Introduction to Linux-like Environments

Reece Hart http://dasher.wustl.edu/~ reece/

1998/03/19

Abstract

This document contains a very brief introduction to Linux-ish operating systems. It is intended primarily for users with little experience with these systems, but I hope that more advanced users may find useful tips as well.

I have four objectives: 1) Sketch features of Linux-ish operating systems; 2) Give an overview of shell interaction; 3) Survey useful commands and techniques; 4) Provide references to more information. I have emphasized shell techniques because it is the primary means of interaction for many users and the source of some confusion for new users.

The presentation is short and it is not possible to teach these topics in the time allotted. Instead, I hope to show the diversity and flexibility of Linux-ish systems and provide encouragement for exploring the environment by supplying plentiful and thought-provoking examples.

These notes were prepared for a St. Louis Linux Users' Group presentation on 1998/03/19. The notes and a PostScript version of them are available at:
http://www.cs.umsl.edu/~ feldt/sluugls/meeting_notes/1998/mar/index.html

1 Overview

Prefatory Note: Although there are important legal and pragmatic differences between Linux and Unix, they are indistinguishable to most users at which this document is targeted. Therefore, I'll use Linux-ish to mean "like Linux or any Unix variant".

1.1 Philosophy of Linix-ish Environments

Linux-ish environments have a dominant theme: Many small, general utilities are better than a few monolithic ones. Most naive users perceive that there are many commands, all of which seem to do very little. *This is correct! It's a feature, not a bug!* What frequently eludes new users is the power that many small utilities has.

For example, suppose one manages a web site and would like to see the number of hits from machines not in the wustl.edu domain, sorted by decreasing frequency. httpd logs contain lines of the form:
machine - - [*date*] "GET *url*" ...

The following pipeline does just what we want:

```
$ cut -f1 -d' ' /local/httpd/logs/access_log \
  | grep -v 'wustl\.edu' \
  | sort \
  | uniq -c \
  | sort -rn \
  | head -7
    851 crawl4.atext.com
    507 cybers65d18.mt.wave.shaw.ca
    427 scooter.pa-x.dec.com
    224 chris.arach-net.com
    136 206.187.18.239
    124 gw.envision.com
    110 www.st-charles-net.com
$
```

1.2 Kernel

The Linux kernel is the heart of the operating system. Its purpose is manage system resources (the CPUs, memory, disk drives, *etc.*). It is also responsible for verifying permissions for access to these resources.

1.3 Processes

The basic execution unit is a process. It is a particular instantiation of a program by a user. Under Linux-ish systems and unlike many others, more than one instantiation of a program may exist.

Processes may spawn other processes. When this occurs, the initiating program becomes the parent of the spawned process.

When a Linux system boots, the init program is started. It, in turn, starts all other processes on the system (kswapd, syslogd, X, *etc.*). When a user logs in, a new process is started for the login shell. When a command is typed in, the shell spawns a new process to run that command. The following shows an excerpt of processes running on my machine using a home-grown perl script:

```
$ ptree
init (S, pid=1)
  |-kflushd (S, pid=2)
  |-kswapd (S, pid=3)
  |-xdm (S, pid=114)
    |-X (S, pid=120)
    |-xdm (S, pid=121)
      |-.xsession (S, pid=132)
        |-fvwm2 (S, pid=217)
  |-emacs (S, pid=24159)
  |-xmcd (S, pid=12813)
```

1.4 Environments

A process "environment" is a set of variables accessible to the program. A program may modify its behavior depending on the value of these variables.

An important aspect of environments is that they are inherited by their children processes. In particular, a process inherits the environment of the login shell or window manager from which it was spawned.

1.5 Preemptive Multitasking

One of the most distinguishing features of Linux-ish systems is preemptive multitasking. This is the means by which the kernel scheduler may suspend a process and continue the execution of another. This gives the impression of simultaneous execution of programs.

Other operating systems (*e.g.,* MacOS, Windows) use cooperative multitasking. Under this scheme, a process *may* allow other processes to run. Those processes may run for indeterminate periods of time before returning control.

