Shell Programming Pitfalls and Style Guide

Sami Tikka

July 8, 2011

1 Introduction

After having written and read a lot of shell scripts, I began to see there are some common mistakes the novice script writers make. Often the scripts seem to work, but they are hard to read, inefficient or they might fail when they are run in a way their author did not expect. Every programming language has its own idioms, ways of doing things, kind of like rules of thumb. It's the same with shell scripts. The following tutorial tries to help the novice to learn these idioms and thus become a better shell scripter.

2 Thanks

There are several people who have had valuable comments or contributions: Jonas Berlin, Timo Metsälä, Juha Autero, Tuukka Pienimäki.

Contents

Part I
Elements of Style

3 True and False Reversed

Every UNIX process has an exit value which is made available to the parent process. A long-standing convention states that the exit value zero indicates success and any other value some kind of error.

Since every command used in a shell script is a UNIX process, the same applies: zero is true and anything else is false. This runs contrary to the established convention from the C language where zero is false and anything else is true. However, if one manages to unlearn what has been taught in the C classes, life becomes much easier for the script writer.

Let's look at some examples. The script below is what a novice shell-scripter usually writes:

```
grep pattern file
ret=$?
if [ $ret -eq 0 ]; then
  echo 'Found it!'
else
  echo 'Not found'
fi
```

Compare that with the script from a more seasoned professional:

```
if grep pattern file; then
  echo 'Found it!'
else
  echo 'Not found'
fi
```

The pro might even pull it off using a one-liner:

```
grep pattern file && echo 'Found it!' || echo 'Not found'
```

4 Commands Usually Have a Useful Exit Value

It is common for the novice scripter to check for the output of the grep command. Look at the code sample below:

```
isdriverloaded() {
    addeddriver=`modinfo | grep fsvpn`
    if [ -z "$addeddriver" ]; then
```

```
        return 0
    fi
    return 1
}
```

What we really want to know is if grep found any lines matching the pattern. The grep command has a useful exit value, it returns zero (success) if one or more matches were found. So, instead of storing the (possibly large) output from grep, which we really don't care about, we should be checking grep's exit value. Note that the script writer has also committed the sin described in the previous section: failing to grasp that zero is true and 1 is false. A better version of the function would read:

```
isdriverloaded() {
  modinfo | grep fsvpn >/dev/null
}
```

To learn why it works, look at the explanation in the next section.

The tty command prints the name of the terminal device the process is attached to, or "not a tty" if there is no terminal. Again, the novice would capture the output of the tty command and check that somehow:

```
if [ "`tty`" = "not a tty" ]; then
    echo "a hard-to-read way of checking for tty"
fi
```

or

```
if tty | grep "not a tty" >/dev/null; then
    echo "just as bad"
fi
```

We are depending on the exact output of that command. In the former example above, we capture the command output, then invoke the test command to compare it. (The test command is also known as [, the opening bracket.) In the latter example, we also invoke two processes, only this time they run at the same time.

A better way would be to notice the tty command has an exit code and using it makes the script very readable and efficient and it would work even when the user's locale settings make tty talk in japanese or tagalog:

```
if ! tty >/dev/null; then
    echo "Look ma! No tty!"
fi
```

5 Invisible Exit Value

Shell functions always return a value, even when there is no return statement anywhere in the function. In such a case the exit value will be the exit value of the last process executed in the shell function. Consider the following function:

```
stop() {
  [ -x /etc/init.d/service ] && /etc/init.d/service stop
}
```

If you are checking the exit value from this function, you might be surprised. The exit value may not be the exit value of the `/etc/init.d/service`. If the file `/etc/init.d/service` was not executable or did not exist at all, the function would return a value indicating failure. You might be well advised to add an explicit return 0 to the end of the function.

