by Peter Collinson, **Hillside Systems**

STEPHEN SCHILDBACH

Over

The main content of this article is shamelessly stolen from *The UNIX Programming Environment* by Brian Kernighan and Rob Pike. Written in 1984, this book remains one of the seminal books on UNIX programming. The book has several gems that can enhance your use of the system. Back in 1985, I typed in one of the scripts and I have been using it ever since. Of course, I had to type it in, there was no Internet back then and CD-ROMs for computer use hadn't been invented. There are just so many things we take for granted as time progresses.

Let's start with the basic problem that the script solves. If you've typed commands into a shell, then you probably know that you can redirect the output of a command or command pipeline to a file using the > syntax. The command

```
$ sort -r /usr/dict/words > \
    reverse
```

will sort the list of words into reverse order and capture the output in a file called `reverse`. The line was split for printing; I've inserted a backslash to maintain the shell syntax. The root of the problem I am examining occurs directly as a result of the way that redirection works. Essentially, when executing the line above, the shell creates the output file `reverse` *before* it invokes the `sort` command. The shell alters the standard output channel of the `sort` command, pointing it at the newly created file. The `sort` command needs to take no special action to generate the output in the file, it just writes data. The shell has arranged that its standard output channel is pointing at the file, and the data will be written there.

For most usage, all this works well. There are often situations, however, where you want to perform a transformation on a file, and place the data back into the original file. So we want to say something like the following:

```
$ sort reverse > reverse
```

This is an instant disaster because the shell creates the output file `reverse` before it calls the command that's intended to sort it. The act of creating a new file will delete its contents, so the redirection will throw away the previous data in `reverse`. The command runs–the `sort` command has an easy time of sorting a zero-length file–and the result is, you guessed it, an empty file. We have waved goodbye to all the data in `reverse`. If we are not aware of what is happening, we probably blame the `sort` command for losing our data, when we've expressly destroyed it by using the redirection system in the shell.

The simple, and ultimately tedious, way out of the problem is to use a temporary file:

```
$ sort reverse > L
$ mv L reverse
```

Temporary files are a pain, because they are litter. Humans are great at dropping litter all over the planet and its environs.

Your file systems are also probably full of junk files you created at some time for some purpose and now you cannot remember whether you need them or not. Mine are.

One common strategy is to always put junk files into `/tmp`. The directory exists for this purpose and is generally cleared automatically when the system is booted. However, on a shared system, someone else might already have a file called `L` in `/tmp` and this will get in the way of what you want to do. If it's writable by you, then you will destroy the old contents, which may not be a social act. If you cannot write to it, then you'll get "Permission denied" and have to think of a new temporary name. Generally, if you use `/tmp`, you need to worry about file name usage in the directory.

Some sites have adopted other strategies for removing litter. On one of the sites I ran, we automatically deleted all files starting with a comma every night, and encouraged the users to create junk file names that started with a comma. This was partially successful. I've personally always used `L`, and so whenever I find a file called `L` lying around the system, I know I can delete it. This again partially works until I need several temporary files.

Anyway, wouldn't it be nice if there was some way of enabling an arbitrary command to overwrite one of its input files, without having to worry about naming the temporary file? One of the scripts in *The UNIX Programming Environment* is designed to provide a solution to this problem. It's called `overwrite`, which is far too much for me to type, so I've always called it `over`.

What follows builds up to a Bourne shell script. If you are running on a publicly available UNIX, then please beware of using the Bourne Shell clone programs. Many of the available clones don't copy the action of the shell exactly. If in doubt, use `bash` instead of `sh` on your system. Actually, `bash` doesn't clone exactly either, but it's close, and for the purpose of this article, it will work.

Thinking About the Problem

The first stab at a solution to the problem makes use of UNIX pipes. Rather than using the redirect shell facilities to create the output file, we make a new command that does the job for us:

```
sort reverse | over reverse
```

The `sort` command will work as before, but this time its output is sent down a pipe into a new command, `over`. The `over` command is given an argument, which is the file name to be used for output storage. The command cannot create the output file until it has received all the input from the pipe. Its fundamental job is to read data from its standard input and stash it somewhere (perhaps on `/tmp`). When it gets an end-of-file indicator on its standard input, it will move all the captured data from the temporary storage location to the file named by its argument.

When you are writing a script that is likely to be part of your working environment, it's a good idea to consider what error management is needed. It's inevitable that one day you will forget the assumptions that were made when you created the script. Even if you've written the script, one day you will become its idiot user. It's important to build in safeguards that prevent damage and problems later.

Arguments

The first point of worry is the arguments you give to the script. I always attempt to test for the validity of arguments, even if the script is "just for a one-off job." I have a pet aphorism about such tasks: "one-off jobs always happen more than once." Simple tests for the validity of the arguments to a script can save much grief.

