36 – Exotica

In this, the final chapter of our journey, we will look at some odds and ends. While we have certainly covered a lot of ground in the previous chapters, there are many `bash` features that we have not covered. Most are fairly obscure and useful mainly to those integrating `bash` into a Linux distribution. However, there are a few that, while not in common use, are helpful for certain programming problems. We will cover them here.

Group Commands and Subshells

`bash` allows commands to be grouped together. This can be done in one of two ways, either with a *group command* or with a *subshell*.

Here are examples of the syntax of a group command:

```
{ command1; command2; [command3; ...] }
```

Here is the syntax of a subshell:

```
(command1; command2; [command3;...])
```

The two forms differ in that a group command surrounds its commands with braces and a subshell uses parentheses. It is important to note that, because of the way `bash` implements group commands, the braces must be separated from the commands by a space and the last command must be terminated with either a semicolon or a newline prior to the closing brace.

So, what are group commands and subshells good for? While they have an important difference (which we will get to in a moment), they are both used to manage redirection. Let's consider a script segment that performs redirections on multiple commands.

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

This is pretty straightforward. Three commands have their output redirected to a file named `output.txt`. Using a group command, we could code this as follows:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

Using a subshell is similar.

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

Using this technique we have saved ourselves some typing, but where a group command or subshell really shines is with pipelines. When constructing a pipeline of commands, it is often useful to combine the results of several commands into a single stream. Group commands and subshells make this easy.

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

Here we have combined the output of our three commands and piped them into the input of `lpr` to produce a printed report.

In the script that follows, we will use groups commands and look at several programming techniques that can be employed in conjunction with associative arrays. This script, called `array-2`, when given the name of a directory, prints a listing of the files in the directory along with the names of the file's owner and group owner. At the end of the listing, the script prints a tally of the number of files belonging to each owner and group. Here we see the results (condensed for brevity) when the script is given the directory `/usr/bin`:

```
[me@linuxbox ~]$ array-2 /usr/bin
/usr/bin/2to3-2.6                       root       root
/usr/bin/2to3                           root       root
/usr/bin/a2p                            root       root
/usr/bin/abrowser                       root       root
/usr/bin/aconnect                       root       root
/usr/bin/acpi_fakekey                   root       root
/usr/bin/acpi_listen                    root       root
/usr/bin/add-apt-repository             root       root
.
.
.
/usr/bin/zipgrep                        root       root
/usr/bin/zipinfo                        root       root
/usr/bin/zipnote                        root       root
```

```
/usr/bin/zip                                root       root
/usr/bin/zipsplit                           root       root
/usr/bin/zjsdecode                          root       root
/usr/bin/zsoelim                            root       root

File owners:
daemon    :     1 file(s)
root      :  1394 file(s)

File group owners:
crontab   :     1 file(s)
daemon    :     1 file(s)
lpadmin   :     1 file(s)
mail      :     4 file(s)
mlocate   :     1 file(s)
root      :  1380 file(s)
shadow    :     2 file(s)
ssh       :     1 file(s)
tty       :     2 file(s)
utmp      :     2 file(s)
```

Here is a listing (with line numbers) of the script:

```
 1    #!/bin/bash
 2
 3    # array-2: Use arrays to tally file owners
 4
 5    declare -A files file_group file_owner groups owners
 6
 7    if [[ ! -d "$1" ]]; then
 8        echo "Usage: array-2 dir" >&2
 9        exit 1
10    fi
11
12    for i in "$1"/*; do
13        owner="$(stat -c %U "$i")"
14        group="$(stat -c %G "$i")"
15        files["$i"]="$i"
16        file_owner["$i"]="$owner"
17        file_group["$i"]="$group"
18        ((++owners[$owner]))
19        ((++groups[$group]))
```

```
20    done
21
22    # List the collected files
23    { for i in "${files[@]}"; do
24        printf "%-40s %-10s %-10s\n" \
25            "$i" "${file_owner["$i"]}" "${file_group["$i"]}"
26    done } | sort
27    echo
28
29    # List owners
30    echo "File owners:"
31    { for i in "${!owners[@]}"; do
32        printf "%-10s: %5d file(s)\n" "$i" "${owners["$i"]}"
33    done } | sort
34    echo
35
36    # List groups
37    echo "File group owners:"
38    { for i in "${!groups[@]}"; do
39        printf "%-10s: %5d file(s)\n" "$i" "${groups["$i"]}"
40    done } | sort
```

Let's take a look at the mechanics of this script:

Line 5: Associative arrays must be created with the `declare` command using the `-A` option. In this script we create five arrays as follows:

- `files` contains the names of the files in the directory, indexed by filename
- `file_group` contains the group owner of each file, indexed by filename
- `file_owner` contains the owner of each file, indexed by file name
- `groups` contains the number of files belonging to the indexed group
- `owners` contains the number of files belonging to the indexed owner

Lines 7-10: These lines check to see that a valid directory name was passed as a positional parameter. If not, a usage message is displayed and the script exits with an exit status of 1.

