The Shell Game

Jeremy Mates

June 25, 2021

While the unix shell is much used it may not be the best choice for programming; some consider it little more than a useful means to orchestrate commands and prototype things. Others have written extensive software with it. I'm more in the former camp.

```
$ (echo one; echo -n two) | while read l; do echo "$l"; done
one
$ (echo pa; echo re; echo -n ci) | while read l; do echo "$l"; done
pa
re
$
```

Where did the last line go? Choices here are to double down on this code (and maybe miss the occasional line) or to detect and handle this condition (longer, slower, and more complicated) or to use a different language (my usual solution). However, the shell is common and may intrude when debugging other problems so some knowledge of the shell and its quirks is essential. Let's review some syntax.

```
$ echo a b c
a b c
$ env letters="a b c" sh -c 'echo "$letters"'
a b c
```

Commands take arguments–in two forms. In addition to the usually visible command line arguments there is a list of usually invisible environment variables passed down by default to every child process. Complications include code that runs before the command is run

```
$ echo a >b c
$ cat b
a c
$ rm b
```

and various order of operation issues:

```
$ unset FOO
$ FOO=bar echo ">$FOO<"
><
```

In the first example standard output is redirected into the file `b` and the remaining arguments `a c` are passed to `echo`. In the second case `$FOO` is first interpolated to nothing because `FOO` is at that point unset, and then the `FOO=bar` assignment happens, and then `echo` runs without any arguments because `$FOO` was already interpolated away. This is akin to breaking eggs over a stove and then putting the pan on the stove and then wondering why there's a mess of eggs under the pan.

`set -x` is useful:

```
$ unset FOO
$ set -x
$ FOO=bar echo ">$FOO<"
+ echo '><'
+ FOO=bar
><
$ set +x
+ set +x
```

This shows that the echo indeed ran without argument. It will not tell someone unfamiliar with the shell why $F00 was already interpolated away; debugging must be paired with at least some understanding of how the software works. Still, the trace output should help as it may better show what is going on, especially as the problems become more complicated and less clear or when requesting help from others.

Such order of operation issues are by no means exclusive to the shell; ECMAScript 6 has a "Temporal Dead Zone" (TDZ) where bindings are not usable before their declaration, or what could be called the "eggs on the stove before pan" pattern.

```
$ node -e 'console.log(typeof x); let x = 42'
...
ReferenceError: Cannot access 'x' before initialization
```

Many other examples can doubtless be found. One solution in the shell is to separate the assignment from the usage (the "avoid doing too many things on one line" pattern). This can take a variety of forms:

```
$ unset F00
$ F00=bar
$ printf "%s\n\n" $F00
bar

$ unset F00
$ F00=bar ; printf "%s\n\n" $F00
bar

$ unset F00
$ F00=bar sh -c 'echo $F00'
bar
```

These adjustments assume familiarity with the shell; `F00=bar echo $F00` produces no error as it is not invalid, much as one can construct a grammatically correct phrase that is flawed in some other way. A perhaps useful set of terms from spoken languages: the vocabulary, the grammar, and the usage. For the shell the vocabulary would in part consist of

```
    cd          while         ls   echo   &      somevar    SECRET_API_KEY
    printf      <             --   $      read   >          =
    /dev/null   /etc/passwd   do   l      -rf    done       rm
```

and the grammar how the vocabulary can be strung together

```
$ >/dev/null ls
$ >&2 echo $somevar
```

while the usage indicates which grammatical forms are most suitable. This is the domain of style guides and best practices–or knowing when not to use the shell. Redirections are more commonly used after rather than before the command even though they may be legal in other locations:

```
$ >/dev/null ls # strange
$ ls >/dev/null # more typical
$ while read l; do echo $l; done </etc/passwd
...
$ printf -- "$somevar\n" >&2
```

An `</etc/passwd` redirection before a `while` loop is legal in `zsh`–a usage choice–and illegal in other shells–a grammar error.

```
$ </etc/passwd while read l; do echo $l; done
/opt/local/bin/mksh: syntax error: unexpected 'do'
$ exec zsh
% </etc/passwd while read l; do echo $l; done
...
```

The conventional form should be used in conventional scripts. While debugging "anything goes" though one may want to clean things up before presenting a problem elsewhere. Or at least to caveat the code as

such. Non-conventional uses may help reveal bugs or portability problems.

