ttys, but aren't usually configured that way by default. ALSO: From X, the sequence is `CTRL+ALT+F n` . Just so you know.

4 The shell

There are many different UNIX shells. The most common Linux one is called `bash`, and it's the one all of you are currently using. It's an updated version of `sh`, the original. (`sh` was the Bourne shell, named for the creator. So `bash` is the Bourne-Again shell. :) Yeah, I know it's a bad joke.) Here we'll go over a small part of its featureset, but keep in mind that there are built-in commands for shell scripting like `if` and `do`. Those are for later.

4.1 Not bash

These actually have nothing at all to do with `bash`, but you should know them anyways. If a command gives a lot of output, you can halt it with the scroll lock key. This will freeze the currently executing process immediately, so you can actually read something. Also, you can activate `scrk` with `^S` and deactivate it with `^Q`. So if your computer suddenly stops responding at a tty, check that LED. Just so you know, anything you type will still go into the buffer with the screen locked: when you unlock it, all the text will appear. Also useful: if some output goes off the screen and you still want to read it, it's all okay. `SHIFT+PGUP/PGDOWN` scrolls through the "scrollback

buffer”, a list of all the lines that just went offscreen. There’s a certain number of lines that get saved (configurable in the kernel), then everything just moves up as more comes in. Use this when the screen is locked, too.

4.2 Basic

You type, and characters show up on the screen. The first token is the command, and the rest are arguments. You finish your command statement, and you press enter. The command executes. You scream in horror because a typo causes the command to erase all your data.

So, keeping in mind the virtue of careful typing, what else can bash do? Press up and down to scroll through the history of commands last typed. Pressing TAB will complete a command you’re typing (if it’s the first token) or a filename/directory (if it’s after that). And with special bash modules, it can complete just about anything. TAB-completion is most definitely your friend. And some keystrokes will help you do things quickly, like erase the previous word or move to the end of the line. But we won’t mention those.

4.3 Foreground, background, redirection, stuff

4.3.1 fg bg

If you just type a command, bash will wait until it finishes before letting you move on. The reason: what you type is attached to the stdin of the program so that it can interact with you. How useful. But what if the program is a server that you want to run all the time? You know it won’t ask for input, so that doesn’t matter. You can’t just run it and leave it on the tty. . . You’ve got to use that tty yourself. So we background it. At the end of the command, add an ampersand. It’ll run in the *background* (the other way was in the *foreground*). When it backgrounds itself, it’ll display its PID so you know how to kill it if need be.

4.3.2 redirs

A command may have output that shows up on your screen. What if you want to put that output in a file? Easily done. *command > filename* will put all of standard out into that file. Nothing shows on your screen. If the file’s already there, it’ll be erased: if you’d rather append, use *>>* instead. Don’t confuse these. The inverse is also possible. Commands that read information on stdin

(you typing), such as an equation to evaluate, can read their input from a file. Put the input into a file, and type `command < filename`. Combine the two? Quite possible. `bc < equationlist.txt > results.txt`. So what if you don't want to go through the intermediary of a file? I quite understand. Frankly, I never use `<` because I like the command line to be a chain data flows through from left to right: I use a command that prints the data in a file to stdout and *pipe* that to the next command. The pipe, by the way, is `|`. So if the program *dict* prints all the words that contain a string passed as an arg (say, *mem*) and *nl* will number the lines of anything it receives and tell me the total number of lines, `dict mem | nl` will tell me how many words were matched. Don't limit yourself to that simple a chain! If I only wanted to know the number of lines, I could add `|tail -n 1` on the end. Incidentally, you can do file redirection with more than just stdin and stdout. Ask if that sounds interesting.

4.3.3 Lines

Sometimes it's much easier to write two statements on one line, especially if you keep repeating that sequence (it's only one hist entry). Separate commands with semicolons. Another thing: sometimes you want to execute one statement only if another one succeeded or failed. Instead of writing an if statement, just use `&&` or `||`. The `and` will execute the second statement only if the first succeeds, and the `or` is the opposite. (Why? Do as little work as possible. If a statement requires both statements to be true to return true, and the first fails, then why execute the second? It doesn't matter, and just wastes time.) And one final thing: You can group statements with parentheses. Probably most useful with the `ands` and `ors`, since you can make a set of commands depend upon that first one.