1.6 Protected Memory

The kernel ensures that a process accesses only memory which it has allocated. This serves two purposes: it protects the privacy of information from other processes (perhaps being run by a different user), and it keeps malicious or wayward programs from writing into another process' data.

When a process attempts to access memory outside of its memory space, it is killed by the kernel via a Segmentation Fault.

2 Shells

A shell is just a program which interprets and executes commands. It may be used for scripting or user interaction (login shell).

For users who spend most of their time running applications with a GUI, the choice of login shell may be immaterial. For those who use a shell as the primary means of interaction with the computer, the choice is critical.

Examples in this section assume BASH, the Bourne Again SHell. Commands appear in `a monospaced font`. Examples usually also show the canonical unprivileged user's prompt (`$`). I've indicated equivalent csh commands when I knew them.

2.1 Shell Flavors

Shells differ in two primary respects: features and command syntax. There are some esoteric difference in capabilities which are of use to only the most sophisticated users..

	Bourne-ish			C-ish	
	Bourne (sh)	Korn (ksh)	BASH (bash)	C (csh)	TC (tcsh)
command history	No	Yes	Yes	Yes	Yes
command alias	No	Yes	Yes	Yes	Yes
shell scripts	Yes	Yes	Yes	Yes	Yes
filename completion	No	Yes*	Yes	Yes*	Yes
command line editing	No	Yes*	Yes	No	Yes
job control	No	Yes	Yes	Yes	Yes

2.2 Shell Choice

The Bourne shell is the only shell which is guaranteed to exist on a system. It is also the smallest shell. For these reasons, it is the preferred shell for scripting. Because it lacks many convenient features, it's also a poor choice for a login shell.

Login shells are like St. Louis: your social status is determined by which school you're in. (And, furthermore, you're an idiot if you use a C shell[2]. –Ed.)

The most prevalent shells are bash and tcsh. There is a strong case to be made for bash: it's available on a large number of systems, it uses the GNU readline package, and it makes sense to script and interact with related (*i.e.*, Bourne-derived) shells. However, the strongest case is for the shell with which you're most comfortable.

2.3 Environments

Shells provide mechanisms by which variables may be modified. In Bourne-ish shells, the syntax is
`$ VARIABLE=VALUE`.
[csh: `setenv` *VARIABLE VALUE*]

In order for this variable to propagated to child processes, it must be exported using
`$ export VARIABLE`.

A child program cannot directly change the environment of any other process.

To see your current environment, type
`$ printenv`

2.4 Executing Commands

Commands in a shell may be one of two basic types: a shell builtin or an executable program. The latter may be a compiled executable binary or a text script written in a scripting language.

To see where a command lives, use the `type` command. For instance,

```
$ type cd mkdir rm
cd is a shell builtin
mkdir is /bin/mkdir
rm is aliased to 'rm -i'
```

[csh: use `which`]

2.5 PATH

Programs may live anywhere and it would be too time consuming for a shell to search all directories. Furthermore, what would happen if there were more than one executable by the same name?

The PATH evironment variable contains a colon-delimited list of directories in which commands may be found. When a user types a command that is not builtin, the shell searches all of the directories in the PATH variable, *in order*. A typical PATH variable is
`$HOME/bin:/usr/local/bin:/usr/bin/X11:/usr/bin`.

You can see your path (or any variable) by typing
`eg1$ echo $PATH`

Once a command is found, its location is remembered in a hash. Subsequent invocations check the hash before searching directories in PATH. If the PATH variable is

changed, the user may have to manually erase the hash with
`$ hash -r`

[csh: `rehash`]

2.6 Command Arguments

Many programs take options which modifies their behavior. Options typically begin with hyphen.

Many packages now support both short (one-letter) and long options. Short options may be combined behind a single hyphen. Long options are preceeded by a double hyphen (–).

Command arguments (*e.g.*, filenames) usually follow the options, if any.