The first assumption this script makes is that it will have a single argument that is the name of the file to be used for output. The Bourne shell gives us a piece of magic that supplies the number of arguments to a script, `$#`. We can use this in a test:

```
if [ $# != 1 ]
then
    echo 'Usage: over filename'
    exit 1
fi
```

There is actually a whole ton of UNIX history in the first line. The `if` statement in the Bourne shell tests the success or failure of a command. All commands return a value, and the UNIX kernel allows the parent that started the command to obtain its value when the command terminates. By convention, a zero value means "success" and a non-zero value means "failure." I'm following the convention in the script above, when I find that the command has been called with less than or more than one parameter, I print an error message and call the `exit` command with a parameter of `1`.

Originally, when an `if` statement using square brackets was run, the shell ran a command called `[`, which lived on `/bin` and was a link to the `test` command. The shell knew about this shorthand syntax and expected to find and delete the trailing `]`. In the original Bourne shell, then, the `if` statement should be written as:

```
if test $# != 1
```

But this never looked nice or felt familiar, so most shell script authors used the square bracket syntax. (Beware of the exclamation point when using `bash`).

Actually, in *The UNIX Programming Environment*, you'll find that the shell's `case` statement is used to test the value:

```
case $# in
1)  ;;
*)  echo 'Usage: over filename'
    exit 1
esac
```

The `case` statement compares the value of the variable after `case` with the expressions to be found before the round brackets in the body of the statement. If the value `$#` is `1`, then the first option is executed. Two adjacent semi-colons are used to indicate the end of a set of commands that are executed when the `case` test string is matched. In the above example, there are no commands before the `;;` so nothing is done, and control falls out from `case` code into the next statement after it. The `*` matches all other values of `$#`, and the error message will be printed.

As you will see later, you can place alternates for matching before the round brackets:

```
case $# in
0|1)   echo 'Error message'...
```

The `echo` command will be executed when `$#` is either `0` or `1`.

The UNIX Programming Environment uses this seemingly overcomplex statement to perform a simple test for efficiency. At the time, all the testing for the matching in the `case` statement was done in the shell. The `if` statement meant that the shell would start a new command (the `[` or `test` command) to obtain a result. Later, the `test` command and its synonym `[` were built into the shell, and the efficiency reason for using `case` rather than `if` evaporated.

Right. We've dealt with making sure the arguments that we give to the program are correct. What next?

Temporary Files

Our `over` script needs to capture data in a temporary file before it creates its output file. In general, it's not a good idea to make a script create a temporary file in the current directory, because you may not have access permission to write in that directory. The current directory could be on read-only media, like a CD-ROM, and we want to ensure that the script will work, irrespective of where the user "is" in the file system. We really want to place temporary files on `/tmp`, and, as I've noted, we need to worry about the file names we use in that directory.

Burning a constant temporary name into the script is not a good idea, because we want to be able to run more than one invocation of the script at anyone time.

Burning a constant temporary name into the script is not a good idea, because we want to be able to run more than one invocation of the script at any one time. Using a constant name would inhibit this desire, because parallel script invocations would use the same name. We'd like to generate a name that is likely to be unique to permit parallel running.

The name doesn't have to be unique for all time. It only needs to be unique while we run the script. We know that each process has an ID number that's guaranteed to be unique while the process is running, and we can use this value to generate a temporary name. The shell gives us another magic variable that contains its process ID: `$$`. We can use this to generate a temporary file name. We'll write something like the following:

```
new=/tmp/overwr.$$
```

The `$$` is replaced by the process ID of the shell that's running the script and that name will be unique enough.

Signals

The next problem to consider is what happens when the user types Control-C to stop a running command. Our `over` script will be in one of two states when this happens. First, it could be getting data from its standard input and stashing it in the temporary file. In this case, when the user types Control-C, we'd like to be good citizens and delete the temporary file.

Second, the script could have received an end-of-file indicator and be copying the data back from `/tmp` to the final destination. In this case, the script will stop soon. The original data has been processed and the output is being written back. We want to ignore the Control-C and finish writing to ensure data integrity.

When the user types Control-C, the terminal interface converts the keystroke into a UNIX "signal" that's sent to all the processes attached to the terminal. When the process is next scheduled to run, it notices the signal and in the normal case will take the default action, which is to die. A UNIX process can "catch" the signal, electing to take some special action when it arrives. It can also ignore the signal. The Bourne shell contains the `trap` statement that allows for both eventualities.