Let us study the "`while` loop drops data" issue some more. Where did the last line go?

```
$ (echo one;echo -n two)|while read l;do echo ">$l<";done
>one<
$ printf ",%s,\n" "$l"
,,
```

It did not escape into `$l` following the loop, and presumably the `do ...` expression was only entered once, which the >< or „ decorators[a] help establish. Therefore the `while` or the `read` or a combination of the two is likely to blame. How can these be tested in isolation? (Another method would be to read the `read` documentation.)

```
$ cat input
one
two
$ cat code
read l; echo "1:$l:"
read l; echo "2:$l:"
read l; echo "3:$l:"
$ chmod +x code
$ ./code < input
1:one:
2:two:
3::
```

The three `read` calls ran despite there being only two lines of input. This points to `while` failing, and if we're relatively awake (which is, alas, not always possible while debugging) the logical answer is because `read` returned a false value. We can test this hypothesis.

```
$ cat code
read l; echo ":$l:$?"
read l; echo ":$l:$?"
read l; echo ":$l:$?"
$ ./code < input
:one:0
:two:0
::1
```

What if when `read` returns non-zero (false) we run something else?

```
$ (echo one;echo -n two)|while read l || echo $l ;do echo $l;done
one
two
two
```

[a]Such "decorators" are useful when there are invisible characters not otherwise seen–various whitespace and control characters in ASCII, or non-breaking spaces many other problematic characters in Unicode and other encodings. A hex dumper may also help. On decorators, use whatever you want, though beware decorators that can be mistaken for valid outputs.

```
^C$ ^C
$ ^C
$
```

Now there are two instances of `two` printed, and we had to recover the terminal, here by mashing **control+c** a lot. Also note how without a decorator it may not be clear which instance of `echo` is printing what. Another good idea is to slow things down. Lacking a "turbo button" to press, we add a `sleep` statement.

```
$ (echo 1;echo -n 2)|while read l || echo "A$l"; do echo "B$l"; sleep 1; done
B1
A2
B2
A
B
^C$
```

With the `|| echo ...` code turned into a logic test, things are looking better:

```
$ (echo one;echo -n two)|while read l || [ -n "$l" ];do echo "$l"; done
one
two
```

There are still problems with this; one may want to set `IFS` and to use the `-r` flag to `read`. Also, `less(1)` has a perhaps interesting `-p` flag.

```
$ man ksh | less -p '^ *read '
...
```

Concerning Globs

```
$ mkdir testdir && cd testdir
$ touch -- -rf
$ ls *f
.   ..   -rf
```

why were `.` and `..` displayed? Use `set -x` to trace the shell.

```
$ set -x
$ ls *f
+ ls -rf
.   ..   -rf
```

This shows that `ls -rf` is run. What do those flags do?

```
$ man ls | col -bx | egrep '^ *-[rf]'
-f  Output is not sorted. This option turns on the -a option.
-r  Reverse the order of the sort ...
$ man ls | col -bx | egrep '^ *-a'
-a  Include directory entries whose names begin with a dot (.).
```

The `*f` glob has matched the file `-rf` which the shell then passes to `ls` which in turn parses `-rf` as a bundle of command line flags. This has various security implications. Option processing should ideally be disabled prior to a glob, which on most systems and for most commands these days is done by supplying `--` after the last flag and before any external, untrusted input. See `getopt(3)` for details.

```
$ ( set -x; ls -- *f )
+ ls -- -rf
-rf
$ rm -- -rf
$ ls
$ cd ..
$ rmdir testdir
```

Globs are handled differently in different shells; in an empty directory (but note that `/var/empty`[b] is not actually empty on all platforms, or may not exist):

```
$ cd /var/empty
$ echo *
*
$ PS1='%% ' zsh -f
% echo *
zsh: no matches found: *
```

I consider bare globs that are assumed to match nothing unsafe as they are a file creation or working directory change away from suddenly matching something and then who knows what goes wrong when a filename or command flag appears instead of the literal glob. This is of less concern during interactive use–"whoa, that was strange"–and of great concern when some forgotten shell workflow is blowing up and nobody knows why.

`find(1)` also accepts globs; the result may vary depending on whether the shell or `find` performs the glob expansion.

```
% mkdir testdir && cd testdir
% mkdir -p {a,b}/{xa,xb}
% find . -name x*
zsh: no matches found: x*
% exec mksh
$ find . -name x*
./a/xa
./a/xb
./b/xa
./b/xb
$ touch xom
$ find . -name x*
./xom
$ find . -name "x*"
./a/xa
./a/xb
./b/xa
./b/xb
./xom
```

A useful use of a shell glob would be to provide a list of input directories (and, maybe, files) to `find`:

```
$ find -- * -name "xa"
a/xa
b/xa
$ echo find -- * -name "xa"
find -- a b xom -name xa
```

Another trick is the `zsh -f` used above. This disables any user-supplied configuration; if a problem persists then it is likely a `zsh` issue; otherwise, a next step would be to bisect the `zsh` configuration to find where the issue is coming from. Similar flags exist for other software, e.g. `vim -u NONE`. This pattern can be abstracted and used to ask whether an issue is, say, at the application or network level, assuming a suitable test exists that would distinguish an application issue from a network one: is the error present in data on the wire, or not? If successfully applied this should help direct attention to the component where the problem is. `zsh -f` may however be too minimal though from a minimal shell one can apply only what is necessary to test the problem.