4.4 A different kind of basic

4.4.1 Killing

CTRL-C is perhaps the most important keystroke ever. If you only know one, know this. It will kill your foreground process (you hope. the process might trap all signals, and you'll have to take the ugly route.). If you make a mistake in your command that you realize after you press enter (and stopping won't corrupt your data), kill it immediately with `^C`. If you're tired of waiting,

press `^C`. If you know the command is going to fail (*eventually*) and you want to stop it, `^C`. So versatile! This is also the accepted way of exiting many, many programs. Especially daemons, which don't take input that might tell them when to stop.

4.4.2 Suspending

Suppose you start a process in `fg`, and want to move it to the back. Or perhaps you want to open a quick `tty`. Simply suspend it with `^Z`, then do whatever. Type `fg` to bring the suspended process back to the fore, and `bg` to background it as if you had started it with `&`. You can have more than one suspended at once: each has a job id counting from one. Use that as an arg to `fg` or `bg` to distinguish; otherwise, they'll just act on the most recently suspended one.

4.4.3 Aliases

I hope you know the definition of the word: as in, say, an alternate identity. The command will link a string to a command, so executing the string is the same as executing the command. Useful for things like `alias la='ls -lA'`.

4.5 Environment variables

These can be specified different ways. With `bash`, it'll be a dollar prefix, then the variable name (traditionally all caps for globally used vars and lowercase for scripting). Example: `$USER`. The shell expands them to their real values if you use them: instead of using `env` to print them all, usually just use `echo $USER`. To set them for the current session (they'll only be used to expand on the command-line), just assign with `=` as follows: `USER=mlowman`. This is one of the few places where whitespace (or lack thereof) really counts. Normally, though, you want to assign with the command `export USER=mlowman`, so that any commands this shell runs (like `bash` scripts) will know the variable, too.

4.6 Paths, working dirs, command sequence

I hope you understand the concept of a filesystem layout with directories and files. Every shell has a working directory, where the shell "is". Pathnames

can either be absolute (specified from the root, like `C:\KQ5\SIERRA.COM`) or relative, specified from the current directory. Two special directories are `“.”`, the current directory, and `“..”`, the directory one up on the hierarchy.

4.6.1 Command search

When you type a command, bash looks several places to execute it. First, it looks in the list of aliases. Then it moves on to the built-in commands (which would vary depending upon the shell), and finally the PATH. PATH is a special environment variable, certainly the most important. It contains a colon-delimited list of directories. Starting at the first dir on the list, bash will look for something named just what you typed. And it will keep going until it finishes off the list. Look at your path now (type `echo $PATH`). Notice anything? There are no relative paths: everything is specified from the root. Reason being, with a relative pathname, directories anyone can write to are searched for executable files. So if root makes a typo, and executes `la` instead of `ls`, it just might run a script to give an attacker root. Also, that means no `“.”`. So anything in the current directory isn't specified either. To run a file in your current dir, the easiest thing to do is to specify it with one addition: as `./executable`. Means the same thing, but won't freak bash out.

5 Users and groups and permissions, oh my!

In the classic Linux system, everyone has a user account and a primary group. These two qualities are the basis for the entire filesystem security model, which is very simple and quite secure (unlike Windows). You log in as your user, so we know what your user is, and you can be part of multiple groups. (Basically, these are specified by three or four files with lists of groups and users and passwords. I want to make it clear that there are many variants on this classic theme. For instance, our lab uses Kerberos, Hesiod, and AFS, all of which have/are alternative methods of authentication, permission-setting, and user storage. But those are for another day.

But I digress. Every file and folder has a number of attributes. Some are unmodifiable, like last access time or size. Others are not, and it is these that determine security. A simple `ls` shows you all your files and dirs, but an `ls -l` (for long) will also list all of the many attributes these files have. Note first, on the far left, a series of 10 dashes mixed with letters. These are

the filetype and permissions. Next comes the number of links (look at the `ln` command if interested), then the owner, then the group owner. Last of all, filesize, modified date, and name. The filetype, by the way, is `d` for dir, dash for file, `b` for block special, `c` for character special, `p` for pipe special, and `s` for socket. Not too important now. Oh, and one last thing: any file or folder with a dot at the beginning of the name is invisible. Add a `-a` option to `ls` to view those, too.

