Shell Environment

CIS 2230 Linux System Administration

Lecture 6

Steve Ruegsegger

Modified (with permission) by Peter Chapin

Review

- Describe the purpose of the shell program.
- Name three popular shells.
- Describe implicit vs explicit command calling
- How does the OS know which command program to run?
- How many arguments can a command have?
- What does $ ls –lrt do?
- What does $!! do? (How do you say that?)
- Most Linux commands are shell built-ins or files?
- What is the env var that has the list of executable directories?
- What are the three ways to combine commands on 1 line?
- How do you start a program in the background?
- How do you put an already running program in the background?

Hello World

- The shell is a program, trying to interpret what you are telling it
- Obviously, it runs programs:

  `$ firefox`
- We can see what it sees through the `echo` command:
  - A string: `$ echo Hello World`

Environment vars

- User Defined:

```
$ A=4
$ PORT=5904
```

  - The key → NO SPACES
- Predefined:

```
$USER, $PATH, $SHELL, $HOME
```

- To see them all:

```
$ env
$ printenv
```

- Use env vars in echo

```
$ echo You are $USER
```

Arithmetic

- Simple math is `$(( ))` in bash

```
$ echo $(( 1 + 5 ))
```

- ONLY integers

Table 7-1: Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (but remember, since expansion only supports integer arithmetic, results are integers.)
%	Modulo, which simply means, " remainder."
**	Exponentiation

Only integers...

```
$ echo Five divided by two equals $((5/2))
Five divided by two equals 2

$ echo with $((5%2)) left over.
with one left over.
```

Options

```
$ echo -n “I'm working...” ; sleep 3 ; echo “done”
```

Command substitution

- The `$()` operator completes the command, then puts the results in its place

```
$ echo The date is $(date)
$ ls -l $(which cp)
```

- A 2nd format for the same thing: back-tick

```
$ echo The date is `date`
```

- My favorite:

```
$ cp /etc/fstab /etc/fstab.`date +%F:%T`
```

(What does that do?)

- Other cool examples:

```
$ echo I am `whoami` on $(hostname).
$ echo `whoami` `hostname` `date +%F:%T` >> userlog.log
```

Globbing

- One of the things the shell does *most* is deal with filenames.
- So, there are some tricks the shell uses to help us with them
- Patterns used to match *filenames* are called globs
  - Process of expanding them is called globbing
- A glob has 3 types of wildcard characters:

  `?` , `*` , `[ ]`

Globbing

1) '*' ("splat") matches <u>any</u> number of characters: 0, 1 or more:

```
$ ls *.txt
accounts.txt letter.txt report.txt
```

- Therefore, a '*' by itself matches all files in the current directory

2) '?' matches *exactly* one character:

```
$ rm -v data?.log
removing data1.log
removing data2.log
removing data3.log
```

3) Globbing with brackets, [ ]

There are a couple of ways to interpret brackets:

- Character classes – allowed possibilities

```
$ ls data_2012_[789]_*.txt
```

- Ranges

```
$ ls drive_[b-f].log
```

- Complement – the next, single character cannot be in this set

```
$ ls /home/[!a-m]*
$ ls -d /home/[!ds]*
```

Absolutely crucial to use globs on file copying, moving & removing

- Move all *.txt files to backup dir

```
$ mv *.txt /data/backup
```

- Copy all *.doc files to another user

```
$ cp ~/docs/*.doc /home/susie/documents
```

- What is the last argument to both commands above? Why?

- Delete all *.bak files

```
$ rm ~/projects/*.bak
```

Who gets the glob?

1. The shell tries to match the glob *first*.
2. If a match is found, the shell 'expands' it.
3. If a match is not found, the glob remains and is passed to the command.

- Example:

```
$ echo *
$ echo *.txt
$ echo *.zzzz
$ echo [a-z].txt
$ echo [a-z].zzzz
```

- You can quote the glob to give it to the command and 'protect' it from the shell

```
$ echo "*"
```

Quoting in the shell

What if we don't want the shell to process some tech:

- e.g., try this:

```
$ echo you owe * me *:  $100.
```

- Double Quotes:
  - Everything ignored by the shell except three things: $, \, and '
  - Mostly used for spaces in filenames

```
$ mv "two words.txt" two_words.txt
```

- Single quotes:
  - Ignores everything – all special characters are ignored by bash

Escaping

Quoting is helpful for including special characters in filenames

```
$ ls 'paris trip (may)'
```

- Escaping is used to tell the shell that the *next character* is actually in the filename and <u>not</u> to be interpreted as a shell function
  - spaces, parens, brackets, quotes, $ | & # * ? < >

```
$ ls steve\'s\ files \(1\)
$ rm a\*\?
```

- Play with Tab auto-completion

```
$ cd paris\ trip\ \(may\)/
$ cd 'paris trip (may)'/
```

- Simplified: double quotes *interpret* $VARs and single quotes do <u>not</u>.

