Use strace To Understand Your Shell (bash)

Harald König

Harald.Koenig2@Bosch-Sensortec.com

koenig@linux.de

The UNIX shell is a very powerful and useful tool. Unfortunately, not only Linux beginners despair at the shell again and again, if you do not know a few simple rules and shell concepts. This talk explains some of the fundamental concepts in UNIX and it's shell, and illustrats and displays them "live" with the help of `strace`. This will also show, how useful `strace` can be in understanding and debugging shell scripts and all other UNIX processes.

1 Introduction

The UNIX/Linux shell (here always using the example of `bash`) is a very simple, but powerful tool. However, many users often have problems that the shell does not quite do they want to: quoting, wildcards, pipes, etc. are again and again cause of surprise and "maloperation".

So, which `echo` is the right one (and when)?

```
$ echo Hallo                                              $ echo  Hallo Dublin
$ echo 'Hallo'                                            $ echo  Hallo     Dublin
$ echo "Hallo"                                            $ echo 'Hallo     Dublin'
```

After many observations of such difficult decisions in the free "command line environment" the final trigger for this lecture was the following command line

```
# insmod *.ko
```

and the question "Does `/sbin/insmod` support using wildcards?".

2 UNIX Basics: `fork()` and `exec()`

One of the universal principles in UNIX is the mechanism for starting all new programs: with the system call `fork()`, the current process gets "duplicated" and the newly created process then starts the new program with the system call `exec()`. This is precisely the main task of any UNIX shell, among many simpler internal shell commands: start and run external programs (tools). For the `exec()` call, the command line arguments for the new program are *individually* passed:

```
main() { execl("/bin/echo", "program", "Hello", "Dublin 1"   , 0); }
main() { execl("/bin/echo", "program", "Hello", "Dublin", "1", 0); }
```

or a bit nicer to read

```
#include <unistd.h>
int main(int arc, char *argv[])
{
  char *const argv[] = { "program", "Hello", "Dublin 2", NULL };
  return execve("/bin/echo", argv , NULL) ;
}
```

It is the task of the shell how the parameters for the new program are passed to this `exec()` system call. How the new program then interpretes and processes these individual arguments (options, file names, etc.) is solely up to the called program.

Since Linux kernel version 2.3.3 in 1999, the existing system call for POSIX `fork()` in the Linux kernel is called `clone()`. This system call `clone()` is called by a function `fork()` which is defined in the `glibc`. All variants of the system call `exec()` call in the `glibc` actually call the kernel call `execve()`.

It is therefore crucial to understand how the shell processes the input command line and then starts the program in an `execve(...)` call. This transfer to the kernel call now can nicely be monitored and displayed with `strace`, which exactly shows what the shell actually did with the input command line and what the call parameters for the program eventually look like.

`strace` offers many command line options options and variants. Here I only show some of the most important ones – for more information and help please have a look into the man page (RTFM: `man strace`).

2.1 First Steps with `strace`

Depending on what you want to analyze, you can (just like with debuggers such as `gdb`) either newly start a command either with `strace`, or you can attach `strace` to an already running process and analyze it:

```
$ strace emacs
```

or for a single `emacs` process which is already running

```
$ ps x | grep emacs
$ strace -p $( pgrep emacs )
```

and if there are multiple processes to be traced at once (e.g. all instances of an running apache `httpd`):

```
$ strace $( pgrep httpd | sed 's/^/-p/' )
```

You can terminate a `strace -p ...` which is attached to a process at any time by typing `CTRL-C`, then `strace` will detach itself and the examined program continues running without any tracing or slowdown.

If you started the program to be tested by `strace ...`, then with `CTRL-C` not only `strace` command gets canceled, but also the launched program is killed.

