Boot Camp for Wizards, Part Three

Jerry Peek

Last month's column, the second in this series of obscure Linux features that wizards should know, introduced Standard I/O. This month we'll see how to take advantage of Standard I/O from a shell — including an example using named pipes (FIFOs).

Duplicating a File Descriptor

The Bourne shells' operator `m>&n` copies a file descriptor. The first digit, *m*, is the file descriptor you want to change. The second digit, *n*, is the file descriptor (f.d.) to copy.

For instance, `2>&1` copies f.d. 2 from f.d. 1. This makes f.d. 2 point to the same open file as f.d. 1 does. Because f.d. 2 is the standard error and f.d. 1 is standard output, after using that operator, *stderr* will point to the same place as *stdout*.

The operator `2>&1` isn't any use in the default situation where both *stderr* and *stdout* point to the terminal. One place it is handy is when you're piping the output of one command to another — for instance, when you're viewing the output of a program with a pager program like *less* (1). Without the `2>&1` operator, like this:

```
$ grep xyz * | less
```

Only the *stdout* of *grep* is piped to *less*. *grep*'s *stderr* goes to the terminal — along with the *stdout* of *less*. So, if *grep* issues an error, it's likely to be mixed into the output text from *less*; you may not notice the error at all. Using `2>&1` fixes that:

```
$ grep xyz * 2>&1 | less
```

With that operator, *grep*'s *stderr* goes to the same place as *grep*'s *stdout*: down the pipe, to be read by (and paged by) *less*. *bash* sets up the pipe first— connecting *grep*'s *stdout* to the *stdin* of *less* — then it redirects *grep*'s *stderr*. The `2>&1` must be written to the left side of the pipe operator; it applies to the command on the left-hand side.

The order of redirections is important: the shell reads a command line and processes redirections from left to right.

Let's see another example: using `2>&1` together with the file-redirection operator `>`. Here's the correct way to use it:

```
$ someprog > $outfile 2>&1
```

Why is that correct?

Reading that command line from left to right, you're first telling the shell to redirect the *stdout* of *someprog* to the file named in `$outfile`. Next, you're telling the shell to make *someprog*'s *stderr* (f.d. 2) go the same place as its *stdout*: that is, to `$outfile`.

Let's compare the previous correct example with the next one, which doesn't do what we want:

```
$ someprog 2>&1 > $outfile
```

What's wrong?

By default, as the shell starts to handle redirections, both f.d. 1 and f.d. 2 are going the same place: to the terminal. As the shell reads left to right, it sees `2>&1` and copies f.d. 1 to f.d. 2. But, at this point, both f.d. 1 and f.d. 2 are still going to the terminal. So, the operator has no effect. Then `> $outfile` redirects *stdout* to `$outfile`, without affecting *stderr*.

In the same way, you can force command substitution to collect both *stdout* and *stderr* — instead of just *stdout*. For instance, to capture all the output of *someprog* into a shell variable *progout*:

```
progout=$(someprog 2>&1)
```

Redirecting a File Descriptor for All Commands: *exec*

Let's start with a little bit of background info. If you pass a command name (and optional arguments) to the shell's built-in *exec* command, that command will replace the shell and continue running within the same process.

For example, if you're running the *bash* shell and you want to run *zsh* instead, type:

```
$ exec zsh
zsh%
```

The original *bash* program is gone, replaced (in the same process) by an instance of *zsh*.

Using *exec* this way was important in the early days of Unix because it saved precious system resources. (The old shell didn't stay around, waiting for a child process to finish.)

In Bourne-type shells, the *exec* command can also redirect open files permanently — that is, until the current shell process exits. It's usually used in shell scripts, though it also works from a shell prompt.

For instance, to redirect the standard output of all following commands to a file named *output*, use this command:

```
exec > output
```

To redirect both *stdout* (file descriptor 1) and *stderr* (f.d. 2) to a file, use the `2>&1` operator:

```
exec > output 2>&1
```

From there until the shell terminates, both *stdout* and *stderr* will go to the file *output* and the script should run silently.

Using Higher File Descriptors

So far we've been using the default file descriptors 0, 1, and 2. File descriptors 3 through 9 generally aren't used in shell scripts. (File descriptors 10 and above may be used internally by the shell, so you shouldn't change them.)

What can you use them for — and how?

One way is to gather the output of certain commands within a loop. As we saw last month, a loop is a shell statement, so the inputs and outputs of commands within the loop can be redirected before or after the loop. For example, let's say you're writing a long-running loop that logs the time each iteration starts into a file *start-times* and logs the directories visited by the loop in a file named *dirnames*. The inefficient way to do this, which opens and closes those log files on each pass through the loop, would be:

```
get-dirnames |
while read dir
do
    cd "$dir"
    date >> start-times
    echo "$dir" >> dirnames
    ...
done
```

It's more efficient to open the two log files once, then write directly to those open files. Here's the improved loop:

```
get-dirnames |
while read dir
do
    cd "$dir"
    date 1>&3
    echo "$dir" 1>&4
    ...
done 3>start-times 4>dirnames
```

As before, `date 1>&3` means "make f.d. 1 go the same place as f.d. 3": to the open file *start-times*, which was opened before the shell started to run the loop.

This is especially useful when more than one command within the loop needs to write to a particular file; it avoids constant re-opening of that file.

Higher file descriptors can also be used to "remember" where another file descriptor is pointing. That's handy if, say, you want to preserve the location of a file descriptor while you temporarily change it. It's also handy for swapping file descriptors.

