Top 3 bugs I make in shell scripts

W. Adam Koszek

Koszek ORG

wojciech@koszek.com

2022-10-05T17:47:57Z

Most people involved in software rarely talk about bugs. Perhaps it's because a lot of software designers are striving for perfection, and aren't willing to talk about the times when the machine conquered them. We all create software bugs, so let's break the silence. Below are my top three bugs which occur while writing shell scripts.

Order of redirection

Let's take an example script which generates "STDOUT" on standard output and "STDERR" on standard error.

```
bash-3.2$ cat script.sh
echo STDERR > /dev/stderr
echo STDOUT > /dev/stdout
```

I often do:

```
bash-3.2$ ./script.sh 2>&1 > /dev/null
STDERR
```

But you must do:

```
bash-3.2$ ./script.sh > /dev/null 2>&1
```

Different usage on different systems

I frequently use `script`, since it's the easiest way to get a trace of whatever is happening in your shell. You start 'script', run commands, hit "exit" and in the current directory the file "typescript" will contain all the output from all the commands that you've run saved. Trick: MacOSX and BSD-based systems:

```
script your_logfile.log command...
```

are different than Linux:

```
script -c 'your command' file
```

That's the common problem, e.g.: I don't like MacOSX not having `readlink -f` support. But oh well..

Value propagation from `while` loops

Very often I forget that everything in shell programming is a command.

Quiz: how do '[' and ']' work?

Looping is so essential in other languages, that sometimes, while scripting in shell, I forget that there are some things I can't do. The other day I tried something like this:

```
#!/bin/sh
X=0
git status -porcelain | while read FILE; do
 # ...
 X=1
done
```

Before you continue reading, another quiz: why won't it work?

Compare the above with this:

```
#!/bin/sh
cnt=0
while [ $cnt -lt 3 ]
do
 echo "In the loop: `./getppid`"
 echo $cnt
 cnt=`expr $cnt + 1`
done
```

Basically when your shell interprets this, it's all contained in one single process. But once you use '|', the pipe between two processes needs to be created, so whatever you have after the pipe sign will be running in a separate process, which the shell has forked for you. The solution to this is to restructure the loop so that everything happens in one single process:

```
#!/bin/sh
X=0
git status -porcelain > _.p
while read FILE; do
 # ...
 X=1
done < _.p
```

This is fairly inefficient, but it works on both Linux and FreeBSD, where `/bin/sh` is an actual BSD shell, not "BASH".

What are your top three mistakes while writing shell scripts?