So, what if a filename begins with a hyphen? The list of options is immediately terminated by a '– ' within the option list. So, you can delete the file '-trash' with a command like:
`$ rm -- -trash` or
`$ rm ./-trash`

Finally, many commands which take filename arguments will take the special filename '-' to mean standard input or standard output. For instance, `$ tar cf - bad-jokes/ | sendmail reece@dasher.wustl.edu`

2.7 Wildcards

Always seeking to emulate MSDOS, Unix provides filename wildcard characters too!

Tokens:

- `?`: any single character
- `*`: zero or more characters
- `[]`: character list
- `{opt1,opt2}`: "brace expansion"
- all other characters are literal

```
eg1$ rm -f .[a-zA-Z0-9]*
eg2$ ls -l [!A-Z]*
eg3$ mkdir -p /usr/local/{bin,src,share,info}
```

2.8 Quoting

Some characters have special meanings to the shell. For instance, spaces are used to delimit arguments. A variety of quoting mechanisms allow special characters to be passed as arguments to a command.

\ quotes the next character. C-v also quotes control characters.

All characters within single quotes are taken literally, except '. ' may not be escaped with a \.

Double quotes allow interpolation: variables and command substitutions are processed normally. They protect whitespace space and filename wildcards.

```
eg1$ touch 'A Filename With Spaces'
eg2$ rm A\ Filename\ With\ Spaces
eg3$ tar cf tar file file2
eg4$ tar cf "tar file" file2
eg5$ X="my data.tar"; tar cvf "$X" data/
```

2.9 Redirection

Output redirection. Output from a program may be redirected to a file using the >, m>, or m>&n operators. m and n refer to file descriptors, where 0 = stdin, 1 = stdout, and 2 = stderr.

```
eg1$ ls >dir-contents
eg2$ tar cvf backup.tar >tar.log 2>&1
eg3$ find / -name \*.c >find.log 2>/dev/null
```

Input Redirection. Input to a program may be obtain from a file by using the <, << *TAG* operators.

```
eg1$ xargs pgrep <file-of-files
```

TIP: Never use `cat` with a pipe when you could've just redirected. People will laugh at you. *e.g.*

```
hip$ somecommand <file
doh$ cat file | somecommand
```

Finally, redirections can occur anywhere on the line:

```
sohip$ <file somecommand
```

2.10 Pipes

Pipes are an extension of output and input redirection: the output of one program is the input of another.

```
eg1$ /bin/ls -1 | sort -f
```

2.11 Running Commands in the Background

Commands may be run in the background by adding an ampersand to the end of the command. Use the `jobs` builtin to see what's running, or `top` to see the most active processes on the system.

```
1$ find ~ -name \*.c >c-files &
[1] 30384
2$ make -C pub >make.log 2>&1 &
[2] 30385
3$ jobs
[1]- 30384 Running                 find ~ -name \*.c >c-files &
[2]+ 30385 Running                 make -C pub >make.log 2>&1 &
```

Jobs can be referred to by their process id (30384 and 30385), or by their job number (%1 and %2). (Job numbers are known only by the parent shell, but process ids are system-wide.)

Foreground jobs may be interrupted with C-z. That job may be placed in the background by typing `bg`. The job will halt when it requests input or displays output (unless it has been redirected). The job may be killed with `kill` *pid*.

```
1$ find / -name \*bogus >|~/tmp/bogus-finds+
  (C-z)
[1]+  Stopped         find / -name \*bogus >|~/tmp/bogus-finds
2$ bg
[1]+ find / -name \*bogus >|~/tmp/bogus-finds &
3$ kill %1
[1]+  Terminated      find / -name \*bogus >|~/tmp/bogus-finds
```

2.12 Subprocesses

Subprocesses execute a series of commands within a subshell. It's particularly useful when you want to group a bunch of commands into a mini script. For instance:

```
eg1$ (set -x; for f in data/*; do processit $f >$f.log; done) 2>errors
```

2.13 Command and Process Substitution

Many shells provide command substitution: the ability to use the output of a command as an argument to another.

```
eg1$ OS=`uname -s`
```

Wake up. This is cool. BASH also provides process substitution: The output of a command is stored in a temporary file, and that filename is substituted on the command line.

```
eg2$ comm <(cd /usr/bin;       /bin/ls) \
          <(cd /usr/local/bin; /bin/ls)
```

3 Scripting

3.1 general notes

One of the most flexible aspects of a Linux-ish environment is the ability to write scripts. Scripts may be written in any scripting language you like. Some common languages are sh, perl, and tcl/tk.