`strace` prints its entire output to `stderr`. However, you can write the output of `strace` to a file or redirect the output, but then including the `stderr` output of the process (here: emacs):

```
$ strace -o OUTFILE emacs
$ strace emacs 2> OUTFILE
```

Usually `strace` outputs one line per system call. This line contains the name of the kernel routine and their parameters, as well as the return value of the system call. The output of `strace` is very similar to the syntax of `C`, which is not very surprising because the kernel API of Linux/UNIX is defined as ANSI-C interface (mostly didactically shorted output):

```
$ strace cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 130 vars */]) = 0
open("/etc/HOSTNAME", O_RDONLY)        = 3
read(3, "harald.linux.de\n", 32768) = 16
write(1, "harald.linux.de\n", 16) = 16
read(3, "", 32768)                     = 0
close(3)                               = 0
close(1)                               = 0
close(2)                               = 0
exit_group(0)                          = ?
```

Even without programming skills in `C` it's possible to understand and reasonably interpret this output. Details about the individual system calls can be found in the appropriate man pages, because all system calls are documented there (`man execve; man 2 open ; man 2 read write`, etc.). Only from the definition of the kernel calls you can know whether the values of the arguments are passed from the process *to* the kernel, or if they are returned back *from* the kernel to the user process (e.g. the string value as the second argument of `read()` and `write()` in the last example).

`strace` creates more verbose output with option `-v` for some system calls (e.g. `stat()` and `execve()`). This can be helpful when looking for more information, like `stat()` details of files or the complete environment in `execve()`.

```
$ strace -e execve cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 130 vars */]) = 0
harald.linux.de
```

```
$ strace -v -e execve cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], ["LESSKEY=/etc/lesskey.bin", "MAN
PATH=/usr/local/man:/usr/loca"..., "XDG_SESSION_ID=195", "TIME=\\t%E real,\\t%
U user,\\t%S sy"..., "HOSTNAME=harald", "GNOME2_PATH=/usr/local:/opt/gnom"...,
 "XKEYSYMDB=/usr/X11R6/lib/X11/XKe"..., "NX_CLIENT=/usr/local/nx/3.5.0/bi"...,
 "TERM=xterm", "HOST=harald", "SHELL=/bin/bash", "PROFILEREAD=true", "HISTSIZE
=5000", "SSH_CLIENT=10.10.8.66 47849 22", "VSCMBOOT=/usr/local/scheme/.sche"..
[ ... many lines deleted ...]
rap"..., "_=/usr/bin/strace", "OLDPWD=/tmp"]) = 0
```

2.2 Reduce the Output of `strace`

The output of `strace` can quickly get *very* large, and the synchronously written output to `stderr` or even to a log file can considerably slow down performance. If you know exactly which system calls are of interest, you can limit the output to one or few kernel calls (or e.g. with `-e file` to all calls with file names). This speeds up the program execution and later considerably simplifies the analysis of the `strace` output. But the option `-e` has many more possibilities. Here are just a few simple examples, which very often are sufficient, everything else is documented in the man page:

```
$ strace -e open            cat /etc/HOSTNAME
open("/etc/HOSTNAME", O_RDONLY)        = 3
```

```
$ strace -e open,read,write  cat /etc/HOSTNAME
open("/etc/HOSTNAME", O_RDONLY)        = 3
read(3, "harald.linux.de\n", 32768) = 16
write(1, "harald.linux.de\n", 16harald.linux.de
) = 16
read(3, "", 32768)                     = 0
```

```
$ strace -e open,read,write  cat /etc/HOSTNAME > /dev/null
open("/etc/HOSTNAME", O_RDONLY)        = 3
read(3, "harald.linux.de\n", 32768) = 16
write(1, "harald.linux.de\n", 16) = 16
read(3, "", 32768)                  = 0
```

```
$ strace -e file            cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 130 vars */]) = 0
open("/etc/HOSTNAME", O_RDONLY)        = 3
```

2.3 Tracing Multiple Processes and Child Processes

`strace` can also simultaneously trace several processes (option `-p PID`) or follow all child processes (with option `-f`). In these cases, each line of `strace` output will start with the PID of corresponding process. Alternatively, you can write the output produced for each process into separate files (with option `-ff`).