Lines 12-20: These lines loop through the files in the directory. Using the `stat` command, lines 13 and 14 extract the names of the file owner and group owner and assign the values to their respective arrays (lines 16 and 17) using the name of the file as the array index. Likewise, the file name itself is assigned to the `files` array (line 15).

Lines 18-19: The total number of files belonging to the file owner and group owner are

incremented by one.

Lines 22-27: The list of files is output. This is done using the `"${array[@]}"` parameter expansion which expands into the entire list of array elements with each element treated as a separate word. This allows for the possibility that a filename may contain embedded spaces. Also note that the entire loop is enclosed in braces thus forming a group command. This permits the entire output of the loop to be piped into the `sort` command. This is necessary because the expansion of the array elements is not sorted.

Lines 29-40: These two loops are similar to the file list loop except that they use the `"${!array[@]}"` expansion which expands into the list of array indexes rather than the list of array elements.

Process Substitution

While they look similar and can both be used to combine streams for redirection, there is an important difference between group commands and subshells. Whereas a group command executes all of its commands in the current shell, a subshell (as the name suggests) executes its commands in a child copy of the current shell. This means the environment is copied and given to a new instance of the shell. When the subshell exits, the copy of the environment is lost, so any changes made to the subshell's environment (including variable assignment) are lost as well. Therefore, in most cases, unless a script requires a subshell, group commands are preferable to subshells. Group commands are both faster and require less memory.

We saw an example of the subshell environment problem in Chapter 28, "Reading Keyboard Input," when we discovered that a `read` command in a pipeline does not work as we might intuitively expect. To recap, if we construct a pipeline like this:

```
echo "foo" | read
echo $REPLY
```

the content of the `REPLY` variable is always empty because the `read` command is executed in a subshell, and its copy of `REPLY` is destroyed when the subshell terminates.

Because commands in pipelines are always executed in subshells, any command that assigns variables will encounter this issue. Fortunately, the shell provides an exotic form of expansion called *process substitution* that can be used to work around this problem.

Process substitution is expressed in two ways.

For processes that produce standard output, it looks like this:

`<(`*`list`*`)`

or, for processes that intake standard input, it looks like this:

`>(`*`list`*`)`

where *list* is a list of commands.

To solve our problem with `read`, we can employ process substitution like this:

```
read < <(echo "foo")
echo $REPLY
```

Process substitution allows us to treat the output of a subshell as an ordinary file for purposes of redirection. In fact, since it is a form of expansion, we can examine its real value.

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

By using `echo` to view the result of the expansion, we see that the output of the subshell is being provided by a file named `/dev/fd/63`.

Process substitution is often used with loops containing `read`. Here is an example of a `read` loop that processes the contents of a directory listing created by a subshell:

```
#!/bin/bash

# pro-sub: demo of process substitution

while read attr links owner group size date time filename; do
    cat <<- EOF
         Filename:   $filename
         Size:       $size
         Owner:      $owner
         Group:      $group
         Modified:   $date $time
         Links:      $links
         Attributes: $attr

    EOF
done < <(ls -l | tail -n +2)
```

The loop executes `read` for each line of a directory listing. The listing itself is produced

on the final line of the script. This line redirects the output of the process substitution into the standard input of the loop. The `tail` command is included in the process substitution pipeline to eliminate the first line of the listing, which is not needed.

When executed, the script produces output like this:

```
[me@linuxbox ~]$ pro-sub | head -n 20
Filename:   addresses.ldif
Size:       14540
Owner:      me
Group:      me
Modified:   2009-04-02 11:12
Links:      1
Attributes: -rw-r--r--

Filename:   bin
Size:       4096
Owner:      me
Group:      me
Modified:   2009-07-10 07:31
Links:      2
Attributes: drwxr-xr-x

Filename:   bookmarks.html
Size:       394213
Owner:      me
Group:      me
```

Traps

In Chapter 10, “Processes,” we saw how programs can respond to signals. We can add this capability to our scripts, too. While the scripts we have written so far have not needed this capability (because they have very short execution times, and do not create temporary files), larger and more complicated scripts may benefit from having a signal handling routine.

