Execution and Redirection

By Jerry Peek

While the title of this month's "Power Tools" is "Execution and Redirection," it's not about about dying and going to heaven. Instead, controlling execution and redirecting input and output is an important part of managing Linux processes.

Let's dig into these two topics, learn the basics, and then see some shell features for managing processes that are little-known, but indispensable nonetheless.

Life as a Process

When Linux runs a program, it "spawns," or starts, a new *process* to keep track of the program's state, including the program's current directory, environment variables, open files, and more. A *process identification number*, or *PID*, uniquely identifies each process, and each process (except for the special seed process named *init*) also has a *parent process ID*, or *PPID*, that refers to the process that spawned it. (A new process is often referred to as a "child," which makes the process that spawned it a "parent.")

You can examine processes with the *ps* (process status) utility. With no arguments, *ps* prints a list of "useful" processes, which may or may not show the process you want. You can expand or refine the list of processes with a number of *ps* options (see its *man* page), or you can look at a specific process just by specifying its PID, using `ps PID` or `ps -p PID`.

For example, your shell is just another program (albeit a complicated one), and a shell running in an *xterm* window is just another process. In most shells, the special variable `$$` ("dollar dollar") represents the PID of the current shell. So, the command `ps -p $$` shows many of the process attributes of your current shell:

```
$ ps -p $$
UID        PID  PPID STIME TTY          TIME CMD
jpeek    23560 23557 07:54 pts/3    0:00 bash
```

From left to right, the output above shows: the user (`jpeek`) that started the process; the process's PID (`23560`) and its *PPID* (`23557`); the time the process started (`7:54 am`); the TTY (`pts/3`) associated with the process; the amount of CPU time consumed by the process (`0` minutes and `0` seconds so far); and the name of the running program (`bash`).

If you want to find more about this *bash* shell's parent process, `23557`, just use *ps* again:

```
$ ps -p 23557
UID        PID  PPID STIME TTY          TIME CMD
jpeek    23557     1 07:54 tty2     0:00 xterm
```

Here, you can see that the shell's parent is *xterm*. That makes sense: when you launch *xterm*, it spawns a new process for the shell running in its window. (What's process `1`? That's the PID of the system seed process, *init*.)

A process can spawn (virtually) any number of new processes. In fact, the shell demonstrates this perfectly: each command you run is a new process, a child of the shell.

For example, if you run the command `sleep 30 & sleep 30 & sleep 30 &`, the shell spawns three new processes, one for each *sleep*. You can see the results by running *ps*:

```
PID        PPID       COMMAND
28925      23560      sleep 30
28926      23560      sleep 30
28927      23560      sleep 30
```

Processes `28925`, `28926`, and `28927` were spawned by `23560`, the shell. Each runs *sleep*, dozing for thirty seconds before exiting.

Swapping Your Shell

A process, like each *sleep* above, typically lives a very brief (but meaningful) life: it's spawned and then runs a program and dies. However, that isn't the only possible sequence of events. It's also possible for a running program to *replace* itself with another. No new process is created.

Indeed, you can even replace the shell you're typing in with another program. Just use the shell's *exec* command (which is based on the features provided by the `exec()` family of system calls). Typing `exec program` at a command-line prompt replaces that shell with *program*.

For example, if you type `exec vi myfile` at the prompt, *vi* replaces *bash*. If you're running in an *xterm*, when you exit *vi*, the *xterm* window closes.

Running *exec* from the shell can be used in other clever

WHY REDIRECTION?

What's good about redirection? A lot. Redirection means that — unless you name a specific file on its command line — a process doesn't need to know what file to read from or write to. It simply writes good data to *fd 1*, reads from *fd 0*, and writes any errors to *fd 2*. You can redirect any or all of those three channels away from the terminal if you want to, but the process doesn't need to worry about that. Linux handles the translation between the character stream and the various types of files and devices.

ways. Perhaps you're a fan of *psh*, the *Perl Shell* (see http://www.focusresearch.com/gregor/psh/index.html), but your system administrator won't let you use it as a standard login shell. No problem. After you log in and get a prompt from your current shell, simply type `exec psh` and presto! A Perl shell appears in your terminal window and keeps running until you exit, then the window closes.

