LPI certification 101 exam prep (release 2), Part 2

Presented by developerWorks, your source for great tutorials

`ibm.com/developerWorks`

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Before you start 2
2. Regular expressions 4
3. FHS and finding files 8
4. Process control 15
5. Text processing 20
6. Kernel modules 26
7. Summary and resources 31

Section 1. Before you start

About this tutorial

Welcome to "Basic administration," the second of four tutorials designed to prepare you for the Linux Professional Institute's 101 exam. In this tutorial, we'll show you how to use regular expressions to search files for text patterns. Next, we'll introduce you to the Filesystem Hierarchy Standard (FSH), and then show you how to locate files on your system. Then, we'll show you how to take full control of Linux processes by running them in the background, listing processes, detaching processes from the terminal, and more. Next, we'll give you a whirlwind introduction to shell pipelines, redirection, and text processing commands. Finally, we'll introduce you to Linux kernel modules.

This particular tutorial (Part 2) is ideal for those who have a good basic knowledge of `bash` and want to receive a solid introduction to basic Linux administration tasks. If you are new to Linux, we recommend that you complete *Part 1* of this tutorial series first before continuing. For some, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way of "rounding out" their basic Linux administration skills.

By the end of this tutorial, you'll have a solid grounding in basic Linux administration and will be ready to begin learning some more advanced Linux system administration skills. By the end of this series of tutorials (eight in all), you'll have the knowledge you need to become a Linux Systems Administrator and will be ready to attain an LPIC Level 1 certification from the Linux Professional Institute if you so choose.

For those who have taken the *release 1 version* of this tutorial for reasons other than LPI exam preparation, you probably don't need to take this one. However, if you do plan to take the exams, you should strongly consider reading this revised tutorial.

About the authors

For technical questions about the content of this tutorial, contact the authors:

- Daniel Robbins, at *drobbins@gentoo.org*
- Chris Houser, at *chouser@gentoo.org*
- Aron Griffis, at *agriffis@gentoo.org*

Residing in Albuquerque, New Mexico, **Daniel Robbins** is the Chief Architect of *Gentoo Linux* an advanced ports-based Linux metadistribution. Besides writing articles, tutorials, and tips for the *developerWorks* Linux zone and Intel Developer Services, he has also served as a contributing author for several books, including *Samba Unleashed* and *SuSE Linux Unleashed*. Daniel enjoys spending time with his wife, Mary, and his daughter, Hadassah. You can contact Daniel at *drobbins@gentoo.org*.

Chris Houser, known to his friends as "Chouser," has been a UNIX proponent since 1994 when he joined the administration team for the computer science network at Taylor University in Indiana, where he earned his Bachelor's degree in Computer Science and Mathematics. Since then, he has gone on to work in Web application programming, user interface design, professional video software support, and now Tru64 UNIX device driver programming at *Compaq*. He has also contributed to various free software projects, most recently to *Gentoo Linux*). He lives with his wife and two cats in New Hampshire. You can contact Chris at *chouser@gentoo.org*.

Aron Griffis graduated from Taylor University with a degree in Computer Science and an award that proclaimed, "Future Founder of a Utopian UNIX Commune." Working towards that goal, Aron is employed by *Compaq* writing network drivers for Tru64 UNIX, and spending his spare time plunking out tunes on the piano or developing *Gentoo Linux*. He lives with his wife Amy (also a UNIX engineer) in Nashua, New Hampshire.

Section 2. Regular expressions

What is a regular expression?

A regular expression (also called a "regex" or "regexp") is a special syntax used to describe text patterns. On Linux systems, regular expressions are commonly used to find patterns of text, as well as to perform search-and-replace operations on text streams.

Glob comparison

As we take a look at regular expressions, you may find that regular expression syntax looks similar to the filename "globbing" syntax that we looked at in Part 1. However, don't let this fool you; their similarity is only skin deep. Both regular expressions and filename globbing patterns, while they may look similar, are fundamentally different beasts.

The simple substring

With that caution, let's take a look at the most basic of regular expressions, the *simple substring*. To do this, we're going to use `grep`, a command that scans the contents of a file for a particular regular expression. `grep` prints every line that matches the regular expression, and ignores every line that doesn't:

```
$ grep bash /etc/passwd
operator:x:11:0:operator:/root:/bin/bash
root:x:0:0::/root:/bin/bash
ftp:x:40:1::/home/ftp:/bin/bash
```

Above, the first parameter to `grep` is a regex; the second is a filename. `Grep` read each line in /etc/passwd and applied the simple substring regex `bash` to it, looking for a match. If a match was found, `grep` printed out the entire line; otherwise, the line was ignored.

Understanding the simple substring

In general, if you are searching for a substring, you can just specify the text verbatim without supplying any "special" characters. The only time you'd need to do anything special would be if your substring contained a `+`, `.`, `*`, `[`, `]`, or `\`, in which case these characters would need to be enclosed in quotes and preceded by a backslash. Here are a few more examples of simple substring regular expressions:

- `/tmp` (scans for the literal string `/tmp`)
- `"\[box\]"` (scans for the literal string `[box]`)
- `"\*funny\*"` (scans for the literal string `*funny*`)
- `"ld\.so"` (scans for the literal string `ld.so`)

Metacharacters

With regular expressions, you can perform much more complex searches than the examples we've looked at so far by taking advantage of metacharacters. One of these metacharacters is the `.` (a period), which matches any single character:

```
$ grep dev.hda /etc/fstab
/dev/hda3       /               reiserfs        noatime,ro 1 1
/dev/hda1       /boot           reiserfs        noauto,noatime,notail 1 2
/dev/hda2       swap            swap            sw 0 0
#/dev/hda4      /mnt/extra      reiserfs        noatime,rw 1 1
```

In this example, the literal text `dev.hda` didn't appear on any of the lines in /etc/fstab. However, `grep` wasn't scanning them for the literal `dev.hda` string, but for the `dev.hda` *pattern*. Remember that the `.` will match *any single character*. As you can see, the `.` metacharacter is functionally equivalent to how the ? metacharacter works in "glob" expansions.

Using []

If we wanted to match a character a bit more specifically than `.`, we could use `[` and `]` (square brackets) to specify a subset of characters that should be matched:

```
$ grep dev.hda[12] /etc/fstab
/dev/hda1       /boot           reiserfs        noauto,noatime,notail 1 2
/dev/hda2       swap            swap            sw 0 0
```

As you can see, this particular syntactical feature works identically to the `[]` in "glob" filename expansions. Again, this is one of the tricky things about learning regular expressions -- the syntax is *similar but not identical* to "glob" filename expansion syntax, which often makes regexes a bit confusing to learn.

Using [^]

You can reverse the meaning of the square brackets by putting a `^` immediately after the `[`. In this case, the brackets will match any character that is **not** listed inside the brackets. Again, note that we use `[^]` with regular expressions, but `[!]` with globs:

```
$ grep dev.hda[^12] /etc/fstab
/dev/hda3       /               reiserfs        noatime,ro 1 1
#/dev/hda4      /mnt/extra      reiserfs        noatime,rw 1 1
```

Differing syntax

It's important to note that the syntax *inside* square brackets is fundamentally different from that in other parts of the regular expression. For example, if you put a `.` inside square brackets, it allows the square brackets to match a literal `.`, just like the `1` and `2` in the examples above. In comparison, a literal `.` outside the square brackets is interpreted as a metacharacter unless prefixed by a `\`. We can take advantage of this fact to print a list of all lines in /etc/fstab that contain the literal string `dev.hda` by typing:

```
$ grep dev[.]hda /etc/fstab
```

Alternately, we could also type:

```
$ grep "dev\.hda" /etc/fstab
```

Neither regular expression is likely to match any lines in your /etc/fstab file.

The "*" metacharacter

Some metacharacters don't match anything in themselves, but instead modify the meaning of a previous character. One such metacharacter is `*` (asterisk), which is used to match zero or more repeated occurrences of the previous character. Note that this means that the `*` has a different meaning in a regex than it does with globs. Here are some examples, and play close attention to instances where these regex matches differ from globs:

- `ab*c` matches `abbbbc` but not `abqc` (if a glob, it would match both strings -- can you figure out why?)
- `ab*c` matches `abc` but not `abbqbbc` (again, if a glob, it would match both strings)
- `ab*c` matches `ac` but not `cba` (if a glob, `ac` would not be matched, nor would `cba`)
- `b[cq]*e` matches `bqe` and `be` (if a glob, it would match `bqe` but not `be`)
- `b[cq]*e` matches `bccqqe` but not `bccc` (if a glob, it would match the first but not the second as well)
- `b[cq]*e` matches `bqqcce` but not `cqe` (if a glob, it would match the first but not the second as well)
- `b[cq]*e` matches `bbbeee` (this would not be the case with a glob)
- `.*` will match any string. (if a glob, it would match any string starting with `.`)
- `foo.*` will match any string that begins with `foo` (if a glob, it would match any string starting with the four literal characters `foo.`.)

Now, for a quick brain-twisting review: the line `ac` matches the regex `ab*c` because the asterisk also allows the preceding expression (c) to appear **zero** times. Again, it's critical to note that the `*` regex metacharacter is interpreted in a fundamentally different way than the `*` glob character.

Beginning and end of line

The last metacharacters we will cover in detail here are the `^` and `$` metacharacters, used to match the beginning and end of line, respectively. By using a `^` at the beginning of your regex, you can cause your pattern to be "anchored" to the start of the line. In the following example, we use the `^#` regex to match any line beginning with the # character:

```
$ grep ^# /etc/fstab
# /etc/fstab: static file system information.
#
```

Full-line regexes

^ and $ can be combined to match an entire line. For example, the following regex will match a line that starts with the # character and ends with the `.` character, with any number of other characters in between:

```
$ grep '^#.*\.$' /etc/fstab
# /etc/fstab: static file system information.
```

In the above example, we surrounded our regular expression with single quotes to prevent $ from being interpreted by the shell. Without the single quotes, the $ will disappear from our regex before grep even has a chance to take a look at it.

Section 3. FHS and finding files

Filesystem Hierarchy Standard

The Filesystem Hierarchy Standard is a document that specifies the layout of directories on a Linux system. The FHS was devised to provide a common layout to simplify distribution-independent software development -- so that stuff is in generally the same place across Linux distributions. The FHS specifies the following directory tree (taken directly from the FHS specification):

- `/` (the root directory)
- `/boot` (static files of the boot loader)
- `/dev` (device files)
- `/etc` (host-specific system configuration)
- `/lib` (essential shared libraries and kernel modules)
- `/mnt` (mount point for mounting a filesystem temporarily)
- `/opt` (add-on application software packages)
- `/sbin` (essential system binaries)
- `/tmp` (temporary files)
- `/usr` (secondary hierarchy)
- `/var` (variable data)

The two independent FHS categories

The FHS bases its layout specification on the idea that there are two independent categories of files: shareable vs. unshareable, and variable vs. static. *Shareable* data can be shared between hosts; *unshareable* data is specific to a given host (such as configuration files). *Variable* data can be modified; *static* data is not modified (except at system installation and maintenance).