When we design a large, complicated script, it is important to consider what happens if the user logs off or shuts down the computer while the script is running. When such an event occurs, a signal will be sent to all affected processes. In turn, the programs representing those processes can perform actions to ensure a proper and orderly termination of the program. Let’s say, for example, that we wrote a script that created a temporary file during its execution. In the course of good design, we would have the script delete the file when the script finishes its work. It would also be smart to have the script delete the file

if a signal is received indicating that the program was going to be terminated prematurely.

`bash` provides a mechanism for this purpose known as a *trap*. Traps are implemented with the appropriately named builtin command, `trap`. `trap` uses the following syntax:

`trap` *`argument signal`* [*`signal...`*]

where *argument* is a string that will be read and treated as a command and *signal* is the specification of a signal that will trigger the execution of the interpreted command.

Here is a simple example:

```
#!/bin/bash

# trap-demo: simple signal handling demo

trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

This script defines a trap that will execute an `echo` command each time either the SIGINT or SIGTERM signal is received while the script is running. Execution of the program looks like this when the user attempts to stop the script by pressing `Ctrl-c`:

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
^CI am ignoring you.
Iteration 3 of 5
^CI am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

As we can see, each time the user attempts to interrupt the program, the message is printed instead.

Constructing a string to form a useful sequence of commands can be awkward, so it is common practice to specify a shell function as the command. In this example, a separate shell function is specified for each signal to be handled:

```
#!/bin/bash

# trap-demo2: simple signal handling demo

exit_on_signal_SIGINT () {
    echo "Script interrupted." 2>&1
    exit 0
}

exit_on_signal_SIGTERM () {
    echo "Script terminated." 2>&1
    exit 0
}

trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

This script features two `trap` commands, one for each signal. Each trap, in turn, specifies a shell function to be executed when the particular signal is received. Note the inclusion of an `exit` command in each of the signal-handling functions. Without an `exit`, the script would continue after completing the function.

When the user presses `Ctrl-c` during the execution of this script, the results look like this:

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
^CScript interrupted.
```

Temporary Files

One reason signal handlers are included in scripts is to remove temporary files that the script may create to hold intermediate results during execution. There is something of an art to naming temporary files. Traditionally, programs on Unix-like systems create their temporary files in the `/tmp` directory, a shared directory intended for such files. However, since the directory is shared, this poses certain security concerns, particularly for programs running with superuser privileges. Aside from the obvious step of setting proper permissions for files exposed to all users of the system, it is important to give temporary files nonpredictable filenames. This avoids an exploit known as a *temp race attack*. One way to create a nonpredictable (but still descriptive) name is to do something like this:

```
tempfile=/tmp/$(basename $0).$$.$RANDOM
```

This will create a filename consisting of the program's name, followed by its process ID (PID), followed by a random integer. Note, however, that the `$RANDOM` shell variable returns only a value in the range of 1-32767, which is not a large range in computer terms, so a single instance of the variable is not sufficient to overcome a determined attacker.

A better way is to use the `mktemp` program (not to be confused with the `mktemp` standard library function) to both name and create the temporary file. The `mktemp` program accepts a template as an argument that is used to build the filename. The template should include a series of "X" characters, which are replaced by a corresponding number of random letters and numbers. The longer the series of "X" characters, the longer the series of random characters. Here is an example:

```
tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

This creates a temporary file and assigns its name to the variable `tempfile`. The "X" characters in the template are replaced with random letters and numbers so that the final filename (which, in this example, also includes the expanded value of the special parameter `$$` to obtain the PID) might be something like this:

```
/tmp/foobar.6593.UOZuvM6654
```

For scripts that are executed by regular users, it may be wise to avoid the use of the `/tmp` directory and create a directory for temporary files within the user's home directory, with a line of code such as this:

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

Asynchronous Execution

It is sometimes desirable to perform more than one task at the same time. We have seen how all modern operating systems are at least multitasking if not multiuser as well. Scripts can be constructed to behave in a multitasking fashion.