Of course, sometimes this behaviour is exactly what you want:

```
isrunning() {
  [ -e /var/run/daemon.pid ] && \
  kill -0 `cat /var/run/daemon.pid` 2>/dev/null
}
```

6 Use true and false Instead of 1 and 0

When you want to use a variable like a Boolean, some people do it like this:

```
upgrade=1
if [ $upgrade -eq 1 ]; then
        do_upgrade
fi
```

The code is much more readable when written like this:

```
upgrade=true
if $upgrade; then
        do_upgrade
fi
```

This is possible because there are commands true and false. true is a real command that lives in `/usr/bin` or possibly in `/bin`. When you execute true, it does nothing except exits with code 0, which means success.

Similarly, there is the command false, which does nothing and exits with code 1 meaning failure.

7 Quoting For Fun and Profit

Quoting is simple, at least in principle. There are just two types of quotes: single quotes (`''`) and double quotes (`""`). Shell variables are expanded inside double quotes. Nothing is expanded inside single quotes. Any quotes disable filename expansion. Regarding filenames, it is usually a good practice to enclose variables that may contain filenames into double quotes. You'll never know when you encounter a filename with a space in it. Leave the filename without double quotes only when you have a filename with wildcard characters in it that you want the shell to expand.

Another difficult area is nested quoting. This kind of situation can arise for example when passing shell variables into an awk or perl script. See this script that prints out the login shell of a user.

```
awk -F: '$1 == "'"$1"'" { print $7 }' /etc/passwd
```

The first single quote is interpreted by the shell, which means the next double quote is passed to awk as is. The second `$1` is found quoted in double quotes, so the shell replaces it with the first command line argument. The double quotes are needed just in case the `$1` contains spaces. Then the single quotes protect the rest of the awk script from expansion. The script which is seen by the awk interpreter is (assuming 'ftp' was the first argument):

```
$1 == "ftp" { print $7 }
```

Sometimes splitting quotes could become overwhelming, especially if the text you want to splice in contains quotes. Then you might want to use a mechanism built into awk precisely for this purpose, the `-v` option:

```
awk -F: -v user=root \
'$1 == user { print "login shell for", user, "is", $7 }' /etc/passwd
```

The `-v` option originated from GNU version of awk and has since spread into other awk implementations, just be aware it might not be everywhere.

Another tricky case of quoting arises from the use of backticks. You may want to store a file name into a shell variable. The file name is produced by running a command and the file name may contain whitespace. How should it be quoted to be safe? A common approach is:

```
docfile="`find $dir -name \*.doc`"
```

Hmm... but what if the variable `$dir` could contain whitespace? Better quote that too. The result would be:

```
docfile="`find "$dir" -name \*.doc`"
```

And now you have quotes within quotes. Shell does not get confused by the nested quotes because the inner quotes are inside the backticks, so they really belong to a separate subshell invocation. But the human reader could well get confused (not to mention some text editors with syntax coloring.)

It turns out the outer quotes are actually not needed. Even though you need quotes in a situation like this:

```
docfile="/Users/simon/Documents/Meeting minutes/Meeting with
    marketing.doc"
```

You do not need the quotes when the path name is produced by backticks.

```
docfile=`find "$dir" -name \*.doc`
```

8 Useless Use of Cat

Almost all commands are able to read files specified on their command line. Even so, one sees a lot of pipes like these:

```
cat somefile | grep somepattern | ...
```

This is the basic form of *useless use of cat* syndrome. The `cat` command is not needed because `grep` can also read files, not just standard input. So the above script should really be written:

```
grep somefile | ...
```

This has several benefits:

- Omit one useless process, so the pipeline starts faster.
- Grep has direct access to its data file (or files), grep can use memory mapping or take other shortcuts that remove the need to read every byte of the file. If reading from standard input, every byte has to be read, otherwise grep cannot access following bytes.
- Reading from files is generally faster than reading from a pipe. Pipes are actually nothing more than a buffer in the kernel and the size of the buffer is usually 4 kilobytes. When reading from disk, it is possible to use larger buffer sizes to minimize the number of context switches needed to read the file.

For those commands that do *not* support reading from files, the shell supports redirection of standard input, making it unnecessary to use `cat` to feed the pipeline.

