

Introduction to UNIX I and II

Computer Science Macintosh & GNU/Linux Lab

www: <http://maclab.cs.uchicago.edu>

e-mail: tutor@cs.uchicago.edu

by Ivan Beschastnikh and Steven Alyari

modified Mon 6 Oct 2003

modified Mon 6 Jan 2009 by Damon Wang

The official location for this document is

http://www.maclab.cs.uchicago.edu/tutorials/unix_tutorial/unix.pdf

Contents

Contents	1
0.1 Purpose	1
0.2 Notation	2
0.3 Introduction	2
0.4 Administrative Details	3
0.5 Basic Commands	3
0.6 Shells	4
0.7 Redirection	5
0.8 Filesystem Hierarchy	5
0.9 Expansion	7
0.10 Permissions	7
0.11 File Transfers	11
0.12 Remote Access	12
0.13 Summary of Commands	12

0.1 Purpose

The purpose of this tutorial is to provide you with an overview of the services offered on UNIX-like systems and especially the CS department's GNU/Linux machines. Another goal

is to provide you with many references to documentation that we have found especially useful for learning.

The tutorial is titled UNIX I and II, simply because beginners seem more familiar with that name and because GNU/Linux is definitely a UNIX-like operating system. Technically, though, UNIX is a trademark of AT&T Bell Labs and an operating system designed in the 1970s which partly laid the conceptual framework for many other operating systems (OSes) such as GNU/Linux, FreeBSD, and even Mac OS X, to name but a few. Though, oddly, it is interesting to note that these OSes have developed so much since the 70s that to continue to use the term UNIX is to rob UNIX of its original meaning. And finally, before we begin, we add one more layer of confusion by telling you that GNU stands for GNU's Not UNIX.

0.2 Notation

- All command lines will be of `monospace` font. Literal portions will be in `upright monospace` font, whereas fields that should be customized will be in *italic monospace* font.
- The regular shell prompt will be denoted “\$”. It is printed by the shell to indicate readiness for a new line of input, *and should not be typed*.
- Commands will be of **bold** font
- C stands for the Ctrl key, M stands for Meta (which is Alt on GNU/Linux), hyphenated keys are pressed simultaneously, and sequences of keystrokes are separated by commas, so that “C-M-A,Enter” means press Ctrl, Alt, Shift, and letter-A all at once and then release, then press Enter.

Note that an alternative and also common notation exists in which capitalization does not count, no space means keys are pressed simultaneously, and Ctrl is \wedge , so that C-c is \wedge C.

- The pictures next to the section topics represent links to the O'Reilly online computer book publishing service, to which the University subscribes. When on the University network, you will have access to these books for intelligent, in-depth, and topical treatments of technical subjects.

0.3 Introduction

There are many different Unices (also spelled Unices and *NIX, using a wildcard which will be covered later). The MacLab offers a variety called Linux, and specifically a customized version of the Debian unstable distribution (“Sid”).

You may interact with the system either via either the graphical interface or the console (also called the “command-line interface (CLI),” the “terminal” or “text-mode”). This particular graphical interface (because Linux likes to offer a sometimes bewildering array of choices for everything) is GNOME by default, but you may choose another if you wish. The consoles are mapped to C-M-F1 through C-M-F6, and a graphical utility called a “terminal

emulator” (**xterm** is a good one) allows access to a CLI side-by-side with graphical applications such as Firefox. *A terminal emulator within the graphical interface is the recommended way to access a CLI.*

This guide is brief and incomplete. UNIX is best learned by experiment

0.4 Administrative Details

The CS Debian machines do not accept CNET IDs but rather require a CS account which may be requested by anyone, regardless of departmental affiliation, via

https://www.cs.uchicago.edu/info/services/account_request

If you need access to the departments CGI server, consult

http://cs.uchicago.edu/info/services/cgi_programming

0.5 Basic Commands

A very important and sometimes dangerous difference between Windows and Mac OS X, and Linux is that the latter assumes that you mean what you typed, and will execute it (usually without confirming or even mentioning the hazards) as soon as you hit Enter, even if this is not actually in your best interest. The canonical example is `rm -rf / path/to/directory`, which is almost certainly a typo. The user meant to delete a single directory somewhere on his system, but his finger slipped, an extra space was inserted, and instead he wiped his entire computer clean.