The following grid summarizes the four possible combinations, with examples of directories that would fall into those categories. Again, this table is straight from the FHS specification:

```
+---------+-----------------+-------------+
|         | shareable       | unshareable |
+---------+-----------------+-------------+
|static   | /usr            | /etc        |
|         | /opt            | /boot       |
+---------+-----------------+-------------+
|variable | /var/mail       | /var/run    |
|         | /var/spool/news | /var/lock   |
+---------+-----------------+-------------+
```

Secondary hierarchy at /usr

Under /usr you'll find a secondary hierarchy that looks a lot like the root filesystem. It isn't critical for /usr to exist when the machine powers up, so it can be shared on a network (shareable), or mounted from a CD-ROM (static). Most Linux setups don't make use of sharing /usr, but it's valuable to understand the usefulness of distinguishing between the primary hierarchy at the root directory and the secondary hierarchy at /usr.

This is all we'll say about the Filesystem Hierarchy Standard. The document itself is quite readable, so you should go take a look at it. You'll understand a lot more about the Linux filesystem if you read it. Find it at *http://www.pathname.com/fhs/*.

Finding files

Linux systems often contain hundreds of thousands of files. Perhaps you are savvy enough to never lose track of any of them, but it's more likely that you will occasionally need help finding one. There are a few different tools on Linux for finding files. This introduction will help you choose the right tool for the job.

The PATH

When you run a program at the command line, `bash` actually searches through a list of directories to find the program you requested. For example, when you type `ls`, `bash` doesn't intrinsically know that the `ls` program lives in /usr/bin. Instead, `bash` refers to an environment variable called `PATH`, which is a colon-separated list of directories. We can examine the value of `PATH`:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin
```

Given this value of `PATH` (yours may differ,) `bash` would first check /usr/local/bin, then /usr/bin for the `ls` program. Most likely, `ls` is kept in /usr/bin, so `bash` would stop at that point.

Modifying PATH

You can augment your `PATH` by assigning to it on the command line:

```
$ PATH=$PATH:~/bin
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin:/home/agriffis/bin
```

You can also remove elements from `PATH`, although it isn't as easy since you can't refer to the existing `$PATH`. Your best bet is to simply type out the new `PATH` you want:

```
$ PATH=/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:~/bin
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/agriffis/bin
```

To make your `PATH` changes available to any future processes you start from this shell, export your changes using the `export` command:

```
$ export PATH
```

All about "which"

You can check to see if there's a given program in your `PATH` by using `which`. For example, here we find out that our Linux system has no (common) sense:

```
$ which sense
which: no sense in (/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin)
```

In this example, we successfully locate `ls`:

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

"which -a"

Finally, you should be aware of the `-a` flag, which causes `which` to show you all of the instances of a given program in your PATH:

```
$ which -a ls
/usr/bin/ls
/bin/ls
```

whereis

If you're interested in finding more information than purely the location of a program, you might try the `whereis` program:

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

Here we see that `ls` occurs in two common binary locations, /bin and /usr/bin. Additionally, we are informed that there is a manual page located in /usr/share/man. This is the man-page you would see if you were to type `man ls`.

The `whereis` program also has the ability to search for sources, to specify alternate search paths, and to search for unusual entries. Refer to the `whereis` man-page for further information.

find

The `find` command is another handy tool for your toolbox. With `find` you aren't restricted to programs; you can search for any file you want, using a variety of search criteria. For example, to search for a file by the name of `README`, starting in /usr/share/doc:

```
$ find /usr/share/doc -name README
/usr/share/doc/ion-20010523/README
/usr/share/doc/bind-9.1.3-r6/dhcp-dynamic-dns-examples/README
/usr/share/doc/sane-1.0.5/README
```

find and wildcards

You can use "glob" wildcards in the argument to `-name`, provided that you quote them or backslash-escape them (so they get passed to `find` intact rather than being expanded by `bash`). For example, we might want to search for `README` files with extensions:

```
$ find /usr/share/doc -name README\*
/usr/share/doc/iproute2-2.4.7/README.gz
/usr/share/doc/iproute2-2.4.7/README.iproute2+tc.gz
/usr/share/doc/iproute2-2.4.7/README.decnet.gz
/usr/share/doc/iproute2-2.4.7/examples/diffserv/README.gz
/usr/share/doc/pilot-link-0.9.6-r2/README.gz
/usr/share/doc/gnome-pilot-conduits-0.8/README.gz
/usr/share/doc/gimp-1.2.2/README.i18n.gz
/usr/share/doc/gimp-1.2.2/README.win32.gz
/usr/share/doc/gimp-1.2.2/README.gz
/usr/share/doc/gimp-1.2.2/README.perl.gz
[578 additional lines snipped]
```

Ignoring case with find

Of course, you might want to ignore case in your search:

```
$ find /usr/share/doc -name '[Rr][Ee][Aa][Dd][Mm][Ee]*'
```

Or, more simply:

```
$ find /usr/share/doc -iname readme\*
```

As you can see, you can use `-iname` to do case-insensitive searching.

find and regular expressions

If you're familiar with regular expressions, you can use the `-regex` option to limit the output to filenames that match a pattern. And similar to the `-iname` option, there is a corresponding `-iregex` option that ignores case in the pattern. For example:

```
$ find /etc -iregex '.*xt.*'
/etc/X11/xkb/types/extra
/etc/X11/xkb/semantics/xtest
/etc/X11/xkb/compat/xtest
/etc/X11/app-defaults/XTerm
/etc/X11/app-defaults/XTerm-color
```

Note that unlike many programs, `find` requires that the regex specified matches the entire path, not just a part of it. For that reason, specifying the leading and trailing `.*` is necessary; purely using `xt` as the regex would not be sufficient.

find and types

The `-type` option allows you to find filesystem objects of a certain type. The possible arguments to `-type` are `b` (block device), `c` (character device), `d` (directory), `p` (named pipe), `f` (regular file), `l` (symbolic link), and `s` (socket). For example, to search for symbolic links in /usr/bin that contain the string `vim`:

```
$ find /usr/bin -name '*vim*' -type l
/usr/bin/rvim
/usr/bin/vimdiff
/usr/bin/gvimdiff
```

find and mtimes

The `-mtime` option allows you to select files based on their last modification time. The argument to `mtime` is in terms of 24-hour periods, and is most useful when entered with either a plus sign (meaning "after") or a minus sign (meaning "before"). For example, consider the following scenario:

```
$ ls -l ?
-rw-------    1 root     root            0 Jan  7 18:00 a
-rw-------    1 root     root            0 Jan  6 18:00 b
-rw-------    1 root     root            0 Jan  5 18:00 c
-rw-------    1 root     root            0 Jan  4 18:00 d
$ date
Mon May  7 18:14:52 EST 2003
```

You could search for files that were created in the past 24 hours:

```
$ find . -name \? -mtime -1
./a
```

Or you could search for files that were created prior to the current 24-hour period:

```
$ find . -name \? -mtime +0
./b
./c
./d
```

The -daystart option

If you additionally specify the `-daystart` option, then the periods of time start at the beginning of today rather than 24 hours ago. For example, here is a set of files created yesterday and the day before:

```
$ find . -name \? -daystart -mtime +0 -mtime -3
./b
./c
$ ls -l b c
-rw-------    1 root     root            0 May  6 18:00 b
-rw-------    1 root     root            0 May  5 18:00 c
```

The -size option

The `-size` option allows you to find files based on their size. By default, the argument to `-size` is 512-byte blocks, but adding a suffix can make things easier. The available suffixes are b (512-byte blocks), c (bytes), k (kilobytes), and w (2-byte words). Additionally, you can prepend a plus sign ("larger than") or minus sign ("smaller than").

For example, to find regular files in /usr/bin that are smaller than 50 bytes:

```
$ find /usr/bin -type f -size -50c
/usr/bin/krdb
/usr/bin/run-nautilus
/usr/bin/sgmlwhich
/usr/bin/muttbug
```

Processing found files

You may be wondering what you can do with all these files that you find! Well, `find` has the ability to act on the files that it finds by using the `-exec` option. This option accepts a command line to execute as its argument, terminated with `;`, and it replaces any occurrences of `{}` with the filename. This is best understood with an example:

```
$ find /usr/bin -type f -size -50c -exec ls -l '{}' ';'
-rwxr-xr-x    1 root     root           27 Oct 28 07:13 /usr/bin/krdb
-rwxr-xr-x    1 root     root           35 Nov 28 18:26 /usr/bin/run-nautilus
-rwxr-xr-x    1 root     root           25 Oct 21 17:51 /usr/bin/sgmlwhich
-rwxr-xr-x    1 root     root           26 Sep 26 08:00 /usr/bin/muttbug
```

As you can see, `find` is a very powerful command. It has grown through the years of UNIX and Linux development. There are many other useful options to `find`. You can learn about them in the `find` manual page.

locate

We have covered `which`, `whereis`, and `find`. You might have noticed that `find` can take a while to execute, since it needs to read each directory that it's searching. It turns out that the `locate` command can speed things up by relying on an external database generated by `updatedb` (which we'll cover in the next panel.)

The `locate` command matches against any part of a pathname, not just the file itself. For example:

```
$ locate bin/ls
```

```
/var/ftp/bin/ls
/bin/ls
/sbin/lsmod
/sbin/lspci
/usr/bin/lsattr
/usr/bin/lspgpot
/usr/sbin/lsof
```

Using updatedb

Most Linux systems have a "cron job" to update the database periodically. If your `locate` returned an error such as the following, then you will need to run `updatedb` as root to generate the search database:

```
$ locate bin/ls
locate: /var/spool/locate/locatedb: No such file or directory
$ su
Password:
# updatedb
```

The `updatedb` command may take a long time to run. If you have a noisy hard disk, you will hear a lot of racket as the entire filesystem is indexed. :)

slocate

On many Linux distributions, the `locate` command has been replaced by `slocate`. There is typically a symbolic link to "locate" so that you don't need to remember which you have. `slocate` stands for "secure locate." It stores permissions information in the database so that normal users can't pry into directories they would otherwise be unable to read. The usage information for `slocate` is essentially the same as for `locate`, although the output might be different depending on the user running the command.

Section 4. Process control

Staring xeyes

To learn about process control, we first need to start a process. Make sure that you have X running and execute the following command:

```
$ xeyes -center red
```

You will notice that an `xeyes` window pops up, and the red eyeballs follow your mouse around the screen. You may also notice that you don't have a new prompt in your terminal.

Stopping a process

To get a prompt back, you could type Control-C (often written as Ctrl-C or ^C):

```
^C
$
```

You get a new `bash` prompt, but the `xeyes` window disappeared. In fact, the entire process has been killed. Instead of killing it with Control-C, we could have just stopped it with Control-Z:

```
$ xeyes -center red
^Z
[1]+  Stopped                 xeyes -center red
$
```

This time you get a new `bash` prompt, and the `xeyes` windows stays up. If you play with it a bit, however, you will notice that the eyeballs are frozen in place. If the `xeyes` window gets covered by another window and then uncovered again, you will see that it doesn't even redraw the eyes at all. The process isn't doing *anything*. It is, in fact, "Stopped."

fg and bg

To get the process "un-stopped" and running again, we can bring it to the foreground with the `bash` built-in `fg`:

```
$ fg
```

`xeyesbashbash`**and**`xeyes`

```
^Z
[1]+  Stopped                 xeyes -center red
$
```

Now continue it in the background with the `bash` built-in `bg`:

```
$ bg
[1]+ xeyes -center red &
$
```

Great! The `xeyes` process is now running in the background, and we have a new, working `bash` prompt.

Using "&"

If we wanted to start xeyes in the background from the beginning (instead of using Control-Z and `bg`), we could have just added an "&" (ampersand) to the end of xeyes command line:

```
$ xeyes -center blue &
[2] 16224
```

Multiple background processes

Now we have both a red and a blue `xeyes` running in the background. We can list these jobs with the `bash` built-in `jobs`:

```
$ jobs -l
[1]- 16217 Running                 xeyes -center red &
[2]+ 16224 Running                 xeyes -center blue &
```

The numbers in the left column are the job numbers `bash` assigned when they were started. Job 2 has a + (plus) to indicate that it's the "current job," which means that typing `fg` will bring it to the foreground. You could also foreground a specific job by specifying its number; for example, `fg 1` would make the red `xeyes` the foreground task. The next column is the process id or `pid`, included in the listing courtesy of the `-l` option to `jobs`. Finally, both jobs are currently "Running," and their command lines are listed to the right.

Introducing signals

To kill, stop, or continue processes, Linux uses a special form of communication called "signals." By sending a certain signal to a process, you can get it to terminate, stop, or do other things. This is what you're actually doing when you type Control-C, Control-Z, or use the `bg` or `fg` built-ins -- you're using `bash` to send a particular signal to the process. These signals can also be sent using the `kill` command and specifying the pid (process id) on the command line:

```
$ kill -s SIGSTOP 16224
$ jobs -l
[1]- 16217 Running                 xeyes -center red &
[2]+ 16224 Stopped (signal)        xeyes -center blue
```

As you can see, `kill` doesn't necessarily "kill" a process, although it can. Using the "-s" option, kill can send any signal to a process. Linux kills, stops or continues processes when they are sent the SIGINT, SIGSTOP, or SIGCONT signals respectively. There are also other signals that you can send to a process; some of these signals may be interpreted in an application-dependent way. You can learn what signals a particular process recognizes by looking at its man-page and searching for a `SIGNALS` section.

SIGTERM and SIGINT

If you *want* to kill a process, you have several options. By default, kill sends SIGTERM, which is not identical to SIGINT of Control-C fame, but usually has the same results:

```
$ kill 16217
$ jobs -l
[1]- 16217 Terminated              xeyes -center red
[2]+ 16224 Stopped (signal)        xeyes -center blue
```

The big kill

Processes can ignore both SIGTERM and SIGINT, either by choice or because they are stopped or somehow "stuck." In these cases it may be necessary to use the big hammer, the SIGKILL signal. A process cannot ignore SIGKILL:

```
$ kill 16224
$ jobs -l
[2]+ 16224 Stopped (signal)        xeyes -center blue
$ kill -s SIGKILL
$ jobs -l
[2]+ 16224 Interrupt               xeyes -center blue
```

nohup

The terminal where you start a job is called the job's controlling terminal. Some shells (not `bash` by default), will deliver a SIGHUP signal to backgrounded jobs when you logout, causing them to quit. To protect processes from this behavior, use the `nohup` when you start the process:

```
$ nohup make &
$ exit
```

Using ps to list processes

The `jobs` command we were using earlier only lists processes that were started from your `bash` session. To see all the processes on your system, use `ps` with the `a` and `x` options together:

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:04 init [3]
    2 ?        SW     0:11 [keventd]
    3 ?        SWN    0:13 [ksoftirqd_CPU0]
    4 ?        SW     2:33 [kswapd]
    5 ?        SW     0:00 [bdflush]
```

I've only listed the first few because it is usually a very long list. This gives you a snapshot of what the whole machine is doing, but is a lot of information to sift through. If you were to leave off the `ax`, you would see only processes that are owned by you, and that have a controlling terminal. The command `ps x` would show you all your processes, even those without a controlling terminal. If you were to use `ps a`, you would get the list of everybody's processes that are attached to a terminal.

Seeing the forest and the trees

You can also list different information about each process. The `--forest` option makes it easy to see the process hierarchy, which will give you an indication of how the various processes on your system interrelate. When a process starts a new process, that new process is called a "child" process. In a `--forest` listing, parents appear on the left, and children appear as branches to the right:

```
$ ps x --forest
  PID TTY      STAT   TIME COMMAND
  927 pts/1    S      0:00 bash
 6690 pts/1    S      0:00  \_ bash
26909 pts/1    R      0:00      \_ ps x --forest
19930 pts/4    S      0:01 bash
25740 pts/4    S      0:04  \_ vi processes.txt
```

The "u" and "l" ps options

The "u" or "l" options can also be added to any combination of "a" and "x" in order to include more information about each process:

```
$ ps au
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
agriffis   403  0.0  0.0  2484   72 tty1     S     2001   0:00 -bash
chouser    404  0.0  0.0  2508   92 tty2     S     2001   0:00 -bash
root       408  0.0  0.0  1308  248 tty6     S     2001   0:00 /sbin/agetty 3
agriffis   434  0.0  0.0  1008    4 tty1     S     2001   0:00 /bin/sh /usr/X
chouser    927  0.0  0.0  2540   96 pts/1    S     2001   0:00 bash
```

```
$ ps al
  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
100  1001   403     1   9   0  2484   72 wait4  S    tty1       0:00 -bash
100  1000   404     1   9   0  2508   92 wait4  S    tty2       0:00 -bash
000     0   408     1   9   0  1308  248 read_c S    tty6       0:00 /sbin/ag
000  1001   434   403   9   0  1008    4 wait4  S    tty1       0:00 /bin/sh
000  1000   927   652   9   0  2540   96 wait4  S    pts/1      0:00 bash
```

Using "top"

If you find yourself running `ps` several times in a row, trying to watch things change, what you probably want is `top`. `top` displays a continuously updated process listing, along with some useful summary information:

```
$ top
 10:02pm  up 19 days,  6:24,  8 users,  load average: 0.04, 0.05, 0.00
75 processes: 74 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  1.3% user,  2.5% system,  0.0% nice, 96.0% idle
Mem:   256020K av,  226580K used,   29440K free,       0K shrd,    3804K buff
Swap:  136544K av,   80256K used,   56288K free                  101760K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
  628 root      16   0  213M  31M  2304 S       0  1.9 12.5  91:43 X
26934 chouser   17   0  1272 1272  1076 R       0  1.1  0.4   0:00 top
  652 chouser   11   0 12016 8840  1604 S       0  0.5  3.4   3:52 gnome-termin
  641 chouser    9   0  2936 2808  1416 S       0  0.1  1.0   2:13 sawfish
```

nice

Each processes has a priority setting that Linux uses to determine how CPU timeslices are shared. You can set the priority of a process by starting it with the `nice` command:

```
$ nice -n 10 oggenc /tmp/song.wav
```

Since the priority setting is called `nice`, it should be easy to remember that a higher value will be nice to other processes, allowing them to get priority access to the CPU. By default, processes are started with a setting of `0`, so the setting of `10` above means `oggenc` will readily give up the CPU to other processes. Generally, this means that `oggenc` will allow other processes to run at their normal speed, regardless of how CPU-hungry `oggenc` happens to be. You can see these niceness levels under the NI column in the `ps` and `top` listings above.

renice

The `nice` command can only change the priority of a process when you start it. If you want to change the niceness setting of a running process, use `renice`:

```
$ ps l 641
  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
000  1000   641     1   9   0  5876 2808 do_sel S    ?          2:14 sawfish
$ renice 10 641
641: old priority 0, new priority 10
$ ps l 641
  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
000  1000   641     1   9  10  5876 2808 do_sel S    ?          2:14 sawfish
```

Section 5. Text processing

Redirection revisited

Earlier in this tutorial series, we saw an example of how to use the > operator to redirect the output of a command to a file, as follows:

```
$ echo "firstfile" > copyme
```

In addition to redirecting output to a file, we can also take advantage of a powerful shell feature called pipes. Using pipes, we can pass the output of one command to the input of another command. Consider the following example:

```
$ echo "hi there" | wc
      1       2       9
```

The | character is used to connect the output of the command on the left to the input of the command on the right. In the example above, the echo command prints out the string `hi there` followed by a linefeed. That output would normally appear on the terminal, but the pipe redirects it into the wc command, which displays the number of lines, words, and characters in its input.

A pipe example

Here is another simple example:

```
$ ls -s | sort -n
```

In this case, `ls -s` would normally print a listing of the current directory on the terminal, preceding each file with its size. But instead we've piped the output into `sort -n`, which sorts the output numerically. This is a really useful way to find large files in your home directory!

The following examples are more complex, but they demonstrate the power that can be harnessed using pipes. We're going to throw out some commands we haven't covered yet, but don't let that slow you down. Concentrate instead on understanding how pipes work so you can employ them in your daily Linux tasks.

The decompression pipeline

Normally to decompress and untar a file, you might do the following:

```
$ bzip2 -d linux-2.4.16.tar.bz2
$ tar xvf linux-2.4.16.tar
```

The downside of this method is that it requires the creation of an intermediate, uncompressed file on your disk. Since `tar` has the ability to read directly from its input (instead of specifying a file), we could produce the same end-result using a pipeline:

```
$ bzip2 -dc linux-2.4.16.tar.bz2 | tar xvf -
```

Woo hoo! Our compressed tarball has been extracted and we didn't need an intermediate file.

A longer pipeline

Here's another pipeline example:

```
$ cat myfile.txt | sort | uniq | wc -l
```

We use cat to feed the contents of `myfile.txt` to the sort command. When the sort command receives the input, it sorts all input lines so that they are in alphabetical order, and then sends the output to `uniq`. `uniq` removes any duplicate lines (and requires its input to be sorted, by the way,) sending the scrubbed output to `wc -l`. We've seen the `wc` command earlier, but without command-line options. When given the `-l` option, it only prints the number of lines in its input, instead of also including words and characters. You'll see that this pipeline will print out the number of unique (non-identical) lines in a text file.

Try creating a couple of test files with your favorite text editor and use this pipeline to see what results you get.