Instead of:

```
cat file | tr 'A-Z' 'a-z' | ...
```

one should use:

```
tr 'A-Z' 'a-z' < file | ...
```

There is yet another form of useless use of cat one commonly sees: cat as the feeder of a while loop.

```
cat file | while read line; do
  # do something to $line
done
```

This can be written more efficiently like this:

```
while read line; do
  # do something to $line
done < file
```

Granted, if the body of the while loop is long, moving the file that is being read to the end of the loop does make it harder to understand what exactly is being read. In such cases using a `cat` could be used to help the reader of the script. When the loop is short, the extra `cat` should be removed.

Another case where `cat` is allowed is when it is being used to concatenate files together. E.g. if you need to process several files in a while loop, you have to use `cat`.

```
cat file1 file2 file3 | while read line; do
  # do something to $line
done
```

Part II
Pitfalls

9 Using Bashisms and Gnuisms

Bash, the Bourne Again shell from the GNU project, is pretty popular these days with Linux everywhere you look. However, there are still a couple of other UNIX flavours hanging around and Bash is not usually installed there. Shell scripts should try to be as operating system independent as possible. The only way to accomplish this is to only use features present in the original Bourne shell, the `/bin/sh`. This can be tricky because these days most script writers have been brought up on Linux. They are used to using the nifty features of bash and other GNU tools. They are absolutely dumbfounded when they have to make their script run on Solaris or HP-UX for the first time. Here is a list of things I've come across that will fail on non-Linux platforms.

Another pitfall these days is the fact that some distributions no longer make `/bin/sh` a symlink to `/bin/bash`. The popular Ubuntu Linux is one of them. `/bin/sh` is actually a very small, POSIX-compatible shell.

10 Arithmetics

In bash you can do

```
a=$((5 * 6 + 2))
```

But in the one true shell the only command that knows arithmetic is expr.

```
a=`expr 5 \* 6 + 2`
```

11 Use of Tilde to Reference the Home Directory

In bash you can use the tilde (`~`) to refer to your home directory and `~user` to refer to a specified user's home directory.

In the one true shell your home directory is $HOME. Someone else's home directory can only be found by looking it up from `/etc/passwd`.

```
awk -F: '$1 == "user" { print $6 }' /etc/passwd
```

12 Redirection shortcuts

In bash (and csh) you can redirect both standard output and standard error streams to a file with a convenient shorthand.

```
command >&file
```

But in the one true shell you have to spell it out in full, do it for both stdout and stderr.

```
command >file 2>&1
```

13 Order of Redirections

What is the difference of the following command lines?

```
command >file 2>&1
command 2>&1 >file
```

The first is the one you probably wanted. It reads like this: "Redirect standard output to a file. Then redirect standard error to where standard output is attached." Both streams will be written to the same file.

The second command line reads: "Redirect standard error to where standard output is attached (probably a terminal). Then redirect standard output to a file." The errors will appear on the terminal and output will be written to the file.

There are two things to remember. One, the redirections are processed from left to right. Two, the notation `2>&1` does not make stderr to go stdout, instead it attaches stderr to where stdout was attached at this moment. In the second case above, the standard output was still connected to the terminal when the first redirection was being processed.

OK, then how do you send both stdout and stderr to another command?

```
make 2>&1 | less
```

This is a special case. Above it was emphasized you first change where stdout goes and only then pipe everything else to stdout. Not here. The reason is you really have 2 commands connected by a pipe. Piping is not the same as redirecting.

14 Backticks and Command Line Length

Many programmers know that the maximum length of a command is very limited in MS-DOS. UNIX programmers seem to think they have an infinite command length or at least the length is so long they don't have to worry about it. The prime example is what happens when you put the find command inside backticks.

```
wc -l `find . -type f`
```

This works fine as long as the number of files printed by the find is less than the maximum command line length of the operating system. It might work for you when you test it. But does it work for someone using your script? Why not be sure and use xargs?

```
find . -type f | xargs wc -l
```

By the way, both of the above will fail miserably if there are filenames with spaces. There is no way to make the first command work properly. The second command can be made to work with minor modifications:

```
find . -type f | xargs -I xxx wc -l xxx
```

xargs will read lines from its standard input and run wc -l with xxx replaced with one line, over and over again until all lines from stdin are exhausted. xargs is free to run wc -l with more than one argument, if the replacement of xxx with multiple lines does not exceed the maximum command line length.