You will never do this, because fortunately the UNIX permissions system forbids it. However, a careless system administrator with root privileges can, and there are many other opportunities for you to do unfortunate things to your personal data. *Do not execute a command unless you know what it will do.*

This mini-course constitutes an exception to the above rule, because the privileges are set up so you cannot damage anything outside your own home directory and the files you create in `/tmp`, and you yet have no personal data in either location. So go ahead and play.

How do you find out what a command will do? Most of them have “online help” in the form of a flag that will print a brief summary of its purpose instead of actually taking action, but more comprehensive and verbose help is available through **man**, which is short for “manual”. Some distributions, including Debian, package part of their documentation via **info**, but **man** is universally available and so complete that some people do not consider a program seriously developed unless it has a manpage.

How do you make a reverse lookup if you know what you want to do, but can’t remember the name of the command? **apropos** and **man -k**.

The basic syntax of a command has three parts:

command [OPTIONS] [arguments].

command This is the name of the command and must be typed exactly. A very few programs will have multiple names and behave different depending on how you call them, but this is unusual.

options Also known as “flags”, these are parameters which modify the command’s action, and can be either long or short. Short flags are a single dash followed by a single letter, and multiple short flags may appear after a single dash, although this may lead to confusion when a flag takes an argument. Long flags are two dashes followed by a more descriptive name, possibly including addition dashes to separate words.

arguments “Args” for short, these provide additional information to the command. For example, **rm**, which removes files and directories, takes at least one file or directory name as an argument. Note that flags can take arguments too, sometimes. The safest way to pass an argument to a flag is to put it immediately afterward.

The syntax for **man** is **man** *[OPTIONS]* *command*. Go ahead and try it. A few other commands are listed below in no particular order. Be warned, some of them start rather formidable programs, some do things you might not want, and one is just silly. Remember, *man is your friend!*

ls	cd	mkdir	pwd	cp	ln	rm
which	less	cat	mv	find	whoami	lp
ps	kill	vim	emacs	sed	awk	grep
find	wget	perl	scp	ssh	w	yes
xargs	echo	fortune	fg	chmod	chown	passwd
which	touch					

Some ‘commands’ are simple keystroke combinations, like the keyboard shortcuts in Windows or Mac OS. Their usage may need some background you don’t yet have, so don’t be worried if the descriptions don’t make sense yet.

C-z Freezes a process. Useful for interrupting something for temporary access to the CLI, but then letting it run again. See **fg**.

C-c Sends a user-interrupt. Most programs will exit immediately. Useful if you don’t know what’s going on and want it to stop.

C-v Marks the next character as a literal. Useful for typing things that are usually displayed, like Enters.

0.6 Shells

The shell acts as a middleman between you and the system itself. Technically the Windows interface is also a shell, a graphical one, but when people speak of UNIX they use “shell” more specifically to mean a command-line interfaces. It provides functions like tab completion, expansions, a search path, job control, and very importantly, scripting. The default shell is **bash**, the Bourne Again Shell.

Knowing about shells, we can distinguish between commands handled within the shell and commands passed to other programs. You may have noticed that some of the commands

previously listed, such as **cd**, didn't have manpages. That's because **cd**, as well as **ls**, and **which**, are handled by the shell, whereas commands like **man** and **vim** are separate programs.

Before we see how the distinction between internal and external commands can matter, we need some commands to play with.

0.7 Redirection

The standard input stream is STDIN, the standard output stream is STDOUT, and the standard error stream is STDERR. By default, STDIN is your keyboard, and STDOUT and STDERR are both your screen, but you can change that. The pipe character — redirects one program's STDOUT to another's STDIN, and `>` and `>>` redirect STDOUT to files, the difference being that `>>` appends to the file's existing contents and `>` deletes them. The colloquial verb to describe these redirections is "piping."

A philosophical digression: Some programs, notably Emacs, Microsoft Office, and its open-source clone OpenOffice, like to have lots of features and do everything. They are anathema. The traditional UNIX style is to do one thing amazingly well, better than anything else, and let other programs do other things. An example is **grep**, which does nothing but search through text, but with such sophistication that its eponymous "regular expressions" are a course and a worship unto themselves. Therefore, it is necessary to bring multiple programs together to accomplish practical goals, and piping is the way to do it.

An important note: one may be tempted to think of piping as data moving through a series of programs, with each acting on it in turn. This mostly works, but the reality is that all the components of a pipeline are started simultaneously. This offers a performance gain in that the second program need not wait for the first program to finish, but can begin its own work as soon as any data becomes available. However, it can produce some surprising results. See Figure 0.1.

