Boot Camp for Wizards, Part Two

Jerry Peek

Last month, this series started with an overview of *Linux* fundamentals and a look at some important concepts behind shells. This month, let's continue looking at obscure features of Linux that books generally don't cover but that wizards should know. Read on to find out more about the standard I/O system. Though the ideas behind standard I/O are common knowledge, let's start with an introduction to get on the same page (so to speak).

What's Behind Standard I/O?

Back in the days when *Unix* was new, the idea of standard I/O wasn't commonplace. It wasn't necessarily simple to write data to different devices — a program might need different code to read or write files, tapes, terminals and printers. Users were accustomed to seeing progress messages and summaries.

A well-behaved Unix/Linux utility outputs only valid data. If there's no trouble and no data, there's no output. For instance, if *grep* doesn't find matching text, it doesn't output "No match found." It simply terminates.

Unix popularized the idea that data can be a stream of bytes, that a program reads and writes without knowing where the data is coming from or where it's going. The operating system and its *device drivers* convert a stream of bytes to and from the correct formats for all of the system's devices (disks, printers, tape drives, and so on), letting the programs handle the data. The Unix operating system and the shell also arrange the "plumbing" that routes data between processes and devices — for instance, by doing the I/O buffering and process stops and starts that make a *pipe* work. Linux has the same concepts, of course.

Unix popularized the idea that data can be a stream of bytes, that a program reads and writes without knowing where the data is coming from or where it's going

As much as possible, a general-purpose Linux program should be able to read data from its standard input (*stdin*), write results to its standard output (*stdout*), and write warnings or errors to the standard error (*stderr*). The *stdout* of a general-purpose program should be only data — data that's ready for writing to a file, sending down a network, or reading by another program. A general-purpose program shouldn't write progress messages to *stdout* ("reading *somefile*... " or "23% done") because, if it did, other programs would need to distinguish between that "out-of-band" text and valid data. Errors and warnings also shouldn't go to *stdout* because *stdout* may be redirected to a file, a printer, or another device. Instead, warnings and errors should go to *stderr*, which almost always goes to the user interface — typically, a terminal — where the messages appear.

Many Unix/Linux utilities act as a *filter*. If you don't tell a filter program where to read and write its data, it defaults to reading from *stdin* and writing to *stdout*. Options and other arguments on the program's command-line can tell the program which file(s) to read and/or write in place of the standard I/O.

At least two types of well-behaved programs won't necessarily act as a filter:

- A program with a graphical user interface (GUI). In addition to *stdin*, *stdout* and *stderr*, it has its GUI.
- Programs like *less* that can't expect to read commands from their standard input because *stdin* could be data from a pipe to be displayed. These programs may read their commands directly from the keyboard, bypassing *stdin*.

Writing a Filter

It's usually straightforward to write a well-behaved filter program. First your program should handle any command-line options. If there are filename arguments left over, read from those files; otherwise, read standard input.

In a shell script, for instance, the parameter `$#` tells how many command-line arguments there are (after any *shift* commands). If `$#` is zero, read standard input with a command like one of these:

```
text=$(cat)    # read into shell variable
text=`cat`     # read into shell variable
text=`prog`    # filter with prog to $text
cat > $tmp     # read into $tmp file
prog > $tmp    # filter with prog into $tmp
```

Terminal Windows: GUIs that Route Standard I/O

You can think of a terminal window as an interface between the standard I/O system and a graphical display. When a terminal window has the GUI's input focus, text you type on the keyboard is available as *stdin* to a program running in that

terminal. The program's *stdout* and *stderr* are displayed there, where they can also be copied and pasted into other GUI windows.

This setup can be useful for more than running a non-GUI program. You can start a GUI program by typing its name (actually, the name of its executable file) at a prompt in the terminal window. You'll probably want to add an ampersand at the end of the command-line so the graphical program runs in the background (and you'll get another shell prompt). Starting a program this way is handy because you can also stop or kill the process using shell job control (which is covered later in this series).

Linux utilities can be useful for processing data from GUI programs. For example, to count the number of words in a GUI window, copy the text with your mouse. Next, in a terminal window, type `wc -w` and press ENTER to start the program. Without a filename to read, *wc* reads its standard input (from the terminal window). Paste text from the other window and press Control-D to signal end-of-input. *wc* counts the words and writes the result to *stdout*.

As another example, you can use *sed* to modify a column of data in an *OpenOffice.org* spreadsheet. *sed* generally outputs an edited line immediately after reading it, so copying and pasting directly (as you did with *wc*) would make a jumble of input and output lines on the screen. Instead, in the terminal window, type the *sed* command and tell the shell to redirect *sed* 's *stdout* to a temporary file:

```
sed 's/this/that/g' > sedout
```

Because *sed* has no filename arguments (the shell reads and removes `> sedout`), *sed* reads its standard input. Use your mouse to copy a column from the spreadsheet and paste it into the terminal window. Type Control-D. Run `cat sedout` to get the column of results, then copy and paste it back into the spreadsheet.