15 How to Refer to All Command Line Arguments?

Often you write a small script to act as a front-end to some other utility. Typically these scripts check invoke some process with all the remaining command line arguments. Some script-writers use the `$*` and some `$@`. Is there a difference? Does it matter? Yes it does.

First of all, both `$@` and `$*` need to be enclosed in double quotes to work properly (`"$@"` and `"$*"`). Unquoted, they behave the same. Let's examine the issue with some examples. First, let's write a script called printargs.sh that only prints it arguments.

```
#!/bin/sh
for a; do
        echo "$a"
done
```

Then write a script called argtest.sh that tries out the different possibilities.

```
#!/bin/sh
echo 'Using $@'
./printargs.sh $@
echo 'Using $*'
./printargs.sh $*
echo 'Using "$@"'
./printargs.sh "$@"
echo 'Using "$*"'
./printargs.sh "$*"
```

Then we execute it and see the following output.

```
./argtest.sh "1 2" 3
Using $@
1
2
3
Using $*
1
```

```
2
3
Using "$@"
1 2
3
Using "$*"
1 2 3
```

As you see, usually you want the behaviour of the "$@". Anything else will bite you in the backside when someone gives your script an argument that contains spaces.

16 Lock files

Sometimes you can only have one instance of your script running. Perhaps you are updating some important system files and you must be sure it all works properly, even if two different root users are logged on and invoke your script with no idea another admin is already doing the same thing. Your script must take some kind of lock before starting to do its work and then release the lock when it is done.

A lock could be the existence of a file. But how do you both create a file and fail to create it if it already existed? You have two possibilities: a hard link and a directory. Creation for both of them is an atomic operation that will fail if the thing already exists.

You could do something like this:

```
echo $$ > lock.$$
if ln lock.$$ lockfile; then
    echo "Lock acquired"
else
    echo "Failed to get the lock"
fi
```

But what if an instance of that script died while holding the lock? You must set up a trap statement that will remove the lockfile when something unexpected happens.

```
trap "rm -f lock.$$ lockfile" EXIT SIGHUP SIGTERM SIGINT
echo $$ > lock.$$
if ln lock.$$ lockfile; then
    echo "Lock acquired"
else
    echo "Failed to get the lock"
fi
```

Something bad might still happen. Your script is killed with "kill -9" or the power fails. Whatever. The fact of the matter is that when your script runs, lockfile exists and it never gets the lock. That's why the lockfile is a link to a file that has the PID of the process that created it. We can simply check if that process exists.

```
echo $$ > lock.$$
trap "rm -f lock.$$ lockfile" EXIT SIGHUP SIGTERM SIGINT
if ln lock.$$ lockfile; then
    echo "Lock acquired"
else
        lockholder=`cat lockfile`
    if kill -0 $lockholder; then
        echo "Failed to get the lock, lock is held by $lockholder"
    else
        rm -f lockfile
        echo "Lock was held by $lockholder (died), try again now"
    fi
fi
```

Something like that. Or you could have a loop trying the hard link over and over as long as it takes for it to succeed.

But what if the process-id read from the lockfile is an existing process, it's just not an instance of your script? Now things get tricky. In general, there might not be a portable solution, but if you can forget portability, there are things you can do.

If you can check when a process was launched, you can compare it against the modification time of lockfile. If they are reasonably close, then the creator of the lockfile is probably still alive.

Handling lock files correctly is a difficult problem that few programs get right. A better way would be to make lockfile be a unix domain socket instead of a file. Then checking if the lock holder was still alive would be a simple matter of connecting to the socket. If the connection was refused, there was no-one home. If the connection was established, you could talk to the other end and ask if the lock holder was still ok. Possibly the lock holder could even reply with an estimate of how long it is going to hold the lock. But, if the connection was refused, you would have to remove the stale socket and try bind a new one.