You can also replace the shell with any kind of program — not just another shell. To run *top* and only *top* as root, type `exec su -c top`. When you (or some bad guy) quits *top*, the window closes, precluding foul play.

REDIRECTION VS. GUIS

Graphical (GUI) programs (ones that open their own window) have a place for input and output other than file descriptors 0, 1, and 2: the window(s) that they open. These windows are distinct from the standard I/O of a process.

So, for example, a GUI program may write an error message on *fd 2*. That message won't appear in the GUI's window, but instead appears on the *stderr* of the GUI process — which may be the terminal window where you started the program, the output of the program (like *startx*) that started the window system, or to another place.

How can you see these messages? You can start your GUI programs by typing their name in a terminal, like:

```
$ gimp &
```

Or, if you start the program from a button or menu, your window system may be able to run it "in a terminal," where a terminal window will open, showing any text on *fd 1* or *fd 2*. If that's not possible, configure the button or menu entry to open a terminal window and run the GUI program from it, like this command that starts the GIMP in verbose mode:

```
$ xterm -title "GIMP console" \
  -geometry 80x9-0+1 -e gimp –verbose
```

Or, finally, you can redirect *stdout* and *stderr* of the program that starts your window system into a log file, then watch that log file.

For instance, assuming that you use *startx* to start X, run one of the following command lines in a Bourne-type shell (like *bash*):

```
$ startx >$HOME/tmp/startx.log 2>&1

$ startx 2>&1 | tee $HOME/tmp/startx.log
```

The first redirects *stdout* and *stderr* to a file. The second does the same, but also displays *stdout* and *stderr* on the terminal where you ran *startx*. Then, in a terminal window on your display, you can run:

```
$ tail -f $HOME/tmp/startx.log
```

The `tail -f` program "watches" the log file and shows any new text. Terminate it with CTRL-C.

(Of course, a bad guy can still kill processes and wreak havoc with a running *top*, so be careful what commands you leave running and unattended. And never use this technique to run a command that can *escape* to a shell. For example, if you ran `exec su -c vi` as root, the *vi* command `:sh` can be used to spawn a new shell *as* root. For details on *su*, see the December 2002 "Power Tools," available online at http://www.linux-mag.com/2002-12/power_01.html.)

Redirecting Standard I/O

In a shell, *redirection* means re-routing the *standard input*, (*stdin*), *standard output* (*stdout*), and *standard error* (*stderr*) from the default location, which is the terminal.

Here are some examples.

```
$ sed 's/^note:/NOTE:/' report
...sed output and errors appear...
$ sed 's/^note:/NOTE:/' report 2>errs | lpr
$
```

In the first command, there's no redirection. So *sed* writes the edited text to *stdout* and writes any error messages to *stderr*. (*sed* can also read text from *stdin*, but here it's reading a file named *report*, ignoring its standard input.)

Running *exec* from the shell can be used in a number of clever ways: redirecting shell I/O and replacing the shell

The second command uses the shell's `2>` operator (which we'll explain shortly) to redirect *sed*'s error messages to a file named *errs*. The command also uses the `|` ("pipe") operator to redirect the edited text to the printer program *lpr*. So there's no output to the terminal.

What's happening here? When you start a process, three file descriptors (*fds*) are associated with the process: *fd 0*, *fd 1*, and *fd 2*. *fd 0* is associated with *stdin*; *fd 1* with *stdout*; and *fd 2* with *stderr*. By default, all of the *fds* point to the device file */dev/tty*, which is the shell's terminal or terminal window.

You can redirect those three file descriptors to other places. For example, the `>` ("greater than") operator sends a process' *stdout* to a plain text file, to *named pipes*, and to any other physical file or Linux device that knows how to handle a stream of characters. The digit `2` in `2>` is the file descriptor for *stderr*. So `sed ... 2> errs` redirects *sed*'s *stderr* to the file named *errs*. (You can actually write `prog 1>` file instead of `prog >` *`file`*, but *fd 1* is the default file descriptor for the `>` operator.)

The `<` ("less than") operator redirects the standard input of a process from a file or device. The pipe operator `|` redirects the

standard input to a process (if you put it before a program name), or the standard output from a process (when it follows the program name).

The sidebars "Why Redirection?" and "Redirection vs. GUIs" have more about this.