The text processing whirlwind begins

Now we embark on a whirlwind tour of the standard Linux text processing commands. Because we're covering a lot of material in this tutorial, we don't have the space to provide examples for every command. Instead, we encourage you to read each command's man page (by typing `man echo`, for example) and learn how each command and it's options work by spending some time playing with each one. As a rule, these commands print the results of any text processing to the terminal rather than modifying any specified files. After we take our whirlwind tour of the standard Linux text processing commands, we'll take a closer look at output and input redirection. So yes, there is light at the end of the tunnel :)

`echo`

`echo` prints its arguments to the terminal. Use the `-e` option if you want to embed backslash escape sequences; for example `echo -e "foo\nfoo"` will print `foo`, then a newline, and then `foo` again. Use the `-n` option to tell echo to omit the trailing newline that is appended to the output by default.

cat, sort, and uniq

`cat`

`cat` will print the *contents* of the files specified as arguments to the terminal. Handy as the first command of a pipeline, for example, `cat foo.txt | blah`.

`sort`

`sort` will print the contents of the file specified on the command line in alphabetical order. Of course, sort also accepts piped input. Type `man sort` to familiarize yourself with its various options that control sorting behavior.

`uniq`

`uniq` takes an *already-sorted* file or stream of data (via a pipeline) and removes duplicate lines.

wc, head, and tail

`wc`

`wc` prints out the number of lines, words, and bytes in the specified file or in the input stream (from a pipeline). Type `man wc` to learn how to fine-tune what counts are displayed.

`head`

Prints out the first ten lines of a file or stream. Use the `-n` option to specify how many lines should be displayed.

`tail`

Prints out the last ten lines of a file or stream. Use the `-n` option to specify how many lines should be displayed.

tac, expand, and unexpand

`tac`

`tac` is like `cat`, but prints all lines in reverse order; in other words, the last line is printed first.

`expand`

Convert input tabs to spaces. Use the `-t` option to specify the tabstop.

`unexpand`

Convert input spaces to tabs. Use the `-t` option to specify the tabstop.

cut, nl, and pr

`cut`

`cut` is used to extract character-delimited fields from each line of an input file or stream.

`nl`

The `nl` command adds a line number to every line of input. Useful for printouts.

`pr`

`pr` is used to break files into multiple pages of output; typically used for printing.

tr, awk, and sed

`vr`

`tr` is a character translation tool; it's used to map certain characters in the input stream to certain other characters in the output stream.

`sed`

`sed` is a powerful stream-oriented text editor. You can learn more about `sed` in the following IBM developerWorks articles:

Sed by example, Part 1

Sed by example, Part 2

Sed by example, Part 3

If you're planning to take the LPI exam, be sure to read the first two articles of this series.

`awk`

`awk` is a handy line-oriented text-processing language. To learn more about `awk`, read the following IBM developerWorks articles:

Awk by example, Part 1

Awk by example, Part 2

Awk by example, Part 3

od, split, and fmt

`od`

`od` is designed to transform the input stream into a octal or hex "dump" format.

`split`

`split` is a command used to split a larger file into many smaller-sized, more manageable chunks.

`fmt`

`fmt` will reformat paragraphs so that wrapping is done at the margin. These days it's less useful since this ability is built into most text editors, but it's still a good one to know.

Paste, join, and tee

paste

`paste` takes two or more files as input, concatenates each sequential line from the input files, and outputs the resulting lines. It can be useful to create tables or columns of text.

`join`

`join` is similar to paste, but it uses a field (by default the first) in each input line to match up what should be combined on a single line.

`tee`

The `tee` command will print its input both to a file and to the screen. This is useful when you want to create a log of something, but you also want to see it on the screen.

Whirlwind over! Redirection

Similar to using > on the `bash` command line, you can also use < to redirect a file *into* a command. For many commands, you can simply specify the filename on the command line, however some commands only work from standard input.

Bash and other shells support the concept of a "herefile." This allows you to specify the input to a command in the lines following the command invocation, terminating it with a sentinal value. This is easiest shown through an example:

```
$ sort <<END
apple
cranberry
banana
END
apple
banana
cranberry
```

In the example above, we typed the words `apple`, `cranberry` and `banana`, followed by "END" to signify the end of the input. The `sort` program then returned our words in alphabetical order.

Using >>

You would expect >> to be somehow analogous to <<, but it isn't really. It simply means to append the output to a file, rather than overwrite as > would. For example:

```
$ echo Hi > myfile
$ echo there. > myfile
$ cat myfile
there.
```

Oops! We lost the "Hi" portion! What we meant was this:

```
$ echo Hi > myfile
$ echo there. >> myfile
$ cat myfile
Hi
there.
```

Much better!

Section 6. Kernel modules

Meet "uname"

The uname command provides a variety of interesting information about your system. Here's what is displayed on my development workstation when I type "uname -a" which tells the "uname" command to print out all of its information in one feel swoop:

```
$ uname -a
Linux inventor 2.4.20-gaming-r1 #1 Fri Apr 11 18:33:35 MDT 2003 i686 AMD Athlon(tm) XP
```

More uname madness

Now, let's look at the information that "uname" provides, in table form:

```
info. option                 arg example
kernel name                 -s "Linux"
hostname                 -n "inventor"
kernel release                 -r "2.4.20-gaming-r1"
kernel version                 -v "#1 Fri Apr 11 18:33:35 MDT 2003"
machine                         -m "i686"
processor                 -p "AMD Athlon(tm) XP 2100+"
hardware platform         -i "AuthenticAMD"
operating system         -o "GNU/Linux"
```

Intriguing! What does your "uname -a" command print out?

The kernel release

Here's a magic trick. First, type "uname -r" to have the uname command print out the release of the Linux kernel that's currently running.

Now, look in the /lib/modules directory and --presto!-- I bet you'll find a directory with that exact name! OK, not quite magic, but now may be a good time to talk about the significance of the directories in /lib/modules and explain what kernel modules are.

The kernel

The Linux kernel is the heart of what is commonly referred to as "Linux" -- it's the piece of code that accesses your hardware directly and provides abstractions so that regular old programs can run. Thanks to the kernel, your text editor doesn't need to worry about whether it is writing to a SCSI or IDE disk -- or even a RAM disk. It just writes to a filesystem, and the kernel takes care of the rest.

Introducing kernel modules

So, what are kernel *modules*? Well, they're parts of the kernel that have been stored in a special format on disk. At your command, they can be loaded into the running kernel and provide additional functionality.

Because the kernel modules are loaded on demand, you can have your kernel support a lot of additional functionality that you may not ordinarily want to be enabled. But once in a blue moon, those kernel modules are likely to come in quite handy and can be loaded -- often automatically -- to support that odd filesystem or hardware device that you rarely use.

Kernel modules in a nutshell

In sum, kernel modules allow for the running kernel to enable capabilities on an on-demand basis. Without kernel modules, you'd have to compile a completely new kernel and reboot in order for it to support something new.

lsmod

To see what modules are currently loaded on your system, use the "lsmod" command:

```
# lsmod
Module                  Size  Used by    Tainted: PF
vmnet                  20520   5
vmmon                  22484  11
nvidia               1547648  10
mousedev                3860   2
hid                    16772   0  (unused)
usbmouse                1848   0  (unused)
input                   3136   0  [mousedev hid usbmouse]
usb-ohci               15976   0  (unused)
ehci-hcd               13288   0  (unused)
emu10k1                64264   2
ac97_codec              9000   0  [emu10k1]
sound                  51508   0  [emu10k1]
usbcore                55168   1  [hid usbmouse usb-ohci ehci-hcd]
```

Modules listing, part 1

As you can see, my system has quite a few modules loaded. the vmnet and vmmon modules provide necessary functionality for my *VMWare* program, which allows me to run a virtual PC in a window on my desktop. The "nvidia" module comes from *NVIDIA corporation* and allows me to use my high-performance 3D-accelerated graphics card under Linux whilst taking advantage of its many neat features.

Modules listing, part 2

Then I have a bunch of modules that are used to provide support for my USB-based input devices -- namely "mousedev," "hid," "usbmouse," "input," "usb-ohci," "ehci-hcd" and "usbcore." It often makes sense to configure your kernel to provide USB support as modules. Why? Because USB devices are "plug and play," and when you have your USB support in modules, then you can go out and buy a new USB device, plug it in to your system, and have the system automatically load the appropriate modules to enable that device. It's a handy way to do things.

Third-party modules

Rounding out my list of modules are "emu10k1," "ac97_codec," and "sound," which together provide support for my SoundBlaster Audigy sound card.

It should be noted that some of my kernel modules come from the kernel sources themselves. For example, all the USB-related modules are compiled from the standard Linux kernel sources. However, the nvidia, emu10k1 and VMWare-related modules come from other sources. This highlights another major benefit of kernel modules -- allowing third parties to provide much-needed kernel functionality and allowing this functionality to "plug in" to a running Linux kernel. No reboot necessary.

depmod and friends

In my /lib/modules/2.4.20-gaming-r1/ directory, I have a number of files that start with the string "modules.":

```
$ ls /lib/modules/2.4.20-gaming-r1/modules.*
/lib/modules/2.4.20-gaming-r1/modules.dep
/lib/modules/2.4.20-gaming-r1/modules.generic_string
/lib/modules/2.4.20-gaming-r1/modules.ieee1394map
/lib/modules/2.4.20-gaming-r1/modules.isapnpmap
/lib/modules/2.4.20-gaming-r1/modules.parportmap
/lib/modules/2.4.20-gaming-r1/modules.pcimap
/lib/modules/2.4.20-gaming-r1/modules.pnpbiosmap
/lib/modules/2.4.20-gaming-r1/modules.usbmap
```

These files contain some lots of dependency information. For one, they record *dependency* information for modules -- some modules require other modules to be loaded first before they will run. This information is recorded in these files.

How you get modules

Some kernel modules are designed to work with specific hardware devices, like my "emu10k1" module which is for my SoundBlaster Audigy card. For these types of modules, these files also record the PCI IDs and similar identifying marks of the hardware devices that they support. This information can be used by things like the "hotplug" scripts (which we'll take a look at in later tutorials) to auto-detect hardware and load the appropriate module to support said hardware automatically.

Using depmod

If you ever install a new module, this dependency information may become out of date. To make it fresh again, simply type "depmod -a". The "depmod" program will then scan all the modules in your directories in /lib/modules and freshening the dependency information. It does this by scanning the module files in /lib/modules and looking at what are called "symbols" inside the modules:

```
# depmod -a
```

Locating kernel modules

So, what do kernel modules look like? For 2.4 kernels, they're typically any file in the /lib/modules tree that ends in ".o". To see all the modules in /lib/modules, type the following:

```
# find /lib/modules -name '*.o'
/lib/modules/2.4.20-gaming-r1/misc/vmmon.o
/lib/modules/2.4.20-gaming-r1/misc/vmnet.o
/lib/modules/2.4.20-gaming-r1/video/nvidia.o
/lib/modules/2.4.20-gaming-r1/kernel/fs/fat/fat.o
/lib/modules/2.4.20-gaming-r1/kernel/fs/vfat/vfat.o
/lib/modules/2.4.20-gaming-r1/kernel/fs/minix/minix.o
[listing "snipped" for brevity]
```

insmod vs. modprobe

So, how does one load a module into a running kernel? One way is to use the "insmod" command and specifying the full path to the module that you wish to load:

```
# insmod /lib/modules/2.4.20-gaming-r1/kernel/fs/fat/fat.o
# lsmod | grep fat
fat                    29272   0  (unused)
```

However, one normally loads modules by using the "modprobe" command. One of the nice things about the "modprobe" command is that it automatically takes care of loading any dependent modules. Also, one doesn't need to specify the path to the module you wish to load, nor does one specify the trailing ".o".

rmmod and modprobe in action

Let's unload our "fat.o" module and load it using "modprobe":

```
# rmmod fat
# lsmod | grep fat
# modprobe fat
# lsmod | grep fat
fat                    29272   0  (unused)
```

As you can see, the "rmmod" command works similarly to modprobe, but has the opposite effect -- it unloads the module you specify.