0.8 Filesystem Hierarchy

(Very nearly) every distribution of Linux uses a standard virtual filesystem that begins with `/` (pronounced "root") and mount not only files on the disk but also devices, areas of RAM, network shares, etc. You can read more about the Filesystem Hierarchy Standard at http://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard, but here are a few important places:

/proc Not an actual place on disk, but rather abstracts certain interfaces as files. For example, instead of requesting a CPU temperature measurement from **acpid**, you can just read the `/proc/acpi/thermal_zone/THM0/temperature` file.

/home Each user should have his local home directory here, under `/home/username`. You don't because your home directories are stored on a central server and mounted over the network wherever you log in.

```

$ ls
$ seq 1 10          # seq just counts
1
2
3
4
5
6
7
8
9
10
$ seq 1 10 > foo
$ ls                # the > created foo
foo
$ grep 1 foo        # note that grep operates blindly on text
1
10
$ cat foo | grep 1  # grep can search files, or STDIN
1
10
$ cat foo | grep 1 > bar      # redirecting the results
$ cat bar
1
10
$ cat foo | grep 1 > bar      # note how > truncates bar
$ cat bar
1
10
$ cat foo | grep 1 >> bar     # but >> appends
$ cat bar
1
10
1
10
$ grep 0 bar
10
10
$ grep 0 bar > bar           # what happened?
$ cat bar
$

```

Figure 0.1: The power and problems with redirections. What happened to bar?

/tmp A place for temporary files, whose permanence is not guaranteed. *Do not assume anything you put here will be available after you log off.*

/dev These are devices, both physical ones such as hard drives and sound cards, and virtual ones like consoles. True random bits can be read from **/dev/random** and pseudorandom bits from **/dev/urandom** if you need unpredictable inputs for testing, and **/dev/null** is a good place to send unwanted output.

- . Always refers to the current directory
- .. Always refers to the directory one above

Some commands for working with directories include **ls**, **cd**, **mkdir**, **rmdir**, and **pwd**. (Which are shell built-ins?) Their use is demonstrated in Figure 0.2.

Some basic commands for working with files include **less**, **rm**, **cp**, **mv**, **cat**, and **ln**. Their use is demonstrated in Figure 0.3.

0.9 Expansion

One of the shell's functions is called expansion, and at its most basic is a process of translating usually easier to remember or easier to type commands, although it can do far more than that. For example, it allows a user to type **cd ~bob** to Bob's home directory, or **touch 'date'** to create a file with the current date. The use of a wildcard expansion is demonstrated in Figure 0.4.

0.10 Permissions

Because UNIX was from the beginning a multi-user system, permissions were very important. UNIX recognizes three kinds of permissions: read, write, and execute, abbreviated **rwX**. On files these are reasonable as long as you remember that programs are also files, and running one is a special privilege.

On directories, the permissions are a little more opaque. Creating a file requires write access to the directory, and listing a directory's contents requires execute permission. However, listing is required for more than just the **ls** command; it is a fundamental operation without which many commands will fail. *For now, let the directory permissions be, and manage permissions at the file level.*

UNIX also recognizes three classes of permissions: user, group, and world or other, specified in that order. Each file is owned by a single user and a single group, and everyone else is considered the "world", and each such category can be granted or revoked its permissions independently via **chmod**. This command accepts two syntaxes, one which specifies the permissions absolutely and one which modifies the current state.

The first syntax takes three digits representing the permissions of the user, group, and world; each digit is a three-bit (octal) number whose bits represent read, write, and execute. So, for example, **chmod 751 foo.cgi** means the user can read, write, and execute, the group can read and execute, and everyone else can only execute. Here are a few common ones:

```

$ cd ~/
# go to home dir
$ pwd
# where am i?
/home/ivan
# i'm in my home dir
$ mkdir tmp
# make directory in home dir named 'tmp'
$ cd tmp
# change current directory to 'tmp'
$ pwd
/home/ivan/tmp
$ ls
# list files in working directory
$ ls -a
. ..
$ ls -a -l
drwxr-xr-x  2 ivan  college      512 Oct  4 19:29 .
drwxr-xr-x  3 ivan  college      512 Oct  4 19:29 ..
$ mkdir tmp2
# make new directory name 'tmp2'
$ ls
# is it there?
tmp2
# \ldots yes it is
$ ls -a tmp2
# show me everything in tmp2
. ..
$ cd tmp2
$ ls -a -l
drwxr-xr-x  2 ivan  college      512 Oct  4 19:38 .
drwxr-xr-x  3 ivan  college      512 Oct  4 19:38 ..
$ pwd
/home/ivan/tmp/tmp2
$ cd ..
$ pwd
/home/ivan/tmp
$ rmdir tmp3
# remove tmp3
rmdir: tmp3: No such file or directory # oops
$ rmdir tmp2
# remove correct directory
'tmp2'
$ ls
# is it still there?
$
# \ldots no