[b] This footnote was intentionally left blank.

POSIX Word Split

POSIX word split is another shell convention that must be guarded against.

```
$ touch "a file"
$ var="a file"
$ echo $var
a file
```

Thanks to POSIX word split `a file` is actually split into two distinct arguments that `echo` hides and `rm` reveals.

```
$ rm -rf $var
rm: a: No such file or directory
rm: file: No such file or directory
```

When `$var` contains something like `/usr/ somedir` this has on many occasions resulted in the `/usr` or `/var` or `Macintosh HD` or other directory trees being completely destroyed. `zsh` does not implement POSIX word split and by default passes `a file` as a single argument to `rm`.

```
$ exec zsh
% var="a file"
% print -l $var
a file
% ( setopt SH_WORD_SPLIT; print -l $var )
a
file
% rm $var
% ls
%
```

POSIX Word Glob

But wait, there's more! In addition to the POSIX word split, a glob is also performed.

```
$ touch c.txt a.txt
$ sleep 3; touch b.txt
$ var="*.txt [abc]*"
$ set -x
$ ls $var
+ ls a.txt b.txt c.txt a.txt b.txt c.txt
a.txt       a.txt   b.txt   b.txt   c.txt   c.txt
```

Here, the words `*.txt` and `[abc]*` are first split, and then each is globbed. Note also how `ls` sorts the results; this may give the false impression that the glob or filesystem impose a specific ordering that is not actually present.

The split can be disabled by quoting the variable:

```
$ set -x
$ var="a file"
+ var='a file'
$ echo "$var" "$var" "$var"
+ echo 'a file' 'a file' 'a file'
a file a file a file
```

quoting also disables the glob:

```
$ var="*.txt"
+ var='*.txt'
$ echo "$var"
+ echo '*.txt'
*.txt
```

```
$ rm $var
+ rm a.txt b.txt c.txt
```

In general POSIX shell variables must be quoted to avoid the often problematic split (and glob!), unless one does want the split (and glob!) behavior. In such a case I would probably insert a comment that states why that variable must not be quoted to help reduce the chance of it being incorrectly quoted by a future maintainer.

The lack of word splitting in zsh can cause problems should a split be desired, though there is a flag to apply it:

```
$ zsh
% p='print -l'
% $p foo bar
zsh: command not found: print -l
% $=p foo bar
foo
bar
```

An important point here is that `print -l` could be a command name; unix filenames are sequences of bytes that disallow only two characters (`nul` and `/`). A `print -l` command could be placed into a `PATH` directory and executed:

```
% mkdir badidea && cd badidea
% path+=‘pwd‘
% print "echo not a good idea" > "print -l"
% chmod +x print\ -l
% print\ -l
not a good idea
% var="print -l"
% $var
not a good idea
%
```

But probably should not be. Unexpected characters in filenames mostly cause potentially hard to debug problems, especially where Windows (or Internet) `\r\n` linefeeds have gotten into a script, in which case the `\r` is considered part of a filename, and not a linebreak:

```
# bash --version | sed -n 1p
GNU bash, version 2.05a.0(1)-release (i686-pc-linux-gnu)
# echo ls$'\r' > cmd
# chmod +x cmd ; ./cmd
: command not found
```

The error reporting for this condition has been improved in modern shells, and in any case a hex viewer will reveal any invisible characters:

```
$ printf '#!/usr/bin/env zsh\nls\r\n' > cmd
$ chmod +x cmd ; ./cmd
./cmd:2: command not found: ls^M
$ od -c cmd
0000000    #   !   /   u   s   r   /   b   i   n   /   e   n   v       z
0000020    s   h  \n   l   s  \r  \n
0000027
```

One could create a `ls\r` command in the `PATH`, but a better idea would be to fix the script to not contain inappropriate invisible characters.

Null Interpolation

Consider a build directory cleanup script, which might contain

```
rm -rf build/*
```

This is far too static–what if we need to vary the build directory? Hence the improved form

```
rm -rf "${build}/*"
```

where care has been taken to properly quote `$build` to prevent the POSIX word split (and glob!) thing from nuking inappropriate directory trees.