Shell Completion

- The shell can complete filenames for you! It's wonderful – use it!
  - Tab for Linux
  - (Esc-Esc for AIX)
- This also works with command names
- If non-unique (file or command), then Tab does 'nothing'
  - Double tab will list all possibilities

```
steve@steveprecise:~$ fi
fi            file          file-roller   find          findaffix     findmnt       firefox
fiascotopnm   filefrag      filetype      find2perl     findfs        findsmb       fitstopnm
```

Command History Interaction

- As expected, the shell keeps a command history
  - Use the Up and Down arrow keys to scroll through the list of previous commands
  - Press Enter to execute the displayed command
- Commands can also be edited/modified before being run
  - Particularly useful for fixing a typo in the previous command

Shell start-up / config scripts

- There are “many” config files
- They are 'sourced' at different times
- They are sourced in a particular order
- *Sometimes* the change carries on to new shells and sometimes *not*
- Confusing...?

Shell Configuration Files

- 1) Login shell:
  - Log in at the console or a new SSH connection
  - Once per connection to the computer
  - Config files:
    - First, it reads the global configuration from `/etc/profile`
    - Then only one of these, in this order:
      - `~/.bash_profile, ~/.bash_login or ~/.profile`
  - Login shells also source `~/.bash_logout` when the user exits

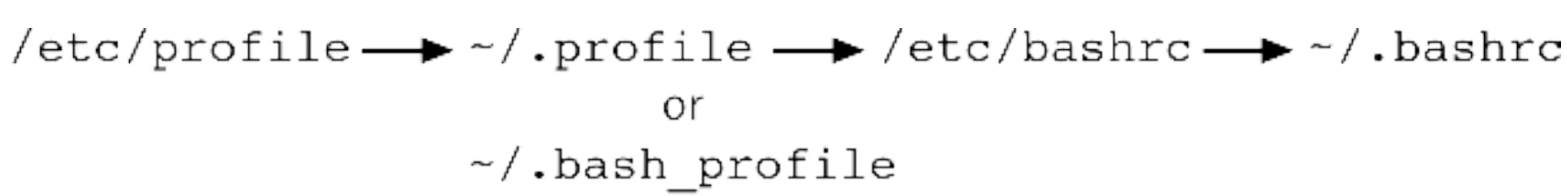


Figure 3-2 The process of executing start-up shell scripts

Shell Configuration Files

- 2) Interactive shell:
  - When you open a new window in a GUI
  - Once per shell (terminal)
  - Config files:
    - global: `/etc/bash.bashrc`
    - local: `~/.bashrc`
  - Hint! Notice the "`rc`" text as a file name ending... Stand for "remote command' files from yester-year

- Comments:
  - The 'distro' usually sets up the global rc's. (That's what a 'distro' does.)
  - If I need 'all my users' to have the same setup, I can easily do that through the 'global' rc's.
  - The default (Ubuntu) setup is to have the login shell source the interactive shell's rc too (note `~/.profile`)

What kind of 'stuff' goes into the 2 types of shell startup rc's?

- `~/.profile` is for things executed once:
  - PATH variables
  - umask
  - graphical desktop session variables
  - one time security/token items
  - application setup (db2)

- `~/.bashrc` is for the configuring each Bash shell
  - aliases
  - setting your favorite editor
  - setting the Bash prompt

"source" as a linux verb

- When you execute a program ("run a script") a new shell is created
- It (the program) does not affect the original, parent shell
- How do we change the current shell?
  We "source" a shell script.
- The command is: . (dot)

```
$ . .profile
$ . .bash_aliases
```

Review:

- 2 shell types:
  - login shell --> think "profile"
  - interactive shell --> think "rc"
- but the "profile" should also *source* the 'rc'

Aliases

- It is often useful to have bash aliases for common commands with preferred options
- An 'alias' is a shortcut or custom command
- Ubuntu is *already* set up to include the aliases in `~/.bash_aliases`
  - This is 'sourced' from `~/.bashrc`
- Note the syntax: "no spaces" around the "="

- The `alias` command with no arguments will show a list of currently defined aliases

```
$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls -F'
alias m='more'
alias rm='rm -i'
```

How to NOT use an alias

- If you want the shell to *'ignore*' or <u>not</u> use an alias, you have 2 options: temporarily, or permanently.

1. Temporarily
   - use a "\" to escape the command

```
$ \ls
```

   - More useful example:

```
alias rm='rm -i'
```

   - to delete without 'interaction': `$ \rm *.txt`
   - (note I could use `rm -f` in this example)

How to NOT use an alias

2. Permanently remove an alias:
   - Use the `unalias` command

```
$ alias foo='echo foobar'
$ foo
foobar

$ \foo
Unknown command

$ foo
foobar

$ unalias foo
$ foo
Unknown command
```

Do you understand this?