Unix domain sockes, unfortunately, are not easily manipulated from a shell script. You could make a Perl script that created the socket and responded to anyone connecting to it. But then you would need to run that Perl script in the background, executing in parallel with your script. Which you can do, just see below.

Another possibility would be to use advisory file locks (flock/fcntl/lockf) but again, they are not accessible to shell scripts, but they are to Perl. The advantage of file locks is they are unlocked by the operating system if the process holding the lock should die.

You could use a helper like this:

```
#!/usr/bin/perl -w
# Lockfile helper by Sami Tikka 2007
#
# Usage from a shell script:
# lockholder=`lockfile PATH_TO_LOCKFILE` || exit 1
# echo "Lock acquired"
# echo "Doing my thing"
# echo "Releasing lock"
# kill $lockholder
#
use strict;
use Fcntl ':flock';

die "USAGE: lockfile PATH_TO_LOCKFILE" if $#ARGV != 0;

my $lockfile = shift @ARGV;
my $LOCK;
my $signaled = 0;
$SIG{'INT'}  = sub { $signaled++ };
$SIG{'TERM'} = sub { $signaled++ };

open($LOCK, '>', $lockfile) or die "Cannot open $lockfile: $!";
flock($LOCK, LOCK_EX|LOCK_NB) or exit 1;
my $lockholder = fork();
# Parent prints the PID of child who will hang around and hold the
    lock
if ($lockholder > 0) {
    print "$lockholder\n";
    exit 0;
}
sleep;
flock($LOCK, LOCK_UN);
close($LOCK);
unlink($lockfile);
exit 0;
```

The shell script would acquire the lock by running the Perl script, capturing Perl's standard output and verifying its exit code. If the exit code was nonzero, lock was not acquired. If the exit code was zero, standard output will contain the id of the process that needs to be killed to release the lock.

17 Doing more than one thing at the same time

Normally your script only has one thread of execution, it runs the commands one at a time, or one pipeline at a time. Sometimes you need to do more. For example, you might want to run a script which takes a long time and you want to make sure it does not take more than 3 hours.

18 Assumption is the mother of all...

18.1 Locale and language

These days users of modern Unix systems expect the operating system and applications to be localized. This is a benefit to the end-users who might not be fluent in English, but it is bad for the script writer: Applications might produce output which is completely different from what you expect. Also, you may get bug reports from your users with error messages in japanese or some other language you don't know that well.

The careful scripter will put these lines somewhere at the beginning of their script:

```
LANG=C
export LANG
```

This will make everything executed after it use the C locale, where programs produce standardized output and error messages, in English.

If your script needs to produce localized output, you can capture the user's locale settings before replacing them with your own:

```
user_locale=`locale`
```

If needed, you can restore them with:

```
eval $user_locale
```

18.2 PATH lookup

If your script needs `/usr/bin/X11` to be in PATH, do not assume it is there. If you need something to be in PATH, add it there.

18.3 UMASK and default permissions

When your script creates files, even temporary files, you might want to be explicit with the permissions they are created with. If your script deals with user's data and the point of the script is not to share the data, set umask to 0066. It is better to create files with tight permissions and then relax for those few files that need relaxing. If you create files that are world-readable and then remove the world-readability after creation, there will be a window of time during which the file will be readable by everyone and sometimes that is all the attacker needs.

18.4 Echo does not support -n or -e

Sometimes you want to be fancy in your printouts, you want to prevent echo from printing the newline at the end of the line or something else: you want to call echo with the -n or -e option. Couple of points:

First, when you call echo, it might be a shell built-in command or it might be `/bin/echo`. They might behave differently.

Second, the echo you call might not support -n or -e or either.

It is best to use echo without options and when you need better control for printout, use printf.

References

[1] S. R. Bourne: “An Introduction to the Unix Shell”, http://www.iki.fi/era/unix/shell.html

[2] Era Eriksson: “Useless Use of Cat Award”, http://www.iki.fi/era/unix/award.html