Catching Some ZZZs

By Jerry Peek

Linux systems have several shells available — some more powerful than others. Perhaps the most powerful shell is *zsh*, the Z shell. To give you an idea of the shell's size and complexity, the *zshall man* page in plain-text format weighs in at more than 16,000 lines. (Rest easy. We won't try to cover all of *zsh* in this month's column.)

Instead, we'll investigate some of the more-obscure *zsh* features that might still be very handy for you. (For a look at some better-known *zsh* features, see the "Power Tools" column in the May, June, and July 2002 issues of Linux Magazine. You can find them online at http://www.linux-mag.com/2002-05/power_01.html, http://www.linux-mag.com/2002-06/power_01.html, and http://www.linux-mag.com/2002-07/power_01.html. If you're new to shells, the things we do here will also show you more of the power of this very useful tool.

By the way, if you want to use *zsh* occasionally, there's no need to change your default shell to *zsh*. This month's Power Tip explains how to switch shells temporarily.

Labeled Secondary Prompts

If you've written shell scripts, you probably know how to use loops (like `for` and `while`) and other complex commands. You may also have written multi-line commands — for instance, using *echo* to output several lines of text at once. All shells let you type these complex commands on a command line, as well as from a script file. But when you nest these (use one complex command "inside" another) on a command line, it can be hard to remember where you are. (Are you typing a command in the inner loop or the outer loop? Do you have an open quote without a closing quote?) Z shell provides helpful cues: it outputs secondary prompts that show which constructs you haven't finished.

For example, you'd like to print all of the files in your current directory. The PNG and JPEG files should be printed with `lpr -Pgfx`; other files should be printed with `lpr -Pps`. You can use a `for` loop to step through all files (from the wildcard `*`) one by one, storing each filename in the shell variable `file`. Inside the `for` loop, a `case` statement tests the filename and runs the right printer command. (You could use `if` instead of the more-cryptic `case`, but it takes longer to type.) In *Listing One*, notice how *zsh* prompts as you go.

Multiple-file I/O

In most shells, the `>` ("greater than") operator overwrites a single file with the output of a command, `>>` appends the command output to a file, and `<` ("less than") takes standard input from a single file. In most shells, if you want to write to or read from more than one file, you need workarounds. Not with *zsh*. Its `MULTIOS` option — which is enabled by default — lets you read from and write to multiple files.

Let's say you're a manager who wants to email a notice to each of the people in your office, telling them how big a pay raise they'll get. Each message has a standard start and finish, but the body is customized for that person. The customized parts are in files named by the person's email address — for instance, the file *amyo* has *amyo*'s custom message. The start and finish come from the files *START* and *END*, respectively.

The *mail* program reads the address from its command line and the message body from its standard input. Since you want to send three files to each person — all merged into one mail message — you'll use three redirection operators, as in *Listing Two*.

Redirecting output to multiple files is handy when you want to write the same information to more than one file. For instance, let's say you have a script that keeps nine log files named *log1* through *log9*. The higher-numbered log files get the most detail (the most-verbose output), and the lowest-numbered files get the least detail (the most-condensed summary).

So, for example, the lowest-priority message would be written only to *log9*, but the highest-priority message would be written to all nine files, *log1* through *log9*.

When `MULTIOS` is set, *zsh* also automatically expands wildcards after the `<`, `>`, and `>>` operators. That's the key to our example. So for instance, your script could append a low-priority message to *log7*, *log8*, and *log9* with the following

LISTING ONE: *zsh* prompts show complex command nesting

```
zsh% for file in *
for> do
for> case "$file" in
for case> *.png|*.jpg) printer=gfx ;;
for case> *) printer=ps ;;
for case> esac
for> echo "printing $file on $printer"
for> lpr -P$printer $file
for> done
printing afile on ps
printing apple.png on gfx
    ...
printing zsh-tips on ps
zsh%
```

LISTING TWO: *zsh* can take input from multiple files

```
zsh% for person in [a-z]*
for> do mail -s "Your raise" \
  $person@ourcorp.xyz \
  < START < $person < END
for> done
zsh%
```

command line (where `$datestr` holds the date and time and `$message` holds the message):

```
zsh% echo "$datestr: $message" >> log[7-9]
```

Or, you can name files individually, without wildcards:

```
zsh% echo "$datestr: $message" >> log8 >> log9
```

And, to append an important message to all files:

```
zsh% echo "$datestr: $message" >> log*
```

Remember that wildcards can only match filenames that already exist! If you want to write to multiple files and only some (or none) of them exist, the handy curly-brace operators `{a,b,c}` will do the trick.

For instance, to quickly initialize all nine log files, this one command suffices:

```
zsh% echo "$datestr: starting job" >
  log{1,2,3,4,5,6,7,8,9}
```

(If you aren't using *zsh*, you can do a similar thing with the *tee* utility, which can write to multiple files. If you want to append to existing files, add the *tee* `-a` switch. Be sure to redirect *tee*'s standard output to */dev/null*.)

Floating-point Numbers, Non-decimal Numbers

All shells can do integer arithmetic. (The most basic shells can call the *expr* program.) In *zsh*, you can write non-decimal numbers (not base 10) by putting *base*`#` before the number, where *base* is the base of the number. For instance, `16#ff` is 255 in hexadecimal (base 16). (You can also use `0xff`.) The *zshmisc man* page explains how to set output bases — and more.

The Z shell's built-in command *float* declares a floating-point variable, and Z shell's built-in operators give double-precision accuracy. As a simple example, let's say you've worked 8 hours and 34 minutes today and 9 hours and 5 minutes yesterday. You can total the hours like this:

```
zsh% float hours
zsh% (( hours = 8.0 + 34.0 / 60.0 ))
zsh% (( hours += 9.0 + 5.0 / 60.0 ))
zsh% echo $hours
1.765000000e+01
```

(This might be more useful in a script than on a command line.) You might want another output format; see the *zshmisc man* page for details. Note also that *zsh* has both integer and floating-point variables. It's safest to declare a variable as floating-point or initialize it with a value that includes a decimal point.

