LPI certification 101 exam prep, 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.

Section 1. About this tutorial

What does this tutorial cover?

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, or FHS, 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. Finally, we'll give you a whirlwind introduction to shell pipelines, redirection, and text processing commands.

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.

The LPI logo is a trademark of Linux Professional Institute.

Should I take this tutorial?

This 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 to "round out" their basic Linux administration skills.

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*

Daniel Robbins lives in Albuquerque, New Mexico, and 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 their daughter, Hadassah.

Chris Houser, known to many of his friends as "Chouser", has been a UNIX proponent since 1994 when 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.

Aron Griffis graduated from Taylor University with a degree in Computer Science and an award that proclaimed him the "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, NH.

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 perform search-and-replace operations on text streams, among other things.

Glob comparison

As we look at regular expressions, you may find that regular expression syntax looks similar to filename "globbing" syntax that we examined in our previous tutorial (see Part 1 listed in the Resources section at the end of this tutorial). 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 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 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 expressions are 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. Here are some examples:

* `ab*c` (matches `abbbbc` but not `abqc`)
* `ab*c` (matches `abc` but not `abbqbbc`)
* `ab*c` (matches `ac` but not `cba`)
* `b[cq]*e` (matches `bqe` but not `eb`)
* `b[cq]*e` (matches `bccqqe` but not `bccc`)
* `b[cq]*e` (matches `bqqcce` but not `cqe`)
* `b[cq]*e` (matches `bbbeee`)
* `.*` (matches any string)
* `foo.*` (matches any string that begins with `foo`)

The line `ac` matches the regex `ab*c` because the asterisk also allows the preceding expression (`c`) to appear zero times. 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 regexps

^ 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. The FHS specifies the following directories (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. We promise you'll understand a lot more about the Linux filesystem if you read it.

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. The following panels will introduce you to them, and help you to 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 elements 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
```

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 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:

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` 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 Jan  7 18:14:52 EST 2002
```

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 Jan  6 18:00 b
-rw-------    1 root     root            0 Jan  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 replacing 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, due to the fact that 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.

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 include a periodic process to update the database. If your system returned an error with the above command such as the following, then you will need to run `updatedb` 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

In 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 wouldn't otherwise be able 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

Starting xeyes

To learn about process control, we first need to start a process:

```
$ 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 on 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
```

xeyes*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 to these 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, in other words, `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]
```

We've listed only the first few because it's usually a very long list. This gives you a snapshot of what the whole machine is doing, but it is a lot of information to sift through. If you were to omit 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 process has a priority setting that Linux uses to determine how fast it should run
relative to the processes on the system. You can set the priority of a process by starting it
with the `nice` command:

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

Because 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

You can 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 -
```

Woohoo! 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, 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 prints only the number of lines in its input, instead of also including words and characters. 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 its 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` prints the *contents* of the files specified as arguments to the terminal. This is handy as the first command of a pipeline, for example, `cat foo.txt | blah`.

sort

`sort` prints 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`

`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`

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

`unexpand`

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

cut, nl, and pr

`cut`

`cut` extracts character-delimited fields from each line of an input file or stream.

`nl`

`nl` adds a line number to every line of input. This is useful for printouts.

`pr`

`pr` breaks files into multiple pages of output; typically used for printing.

---

tr, sed, and awk

`tr`

`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 about `sed` in the following *developerWorks* articles:
   * *Sed by example, Part 1: Get to know the powerful UNIX editor*
   * *Sed by example, Part 2: Taking further advantage of the UNIX text editor*
   * *Sed by example, Part 3: Data crunching, sed style*

`awk`

`awk` is a handy line-oriented text-processing language. To learn about `awk`, read the following IBM developerWorks articles:
   * *Awk by example, Part 1: Intro to the great language with the strange name*
   * *Awk by example, Part 2: Records, loops, and arrays*
   * *Awk by example, Part 3: String functions and ... checkbooks?*

---

od, split, and fmt

`od`

`od` transforms the input stream into an octal or hex "dump" format.

`split`

`split` splits a larger file into many smaller-sized, more manageable chunks.

`fmt`

`fmt` reformats paragraphs so that wrapping is done at the margin. 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 (the first, by default) in each input line to match up what should be combined on a single line.

`tee`

`tee` prints 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 work only 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 sentinel value. Here's an example:

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

In our example, 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! We meant this:

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

```
there.
```

Much better!

Section 6. Resources and feedback

Resources and homework

Congratulations; you've reached the end of "Basic administration". We hope that this tutorial has helped you to firm up your foundational Linux knowledge. Please join us in our next tutorial, "Intermediate administration," where we'll 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 prepare yourself to attain your LPIC Level 1 Certification from the Linux Professional Institute. Speaking of LPIC certification, if this is something you're interested in, then we recommend that you study the following resources, which augment the material covered in this tutorial:

There are a number of good regular expression resources on the Web. Here are a couple that we've found:
  * *Regular Expressions Reference*
  * *Regular Expressions Explained*

Be sure to read up on the *Filesystem Hierarchy Standard*.

In the *Bash by example* article series on *developerWorks*, Daniel shows you how to use `bash` programming constructs to write your own `bash` scripts. This bash series (particularly Parts 1 and 2) 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 *developerWorks* articles (Parts 1 and 2 will be good preparation for the LPIC Level 1 exam):
  * *Sed by example, Part 1: Get to know the powerful UNIX editor*
  * *Sed by example, Part 2: Taking further advantage of the UNIX text editor*
  * *Sed by example, Part 3: Data crunching, sed style*

To learn more about `awk`, read the following *developerWorks* articles:
  * *Awk by example, Part 1: Intro to the great language with the strange name*
  * *Awk by example, Part 2: Records, loops, and arrays*
  * *Awk by example, Part 3: String functions and ... checkbooks?*

We highly recommend the *Technical FAQ by Linux Users*, a 50-page in-depth list of frequently-asked Linux questions, along with detailed answers. The FAQ itself is in PDF (Adobe 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 familiar with the `vi` editor, we strongly recommend that you check out Daniel's *Vi intro -- 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

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact Daniel Robbins directly at *drobbins@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.