The disaster requires two conditions. First, that the code is run as `root`. Second, that someone forgets to set `$build`. In this case the command that is run is

```
rm -rf /*
```

This might be a good time to mention backups, snapshots, configuration management, and various other ways to restore or rebuild a working environment. The shell has long contained features that guard against such null interpolation; the "Heirloom Bourne Shell" supports:

```
    set [--aefhknptuvx [arg ...]]
    ...
      -u  Treat unset variables as an error when substituting.

    ${parameter:?word}
      If parameter is set and not empty then substitute its value;
      otherwise, print word and exit from the shell. ...
```

or manual checks that necessary variables are set can be made. The following is fairly similar to a `${parameter:?word}` expansion.

```
#!/bin/sh
if [ -z "$build" ]; then
   echo >&2 "$0: 'build' variable is not set"
   exit 1
fi
...
```

Subshells and Processes

Earlier a subshell was used without naming it:

```
$ ( set -x; ls -- *f )
```

This isolates the `set -x` change to a child process. The isolation also applies to shell variables:

```
$ var=orig
$ ( var=new; echo "$var" )
new
$ echo "$var"
orig
```

This isolation can manifest in unexpected and perhaps undesirable ways, for instance in a `while` loop:

```
$ var=orig
$ echo new | while read l; do var=$l; echo $var; done
new
$ echo "$var"
orig
```

Child processes cannot alter the parent process; this is why the `orig` value in `var` reappears after the subshell completes. Actually, the previous statement is slightly false; a child process can change the parent because certain resources are shared between them. Identification of subshells is possible with process listing tools, or via methods specific and internal to the shell, or by various system call or kernel tracing methods. (Or when your shell code does something unexpected that reveals a subshell.)

Process listings are transitory and shell commands may complete very quickly, especially when they fail. One trick is to add `sleep` calls to slow down the speed of execution:

```
% ( sleep 31; echo new ) \
| while read l; do var=$l; echo $var; sleep 33; done
```

In another terminal inspect the process listing; `pstree(1)` is useful for this:

```
% pstree > x
% grep -2 sleep x
 | |-+= 00528 jmates zsh -l
 | | \-+= 30192 jmates zsh -l
 | |   \--- 30193 jmates sleep 31
 | |--= 00531 jmates zsh -l
 | |--= 00534 jmates zsh -l
```

As an exercise, consider why the `sleep 31` but not the `sleep 33` appears in the listing. What happens if the `echo new` command is placed before the `sleep 31`?

The commands can be better tied to the process table by logging the process ID somewhere; this may be necessary on busy systems where there are many other processes that may match what is being searched for:

```
( echo >&2 $$; sleep 31; ... ) | ...
```

`$$` however does not identify subshells:

```
$ (echo >&2 p$$; echo foo) | while read l; do echo s$$; done
p22531
s22531
```

A shell may have variables that will display the ID of the subshell.

```
$ exec mksh
$ echo $BASHPID ; ( echo $BASHPID )
30300
30303
$ exec zsh
% zmodload zsh/system
% print $sysparams[pid]; (print $sysparams[pid])
30294
30296
```

Another option may be to change the name of a process to make it easier to find; various languages also have `sleep` calls that avoid the complication of a `( sleep ... ; command ... )` subshell:

```
% perl -e '$0 = "asdf"; sleep 99' &
[1] 30390
% ruby -e '$0 = "asdf"; sleep 99' &
[2] 30391
% ps o pid,command | grep 'asd[f]'
30390 asdf
30391 asdf
```

Note that process naming is not portable; the above is from Mac OS X; on OpenBSD the name includes the name of the interpreter involved.

```
$ perl -e '$0 = "asdf"; sleep 99' &
[1] 15860
$ ruby -e '$0 = "asdf"; sleep 99' &
[2] 52604
$ ps o pid,command | grep 'asd[f]'
15860 perl: asdf (perl)
52604 ruby: asdf (ruby30)
```

The use of the `pgrep(1)` and `pkill(1)` commands instead of `ps(1)` may be beneficial; these do not need the `asd[f]` trick that prevents `grep` from sometimes matching itself in the process table.

```
% sleep 99 &
[1] 30439
% ps o pid,command | grep sleep
30439 sleep 99
30441 grep sleep
% pgrep sleep
30439
```

Process listing tools are unportable; review the documentation specific to the commands for the system in question. A coworker once wrote a `killall` command for Digital UNIX, but ended up running the vendor-provided `killall` on Solaris when they moved their script there. Ooops.