Process Substitution

The Z shell isn't the only shell that can do process substitution, but *zsh* does it in a really flexible way — especially when its `MULTIOS` option is set. Process substitution is new to a lot of Linux users, though, so let's start by reviewing what it's for.

Standard GNU/Linux utilities (*grep*, *sort*, and so on) can read from one or many files that you provide as arguments (parameters) on the command line. So, for instance, to search the files *foo*, *bar*, and *baz* for any line containing *burger*, you'd type:

```
zsh% grep burger foo bar baz
```

But what if you don't want to search files — and, instead, you want to search the output of *processes* (other running programs)? Well, because *grep* is a well-behaved Linux program (it acts like a *filter*), it reads its standard input. That lets you read the output of those programs one-by-one:

```
zsh% prog1 | grep burger
zsh% prog2 | grep burger
zsh% prog3 | grep burger
  ...
```

But how tedious. And, in some cases, you really *need* to read the output of several processes all at once. You can work around the problem by storing the output of each process in a temporary file, then reading all of the files at once:

```
zsh% prog1 > temp1
zsh% prog2 > temp2
  ...
zsh% paste temp1 temp2 ... > pasteout
zsh% rm temp1 temp2 ...
```

The GNU *paste* utility creates a multi-column output by reading each of its input files line-by-line. So, the *paste* command above makes a file named *pasteout* where the first column of the first line of *pasteout* has the first line of the *temp1* file (which is the first line of *prog1* output), and the second column of the first

line of *pasteout* has the first line of the *temp2* file (the first line of *prog2* output). Got it? If you haven't done this before, it's a bit tricky to visualize — but hang in there and you'll see where we're going!

Now let's see where process substitution fits in. There are three kinds; we'll use the one we need here, then mention the other two.

When *zsh* (and other shells that have process substitution) see the operator `(process)`, they run the program called *process*; store its output in a temporary location, and then substitute that operator with the pathname of the temporary storage location.

Huh? If you look at the previous example, written with process substitution, this should be clearer:

```
zsh% paste <(prog1) <(prog2) ... > pasteout
```

Presto! No temporary files! Well, there *are* temporary files, or named pipes, or (in the case of many Linux systems), the */dev/fd* file descriptor device "files." The Z shell is especially good at deciding which of these temporary storage locations to use, and you don't have to worry about it; just use the `(process)` operator and the shell does the rest.

Now let's turn to the rest of this example (which we haven't seen yet). Actually, the *pasteout* file is a temporary file, and we don't really want it. We want to send the output of *paste* to the high-speed laser printer down the hall — and also see the first ten lines of the output so we can recognize the first page of the print job. If we have the *pasteout* file sitting around, the obvious way to do this is:

```
zsh% lpr -Phs pasteout
zsh% head pasteout
   ...first ten lines of pasteout...
zsh% rm pasteout
```

But that's tedious, too. Any guesses about how to get rid of the temporary file? Right! It's process substitution again. What you want to do in this case is to write the standard output of the *paste* process to *two* other processes. Shells with process substitution have an operator `>(process)`. In that case, it runs the program called *process*, stores the output of that file in a temporary location, and replaces that operator with the pathname of that temporary location. The *zsh* `MULTIOS` option lets you use this operator multiple times.

Huh? Yes, it's time again for an example! We've broken this across multiple lines for printing, but you can type it all on one command line:

```
zsh% paste <(prog1) <(prog2) \
  > >(lpr -Phs) > >(head)
   ...first ten lines of paste output...
```

POWER TIP: CHANGING SHELLS TEMPORARILY

If you want to use some feature of another shell (*zsh*, for instance), you don't need to change your default shell. Simply type the new shell's name at your current shell prompt. The new shell will prompt you and interpret your command lines until you type *exit* to end it. Then you'll get a prompt from your usual shell. (Shells are simply programs. You can run a new shell just as you'd run any other Linux program.)

```
bash$ zsh
zsh% ...run commands...
zsh% exit
bash$
```

There's one wrinkle in this picture. Some programs don't work well with the `(process)` operator. Some of these problem programs need to "seek" around the temporary file with a system call like `lseek()` (re-reading the file several times, for instance, or reading various locations). On systems where the temporary storage is actually a FIFO (a named pipe), the program reading the storage location can block and may have to be killed manually.

If all of that sounds like gibberish to you, here's the *zsh* answer. If the `(process)` operator doesn't work, use the `=(process)` operator instead. This tells the shell to use an actual file as the temporary storage location. (The shell will remove the temporary file after you're done with it.)

Much more about *zsh*

The *zsh* examples you've seen here are just a few things this amazing shell can do, picked more or less at random (by reading through the first ten percent or so of the *zshall man* page!). There's much more.

A good place to get snapshots of other things people are doing with this versatile shell is the *zsh* mailing list archives at http://www.zsh.org/mla. There's also a frequently-asked questions list at http://www.zsh.org/FAQ.

For a complete wrapup of this amazing shell (as well as *bash* and a general introduction to what shells can do for you), watch for (shameless plug coming...) *From bash to Z Shell: Conquering the Command Line* by Oliver Kiddle, Peter Stephenson (two *zsh* developers), and Jerry Peek. It's due to be published by Apress (http://www.apress.com) in June 2004.

And a footnote: Many thanks to La Laundry, the Linux-powered laundromat and Internet point in Barcelona, Spain, where your author finished this column while he was doing his wash.

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 at jpeek@jpeek.com.