Your friend modinfo and modules.conf

You can use the "modinfo" command to learn interesting things about your favorite modules:

```
# modinfo fat
filename:    /lib/modules/2.4.20-gaming-r1/kernel/fs/fat/fat.o
description: <none>
author:      <none>
license:     "GPL"
```

And make special note of the /etc/modules.conf file. This file contains configuration information for modprobe. It allows you to tweak the functionality of modprobe by telling it to load modules before/after loading others, run scripts before/after modules load, and more.

modules.conf gotchas

The syntax and functionality of modules.conf is quite complicated, and we won't go into its syntax now (type "man modules.conf" for all the gory details), but here are some things that you *should* know about this file.

For one, many distributions generate this file automatically from a bunch of files in another directory, like /etc/modules.d/. For example, Gentoo Linux has an /etc/modules.d/ directory, and running the update-modules command will take every file in /etc/modules.d/ and concatenate them to produce a new /etc/modules.conf. Therefore, make your changes to the files in /etc/modules.d/ and run update-modules if you are using Gentoo. If you are using Debian, the procedure is similar except that the directory is called /etc/modutils/.

Section 7. Summary and resources

Summary

Congratulations; you've reached the end of this tutorial on basic Linux administration! We hope that it has helped you to firm up your foundational Linux knowledge. Please join us in our next tutorial covering intermediate administration, where we will build on the foundation laid here, covering topics like the Linux permissions and ownership model, user account management, filesystem creation and mounting, and more. And remember, by continuing in this tutorial series, you'll soon be ready to attain your LPIC Level 1 Certification from the Linux Professional Institute.

Resources

Speaking of LPIC certification, if this is something you're interested in, then we recommend that you study the following resources, which have been carefully selected to augment the material covered in this tutorial:

There are a number of good regular expression resources on the 'net. Here are a couple that we've found:

- *Regular Expressions Reference*
- *Regular Expressions Explained*

Be sure to read up on the Filesystem Hierarchy Standard at *http://www.pathname.com/fhs/*.

In the *Bash by example* article series, I show you how to use `bash` programming constructs to write your own `bash` scripts. This series (particularly parts one and two) will be good preparation for the LPIC Level 1 exam:

- *Bash by example, part 1: Fundamental programming in the Bourne-again shell*
- *Bash by example, part 2: More bash programming fundamentals*
- *Bash by example, part 3: Exploring the ebuild system*

You can learn more about `sed` in the following IBM developerWorks articles:

Sed by example, Part 1

Sed by example, Part 2

Sed by example, Part 3

If you're planning to take the LPI exam, be sure to read the first two articles of this series.

To learn more about `awk`, read the following IBM developerWorks articles:

Awk by example, Part 1

Awk by example, Part 2

Awk by example, Part 3

We highly recommend the *Technical FAQ for Linux users*, a 50-page in-depth list of frequently-asked Linux questions, along with detailed answers. The FAQ itself is in PDF (Acrobat) format. If you're a beginning or intermediate Linux user, you really owe it to yourself to check this FAQ out.

If you're not too familiar with the `vi` editor, I strongly recommend that you check out my *Vi -- the cheat sheet method tutorial*. This tutorial will give you a gentle yet fast-paced introduction to this powerful text editor. Consider this must-read material if you don't know how to use `vi`.

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered in *developerWorks* tutorials.

For questions about the content of this tutorial, contact the authors:

- Daniel Robbins, at *drobbins@gentoo.org*
- Chris Houser, at *chouser@gentoo.org*
- Aron Griffis, at *agriffis@gentoo.org*

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.

Common threads: Sed by example, Part 1

Get to know the powerful UNIX editor

Daniel Robbins (drobbins@gentoo.org)
President/CEO, Gentoo Technologies, Inc.
01 Sep 2000

In this series of articles, Daniel Robbins will show you how to use the very powerful (but often forgotten) UNIX stream editor, sed. Sed is an ideal tool for batch-editing files or for creating shell scripts to modify existing files in powerful ways.

Contents:

Pick an editor
Enter sed
GNU sed
The newest GNU sed
The right sed
Sed examples
Another sed example
Address ranges
Addresses with regular expressions
Regular expression refresher
More on addresses
C source example
Next time
Resources
About the author
Rate this article

Pick an editor

In the UNIX world, we have a lot of options when it comes to editing files. Think of it -- vi, emacs, and jed come to mind, as well as many others. We all have our favorite editor (along with our favorite keybindings) that we have come to know and love. With our trusty editor, we are ready to tackle any number of UNIX-related administration or programming tasks with ease.

While interactive editors are great, they do have limitations. Though their interactive nature can be a strength, it can also be a weakness. Consider a situation where you need to perform similar types of changes on a group of files. You could instinctively fire up your favorite editor and perform a bunch of mundane, repetitive, and time-consuming edits by hand. But there's a better way.

Enter sed

It would be nice if we could automate the process of making edits to files, so that we could "batch" edit files, or even write scripts with the ability to perform sophisticated changes to existing files. Fortunately for us, for these types of situations, there is a better way -- and the better way is called "sed".

sed is a lightweight stream editor that's included with nearly all UNIX flavors, including Linux. sed has a lot of nice features. First of all, it's very lightweight, typically many times smaller than your favorite scripting language. Secondly, because sed is a *stream* editor, it can perform edits to data it receives from stdin, such as from a pipeline. So, you don't need to have the data to be edited stored in a file on disk. Because data can just as easily be piped to sed, it's very easy to use sed as part of a long, complex pipeline in a powerful shell script. Try doing that with your favorite editor.

GNU sed

Fortunately for us Linux users, one of the nicest versions of sed out there happens to be GNU sed, which is currently at version 3.02. Every Linux distribution has GNU sed, or at least should. GNU sed is popular not only because its sources are freely distributable, but because it happens to have a lot of handy, time-saving extensions to the POSIX sed standard. GNU sed also doesn't suffer from many of the limitations that earlier and proprietary versions of sed had, such as a limited line length -- GNU sed handles lines of any length with ease.

The newest GNU sed

While researching this article, I noticed that several online sed aficionados made reference to a GNU sed 3.02a. Strangely, I couldn't find sed 3.02a on **ftp.gnu.org** (see Resources for these links), so I had to go look for it elsewhere. I found it at **alpha.gnu.org**, in /pub/sed. I happily downloaded it, compiled it, and installed it, only to find minutes later that the most recent version of sed is 3.02.80 -- and you can find its sources right next to those for 3.02a, at **alpha.gnu.org**. After getting GNU sed 3.02.80 installed, I was finally ready to go.

The right sed

In this series, we will be using GNU sed 3.02.80. Some (but very few) of the most advanced examples you'll find in my upcoming, follow-on articles in this series will not work with GNU sed 3.02 or 3.02a. If you're using a non-GNU sed, your results may vary. Why not take some time to install GNU sed 3.02.80 now? Then, not only will you be ready for the rest of the series, but you'll also be able to use arguably the best sed in existence!

alpha.gnu.org
alpha.gnu.org (see Resources) is the home of new and experimental GNU source code. However, you'll also find a lot of nice, stable source code there, too. For some reason a lot of the GNU developers either forget to move stable sources over to ftp.gnu.org, or they have unusually long (2 years!) "beta" periods. As an example, sed 3.02a is two years old, and even 3.02.80 is one year old, but they're still (at the time this article was written, August 2000) not available on ftp.gnu.org!

Sed examples

Sed works by performing any number of user-specified editing operations ("commands") on the input data. Sed is line-based, so the commands are performed on each line in order. And, sed writes its results to standard output (stdout); it doesn't modify any input files.

Let's look at some examples. The first several are going to be a bit weird because I'm using them to illustrate how sed works rather than to perform any useful task. However, if you're new to sed, it's very important that you understand them. Here's our first example:

```
$ sed -e 'd' /etc/services
```

If you type this command, you'll get absolutely no output. Now, what happened? In this example, we called sed with one editing command, 'd. Sed opened the /etc/services file, read a line into its pattern buffer, performed our editing command ("delete line"), and then printed the pattern buffer (which was empty). It then repeated these steps for each successive line. This produced no output, because the "d" command zapped every single line in the pattern buffer!

There are a couple of things to notice in this example. First, /etc/services was not modified at all. This is because, again, sed only reads from the file you specify on the command line, using it as input -- it doesn't try to modify the file. The second thing to notice is that sed is line-oriented. The 'd' command didn't simply tell sed to delete all incoming data in one fell swoop. Instead, sed read each line of /etc/services one by one into its internal buffer, called the pattern buffer. Once a line was read into the pattern buffer, it performed the 'd' command and printed the contents of the pattern buffer (nothing in this example). Later, I'll show you how to use address ranges to control which lines a command is applied to -- but in the absence of addresses, a command is applied to *all lines*.

The third thing to notice is the use of single quotes to surround the 'd' command. It's a good idea to get into the habit of using single quotes to surround your sed commands, so that shell expansion is disabled.

Another sed example

Here's an example of how to use sed to remove the first line of the /etc/services file from our output stream:

```
$ sed -e '1d' /etc/services | more
```

As you can see, this command is very similar to our first 'd' command, except that it is preceded by a '1'. If you guessed that the '1' refers to line number one, you're right. While in our first example, we used 'd' by itself, this time we use the 'd' command preceded by an optional numerical address. By using addresses, you can tell sed to perform edits only on a particular line or lines.

Address ranges

Now, let's look at how to specify an address *range*. In this example, sed will delete lines 1-10 of the output:

```
$ sed -e '1,10d' /etc/services | more
```

When we separate two addresses by a comma, sed will apply the following command to the range that starts with the first address, and ends with the second address. In this example, the 'd' command was applied to lines 1-10, inclusive. All other lines were ignored.

Addresses with regular expressions

Now, it's time for a more useful example. Let's say you wanted to view the contents of your /etc/services file, but you aren't interested in viewing any of the included comments. As you know, you can place comments in your /etc/services file by starting the line with the '#' character. To avoid comments, we'd like sed to delete lines that start with a '#'. Here's how to do it:

```
$ sed -e '/^#/d' /etc/services | more
```

Try this example and see what happens. You'll notice that sed performs its desired task with flying colors. Now, let's figure out what happened.