How does one make a script? Easy. The first line should be:

`#!/path/to/script/interpreter`

In addition, it's convenient to make the script executable:

`$ chmod 755` *filename*

3.2 shell

As an example, let's say you want a script to open a URL from the command line. Put the following in a file called open-url.sh:

```
#!/bin/sh
# open-url -- the Bourne-shell version
# usage: open-url URL
browser=${BROWSER:-netscape}
if [ $# -lt 1 ]; then
    echo usage: open-url URL 1>&2
    exit 1
fi
$browser -remote "openUrl($1)" || $browser $1
```

Then type

`$ chmod 755 open-url.sh`

And now: `$ open-url.sh http://dasher.wustl.edu/~reece/ &`

If a netscape is running, it'll get sent the remote command; if not, a new one will be started.

3.3 perl

Now let's try perl. This version allows you to enter the relative path of a local file. It'll automatically prepend `file:` and the current directory if there's no protocol specified.

```
#!/usr/local/bin/perl -w
use Cwd;
die("usage: open-url URL\n") if ($#ARGV != 0);
$browser = exists $ENV{'BROWSWER'} ? $ENV{'BROWSWER'} : 'netscape';
```

```
$url = pop(@ARGV);
if (not $url=~m/^(?:file|gopher|https?|mailto|news):/)
  {$url = 'file:' . ($url=~m|^/| ? $url : cwd."/$url")}
(system($browser,@ARGV,'-remote',"openUrl($url)")>>8)
  and exec($browser,@ARGV,$url);
```

Soapbox: perl has oodles of very attractive features: unlimited length arrays and hashes; supports object oriented modules; protected namespaces; turbo studly regular expressions; system interface commands; taint security checking; simplified data types; and ... (time passes, people fall asleep) ... and on-the-fly compilation.

3.4 make

Not many people think of make as a scripting language, but it's ideal when a series of steps are used to process files repeatedly.

Make builds files according to rules. A rule may describe how to build one type of file from another type , or how to build a specific file from a specific dependency. Make builds any intermediate dependencies required to achieve

Suppose your new OpenSource program depends on 3 files: file1.c, file2.c, and main.c. To compile the program, you'd type:

```
$ gcc -c file1.c
$ gcc -c file2.c
$ gcc -c main.c
$ gcc -o myprogram file1.o file2.o file3.o
```

That would get old fast. You'd have to remember which .o files were current with respect to their sources, and you'd have to know whether the executable was current with respect to the objects.

Make automates this. By default, it knows how to build a .o file from a .c file. All you need to tell it is that myprogram depends on three object files:

```
# Makefile for myprogram
myprogram : file1.o file2.o main.o
    gcc -o $@ $^
```

Make's not limited to programming. How 'bout this for counting url hits and hosts from our web server access log:

```
# Makefile for access_logs
LOG:=/local/httpd/logs/access_log
```

```
default : host.counts url.counts

host.list : ${LOG}
	cut -f1 -d' ' $< >$@

url.list : ${LOG}
	perl -ne 'm/"GET (\S+)/ && print($$1,"\n")' $< >$@

%.sorted : %.list
	sort $< >$@

%.counts : %.sorted
	uniq -c $< >$@
```

4 Regular Expressions

Regular expressions represent patterns of text. Here are a few examples to get us started. I'm looking for a love song in my guitar archives:

```
$ grep -l love guitar/*
guitar/50-Ways.crd
guitar/American-Pie.pro
guitar/At-17.pro
guitar/Best-Of-My-Love.crd
guitar/Boxer.tab
$
```

Boxer? Love? Oops, that's *glove*. Not only that, I missed the case variants of that. Besides, love's too mooshy. I want all the songs with 'romance' and 'candles' in the lyrics. This is a job for perl:

```
$ perl -n0e 'print($ARGV,"\n") if (/\Wlove\W/i and /\Wwife\W/i and \
/\Wmoon/i)' guitar/*
guitar/space-oddity
$
```

Hmm. Not exactly the kind of song I was looking for.