Open Files

When a Linux program needs to read from or write to a file, it uses a system call like `open()` or a higher-level library function to open the file. This makes several things happen:

1. By reading the file's name or pathname, the system locates the file's inode and finds the file's actual physical disk blocks (unless the file doesn't exist and needs to be created).

2. The file is opened for reading and/or writing; a *file descriptor* number is assigned. Each file descriptor is a small integer (3, 4, 5,…) that you can use to refer to the open file from within the process that opened it. (The first three file descriptors, numbered 0, 1, and 2, are already used for the three default open files: *stdin*, *stdout*, and *stderr*, respectively.)

Also, a *file offset* points to a location within the file. For reading a file, writing a new file, or overwriting an existing file, the pointer is set to the start of the file. For appending to a file, the offset points to the end of the file. Each time data is read or written to the file, the offset changes to keep track of the next location to read or write.

3. Now that the file is open, its filename isn't needed anymore.

The file stays open until it's explicitly closed or the process terminates.

If you don't tell a filter program where to read and write its data, it defaults to reading from *stdin* and writing to *stdout*. "Out of band" data is written to *stderr*.

Using Open Files from the Shell

Remember from last month that, when you start a child process, the child inherits the parent's open files. So, if you start your shell script named *prog* from a shell prompt in your terminal…

```
$ prog
```

… since you haven't redirected *prog*' s output or input, each child process that *prog* starts also has *stdin*, *stdout* and *stderr* connected to your terminal. On the other hand, if you've redirected *prog*' s *stdout* to a file…

```
$ prog > outfile
```

… then, if a child (the commands that *prog* runs) writes to *stdout*, its *stdout* is also redirected to *outfile* and its output goes there by default. (Note that each child process doesn't know anything about redirection to *outfile*. Each simply writes to its standard output, as always.) The childrens' *stderr* and *stdin* are still your terminal.

By itself, the shell's `>` operator redirects file descriptor 1. In Bourne-type shells like *bash*, you can also write a file descriptor number before `>` to redirect that open file. For instance, `2>` redirects *stderr*. So, if you're going to start a program named *cruncher* that runs unattended, you might want to collect its *stdout* and *stderr* into two different files, like this:

```
$ cruncher >cruncher_out 2>cruncher_err
```

Adding a final `&` lets *cruncher* crunch in the background while you do something else. You won't see any error messages, of course. So you might want to use `tail -f cruncher_err` — possibly from a different terminal window — to see any new error messages.

Redirecting Multiple Command Outputs to One File

The shells' `>>` operator appends *stdout* to an existing file. That is, it opens the file for writing, moves the file offset pointer to the end, and writes text there. So, if you had three programs named *crunch1*, *crunch2* and *crunch3*, you could capture their standard outputs into a single file like this:

```
$ crunch1 > crunches_out
$ crunch2 >> crunches_out
$ crunch3 >> crunches_out
```

The Bourne shell's curley-brace operator {} lets you redirect the input or output of a series of commands. Use the semicolon to separate each command.

(`2>>` would append the standard error to a file.) That's a bit inefficient, though, because you're opening and re-opening the same file. Here's where your understanding of open files starts to pay off. The Bourne shell's curly-brace operator `{}` lets you redirect the input or output of a series of commands. Inside the braces, end each command with a semicolon (`;`). A more efficient way to write the previous example is:

```
$ { crunch1; crunch2; crunch3; } > \
  crunches_out
```

In a shell script, using several lines may be clearer:

```
{
   crunch1;
   crunch2;
   crunch3;
} > crunches_out
```

Here's another way:

```
for prog in crunch {1,2,3}
do
   $prog
done > crunches_out
```

What's happening? The *for* loop is actually a single shell statement. We've redirected the output of the statement— and the child processes that it runs— into a file.

In the same way, you can redirect the standard input of a loop by using a pipe operator before the loop or the shell's `<` operator after the *done* command. This is commonly used to read the output of a command line-by-line:

```
ls /some/directory |  \
while read filename
do
   … process $filename…
done
```

(The *read* command reads a single line from its standard input and copies it into a shell variable.) However, a loop with redirected output or input *may* be run in a child shell process. If it is, any changes to environment variables, shell variables, current directory, etc., *within the loop* may not affect the parent shell, that is, the shell running the script. (As mentioned last month, changes in a child process don't affect the parent.)

To guarantee that a set of commands is run in a child process, use the parenthesis operators `()`. Watch:

```
$ pwd
/foo/bar

$ (cd d1; prog1; cd d2; prog2) > progs_out

$ pwd
/foo/bar
```

The *cd* commands inside the subshell operators change the subshell's current directory. So `prog1` ran in directory *d1*, and `prog2` ran in directory *d1/d2*. But the parent's current directory hasn't changed! Also, the output of both `prog1` and `prog2` has gone into the file `progs_out` in the parent shell's current directory */foo/bar*.

Until Next Time…

Standard I/O pictures all data sources as a stream of bytes, and is a fundamental, differentiating feature of Linux- and Unix-based systems. Standard I/O operands, such as the pipe and redirection, allow you to instantly create a new application simply by combining existing utilities and features of the shell.

Next month covers more about manipulating standard I/O from the shell.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for more than 25 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.