To understand the '/^#/d' command, we first need to dissect it. First, let's remove the 'd' -- we're using the same delete line command that we've used previously. The new addition is the '/^#/' part, which is a new kind of *regular expression* address. Regular expression addresses are always surrounded by slashes. They specify a *pattern*, and the command that immediately follows a regular expression address will only be applied to a line if it happens to match this particular pattern.

So, '/^#/' is a regular expression. But what does it do? Obviously, this would be a good time for a regular expression refresher.

Regular expression refresher

We can use regular expressions to express patterns that we may find in the text. If you've ever used the '*' character on the shell command line, you've used something that's similar, but not identical to, regular expressions. Here are the special characters that you can use in regular expressions:

Character	Description
^	Matches the beginning of the line
$	Matches the end of the line

.	Matches any single character
*	Will match zero or more occurrences of the *previous* character
[ ]	Matches all the characters inside the [ ]

Probably the best way to get your feet wet with regular expressions is to see a few examples. All of these examples will be accepted by sed as valid addresses to appear on the left side of a command. Here are a few:

Regular expression	**Description**
/./	Will match any line that contains at least one character
/../	Will match any line that contains at least two characters
/^#/	Will match any line that begins with a '#'
/^$/	Will match all blank lines
/}^/	Will match any lines that ends with '}' (no spaces)
/} *^/	Will match any line ending with '}' followed by *zero* or more spaces
/[abc]/	Will match any line that contains a lowercase 'a', 'b', or 'c'
/^[abc]/	Will match any line that *begins* with an 'a', 'b', or 'c'

I encourage you to try several of these examples. Take some time to get familiar with regular expressions, and try a few regular expressions of your own creation. You can use a regexp this way:

```
$ sed -e '/regexp/d' /path/to/my/test/file | more
```

This will cause sed to delete any matching lines. However, it may be easier to get familiar with regular expressions by telling sed to *print* regexp matches, and delete non-matches, rather than the other way around. This can be done with the following command:

```
$ sed -n -e '/regexp/p' /path/to/my/test/file | more
```

Note the new '-n' option, which tells sed to not print the pattern space unless explicitly commanded to do so. You'll also notice that we've replaced the 'd' command with the 'p' command, which as you might guess, explicitly commands sed to print the pattern space. Voila, now only matches will be printed.

More on addresses

Up till now, we've taken a look at line addresses, line range addresses, and regexp addresses. But there are even more possibilities. We can specify two regular expressions separated by a comma, and sed will match all lines starting from the first line that matches the first regular expression, up to and including the line that matches the second regular expression. For example, the following command will print out a block of text that begins with a line containing "BEGIN", and ending with a line that contains "END":

```
$ sed -n -e '/BEGIN/,/END/p' /my/test/file | more
```

If "BEGIN" isn't found, no data will be printed. And, if "BEGIN" is found, but no "END" is found on any line below it, all subsequent lines will be printed. This happens because of sed's stream-oriented nature -- it doesn't know whether or not an

"END" will appear.

C source example

If you want to print out only the main() function in a C source file, you could type:

```
$ sed -n -e '/main[[:space:]]*(/,/^}/p' sourcefile.c | more
```

This command has two regular expressions, '/main[[:space:]]*(/' and '/^}/', and one command, 'p'. The first regular expression will match the string "main" followed by any number of spaces or tabs, followed by an open parenthesis. This should match the start of your average ANSI C main() declaration.

In this particular regular expression, we encounter the '[[:space:]]' character class. This is simply a special keyword that tells sed to match either a TAB or a space. If you wanted, instead of typing '[[:space:]]', you could have typed '[', then a literal space, then Control-V, then a literal tab and a ']' -- The Control-V tells bash that you want to insert a "real" tab rather than perform command expansion. It's clearer, especially in scripts, to use the '[[:space:]]' command class.

OK, now on to the second regexp. '/^}' will match a '}' character that appears at the beginning of a new line. If your code is formatted nicely, this will match the closing brace of your main() function. If it's not, it won't -- one of the tricky things about performing pattern matching.

The 'p' command does what it always does, explicitly telling sed to print out the line, since we are in '-n' quiet mode. Try running the command on a C source file -- it should output the entire main() { } block, including the initial "main()" and the closing '}'.

Next time

Now that we've touched on the basics, we'll be picking up the pace for the next two articles. If you're in the mood for some meatier sed material, be patient -- it's coming! In the meantime, you might want to check out the following sed and regular expression resources.

Resources

- Read Daniel's other sed articles on *developerWorks*: Common threads: Sed by example, Part 2 and Part 3.
- Check out Eric Pement's excellent sed FAQ.
- You can find the sources to sed 3.02 at ftp://ftp.gnu.org/pub/gnu/sed.
- You'll find the nice, new sed 3.02.80 at alpha.gnu.org.
- Eric Pement also has a handy list of sed one-liners that any aspiring sed guru should definitely look at.
- If you'd like a good old-fashioned book, O'Reilly's sed & awk, 2nd Edition would be wonderful choice.
- Maybe you'd like to read 7th edition UNIX's sed man page (circa 1978!).
- Take Felix von Leitner's short sed tutorial.
- Read David Mertz's article on "Text processing in Python" on *developerWorks*.

- Brush up on using regular expressions to find and modify patterns in text in this free, dW-exclusive tutorial.
- See the regular expressions how-to document from Python.org.
- Refer to an overview of regular expressions from the University of Kentucky.

About the author

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of Gentoo Technologies, Inc., the creator of Gentoo Linux, an advanced Linux for the PC, and the Portage system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at SONY Electronic Publishing/Psygnosis. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah. You can contact Daniel at drobbins@gentoo.org.

PDF e-mail it!

Rate this article

This content was helpful to me:

○ Strongly disagree (1) ○ Disagree (2) ○ Neutral (3) ○ Agree (4) ○ Strongly agree (5)

Comments?

Submit feedback

Common threads: Sed by example, Part 2

How to further take advantage of the UNIX text editor

Daniel Robbins (drobbins@gentoo.org)
President/CEO, Gentoo Technologies, Inc.
01 Oct 2000

Sed is a very powerful and compact text stream editor. In this article, the second in the series, Daniel shows you how to use sed to perform string substitution; create larger sed scripts; and use sed's append, insert, and change line commands.

Contents:
Substitution!
Regexp snafus
More character matching
Advanced substitution stuff
Those wonderful backslashed parentheses
Mixing things up
Multiple commands for one address
Append, insert, and change line
Next time
Resources
About the author
Rate this article

Sed is a very useful (but often forgotten) UNIX stream editor. It's ideal for batch-editing files or for creating shell scripts to modify existing files in powerful ways. This article builds on my previous article introducing sed.

Substitution!

Let's look at one of sed's most useful commands, the substitution command. Using it, we can replace a particular string or matched regular expression with another string. Here's an example of the most basic use of this command:

```
$ sed -e 's/foo/bar/' myfile.txt
```

The above command will output the contents of myfile.txt to stdout, with the first occurrence of 'foo' (if any) on each line replaced with the string 'bar'. Please note that I said *first occurrence on each line*, though this is normally not what you want. Normally, when I do a string replacement, I want to perform it globally. That is, I want to replace *all* occurrences on every line, as follows:

```
$ sed -e 's/foo/bar/g' myfile.txt
```

The additional 'g' option after the last slash tells sed to perform a global replace.

Here are a few other things you should know about the 's///' substitution command. First, it is a command, and a command only; there are no addresses specified in any of the above examples. This means that the 's///' command can also be used with addresses to control what lines it will be applied to, as follows:

```
$ sed -e '1,10s/enchantment/entrapment/g' myfile2.txt
```

The above example will cause all occurrences of the phrase 'enchantment' to be replaced with the phrase 'entrapment', but only on lines one through ten, inclusive.

```
$ sed -e '/^$/,/^END/s/hills/mountains/g' myfile3.txt
```

This example will swap 'hills' for 'mountains', but only on blocks of text beginning with a blank line, and ending with a line beginning with the three characters 'END', inclusive.

Another nice thing about the 's///' command is that we have a lot of options when it comes to those '/' separators. If we're performing string substitution and the regular expression or replacement string has a lot of slashes in it, we can change the separator by specifying a different character after the 's'. For example, this will replace all occurrences of /usr/local with /usr:

```
$ sed -e 's:/usr/local:/usr:g' mylist.txt
```

In this example, we're using the colon as a separator. If you ever need to specify the separator character in the regular expression, put a backslash before it.

Regexp snafus

Up until now, we've only performed simple string substitution. While this is handy, we can also match a regular expression. For example, the following sed command will match a phrase beginning with '<' and ending with '>', and containing any number of characters inbetween. This phrase will be deleted (replaced with an empty string):

```
$ sed -e 's/<.*>//g' myfile.html
```

This is a good first attempt at a sed script that will remove HTML tags from a file, but it won't work well, due to a regular expression quirk. The reason? When sed tries to match the regular expression on a line, it finds the *longest* match on the line. This wasn't an issue in my previous sed article, because we were using the 'd' and 'p' commands, which would delete or print the entire line anyway. But when we use the 's///' command, it definitely makes a big difference, because the entire portion that the regular expression matches will be replaced with the target string, or in this case, deleted. This means that the above example will turn the following line:

```
<b>This</b> is what <b>I</b> meant.
```

into this:

```
meant.
```

rather than this, which is what we wanted to do:

```
This is what I meant.
```

Fortunately, there is an easy way to fix this. Instead of typing in a regular expression that says "a '<' character followed by any number of characters, and ending with a '>' character", we just need to type in a regexp that says "a '<' character followed by any number of non-'>' characters, and ending with a '>' character". This will have the effect of matching the shortest possible match, rather than the longest possible one. The new command looks like this:

```
$ sed -e 's/<[^>]*>//g' myfile.html
```

In the above example, the '[^>]' specifies a "non-'>'" character, and the '*' after it completes this expression to mean "zero or more non-'>' characters". Test this command on a few sample html files, pipe them to more, and review their results.

More character matching

The '[ ]' regular expression syntax has some more additional options. To specify a range of characters, you can use a '-' as long as it isn't in the first or last position, as follows:

```
'[a-x]*'
```

This will match zero or more characters, as long as all of them are 'a','b','c'...'v','w','x'. In addition, the '[:space:]' character class is available for matching whitespace. Here's a fairly complete list of available character classes:

Character class	Description
[:alnum:]	Alphanumeric [a-z A-Z 0-9]
[:alpha:]	Alphabetic [a-z A-Z]
[:blank:]	Spaces or tabs
[:cntrl:]	Any control characters
[:digit:]	Numeric digits [0-9]
[:graph:]	Any visible characters (no whitespace)
[:lower:]	Lower-case [a-z]
[:print:]	Non-control characters
[:punct:]	Punctuation characters
[:space:]	Whitespace
[:upper:]	Upper-case [A-Z]

[:xdigit:]	hex digits [0-9 a-f A-F]

It's advantageous to use character classes whenever possible, because they adapt better to nonEnglish speaking locales (including accented characters when necessary, etc.).