Usually this involves launching a script that, in turn, launches one or more child scripts to perform an additional task while the parent script continues to run. However, when a series of scripts runs this way, there can be problems keeping the parent and child coordinated. That is, what if the parent or child is dependent on the other and one script must wait for the other to finish its task before finishing its own?

`bash` has a builtin command to help manage *asynchronous execution* such as this. The `wait` command causes a parent script to pause until a specified process (i.e., the child script) finishes.

`wait`

We will demonstrate the `wait` command first. To do this, we will need two scripts. First a parent script.

```
#!/bin/bash

# async-parent: Asynchronous execution demo (parent)

echo "Parent: starting..."

echo "Parent: launching child script..."
async-child &
pid=$!
echo "Parent: child (PID= $pid) launched."

echo "Parent: continuing..."
sleep 2

echo "Parent: pausing to wait for child to finish..."
wait "$pid"

echo "Parent: child is finished. Continuing..."
echo "Parent: parent is done. Exiting."
```

The second is a child script.

```
#!/bin/bash

# async-child: Asynchronous execution demo (child)

echo "Child: child is running..."
sleep 5
echo "Child: child is done. Exiting."
```

In this example, we see that the child script is simple. The real action is being performed by the parent. In the parent script, the child script is launched and put into the background. The process ID of the child script is recorded by assigning the `pid` variable with the value of the `$!` shell parameter, which will always contain the process ID of the last job put into the background.

The parent script continues and then executes a `wait` command with the PID of the child process. This causes the parent script to pause until the child script exits, at which point the parent script concludes.

When executed, the parent and child scripts produce the following output:

```
[me@linuxbox ~]$ async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child: child is running...
Parent: pausing to wait for child to finish...
Child: child is done. Exiting.
Parent: child is finished. Continuing...
Parent: parent is done. Exiting.
```

Named Pipes

In most Unix-like systems, it is possible to create a special type of file called a *named pipe*. Named pipes are used to create a connection between two processes and can be used just like other types of files. They are not that popular, but they’re good to know about.

There is a common programming architecture called *client-server*, which can make use of a communication method such as named pipes, as well as other kinds of *interprocess communication* such as network connections.

The most widely used type of client-server system is, of course, a web browser communi-

cating with a web server. The web browser acts as the client, making requests to the server, and the server responds to the browser with web pages.

Named pipes behave like files but actually form first-in first-out (FIFO) buffers. As with ordinary (unnamed) pipes, data goes in one end and emerges out the other. With named pipes, it is possible to set up something like this:

```
process1 > named_pipe
```

and this:

```
process2 < named_pipe
```

and it will behave like this:

```
process1 | process2
```

Setting Up a Named Pipe

First, we must create a named pipe. This is done using the `mkfifo` command.

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me   me    0 2009-07-17 06:41 pipe1
```

Here we use `mkfifo` to create a named pipe called `pipe1`. Using `ls`, we examine the file and see that the first letter in the attributes field is “p”, indicating that it is a named pipe.

Using Named Pipes

To demonstrate how the named pipe works, we will need two terminal windows (or alternately, two virtual consoles). In the first terminal, we enter a simple command and redirect its output to the named pipe.

```
[me@linuxbox ~]$ ls -l > pipe1
```

After we press the `Enter` key, the command will appear to hang. This is because there is nothing receiving data from the other end of the pipe yet. When this occurs, it is said that the pipe is *blocked*. This condition will clear once we attach a process to the other end and it begins to read input from the pipe. Using the second terminal window, we enter this command:

```
[me@linuxbox ~]$ cat < pipe1
```

The directory listing produced from the first terminal window appears in the second terminal as the output from the `cat` command. The `ls` command in the first terminal successfully completes once it is no longer blocked.

Summing Up

Well, we have completed our journey. The only thing left to do now is practice, practice, practice. Even though we covered a lot of ground in our trek, we barely scratched the surface as far as the command line goes. There are still thousands of command line programs left to be discovered and enjoyed. Start digging around in `/usr/bin` and you'll see!

Further Reading

- The "Compound Commands" section of the bash man page contains a full description of group command and subshell notations.
- The EXPANSION section of the bash man page contains a subsection covering process substitution.
- *The Advanced Bash-Scripting Guide* also has a discussion of process substitution: http://tldp.org/LDP/abs/html/process-sub.html
- *Linux Journal* has two good articles on named pipes. The first, from September 1997: http://www.linuxjournal.com/article/2156
- and the second, from March 2009: http://www.linuxjournal.com/content/using-named-pipes-fifos-bash