5.1 Permissions

These define access privileges. `rw` stand for read, write, and execute. Read means can read the file or list the dir's contents (and the files in the dir's permissions). Write means you can edit the file or modify the dir. And execute makes a file executable, or makes it possible to `cd` to a dir. Important: write on a dir lets you modify its contents: even though you can't edit a file without `w` access to that file, you can delete it if the dir is `w`. (Or create new files).

Reading from left to right in groups of three, the groups are user, group, and other. The user permissions apply if you own the file. The group permissions apply if your group owns the file. And the other permissions apply if you're none of the above. Of course, the least restrictive permissions are the ones that control what you can do. See how to change permissions in the section on must-know commands.

5.2 Root

Root is God. (unoffensively)

Root is no normal user. Everyone should have a normal account, and only assume root access when needed (and drop it as soon as the task is complete). Root should never be a replacement for a normal account. Why? Root is God. The restrictions of the filesystem permissions are swept aside by root's all-powerful fist of doom. Normally you can't remove, say, all the binaries of your system. Like `bash`, or `ls`, or... You get it. The permissions don't allow that, but no petty permissions are going to stop root. Root can change the permissions of a file. Even if a file's permissions are ——— and owned by `omkay`, root can view this file, edit it, and change whatever else about it root pleases. Not to mention that root can change any user's password, or even replace `ls` with a nefarious trojan horse. In short: you can hurt your

system, and if you carelessly leave root access available (say, an open root shell) anyone can completely compromise your security.

6 Filesystem

Now for the structure. Everything is set up around common points. Everyone agrees where to put certain types of files. So you know where to look for them. First, / is the root. The top level. Nothing above it. Points of interest:

- /home is the traditional location for every user's homedir. What's in a homedir? User-specific confs and user files. It's your base of operations.
- /boot contains the boot information. The kernel, the bootloader configuration file, the additional files a bootloader might need...It's all here.
- /mnt is a usually mountpoint. Normally you make dirs under /mnt for all your mounts.
- /media is also a mountpoint, but I like /mnt more. /media is used by HAL, so your system'll still have it.
- /var is for variable data. /var/log has logfiles, /var is a traditional place to put webserver files, bind confs, and pid files.
- /root is root's homedir. Just like the /home dirs, but off of / so it can have different perms.
- /lib has libraries. Woo.
- /lost+found is for orphaned inodes, and may or may not be there dependent upon fs type. It'll be on the root dir of every mount if there is one.
- /proc is a special filesystem, which contains information about the system: disk stats, process calling strings, memory usages, bus populations... Everything.
- /sys is a kind of special kernel-user interface. I've never used it, and I'm pretty sure it's deprecated.

- `/bin` is all the binaries, the generic utilities like `ln`, `ls`, `bash`, etc.
- `/sbin` is all the system binaries, the generic utilities used for administration like `init` or `shutdown`.
- `/etc` is the configuration directory. Every configurable piece of software has at least one file or folder here, possibly both. Also, system configuration files are put here.
- `/dev` has all the device nodes. These are special files that act as direct interfaces to devices. Pretty low-level stuff, and very powerful.
- `/usr` has everything related to user-installed software.
- `/usr/bin` and `/usr/sbin` are like `/bin` and `/sbin`, but they are user-installed stuff.
- `/usr/games` has games.
- `/usr/local` is usually for packages compiled from source and their trap-pings, generally testing stuff.
- `/usr/src` is for source files. Most important, of course, being `/usr/src/linux` (the kernel sources). On a Debian system these won't be installed by default.
- `/usr/lib`, `/usr/include`, and `/usr/libexec` are libs or supporting files for the programs
- `/usr/man` has man pages.
- `/usr/share` has supporting files, like icon sets packaged with a program.
- `/usr/share/applications` has `.desktop` files (used for, say, menu generation).
- `/usr/share/doc` has documentation for almost every program you've installed.