Another concern is that process listings are only a snapshot; processes can be created and destroyed while `ps` is running and thus may not be seen by `ps`. `CVE-2018-1121` details a trick where a process would receive a notification when the process table was being scanned and would then fork itself to avoid being listed in the process table.

Process tracing facilities can also track what goes on in a shell pipeline; `acct(2)` appeared in Version 7 AT&T UNIX. Since then more elaborate tracing facilities have been created, notably DTrace, though other tools such as `ktrace` or `strace` or `sysdig` are useful as well. On Linux `strace(1)` collects information on all the system calls made; the above shell pipeline run under `strace` reveals that three unique processes were traced.

```
$ cat cmd
#!/bin/sh
echo new | while read l; do var=$l; echo "x$var"; done
$ chmod +x cmd
$ strace -ff -o out ./cmd
xnew
$ echo out*
out.22172 out.22173 out.22174
```

System calls operate at a different level than the shell code; the `new` string appears in `read(2)` and `write(2)` system calls

```
$ grep new out*
out.22172:read(3, "#!/bin/sh\necho new | while read "...
out.22172:read(255, "#!/bin/sh\necho new | while read "...
out.22173:write(1, "new\n", 4) = 4
out.22174:write(1, "xnew\n", 5) = 5
$ man 2 read | col -bx | awk '/read\(/{print;exit}'
ssize_t read(int fd, void *buf, size_t count);
```

made by the main shell process, PID 22172, that reads the script `cmd` off of the filesystem, and two subshell processes, one that writes the string (PID 22173, corresponding to the `echo new` shell code) and the other that carries out the code for the `while` loop (PID 22174, shell code `echo "x$var"`). Note that PID 22174 must at some point read in the string `new`, written to it by PID 22173. . . why does that not appear? An inspection of the trace file reveals single character reads of the string via standard input (file descriptor `0`):

```
$ grep -A 3 '"n' out.22174
read(0, "n", 1)                         = 1
read(0, "e", 1)                         = 1
read(0, "w", 1)                         = 1
read(0, "\n", 1) = 1
```

String and file path tracing with `strace` may require suitable `-s` and `-y` flags especially when large amounts of data–where "large" is anything longer than 32 characters–or many different files are being used. If this sounds all very new and strange, "Advanced Programming in the Unix Environment"[1] is a classic text; study the C programs therein and trace them as they run.

Exec Wrappers

Less complicated than tracing and useful for other purposes are small wrapper programs that carry out some task and then `exec` another program. These are typically but need not be shell scripts. Instead of tracing a huge application, one might have that application call the following `exec` wrapper:

```
#!/bin/sh
TMPFILE=‘mktemp -t /tmp/log.XXXXXXXXXX‘ && \
  ( id ; tty ; env ; echo "$@" ) >> "$TMPFILE"
exec /path/to/the/real/program "$@"
```

Then the huge application can be run and the resulting `/tmp/log.*` files inspected for the details logged. Targeted tracing could also be accomplished with `sysdig` on Linux, among various other options.

The wrapper program should not alter the standard input streams; reading from standard input would by default prevent what was read from reaching the real program.

A common use for a wrapper is to alter environment variables:

```
$ cat wrapper
#!/bin/sh
unset SECRET_API_KEY
export PATH=/plugin/dir:$PATH
exec "$@"
$ chmod +x ./wrapper
$ export SECRET_API_KEY=muffins-dreamspork-hodgepodge
$ ./wrapper sh -c 'echo ">$SECRET_API_KEY<"'
><
```

The downside of this method is that there is an additional fork and exec of a process; the environment could instead be customized in the parent process between the `fork` and subsequent `exec`. On the other hand, the parent process may not be possible to modify, and in many cases such wrapper programs are not run where performance is critical. Shell functions can also be used to replace wrapper scripts. Care must be taken that the shell function does not call itself.

```
# bad! calls itself
function info { info "$@" 2>/dev/null | $PAGER; }

# better (emacs users might disagree)
function info { command info "$@" 2>/dev/null | $PAGER; }
```

Wrappers may be necessary for security where a process is unable to properly clean up the environment. For example, one may wish to ensure that a program is not passed duplicate environment variables. For this I use the `nodupenv` wrapper program. Also required is `dupenv` to create duplicate environment variables for testing purposes.

```
$ dupenv FOO=x FOO=y env | grep FOO
FOO=x
FOO=y
$ dupenv FOO=x FOO=y FOO=z nodupenv env | grep FOO
FOO=x
```

Wait, environment variables can be duplicated? How does that work? Now might be a good time to talk about programs and their arguments in some detail, but so ends the shell game.

References

1. W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the Unix Environment*. Addison Wesley, 2013.