

Bash and the process tree

The process tree

The processes in UNIX® are - unlike other systems - **organized as a tree**. Every process has a parent process that started, or is responsible, for it. Every process has its own **context memory** (Not the memory where the process stores its data, rather, the memory where data is stored that doesn't directly belong to the process, but is needed to run the process) i.e. **The environment**.

Every process has its **own** environment space.

The environment stores, among other things, data that's useful to us, the **environment variables**. These are strings in common `NAME=VALUE` form, but they are not related to shell variables. A variable named `LANG`, for example, is used by every program that looks it up in its environment to determinate the current locale.

Attention: A variable that is set, like with `MYVAR=Hello`, is **not** automatically part of the environment. You need to put it into the environment with the bash builtin command `export`:

```
export MYVAR
```

Common system variables like `PATH` or `HOME` are usually part of the environment (as set by login scripts or programs).

Executing programs

All the diagrams of the process tree use names like "`xterm`" or "`bash`", but that's just to make it easier to understand what's going on, it doesn't mean those processes are actually executed.

Let's take a short look at what happens when you "execute a program" from the Bash prompt, a program like "`ls`":

```
$ ls
```

Bash will now perform **two steps**:

- It will make a copy of itself
- The copy will replace itself with the "`ls`" program

The copy of Bash will inherit the environment from the "main Bash" process: All environment variables will also be copied to the new process. This step is called **forking**.

For a short moment, you have a process tree that might look like this...

```
xterm ----- bash ----- bash(copy)
```

...and after the "second Bash" (the copy) replaces itself with the `ls` program (the copy execs it), it might look like

```
xterm ----- bash ----- ls
```

If everything was okay, the two steps resulted in one program being run. The copy of the environment from the first step (forking) becomes the environment for the final running program (in this case, `ls`).

What is so important about it? In our example, what the program `ls` does inside its own environment, it can't affect the environment of its parent process (in this case, `bash`). The environment was copied when `ls` was executed. Nothing is "copied back" to the parent environment when `ls` terminates.

Bash playing with pipes

Pipes are a very powerful tool. You can connect the output of one process to the input of another process. We won't delve into piping at this point, we just want to see how it looks in the process tree. Again, we execute some commands, this time, we'll run `ls` and `grep` :

```
$ ls | grep myfile
```

It results in a tree like this:

```

                +-- ls
xterm ----- bash --|
                +-- grep
```

Note once again, `ls` can't influence the `grep` environment, `grep` can't influence the `ls` environment, and neither `grep` nor `ls` can influence the `bash` environment.

How is that related to shell programming?!?

Well, imagine some Bash code that reads data from a pipe. For example, the internal command `read`, which reads data from *stdin* and puts it into a variable. We run it in a loop here to count input lines:

```
counter=0

cat /etc/passwd | while read; do ((counter++)); done
echo "Lines: $counter"
```

What? It's 0? Yes! The number of lines might not be 0, but the variable `$counter` still is 0. Why? Remember the diagram from above? Rewriting it a bit, we have:

```

+-- cat /etc/passwd
xterm ----- bash --|
+-- bash (while read; do ((counter++)); done)

```

See the relationship? The forked Bash process will count the lines like a charm. It will also set the variable `counter` as directed. But if everything ends, this extra process will be terminated - **your "counter" variable is gone**. You see a 0 because in the main shell it was 0, and wasn't changed by the child process!

So, how do we count the lines? Easy: **Avoid the subshell**. The details don't matter, the important thing is the shell that sets the counter must be the "main shell". For example:

```

counter=0

while read; do ((counter++)); done </etc/passwd
echo "Lines: $counter"

```

It's nearly self-explanatory. The `while` loop runs in the **current shell**, the counter is incremented in the **current shell**, everything vital happens in the **current shell**, also the `read` command sets the variable `REPLY` (the default if nothing is given), though we don't use it here.

Actions that create a subshell

Bash creates **subshells** or **subprocesses** on various actions it performs:

Executing commands

As shown above, Bash will create subprocesses everytime it executes commands. That's nothing new.

But if your command is a subprocess that sets variables you want to use in your main script, that won't work.

For exactly this purpose, there's the `source` command (also: the *dot* `.` command). Source doesn't execute the script, it imports the other script's code into the current shell:

```

source ./myvariables.sh
# equivalent to:
. ./myvariables.sh

```

Pipes

The last big section was about pipes, so no example here.

Explicit subshell

If you group commands by enclosing them in parentheses, these commands are run inside a subshell:

```
(echo PASSWD follows; cat /etc/passwd; echo GROUP follows; cat /etc/g
roup) >output.txt
```

Command substitution

With command substitution you re-use the output of another command as text in your command line, for example to set a variable. The other command is run in a subshell:

```
number_of_users=$(cat /etc/passwd | wc -l)
```

Note that, in this example, a second subshell was created by using a pipe in the command substitution:

```
xterm ----- bash ----- bash (cmd. subst.) --|
                                     +-- cat /etc/passwd
                                     +-- wc -l
```

 to be continued

Discussion



Rob Hubbard, [2015/06/26 09:25 \(\)](#)

I had a question about how to tell which variables are part of the environment. This article answered that question indirectly.

So, explicitly:

The command `set` lists all the variables that have been set with their values (the output tends to be verbose as it includes functions).

On the other hand, the command `env` shows only those variables (again with their values) that have been `export` ed to the environment.

 [scripting/processtree.txt](#)  Last modified: 2019/08/30 14:55 by ersen

This site is supported by Performing Databases - your experts for database administration

Bash Hackers Wiki



Except where otherwise noted, content on this wiki is licensed under the following license:
GNU Free Documentation License 1.3