If you don't yet exactly know what you really need to trace, then you should use `-f` or `-ff`, since there might be some (sub-) processes or scripts being involved. Without `-f` or `-ff` you could otherwise miss important information in the trace log from other processes! In my simple examples with `emacs` this is itself already a wrapper script. Here are a few variations as little brainteaser to guess how the `bash` is internally ticking:

```
$ strace -e execve bash -c true
execve("/bin/bash", ["bash", "-c", "true"], [/*...*/]) = 0
```

```
$ strace -e execve bash -c /bin/true
execve("/bin/bash", ["bash", "-c", "/bin/true"], [/*...*/]) = 0
execve("/bin/true", ["/bin/true"], [/*...*/]) = 0
```

```
$ strace -e execve bash -c "/bin/true ; /bin/false"
execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/*...*/]) = 0
```

```
$ strace -f -e execve bash -c "/bin/true ; /bin/false"
execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/*...*/]) = 0
execve("/bin/true", ["/bin/true"], [/*...*/]) = 0
execve("/bin/false", ["/bin/false"], [/*...*/) = 0
```

```
$ strace -o OUT -f -e execve bash -c "/bin/true ; /bin/false"
$ grep execve OUT
1694 execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], []) = 0
1695 execve("/bin/true", ["/bin/true"], [/*...*/]) = 0
1696 execve("/bin/false", ["/bin/false"], [/*...*/]) = 0
```

```
$ strace -o OUT -ff -e execve bash -c "/bin/true ; /bin/false"
$ grep execve OUT*
OUT.2155:execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"],[]) = 0
OUT.2156:execve("/bin/true", ["/bin/true"], [/*...*/]) = 0
OUT.2157:execve("/bin/false", ["/bin/false"], [/*...*/]) = 0
```

2.4 And now let's trace `bash`

In order to observe an interactive shell, we start two terminals. In the first window, we output the PID of the shell, which we're going to monitor now

```
$ echo $$
12345
```

and in the second window we start `strace` to attach to the first shell and monitor all called programs (with or without option -v, depending on whether the process environment and long numerous arguments are important or not):

```
$ strace -f -e execve -p 12345
$ strace -f -e execve -p 12345 -v
```

Below in the examples, first the shell input and command output is shown, then followed by the associated output of strace.

And off we go . . .

3 Globbing (Wildcards)

First, let's look at the content of a directory with the command ls. Some file names have an appended asterisk (*). The reason for this is the magically appearing option "-F" for ls, which origins from an alias for ls (as type -a can show):

```
$ ls
echo1*  echo1.c  hello*  hello.c  hello.cpp  hello.h  test.c  test.ko
execve("/bin/ls", ["ls", "-F"], [/*...*/]) = 0

$ type -a ls
ls is aliased to `ls -F'
ls is /bin/ls
```

In order to avoid this alias, you can either directly call /bin/ls, or type \ls to escape that alias, both has the same effect. If you only want to see the C source files, you can run ls with the argument *.c

```
$ /bin/ls
echo1  echo1.c  hello  hello.c  hello.cpp  hello.h  test.c  test.ko
execve("/bin/ls", ["ls"], [/*...*/]) = 0

$ \ls *.c
echo1.c  hello.c  test.c
execve("/bin/ls", ["ls", "echo1.c", "hello.c", "test.c"], [/*...*/]) = 0
```

Now with strace we can see that ls does not get *.c as it's single parameter, but a list of all file names of the C sources. The expansion of wildcards (globbing) is already handled by the shell, and therefore ls and all other programs no longer have to care about expanding wildcards.

If you prevent the expansion of the file name pattern in the shell with quotes, then ls really gets the argument *.c and doesn't know how to deal with it:

```
$ \ls "*.c"
ls: cannot access *.c: No such file or directory
execve("/bin/ls", ["ls", "*.c"], [/*...*/]) = 0
```

This answers the question about /sbin/insmod in the very beginning:

```
# insmod *.ko
execve("/sbin/insmod", ["/sbin/insmod", "test.ko"], [/*...*/]) = 0

# insmod "*.ko"
insmod: can't read '*.ko': No such file or directory
execve("/sbin/insmod", ["/sbin/insmod", "*.ko"], [/*...*/]) = 0
```

/sbin/insmod can handle wildcards as little as /bin/ls can, the command insmod expects a correct filename of a kernel module as an argument on the command line. For insmod you should note that this command expects only a single kernel module on the command line. Therefore, you should run insmod *.ko *only* when you are sure that exactly one kernel module *.ko is in the current directory!