```

Figure 0.2: A transcript demonstrating some common directory-handling commands. Note that “#” is a comment character and tells the shell to ignore the rest of the line.

```

$ cd /home/ivan/tmp
$ ls -la
drwxr-xr-x  2 ivan  college  512 Oct  4 19:42 .
drwxr-xr-x  3 ivan  college  512 Oct  4 19:42 ..
-rw-r--r--  1 ivan  college   55 Oct  4 19:42 hello_world
$ more hello_world          # what's in this file?
Welcome to the MATRIX!
Thou wilt be programmed!

- neo
$ less hello_world          # what's in this file?
Welcome to the MATRIX!
Thou wilt be programmed!

- neo
hello_world (END)          # press 'q' to exit
$ cp hello_world h2        # copy 'hello_world' to 'h2'
$ ls
h2 hello_world             # now there are two files
$ mv hello_world h3        # rename 'hello_world'
with 'h3'
$ ls
h2 h3                      # now what's there?
$ rm h3                     # two files: 'h2' \& 'h3'
$ ls                         # remove h3
h2                           # now there's only h2
$ mv h2 ../                 # move h2 one level up
the directory tree
$ ls                         # now there are no files here
$ cd ..                      # go up one
level
$ ls
h2                           # here's the file we moved
$

```

Figure 0.3: A transcript demonstrating some common file-handling commands. Once again, “#” is a comment character.

```

$ ls -F                                # no output because there are no files
                                        # what does -F do?

$ mkdir foo

$ ls -F                                # -F appends a / to directory names
foo/

$ touch bar                            # what does touch do?

$ ls -F
bar  foo/

$ rm *                                  # rm doesn't like to remove directories
rm: cannot remove 'foo': Is a directory

$ ls -F                                # but it removed bar just fine
foo/

$ touch -r                              # touch thinks -r is a flag
touch: option requires an argument -- r
Try 'touch --help' for more information.

$ touch ./-r                           # ./-r forces -r to be a filename

$ ls -F
foo/  -r

$ rm *

$ ls -F                                # Where's foo/? And why is -r still here?
-r

```

Figure 0.4: A transcript demonstrating both the power and the danger of expansions.

755 a standard executable

644 a standard non-executable

700 a sensitive executable you don't want anyone else to see, modify, or use

600 a sensitive non-executable, maybe containing passwords

The second syntax is very well explained in the manpage, but invokes three new bits:

sticky Traditionally an optimization technique, it is now applied to directories to mean that each file can be renamed or deleted only by the file owner or the directory owner. (The superuser, of course, can do anything he wants.)

setuid Useful on executable files only, it permits the file to set its effective user identity to that of its owner rather than that of its caller. This permits, for example, **ping** to run as root and work with the network interface's control packets, a privilege denied to regular users.

setgid On files it acts like setuid, except for changing the effective group identity. On directories, it forces new files to be created using the directory's group identity rather than the creator's.

Note that `setuid` and `setgid` on a flawed program constitutes a major security flaw. *Do not set these bits unless you really must!*

What if you want to change a file's owner or group? **chown** (and **chgrp** if you like).

0.11 File Transfers

A secure and powerful way to move files between computers is **scp** (secure copy). A non-secure alternative is **ftp**, which offers a little more in the way of remote file manipulations. However, we will see another access method that can also accomplish these manipulations, and far more. Mac OS X can perform SCP operations either via **scp** in Terminal or by a graphical client such as Fugu. Windows requires a graphical client such as WinSCP.

A vocabulary moment: you sit at the local machine, and everything else is remote.

SCP optionally (but conveniently) allows passwordless identification using an SSH key (also called an RSA or DSA key for the algorithms that generate it). The command **ssh-keygen** will create two files, the public and private keys, distinguished by the `.pub` extension on the public version. The public key must be appended to the remote `~/.ssh/authorized_keys` file, where `~` refers to your home directory on the remote machine, and *the private key must never be passed over a network, or printed out to a logged screen, or in any other way exposed to eavesdroppers.*

Figure 0.5 shows **scp** in use. Note that Ivan apparently uses a different username on the local machine, or else he wouldn't need to specify `ivan@`. Also, note that the difference between uploading and downloading was just the order of arguments. Whether a file is local or remote makes no difference to **scp**, and in fact `scp john@jacob:/home/john/john.file jingleheimer@schmidt:/home/jingleheimer/` is a perfectly legal command. (Why don't I use the `~` expansion?)