For instance, you might want to use command substitution to capture the standard error of a process instead of the default, the standard output. Let's look at an example.

In this situation, the variable *text* gets the *stdout* of the program *prog*. *prog*'s *stderr* goes to the default location (typically, the terminal):

```
text=$(prog)
```

Here's how to route *prog*'s *stderr* to *text* and its *stdout* to wherever the *stderr* had been going (typically, the terminal):

```
text=$(prog 3>&2 2>&1 1>&3)
```

Reading that left to right, we're using f.d. 3 to remember where f.d. 2 (*stderr*) was pointing.

In Bourne-type shells, the *exec* command can also redirect open files permanently — that is, until the current shell process exits

Next we're making f.d. 2 point to the place that f.d. 1 is currently pointing — which is to the shell variable `$text`, via command substitution. Finally, we make f.d. 1 (the *stdout* of *prog*) point to the place that f.d. 3 has been pointing — which is the previous location of f.d. 2.

If that makes your head hurt, try drawing a table, or a series of diagrams, showing where each file descriptor points after each `m>&n` operator.

To close an open output file, use `m>&-`, where *m* is the file descriptor.

Finally: to redirect an open input file, use `m<&n`. To close an open input file, use `m<&-`.

Filenames? Who Needs Filenames?

We mentioned last month that, once the shell has opened a file, the filename isn't needed anymore.

You use the file descriptor instead. This fact leads to an obscure, and potentially useful, trick that shows a lot about how open files work. To demonstrate this, here's a shell script that creates a small file named *afile* with ten lines from the system password file.

Next, the script opens the file for reading on the standard input, uses `head -1` to read and output a line from the file (via *head*'s standard input), removes the file, uses *ls* to confirm that the file is gone, then reads and outputs another line from the file:

```
head /etc/passed > afile
exec < afile
head -1
rm afile
ls afile
head -1
```

Running that script gives output like this:

```
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
ls: afile: No such file or directory
bin:x:2:2:bin:/bin:/bin/sh
```

The important thing to understand here is that *rm* (1) doesn't actually remove the physical file blocks from the disk. Instead, it removes the *link* to the file — that is, the filename that allows the system to locate the corresponding disk blocks. The disk blocks themselves aren't removed (or added to a list of free disk blocks) until all processes with that file open have terminated.

You can see the open file by adding calls to `lsof -p $$` before the *exec* command and after the *rm* command. (`lsof -p $$` lists all files open by process `$$`, the current shell process.) After *rm*, the *lsof* output will show `/.../ afile(deleted)`. You'll also see the file descriptor numbers for *stdin*, *stdout* and *stderr* in the `FD` column — and the terminal's filename, like `/dev/pts/35`, in the `NAME` column for the standard I/O files.

Dangers of Nameless Files

You might think of using this trick for extra security with temporary files in */tmp*: if the temporary file doesn't have a name, it's hard for an attacker to replace or corrupt the file. Before you use this trick, though, consider the problems it can cause for a system administrator (maybe that's you!) if a filesystem fills up. The open file is occupying disk space, but there's no link to the file — that is, no filename — that the sysadmin can give to *rm* to free disk space. If the sysadmin knows what's happening, she can use *lsof* to find the process that has the file open and kill it.

This trick can also cause problems on some versions of NFS, where NFS has to make a temporary filename to stand in for the missing filename.

Named Pipes

Typing `proga | progb` tells a shell to create a pipe that connects the standard output of `proga` to the standard input of `progb`. The system also manages the two processes so that `proga` will be stopped when it's written some data for the `progb` to read — until `progb` has actually read that data. This kind of pipe is unnamed or anonymous; it connects two specific processes.

A *named pipe* or *FIFO* (first in, first out) is similar. But it's a reusable pipe in the filesytem, with a name like any other file. A FIFO is handy in cases when you want two arbitrary processes to communicate. The first process writes to the FIFO with the shell's `>` operator — or by writing directly to that filename. The second process reads from the FIFO by opening it as it would open a plain text file — or by using the shell's `<` operator.

To make a FIFO, use *mkfifo* (1). When you list a FIFO with `ls-l`, the file type is `p`. Using `ls-F`, a FIFO is marked with a trailing `|` character:

```
$ mkfifo readme
$ ls -l readme
prw-r—r—  1 jpeek users 0 ... readme
$ ls -F readme
readme|
```

One place you can use a FIFO is when you want to watch the output of a process running on one terminal from a window on a different terminal. Here's a simple shell script named *writer* that runs the *date* command every three seconds, sending its output both to the terminal (via *stdout*) and to our FIFO named *readme* (via f.d. 3):

```
while sleep 3
do
    dateout=$(date)
    echo "$dateout"
    echo "$dateout" 1>&3
done 3> readme
```

In one terminal, run *writer*. Not much happens because the process is being blocked, waiting for the write to the FIFO to succeed.

In the other terminal, run `cat readme` and wait a moment. Now you should start seeing output in both terminals every three seconds: via *stdout* on the terminal running *writer*, and via the FIFO on the terminal running *cat*. If you kill either process (either *writer* or *cat*) with `CTRL-C`, both processes exit. But the FIFO is still in the filesystem, so you can use it again. To remove the FIFO, use *rm*. It may take some time to familiarize yourself with all of the shell's file descriptors and such, but it's well worth it.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for 25+ years. (http://www.jpeek.com/contact.html.)