In our first phase, we want to catch the signal, remove the temporary file and exit. So our script will contain something like the following:

```
trap 'rm -f $new; exit 1' 1 2 15
```

The word `trap` is followed by a command sequence that is actioned when an appropriate signal is received. We will force the removal of the temporary file and then exit. The command sequence is followed by a list of numbers that map onto the actual numbers used to code the signals. Control-C generates the "Interrupt" signal, `1`; Control-Shift-Backslash usually generates the "Quit" signal, `2`; and the `kill` command generates the "Terminate" signal by default, `15`. By default, the "Quit" signal causes a core dump of the program, and because this is mostly used only by programmers, it's often turned off.

After we have created the temporary file and received the end-of-file indicator on input, we'll ignore signals while we overwrite the original file, using

```
trap '' 1 2 15
```

Pulling it Together

Well, I've covered most of the tricky bits. There remains one fatal flaw in the planning. I've said that we will use the closure of a data stream coming down a pipe to signal the end of input. At the end of input, we'll move the information that we've stored back into the original file. The flaw in the thinking is that the command that is driving the pipe may itself fail, as in:

```
sed -e s/broken/ datafile | over datafile
```

Here, I've mistyped the editing command for `sed`, and the command will say:

```
sed: command garbled: s/broken/
```

The `over` command cannot see this message because it will be put out on the standard error channel pointing at the terminal. The script will think the command that's driving it has finished and will overwrite data file with the new contents, which will be empty. Ideally, we want the `over` command to know whether or not the command that's driving it has succeeded before copying the data over. The best and easiest way of determining whether the command has finished is to run the command under the control of `over` itself. Because the `over` script will be the parent of the commands being run, it can obtain and test the exit status of each command.

We need to recast our syntax to pass the command and the file into the `over` script:

```
$ over reverse sort -r reverse
```

We place the command at the end of the command line because we know there'll be a variable number of arguments to the command. We really are now able to write the script (see Listing 1).

The first two active lines are defensive. They preserve the user's search `PATH` variable for later use and then set one that's local to the script. This way, we ensure that our script will run using the standard UNIX utility set and not use any local commands established by the user. We then check whether we have more than two arguments and complain when we have zero or one. The error message here contains the syntax `1>&2`. This is the Bourne shell way of ensuring the error output from the message printed by `echo` is placed on the standard error channel.

We then pick up the final target file name from the argument list and use the `shift` command to leave only the command in the argument list. The `shift` command moves all the arguments up the list by one, so the file name will "drop off." After setting up the `trap` call to capture any signals the user may generate, we execute the command using the following:

```
if PATH=$opath "$@" > $new
```

We are using the general capability of the `if` statement to run a command and test its returned status when the command has finished. The command is contained in the argument list to the script; remember that we've removed the output file name from the start of the list. We invoke the command using `"$@"` and precede it by a local environment setting to reset the search path back to that of the user. You can always set an environment variable that's set only for that command in the Bourne shell by placing the appropriate variable setting statement before the command to be executed.

Listing 1. Over

```
#!/bin/sh
# overwrite: copy standard input to output
# after an EOF (from Kernighan and Pike)
opath=$PATH
PATH=/bin:/usr/bin
case $# in
0|1)   echo 'Usage: over file cmd [args]' 1>&2; exit 2
esac

file=$1; shift
new=/tmp/over.$$

trap 'rm -f $new;exit 1' 1 2 15       # clean up files

if PATH=$opath "$@" > $new    # collect input
then
  trap '' 1 2 15              # ignore signals
  cp $new $file
else
  echo "over: $1 failed, $file unchanged" 1>&2
  exit 1
fi

rm -f $new
exit 0
```

The `"$@"` is a little more shell magic. It ensures that a quoted argument on the input line is passed through intact as a single argument to the command. Basically, it means that all the funny things the user has typed will be passed through to the final command that's executed, as if the user had typed the command. So, the user types the command to be executed as arguments to the `over` script. The command will be run by the script using the user's search path, and its output will be captured in the temporary file.

When the command terminates, it will have either succeeded or failed. In the case of success, signals are ignored and the temporary file is copied back. The `cp` command is used to preserve ownership and permission bits on the output file. For failure, we print an error message and die with a non-zero exit status.

The `over` command is robust, easy to use, and, as I said at the beginning, I've been using it for years. It copes well with human error, flagging problems with input, handling the emergency use of Control-C and the failure of the source command. It's also a good demonstration of what you need to think about when creating shell scripts for your own use. There is good mileage in spending the time to make scripts into robust entities. I suspect it's one of those things you don't really notice until you don't take care to write a "good" script and are bitten.

Further Reading

The UNIX Programming Environment by Brian W. Kernighan and Rob Pike is published by Prentice-Hall Inc., ISBN 0-13-937681-X.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.