Finding an alias: type vs which

- `which` only looks at the `$PATH`
- `type` looks "into the shell"
  - no man page for `type` because it's a shell command

```
$ type rm
rm is aliased to `rm -i'
$ which rm
/bin/rm
$ type cat
cat is hashed (/bin/cat)
```

"all"

```
$ type -a rm
rm is aliased to `rm -i'
rm is /bin/rm
```

shell vs env variables

Shell Variables vs Environment Variables

- We've used shell variables in our scripting:
  - By default, they are *private* to the shell
- However, environment variables are passed to all programs run from the shell
  - This is called "*the environment*" of the session
- In Bash, use `export` to push a shell variable from being '*private*' to the shell into the environment:

```
$ files="notes.txt report.txt"
$ export files
```

  - Or combine those into one line:

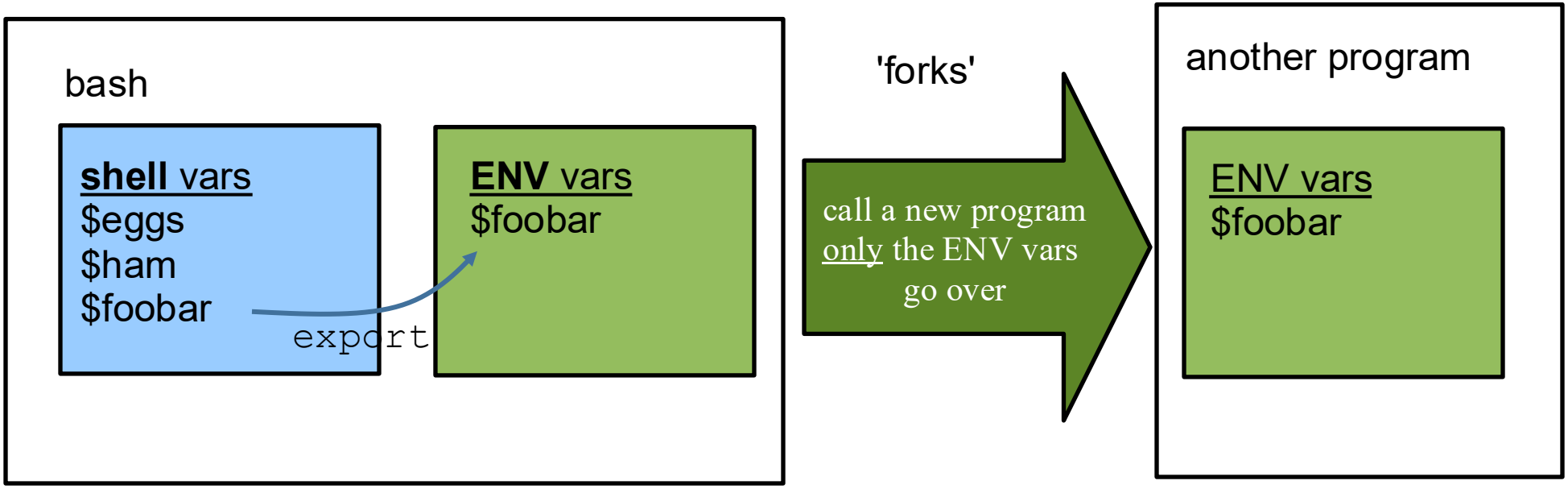
```
$ export files="notes.txt report.txt"
```

- The `env` command lists all environment variables

export example

```
$ foobar="this is foobar text"
$ echo $foobar
this is foobar text
$ env | grep foo
$ export foobar
$ env | grep foo
foobar=this is foobar text
```

Only the "env" vars are copied/available to *new* processes



(review) execute vs source

1. Execute a command:
   - *fork* a new process/shell
   - Only ENV vars are copied over (inherited)
   - run the script
   - maybe new shell variables created → they don't affect the parent
   - exit that child process/shell
   - the child did not change anything in the parent shell
2. Source a file (dot command):
   - does not fork off a new shell/environment
   - environment var changes affects the *current* shell

```
$ . <script_file>
```

misc

Always fun to change the prompt

- The variable called `$PS1` (Prompt String 1) specifies how to display the shell prompt
- It's cryptic!

```
$ echo $PS1
[\u@\h \W]\$
```

- The default Ubuntu PS1 (complicated...):

```
$ echo $PS1 \[\e]0;\u@\h:
\w\a\]${debian_chroot:+($debian_chroot)}\u@\h:\w\$
```

- The special characters:
  - \u, \h, and \W represent your user/login name, the machine's hostname, and the current working directory
  - $USER, $HOSTNAME, $PWD
- Google "Prompt string $PS1" to learn more

BASH Prompt String Settings

- There is lots of help for prompt string settings
- The following list shows the meanings of the special characters used to define the $PS1 prompt strings.
  - \t - time
  - \d - date
  - \n - newline
  - \s - Shell name
  - \W - The current working directory
  - \w - The full path of the current working directory.
  - \u - The username
  - \h - Hostname
  - \# - The command number of this command.
  - \! - The history number of the current command

Have fun and be creative!

byobu

- "text window manager"
- Just a way to have multiple shells in one (ssh) window
- Japanese word for the screen to change behind – a separator
- Written by a Ubuntu programmer. why?
- How I remember this: byob, u (cheesy)
- another, similar program is `screen`, but `byobu` is 'fancier'
- Key key-bindings:
  - F2 - new window
  - F3 - prev shell window
  - F4 - next shell window