Advanced substitution stuff

We've looked at how to perform simple and even reasonably complex straight substitutions, but sed can do even more. We can actually refer to either parts of or the entire matched regular expression, and use these parts to construct the replacement string. As an example, let's say you were replying to a message. The following example would prefix each line with the phrase "ralph said: ":

```
$ sed -e 's/.*/ralph said: &/' origmsg.txt
```

The output will look like this:

```
ralph said: Hiya Jim,
ralph said:
ralph said: I sure like this sed stuff!
ralph said:
```

In this example, we use the '&' character in the replacement string, which tells sed to insert the entire matched regular expression. So, whatever was matched by '.*' (the largest group of zero or more characters on the line, or the entire line) can be inserted anywhere in the replacement string, even multiple times. This is great, but sed is even more powerful.

Those wonderful backslashed parentheses

Even better than '&', the 's///' command allows us to define *regions* in our regular expression, and we can refer to these specific regions in our replacement string. As an example, let's say we have a file that contains the following text:

```
foo bar oni
eeny meeny miny
larry curly moe
jimmy the weasel
```

Now, let's say we wanted to write a sed script that would replace "eeny meeny miny" with "Victor eeny-meeny Von miny", etc. To do this, first we would write a regular expression that would match the three strings, separated by spaces:

```
'.* .* .*'
```

There. Now, we will define regions by inserting backslashed parentheses around each region of interest:

```
'\(.*\) \(.*\) \(.*\)'
```

This regular expression will work the same as our first one, except that it will define three logical regions that we can refer to in our replacement string. Here's the final script:

```
$ sed -e 's/\(.*\) \(.*\) \(.*\)/Victor \1-\2 Von \3/' myfile.txt
```

As you can see, we refer to each parentheses-delimited region by typing '\x', where x is the number of the region, starting at one. Output is as follows:

```
Victor foo-bar Von oni
Victor eeny-meeny Von miny
Victor larry-curly Von moe
Victor jimmy-the Von weasel
```

As you become more familiar with sed, you will be able to perform fairly powerful text processing with a minimum of effort. You may want to think about how you'd have approached this problem using your favorite scripting language -- could you have easily fit the solution in one line?

Mixing things up

As we begin creating more complex sed scripts, we need the ability to enter more than one command. There are several ways to do this. First, we can use semicolons between the commands. For example, this series of commands uses the '=' command, which tells sed to print the line number, as well as the 'p' command, which explicitly tells sed to print the line (since we're in '-n' mode):

```
$ sed -n -e '=;p' myfile.txt
```

Whenever two or more commands are specified, each command is applied (in order) to every line in the file. In the above example, first the '=' command is applied to line 1, and then the 'p' command is applied. Then, sed proceeds to line 2, and repeats the process. While the semicolon is handy, there are instances where it won't work. Another alternative is to use two -e options to specify two separate commands:

```
$ sed -n -e '=' -e 'p' myfile.txt
```

However, when we get to the more complex append and insert commands, even multiple '-e' options won't help us. For complex multiline scripts, the best way is to put your commands in a separate file. Then, reference this script file with the -f options:

```
$ sed -n -f mycommands.sed myfile.txt
```

This method, although arguably less convenient, will always work.

Multiple commands for one address

Sometimes, you may want to specify multiple commands that will apply to a single address. This comes in especially handy when you are performing lots of 's///' to transform words or syntax in the source file. To perform multiple commands per address, enter your sed commands in a file, and use the '{ }' characters to group commands, as follows:

```
1,20{
  s/[Ll]inux/GNU\/Linux/g
  s/samba/Samba/g
  s/posix/POSIX/g
}
```

The above example will apply three substitution commands to lines 1 through 20, inclusive. You can also use regular expression addresses, or a combination of the two:

```
1,/^END/{
        s/[Ll]inux/GNU\/Linux/g
        s/samba/Samba/g
        s/posix/POSIX/g
  p
}
```

This example will apply all the commands between '{ }' to the lines starting at 1 and up to a line beginning with the letters "END", or the end of file if "END" is not found in the source file.

Append, insert, and change line

Now that we're writing sed scripts in separate files, we can take advantage of the append, insert, and change line commands. These commands will insert a line after the current line, insert a line before the current line, or replace the current line in the pattern space. They can also be used to insert multiple lines into the output. The insert line command is used as follows:

```
i\
This line will be inserted before each line
```

If you don't specify an address for this command, it will be applied to each line and produce output that looks like this:

```
This line will be inserted before each line
line 1 here
This line will be inserted before each line
line 2 here
This line will be inserted before each line
line 3 here
This line will be inserted before each line
line 4 here
```

If you'd like to insert multiple lines before the current line, you can add additional lines by appending a backslash to the previous line, like so:

```
i\
insert this line\
and this one\
and this one\
and, uh, this one too.
```

The append command works similarly, but will insert a line or lines after the current line in the pattern space. It's used as follows:

```
a\
insert this line after each line.  Thanks! :)
```

On the other hand, the "change line" command will actually *replace* the current line in the pattern space, and is used as follows:

```
c\
You're history, original line! Muhahaha!
```

Because the append, insert, and change line commands need to be entered on multiple lines, you'll want to type them in to text sed scripts and tell sed to source them by using the '-f' option. Using the other methods to pass commands to sed will result in problems.

Next time

Next time, in the final article of this series on sed, I'll show you lots of excellent real-world examples of using sed for many different kinds of tasks. Not only will I show you what the scripts do, but *why* they do what they do. After you're done, you'll have additional excellent ideas of how to use sed in your various projects. I'll see you then!

Resources

- Read Daniel's other sed articles on *developerWorks*: Common threads: Sed by example, Part 1 and Part 3.
- Check out Eric Pement's excellent sed faq.
- You can find the sources to sed 3.02 at ftp.gnu.org.
- You'll find the nice, new sed 3.02.80 at alpha.gnu.org.
- Eric Pement also has a handy list of sed one-liners that any aspiring sed guru should definitely take a look at.
- If you'd like a good old-fashioned book, O'Reilly's sed & awk, 2nd Edition would be wonderful choice.
- Maybe you'd like to read 7th edition UNIX's sed man page (circa 1978!).
- Take Felix von Leitner's short sed tutorial.

- Brush up on using regular expressions to find and modify patterns in text in this free, dW-exclusive tutorial.

About the author

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of Gentoo Technologies, Inc., the creator of Gentoo Linux, an advanced Linux for the PC, and the Portage system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at SONY Electronic Publishing/Psygnosis. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah. You can contact Daniel at drobbins@gentoo.org.

Common threads: Sed by example, Part 3

Taking it to the next level: Data crunching, sed style

Daniel Robbins (drobbins@gentoo.org)
President/CEO, Gentoo Technologies, Inc.
01 Nov 2000

In this conclusion of the sed series, Daniel Robbins gives you a true taste of the power of sed. After introducing a handful of essential sed scripts, he'll demonstrate some radical sed scripting by converting a Quicken .QIF file into a text-readable format. This conversion script is not only functional, it also serves as an excellent example of sed scripting power.

Contents:
Muscular sed
Text translation
Reversing lines
Reversal explained
sed QIF magic
A tale of two formats
Starting the process
Refinement
Finishing touches
Don't get confuSed
Resources
About the author
Rate this article

Muscular sed

In my second sed article, I offered examples that demonstrated how sed works, but very few of these examples actually did anything particularly *useful*. In this final sed article, it's time to change that pattern and put sed to good use. I'll show you several excellent examples that not only demonstrate the power of sed, but also do some really neat (and handy) things. For example, in the second half of the article, I'll show you how I designed a sed script that converts a .QIF file from Intuit's Quicken financial program into a nicely formatted text file. Before doing that, we'll take a look at some less complicated yet useful sed scripts.

Text translation

Our first practical script converts UNIX-style text to DOS/Windows format. As you probably know, DOS/Windows-based text files have a CR (carriage return) and LF (line feed) at the end of each line, while UNIX text has only a line feed. There may be times when you need to move some UNIX text to a Windows system, and this script will perform the necessary format conversion for you.

```
$ sed -e 's/$/\r/' myunix.txt > mydos.txt
```

In this script, the '$' regular expression will match the end of the line, and the '\r' tells sed to insert a carriage return right before it. Insert a carriage return before a line feed, and presto, a CR/LF ends each line. Please note that the '\r' will be replaced with a CR only when using GNU sed 3.02.80 or later. If you haven't installed GNU sed 3.02.80 yet, see my first sed article for instructions on how to do this.

I can't tell you how many times I've downloaded some example script or C code, only to find that it's in DOS/Windows format. While many programs don't mind DOS/Windows format CR/LF text files, several programs definitely do -- the most notable being bash, which chokes as soon as it encounters a carriage return. The following sed invocation will convert DOS/Windows format text to trusty UNIX format:

```
$ sed -e 's/.$//' mydos.txt > myunix.txt
```

The way this script works is simple: our substitution regular expression matches the last character on the line, which happens to be a carriage return. We replace it with nothing, causing it to be deleted from the output entirely. If you use this script and notice that the last character of every line of the output has been deleted, you've specified a text file that's already in UNIX format. No need for that!

Reversing lines

Here's another handy little script. This one will reverse lines in a file, similar to the "tac" command that's included with most Linux distributions. The name "tac" may be a bit misleading, because "tac" doesn't reverse the position of characters on the line (left and right), but rather the position of lines in the file (up and down). Tacing the following file:

```
foo
bar
oni
```

....produces the following output:

```
oni
bar
foo
```

We can do the same thing with the following sed script:

```
$ sed -e '1!G;h;$!d' forward.txt > backward.txt
```

You'll find this sed script useful if you're logged in to a FreeBSD system, which doesn't happen to have a "tac" command. While handy, it's also a good idea to know why this script does what it does. Let's dissect it.

Reversal explained

First, this script contains three separate sed commands, separated by semicolons: '1!G', 'h' and '$!d'. Now, it's time to get an good understanding of the addresses used for the first and third commands. If the first command were '1G', the 'G' command would be applied only to the first line. However, there is an additional '!' character -- this '!' character *negates* the address, meaning that the 'G' command will apply to *all but* the first line. For the '$!d' command, we have a similar situation. If the command were '$d', it would apply the 'd' command to only the last line in the file (the '$' address is a simple way of specifying the last line). However, with the '!', '$!d' will apply the 'd' command to *all but* the last line. Now, all we need to to is understand what the commands themselves do.

When we execute our line reversal script on the text file above, the first command that gets executed is 'h'. This command tells sed to copy the contents of the pattern space (the buffer that holds the current line being worked on) to the hold space (a temporary buffer). Then, the 'd' command is executed, which deletes "foo" from the pattern space, so it doesn't get printed after all the commands are executed for this line.