However, **scp** will create the files using your remote login, which may not be what you want. For example, when developing web applications it is common to let the files be owned by the webserver. Other circumstances may require even more elaborate arrangements. Rather than manually correct all the permissions and ownerships after the transfer, use **tar**, which, incidentally, is also great for bundling directories up as neat, compressed packages called "tarballs."

A note on courteous **tar** usage: if you tar up a number of individual files, upon extraction they will be mixed into the remote working directory. This is called a "tarbomb," and while it seems to be standard practice with ZIP archives it is rightly much abhorred. *Always put your files into a single directory first.* If you intend to extract directly into the destination directory, then tar up a containing directory of the same name, and extract one directory higher. The system will copy in the newly extracted files on top of the existing directory contents.

Common ways of calling **tar** include `tar xzvf file.tar.gz -C destination`, `tar tjvpf file.tar.gz`, and `tar cf file.tar.gz source`. You tell me what they do.

```

$ scp ivan@harper.uchicago.edu:~/hello_world from_harper
# get file hello_world from harper and put name it as from_harper
ivan@harper.uchicago.edu's password:
scp: warning: Executing scp1 compatibility.
hello_world                100%   51      0.0KB/s   00:00
# download information
$ scp -r tmp harper.uchicago.edu:~/new_tmp
# upload the directory tmp to harper and name it as new_tmp
# make SURE that the -r flag is present, this means to scp recursively
ivan@harper.uchicago.edu's password:
scp: warning: Executing scp1 compatibility.
# some warning, ignore
stupid.sh                   100%   85      0.0KB/s   00:00
helloworld                  100%  100      0.0KB/s   00:00
helloworld2                 100%  471      0.0KB/s   00:00
job_latex.tex               100% 1110      0.0KB/s   00:00
h2                           100%   55      0.0KB/s   00:00
flight.tex                  100% 3604      0.0KB/s   00:00
flight.tex~                 100%  885      0.0KB/s   00:00
hello_world                  100%   55      0.0KB/s   00:00

```

Figure 0.5: A transcript demonstrating the use of `scp`.

0.12 Remote Access

I mentioned before that there was a better way to accomplish the file manipulations that `ftp` offers, without exposing your login and your data. This is `ssh`, which stands for “secure shell” and is in every way superior to its predecessors `rsh` and `telnet`. In particular, unlike `ftp` which allows only a limited set of commands, `ssh` lets you do anything you could do sitting in front of the computer, including run graphical applications via X-forwarding.

Techstaff offers a tool which lists machines to which you may connect at

http://tools.cs.uchicago.edu/find_cs_hosts/find.cgi

and NSIT allows connections to its Solaris server

harper.uchicago.edu.

A machine's popularity among remote users correlates positively with its specifications and negatively with the length of its host name. Publication of the previous statement will therefore be an interesting social experiment.

0.13 Summary of Commands

Table 0.1: A table of commonly-used commands, with some keystrokes or flags

cat	Prints the file contents, one after the next, to STDOUT
cd	Change the working directory, by default to your homedir.
chmod	Change modes (permissions) -R recurse through subdirectories
chown	Change owner and group -R Recurse through subdirectories
cp	Copy a file -r copy directories recursively
less	Page through the contents of a file. Space scroll down one page b scroll one page back / search q quit
ls	List the contents of a directory -F append / to directories, * to executables, etc. -l long format, showing size, permissions, owner, etc. -h use appropriate human-readable units (e.g., MB, not B) -a include hidden files
mv	Move (which can mean rename) a file.
pwd	print the working directory
rm	Remove files -r recurse through directories -f force your way through warnings
mkdir	Make a directory -p if the given path involves nonexistent parent directories, create them
scp	Securely copy files between machines -r copy directories recursively -C enable compression (not worth the processing time on fast lines) -p preserve modification and access times and permissions
ssh	Make an SSH connection -X forward X connections (-Y for trusted connections, like for CS machines)
tar	Create tape archives, a.k.a. tarballs -xv?f verbosely extract from a file. ? is z (gzip), j (bzip2), or nothing -c create the archive -p preserve permissions -C change to the given directory first