The classic counter-example of this otherwise normal UNIX behavior is the command `find` with its option `-name`. Here you can specify a "pattern" after `-name` to be searched which also can include wildcards. In this case it leads to an error or surprising behavior when the wildcards are not quoted and therefore get expanded by the shell. In the first example `find` will exclusively look for `test.ko` and not all kernel modules (surprise), and in the second instance, you'll get a cryptic error message from `find`, since you didn't follow the (somewhat strange) syntax of this command:

```
$ find .. -name *.ko
../dir/test.ko
execve("/usr/bin/find", ["find", "..", "-name", "test.ko"], [/*...*/]) = 0

$ find .. -name *.c
find: paths must precede expression: hello.c
Usage: find [-H] [-L] [-P] [-Olevel] [-D help|tree|search|stat|rates|opt|exec]
            [path...] [expression]
execve("/usr/bin/find",["find", "..", "-name", "echo1.c", "hello.c", "test.c"],[]) = 0
```

In this case you have to use quotes or backslash to avoid the wildcard expansion of the shell, so that `find` will get the required wildcard pattern for `-name`:

```
$ find .. -name \*.ko
execve("/usr/bin/find", ["find", "..", "-name", "*.ko"], [/*...*/]) = 0
../dir/test.ko
../dir2/test2.ko

$ find .. -name "*.c"
../dir/echo1.c
../dir/hello.c
../dir/test.c
../dir2/test2.c
execve("/usr/bin/find", ["find", "..", "-name", "h*.c"], [/*...*/]) = 0
```

4 Quoting

The examples with wildcards show that in the shell sometimes you have to "quote". The question is when and how to do it right. In this context, you must know the characters with special functions in the shell, which you want to deactivate in some cases.

These special characters include, as already seen, the wildcards for pattern (`*` and `?`), the dollar sign (`$`) for variable names etc., greater than, less than and pipe character (`< > |`) for input and output redirection and pipes, parentheses for subshells, the ampersand (`&`) for background processes, the exclamation mark (`!`) for the shell history, the so-called white space characters (space, tab, newline) the backquote (`` ` ``) for command substitution, and of course the quoting characters themself (`\ ' "`), and probably a few more forgotten characters.

The shell offers three methods to "quote" those special characters, such as the wildcards: the backslash (`\`), the single (`'`) and double (`"`) quotes. The function of the three quoting styles is easily explained:

The backslash (`\`) acts on the following character and disables its special function.

In strings within single quotation marks (`'`), there are no special features anymore, with the sole exception of the (`'`) itself, thereby ending the quoted string. Also, the backslash has no special function. Therefore, you cannot quote the single quote within such strings. Explicit single quotes can only used outside of quoted strings or inside double quotes.

In strings within double quotes (") there are still a few active special characters: the dollar sign ($) for variable names, the back quote (`) for command subsitution, the exclamation mark (!) in the interactive input, and the backslash (\) to quote all those characters and also itself. Spaces, tabular character and line endings are preserved in both types of quotes.

echo is an *internal* command in the bash, and is thus not monitored by strace, not even with a backslash in front of it, as in the casse of the alias for ls:

```
$ echo Hello    Dublin
Hello Dublin

$ \echo Hello             Dublin
Hello Dublin
```

In order to obtain spaces you must quote them, whether with single or double quotes or backslashes doesn't matter:

```
$ /bin/echo Hello    Dublin
Hello Dublin
execve("/bin/echo", ["/bin/echo", "Hello", "Dublin"], [/*...*/]) = 0

$ /bin/echo "Hello    Dublin"
Hello    Dublin
execve("/bin/echo", ["/bin/echo", "Hello    Dublin"], [/*...*/]) = 0

$ /bin/echo 'Hello    Dublin'
Hello    Dublin
execve("/bin/echo", ["/bin/echo", "Hello    Dublin"], [/*...*/]) = 0

$ /bin/echo Hello\ \ \ \ Dublin
Hello    Dublin
execve("/bin/echo", ["/bin/echo", "Hello    Dublin"], [/*...*/]) = 0
```