5 Tips, Tricks, and Tools

5.1 Applying a command to a list of files

One frequently wants to execute a command on a list of files. You might imagine doing
`eg1$ gzip -9v` ‘find / -name .log‘

Unfortunately, some systems have limits on the size of a command line. (Does linux?) xargs assembles and executes command lines from arguments provided on stdin. For instance:

```
eg2$ find / -name \*.log | xargs gzip -9v
```

5.2 Comparing Files with md5

md5 is an algorithm for computing a “message digest” – a fingerprint for a file. md5sum (part of the GNU fileutils package) computes md5 digests for files given on the command line.

```
eg1$ md5sum sluug.tex
407e6f4d8c2d008e0a46def8595e8671  sluug.tex
```

Suppose two large files reside on distinct computers separated by a slow link and you wish to know whether they're identical or not. It's much easier to compare the digest than to download a file and `cmp` them.

md5 is frequently used to ensure that packages are distributed unmolested. Many sites distribute a file formatted like

```
2c43e710dcea2089bcf139f5b1d3ecdd  README
b7d07cc5545baec15aa2539cf7e44e83  file1.c
9e68832ade6156495b5377a2bf2bdc3a  file2.c
```

To verify that the package has not been modified, simply type

```
$ md5sum -c md5-file
README: OK
file1.c: FAILED
file2.c: OK
```

I use md5 to compare files at home with files at lab with

```
$ rsh dasher 'find src -type f | xargs md5sum | gzip -c' \
  | gzip -cd | md5sum -c
```

5.3 Configuration

Most programs follow the same paradigm for configuration. The effective configuration for a program is assembled by layering parameters from the following sources in order:

Default parameters Stored within the program

System-wide parameters Stored in a shared directory

User parameters Stored in the user's directory

Invocation parameters Specified at invocation

For example, Xdefaults are typically compiled into the program and augmented by specifications in /usr/lib/X11/app-defaults/, $HOME/.Xdefaults, and the command line. The default color of the xclock background is white; a line in /usr/lib/X11/app-defaults/XClock like
`XClock*background: orange`
shows no sense of aesthetics and could be overridden with a line in $HOME/.Xdefaults:
`XClock*background: cornsilk`.
You might use the following to remind yourself to go home at 9pm:
`$echo "xclock -display :0 -geometry 500x500 -bg red" | at 9pm`

References

[1] The linux journal. www.linuxjournal.com. ISBN:1075-3583.

[2] CHRISTIANSON, T. csh programming considered harmful. http://faq.sph.umich.edu/faq/files/unix-faq/shell/csh-whynot.

[3] freshmeat. http://freshmeat.unreal.org.

[4] GILLY, D. *UNIX in a Nutshell.* O'Reilly & Associates, Inc., http://www.ora.com, 1992.

[5] HART, R. K. Miscellaneous scripts and configuration files. http://dasher.wustl.edu/ reece/setup,src/.

[6] INGLIS, G., FARVIS, K., SMITH, J., AND MCGONIGLE, B. Unixhelp 1.3. http://unixhelp.ed.ac.uk/, January 1996. University of Edinburgh.

[7] Linux headquarters. http://www.linuxhq.com.

[8] RUSLING, D. A. *The Linux Kernel 0.8.* The Linux Documentation Project, 1998.

[9] SRINIVASAN, S. *Advanced Perl Programming.* O'Reilly & Associates, Inc., http://www.ora.com, 1997.

[10] WALL, L., CHRISTIANSEN, T., AND SCHWARTZ, R. L. *Programming Perl.* O'Reilly & Associates, Inc., http://www.ora.com, 1996.