Now, line two. After "bar" is read into the pattern space, the 'G' command is executed, which appends the contents of the hold space ("foo\n") to the pattern space ("bar\n"), resulting in "bar\n\foo\n" in our pattern space. The 'h' command puts this back in the hold space for safekeeping, and 'd' deletes the line from the pattern space so that it isn't printed.

For the last "oni" line, the same steps are repeated, except that the contents of the pattern space aren't deleted (due to the '$!' before the 'd'), and the contents of the pattern space (three lines) are printed to stdout.

Now, it's time to do some powerful data conversion with sed.

sed QIF magic

For the last few weeks, I've been thinking about purchasing a copy of Quicken to balance my bank accounts. Quicken is a very nice financial program, and would certainly perform the job with flying colors. But, after thinking about it, I decided that I could easily write some software that would balance my checkbook. After all, I reasoned, I'm a software developer!

I developed a nice little checkbook balancing program (using awk) that calculates by balance by parsing a text file containing all my transactions. After a bit of tweaking, I improved it so that I could keep track of different credit and debit categories, just like Quicken can. But, there was one more feature I wanted to add. I recently switched my accounts to a bank that has an online Web account interface. One day, I noticed that my bank's Web site allowed me to to download my account information in Quicken's .QIF format. In very little time, I decided that it would be really neat if I could convert this information into text format.

A tale of two formats

Before we look at the QIF format, here's what my checkbook.txt format looks like:

```
28 Aug 2000     food    -       -       Y     Supermarket             30.94
25 Aug 2000     watr    -       103     Y     Check 103               52.86
```

In my file, all fields are separated by one or more tabs, with one transaction per line. After the date, the next field lists the type of expense (or "-" if this is an income item). The third field lists the type of income (or "-" if this is an expense item). Then, there's a check number field (again, "-" if empty), a transaction cleared field ("Y" or "N"), a comment and a dollar amount. Now, we're ready to take a look at the QIF format. When I viewed my downloaded QIF file in a text viewer, this is what I saw:

```
!Type:Bank
D08/28/2000
T-8.15
N
PCHECKCARD SUPERMARKET
^
D08/28/2000
T-8.25
N
PCHECKCARD PUNJAB RESTAURANT
^
D08/28/2000
T-17.17
N
PCHECKCARD SUPERMARKET
```

After scanning the file, wasn't very hard to figure out the format -- ignoring the first line, the format is as follows:

```
D<date>
T<transaction amount>
N<check number>
P<description>
^
 (this is the field separator)
```

Starting the process

When you're tackling a significant sed project like this, don't get discouraged -- sed allows you to gradually massage the data into its final form. As you progress, you can continue to refine your sed script until your output appears exactly as intended. You don't need to get it exactly right on the first try.

To start off, I created a file called "qiftrans.sed", and started massaging the data:

```
1d
/^^/d
s/[[:cntrl:]]//g
```

The first '1d' command deletes the first line, and the second command removes those pesky '^' characters from the output. The last line removes any control characters that may exist in the file. Since I'm dealing with a foreign file format, I want to eliminate the risk of encountering any control characters along the way. So far, so good. Now, it's time to add some processing punch to this basic script:

```
1d
/^^/d
s/[[:cntrl:]]//g
/^D/ {
  s/^D\(.*\)/\1\tOUTY\tINNY\t/
        s/^01/Jan/
        s/^02/Feb/
        s/^03/Mar/
        s/^04/Apr/
        s/^05/May/
        s/^06/Jun/
        s/^07/Jul/
        s/^08/Aug/
        s/^09/Sep/
        s/^10/Oct/
        s/^11/Nov/
        s/^12/Dec/
        s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
}
```

First, I add a '/^D/' address so that sed will only begin processing when it encounters the first character of the QIF date field, 'D'. All of the commands in the curly braces will execute in order as soon as sed reads such a line into its pattern space.

The first line in the curly braces will transform a line that looks like:

```
D08/28/2000
```

into one that looks like thist:

```
08/28/2000  OUTY  INNY
```

Of course, this format isn't perfect right now, but that's OK. We'll gradually refine the contents of the pattern space as we go. The next 12 lines have the net effect of transforming the date to a three-letter format, with the last line removing the three slashes from the date. We end up with this line:

```
Aug 28 2000 OUTY  INNY
```

The OUTY and INNY fields are serving as placeholders and will get replaced later. I can't specify them just yet, because if the dollar amount is negative, I'll want to set OUTY and INNY to "misc" and "-", but if the dollar amount is positive, I'll want to change them to "-" and "inco" respectively. Since the dollar amount hasn't been read yet, I need to use placeholders for the time being.

Refinement

Now, it's time for some further refinement:

```
1d
/^^/d
s/[[:cntrl:]]//g
/^D/ {
        s/^D\(.*\)/\1\tOUTY\tINNY\t/
        s/^01/Jan/
        s/^02/Feb/
        s/^03/Mar/
        s/^04/Apr/
        s/^05/May/
        s/^06/Jun/
        s/^07/Jul/
        s/^08/Aug/
        s/^09/Sep/
        s/^10/Oct/
        s/^11/Nov/
        s/^12/Dec/
        s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
        N
        N
        N
        s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
        s/NUMNUM/-/
        s/NUM\([0-9]*\)NUM/\1/
        s/\([0-9]\),/\1/
}
```

The next seven lines are a bit complicated, so we'll cover them in detail. First, we have three 'N' commands in a row. The 'N' command tells sed to read in the *next line in the input* and append it to our current pattern space. The three 'N' commands cause the next three lines to be appended to our current pattern space buffer, and now our line looks like this:

```
28 Aug 2000 OUTY  INNY  \nT-8.15\nN\nPCHECKCARD SUPERMARKET
```

Sed's pattern space got ugly -- we need to remove the extra newlines and perform some additional formatting. To do this, we'll use the substitution command. The pattern we want to match is:

```
'\nT.*\nN.*\nP.*'
```

This will match a newline, followed by a 'T', followed by zero or more characters, followed by a newline, followed by an 'N', followed by any number of characters and a newline, followed by a 'P', followed by any number of characters. Phew! This regexp will match the entire contents of the three lines we just appended to the pattern space. But we want to reformat this region, not replace it entirely. The dollar amount, check number (if any) and description need to reappear in our replacement string. To do this, we surround those "interesting parts" with backslashed parentheses, so that we can refer to them in our replacement string (using '\1', '\2\, and '\3' to tell sed where to insert them). Here is the final command:

```
s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
```

This command transforms our line into:

```
28 Aug 2000  OUTY  INNY  NUMNUM    Y     CHECKCARD SUPERMARKET   AMT-8.15AMT
```

While this line is getting better, there are a few things that at first glance appear a bit...er...interesting. The first is that silly "NUMNUM" string -- what purpose does that serve? You'll find out as you inspect the next two lines of the sed script, which will replace "NUMNUM" with a "-", while "NUM"<number>"NUM" will be replaced with <number>. As you can see, surrounding the check number with a silly tag allows us to conveniently insert a "-" if the field is empty.

Finishing touches

The last line removes a comma following a number. This converts dollar amounts like "3,231.00" to "3231.00", which is the format I use. Now, it's time to take a look at the final, production script:

```
1d
/^^/d
s/[[:cntrl:]]//g
/^D/ {
  s/^D\(.*\)/\1\tOUTY\tINNY\t/
  s/^01/Jan/
  s/^02/Feb/
  s/^03/Mar/
  s/^04/Apr/
  s/^05/May/
  s/^06/Jun/
  s/^07/Jul/
  s/^08/Aug/
  s/^09/Sep/
  s/^10/Oct/
  s/^11/Nov/
  s/^12/Dec/
  s:^\(.*\)/\(.*\)/\(.*\):\2 \1 \3:
  N
  N
  N
  s/\nT\(.*\)\nN\(.*\)\nP\(.*\)/NUM\2NUM\t\tY\t\t\3\tAMT\1AMT/
  s/NUMNUM/-/
  s/NUM\([0-9]*\)NUM/\1/
  s/\([0-9]\),/\1/
  /AMT-[0-9]*.[0-9]*AMT/b fixnegs
  s/AMT\(.*\)AMT/\1/
  s/OUTY/-/
  s/INNY/inco/
  b done
:fixnegs
  s/AMT-\(.*\)AMT/\1/
  s/OUTY/misc/
  s/INNY/-/
:done
}
```

The additional eleven lines use substitution and some branching functionality to perfect the output. We'll want to take a look at this line first:

```
        /AMT-[0-9]*.[0-9]*AMT/b fixnegs
```

This line contains a branch command, which is of the format "/regexp/b label". If the pattern space matches the regexp, sed will branch to the fixnegs label. You should be able to easily spot this label, which appears as ":fixnegs" in the code. If the regexp doesn't match, processing continues as normal with the next command.

Now that you understand the workings of the command itself, let's take a look at the branches. If you look at the branch regular expression, you'll see that it will match the string 'AMT', followed by a '-', followed by any number of digits, a '.', any number of digits and 'AMT'. As I'm sure you've figured out, this regexp deals specifically with a negative dollar amount. Earlier, we surrounded our dollar amount with 'AMT' strings so we could easily find it later. Because the regexp only matches dollar amounts that begin with a '-', our branch will only happen if we happen to be dealing with a debit. If we are dealing with a debit, OUTY should be set to 'misc', INNY should be set to '-', and the negative sign in front of the debit amount should be removed. If you follow the code, you'll see that this is exactly what happens. If the branch isn't executed, OUTY gets replaced with '-', and INNY gets replaced with 'inco'. We're finished! Our output line is now perfect:

```
28 Aug 2000 misc  - -       Y     CHECKCARD SUPERMARKET  -8.15
```

Don't get confuSed

As you can see, converting data using sed isn't all that hard, as long as you approach the problem incrementally. Don't try to do everything with a single sed command, or all at once. Instead, gradually work your way toward the goal, and continue to enhance your sed script until your output looks just the way you want it to. Sed packs a lot of punch, and I hope that you've become very familiar with its inner workings and that you'll continue to grow in your sed mastery!

Resources

- Read Daniel's previous sed articles on *developerWorks*: Common threads: Sed by example, Part 1 and Part 2.
- Check out Eric Pement's excellent sed faq.
- You can find the sources to sed 3.02 at ftp.gnu.org.
- You'll find the nice, new sed 3.02.80 at alpha.gnu.org.
- Eric Pement also has a handy list of sed one-liners that any aspiring sed guru should definitely take a look at.

- If you'd like a good old-fashioned book, O'Reilly's sed & awk, 2nd Edition would be a wonderful choice.
- Maybe you'd like to read 7th edition UNIX's sed man page (circa 1978!).
- Take Felix von Leitner's short sed tutorial.
- Brush up on using regular expressions to find and modify patterns in text in this free, dW-exclusive tutorial.

About the author

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of Gentoo Technologies, Inc., the creator of Gentoo Linux, an advanced Linux for the PC, and the Portage system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at SONY Electronic Publishing/Psygnosis. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah. You can contact Daniel at drobbins@gentoo.org.