But even correctly quoted spaces can quickly disappear, once you forget the quotes:

```
$ a="A  B     C"
$ /bin/echo $a
A B C
execve("/bin/echo", ["/bin/echo", "A", "B", "C"], [/*...*/]) = 0

$ /bin/echo "$a"
A  B     C
execve("/bin/echo", ["/bin/echo", "A  B     C"], [/*...*/]) = 0

$ /bin/echo '$a'
$a
execve("/bin/echo", ["/bin/echo", "$a"], [/*...*/]) = 0
```

and wildcards always must be tamed and kept in "quote bridle", as long as you do not want their expansion:

```
$ b="with find one can search for *.c ..."
$ /bin/echo "$b"
with find one can search for *.c ...
execve("/bin/echo", ["/bin/echo", "with find one can search for *.c ..."], []) = 0

$ /bin/echo $b
with find one can search for echo1.c hello.c test.c ...
execve("/bin/echo", ["/bin/echo", "with", "find", "one", "can", "search", "for",
                    "echo1.c", "hello.c", "test.c", "..."], []) = 0
```

If you need both the effect of simple and double quotes on a single command line, then you have to open and close them alternately accordingly, or you can use backslashes where required:

```
$ echo 'The variable "a" is read with $a and has the value '"'$a'"
The variable "a" is read with $a and has the value 'A  B     C'
execve("/bin/echo", ["/bin/echo", "The variable \"a\" is read with $a
                      and has the value 'A  B     C'"], [/*...*/]) = 0
```

```
$ echo "The variable \"a\" is read with \$a and has the value '$a'"
The variable "a" is read with $a and has the value 'A  B     C'
execve("/bin/echo", ["/bin/echo", "The variable \"a\" is read with $a
                      and has the value 'A  B     C'"], [/*...*/]) = 0
```

```
$ /bin/echo The variable \"a\" is read with \$a and has the value \'"$a"\'
The variable "a" is read with $a and has the value 'A  B     C'
execve("/bin/echo", ["/bin/echo", "The", "variable", "\"a\"", "is", "read",  "with",
       "$a", "and", "has", "the", "value", "'A  B     C'"], []) = 0
```

and again briefly the already discussed wildcards

```
$ /bin/echo *.c
echo1.c hello.c test.c
execve("/bin/echo", ["/bin/echo", "echo1.c", "hello.c", "test.c"], [/*...*/]) = 0
```

```
$ /bin/echo \*.c
*.c
execve("/bin/echo", ["/bin/echo", "*.c"], [/*...*/]) = 0
```

```
$ /bin/echo '*.c'
*.c
execve("/bin/echo", ["/bin/echo", "*.c"], [/*...*/]) = 0
```

5 Which login scripts have been executed

... can quite easily and generally be tested and verified with strace. If you trust the appropriate shell (docs), that the shell, for example, with the option -login really behaves identical to a “real” login. With this assumption, it is enough to create and analyze list of files. Below the FILELISTs remain unsorted, since the order of the login scripts obviously is important. Otherwise, these lists of filenames are usually easier to read after being sorted with sort -u.

```
$ strace -o OUT -e open -f bash --login -i
exit
$ grep -v ENOENT OUT | grep \" | cut -d\" -f2 > FILELIST1
$ strace -o OUT -e file -f bash --login -i
exit
$ grep \" OUT | cut -d\" -f2 > FILELIST2
```

```
$ egrep "^/etc|$HOME" FILELIST1
/etc/profile
/etc/bash.bashrc
/etc/bash.local
/home/harald/.bashrc
/etc/bash.bashrc
/etc/bash.local
/home/harald/.bash_profile
/home/harald/.profile.local
/home/harald/.bash_history
/etc/inputrc
/home/harald/.bash_history
```

If you don’t fully trust the shell, then you should trace the “source’ of a a login yourself! As a non-root user, you can e.g. try this with xterm -ls. Or as root you directly trace the SSH daemon on port 22 or a getty process with all subsequent child processes:

```
$ lsof -i :22
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
sshd    2732 root    3u  IPv4  15551      0t0  TCP *:ssh (LISTEN)
$ strace -p 2732 -ffv -e file -o sshd.strace ....
```

6 Conclusion

There is a lot more to be learned about strace with shell script execution and the entire UNIX/Linux system. Here the fantasy can offer plenty of space for other uses and can be material for further talks…

Now have fun with the shell, and success and new UNIX findings when using strace!

References

[1] RTFM: man strace bash

[2] http://linux.die.net/man/1/bash man page of bash

[3] http://linux.die.net/man/1/strace man page of strace