We've seen that `sed ... 2>errs` redirects the *stderr* of the *sed* process. What if you'd like to redirect the *stderr* of all processes run from your shell? Use the shell's *exec* command with the redirection operators you want.

For example, `exec 2> errorlog` redirects the *stderr* of the shell itself into the file *errorlog*. Because the shell's *stderr* is redirected to a file, the *stderr* of its children (the processes it spawns) also go to that file. (A child process inherits its parent's open file descriptors.) This technique is very useful in shell scripts.

Mixing It Up

Another useful Bourne-type shell operator is *m*`&`*n*. It duplicates file descriptor *m* from file descriptor *n*. In other words, it makes *fd* m "point to" the same file as *fd* n.

So, how can you redirect both stdout and stderr to the same file? As we did for *startx* in the sidebar "Redirection vs. GUIs":

```
$ startx >logfile 2>&1
```

The shell reads left-to-right. First, the `>` operator redirects *stdout* to *logfile*. Next, the `2>&1` operator duplicates *fd 2* from *fd 1* — that is, it makes *stderr* go the same place as *stdout*, which is *logfile*.

exec 2> errorlog redirects the stderr of the entire shell to *errorlog*

(The order of the redirections is important! If you type `2>&1 >logfile`, that would first make *fd 2* go the same place as *fd 1* — to */dev/tty*, which is no change. Then it would redirect *fd 1* to *logfile*, leaving *fd 2* on */dev/tty*.)

This is very useful for sending both *stderr* and *stdout* down a pipe. A pipe redirects only *fd 1*. You can use `2>&1` to make *stderr* go the same place as *fd 1:* that's the pipe!

So, to use *less* to page through both the *stdout* and *stderr* of *make*:

```
$ make 2&1 | less>
```

The opposite is handy in shell scripts, which should write error messages to *stderr*. But *echo* writes messages to *stdout*. How to fix it? Add `1>&2` to the *echo* command line:

```
echo "$progname: can't read $file" 1>&2
```

POWER TIP: Suspending "Chained" *ssh* Sessions

The November 2003 Power Tip (available online at http://www.linux-mag.com/2003-11/power_01.html) showed how to use the sequence RETURN ~ ("tilde") CTRL-Z to suspend an *ssh* or *rsh* login session. But using this on a "chained" session — for instance, `ssh host1` and then (from *host1*) `ssh host2` — suspends the entire session.

To drop back to the intermediate host — in our example, to suspend the connection to *host2* and get a shell prompt from *host1* — use *two* tildes, as in RETURN ~~ CTRL-Z.

Typing two tildes (~~ sends a literal single tilde ~) to the *ssh* process on *host1*. The *host1 ssh* will see this and suspend the connection to *host2*, leaving you with a prompt on *host1*.

Other File Descriptors

Other file descriptors, *fd 3* through *fd 9*, are available, but are generally not used. With *exec*, you can open files and associate them to one of those *fd*s.

For instance, use the command `exec 3>logfile` to open *logfile* as *fd 3*. Then you can give commands like `ls 1>&3` and `cal 1>&3` to append the outputs of *ls* and *cal* to *logfile*. *logfile* stays open until the shell exits or until you close it with the operator `3>&-`. (This is more efficient for writing multiple command outputs to a file than repeatedly opening a file with the shell's "append" operator `>>`.)

Last month's column (available online, beginning August 2004, at http://www.linux-mag.com/2004-05/power_01.html), showed how to redirect the *stdout* and *stderr* of a loop. You can also write to *fd 3* through *fd 9* within a loop, then redirect that at the end of the loop. You might use this for progress messages or debugging output that you want to put into a file without affecting *stdout* or *stderr*.

For instance, you could write a `for` loop like this:

```
for file in /home/joe/*
do
  echo "Doing $file at $(date)" 1>&3
  ls -l "$file" 1>&4
  ...
done 3>logfile 4>ls_listings
exec 3>&- 4>&-
```

After the loop finishes, *logfile* contains the output of all the *echo* commands, and *ls_listings* has all of the `ls -l` listings.

These extra file descriptors can be used in other ways, too — to swap *stdout* and *stderr*, for instance. But we'll have to leave that for another column.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.