Bash and Tmux

Joshua Kohn Chris Daw

June 4, 2019

Contents

1 Bash

Bash is the most common shell environment pre-installed on *nix machines. An acronym for the **B**ourne **a**gain **sh**ell, named after Stephen Bourne, the creator of its predecessor, the sh shell.

1.1 Background

1.1.1 What is a shell?

A shell is the environment through which system commands are issued. It handles and abstracts kernel access switching and is the primary source of contact between a user and the capabilities of their system. It is the *nix equivalent to the Windows based ‘command prompt‘

1.1.2 Why does it matter?

A natural next question might be ”Why do I need to know anything about this? Can’t I just give my commands and it gives me a response and then we’re done?”

Technically, yes, you can use your shell this way and that’s fine. But it’s also just the tip of the iceberg. Shells are more than just basic input and output, they are full-fledged languages. Part of a subclass that we can refer to as ”command languages”. There are dozens of different shells (zsh, csh, ksh, bash, etc.) and each of them can be thought of as a different language that at its core accomplishes the same task - namely, system IO. This is very similar to how, for example, Java and Python are both ”imperative languages” that each try to accomplish the same thing but approach it in syntactically different ways.

1.1.3 What’s with all these symbols?

It can feel like hieroglyphics at first but you’ll get used to it. Here’s a brief cheat sheet to help out:

- `$` - Starts the terminal output. You can change this if you really don’t like it but it’s definitely standard. Try to make peace :P
- `.` - Refers to the current directory
- `..` - Refers to the directory above the directory you’re currently in
- `~` - Refers to the ‘home‘ directory

1.2 Syntactic Sugar

This is a tips and tricks section. Without any configuration or customization, these are some things that you can always expect to work in a bash terminal that can save you a significant amount of time and frustration.

- `&&` - Links 2 commands together to be run with a single carriage return
- `!!` - Repeats the last command entered (extra useful for when you forget to add 'sudo' to the front of a command)
- `!<string>` - Repeats the last command run starting with `<string>`
- `!$` - Repeats the last command with new first argument
- `$()` - Allows nested commands

1.3 Bash Files

There are a series of files that the system knows to check which allow you to access this language and manipulate your environment. Unlike the languages you are likely used to, you don't explicitly tell these program files to compile and then run them by hand since, of course, you need to USE the shell to run them. A (non-exhaustive) list of the typical files and their locations are given below. They are listed in the order of their execution.

1. Loaded on Login
   - /etc/profile
   - ~/.bash_profile
   - ~/.bash_login
   - ~/.profile
2. Loaded Interactively
   - ~/.bashrc
3. Loaded on Logout
   - ~/.bash_logout

This tutorial will focus mainly on the ~/`.bashrc` file as most any customization that you'd like to accomplish within your shell can be achieved from this file. If you are certain something is better served to only be run once and only on login or only once and only on logout there are clearly better options but these are edge cases and not the general use case. The skills you learn through ~/`.bashrc` customization will be applicable to the other files as well so if a situation arises, you can migrate your code to the necessary location.

1.4 Aliases

Aliasing is the most common use of the ~/`.bashrc` file. Many people who don't understand the full depth of bash still utilize aliasing. An alias is essentially just a shorthand version of a command - like a command nickname.

Example

Say you frequently find yourself needing to navigate to a deep location in your file tree and you don't feel like going through the process of multiple tab completions or half a dozen different `cd` commands 20 times a day. Bash makes this very simple:
`alias go='cd ~/my/directory/location/goes/here'`. After placing this inside of your `~/.bashrc` file, anytime you write `go`, the shell will interpret that to mean the entire command:
`cd ~/my/directory/location/goes/here`. This can be broken down to:
`alias <nickname>='<command to call when nickname is entered>'`

note:

You must tell your shell that the environment has changed! In order to reload your `~/.bashrc` file, use this command: `source ~/.bashrc`.
You can even re-assign commands that you're used to in order to give them more zest. Try this: `alias ls='ls --color=auto --group-directories-first'`

1.5 Functions

Because bash is a full-fledged language, of course you can write functions! Functions allow you to do things that simple aliases can't, such as conditional execution, variable assignment and argument addressing. Before we get to an example, here is some basic reference material that can help you understand what's going on:

- `$1` - Refers to the first command entered (can be used with any positive number 'n' to refer to the nth command entered – 0 refers to the function name)
- `$#` - Refers to the integer 'number of commands entered'
- `$@` - Refers to every unaddressed command entered
- `$?` - Refers to the last 'return' value
- `-eq` - Equals (similarly `-ne` means 'not equals')
- `command` - Keyword that tells the shell to run a command (as opposed to addressing a variable or statement in a function)

Example

Goal: get a file from your account on the Western machines to your local machine.

```
get_from_wwu() {
  if [ "$#" -ne 2 ]; then
    echo "2 arguments: 1st is file location; 2nd is arrival location"
  else
    command scp -P 922 -r username@linux.cs.wwu.edu:"$1" "$2"
  fi
}
```

syntax

- `if [ <condition to check> ]; then` (elif can be used as well)
- `if` conditions end with `fi`

Voila! With this in your ~/.bashrc file you can now simply type:
`get_from_wwu ~/file/i/want ~/where/it/goes/` in your shell environment to get any file from the Western machines to your personal machine! No more running back to campus frantically to get your homework or lab files!

1.6 Scripts

Remember when we said you don't explicitly tell bash files to run? Well that's not all together true. The code that we put inside of our known bash files is added to your bash environment and is integrated into your terminal! If you're in a situation where you want a bash program to run just once and not affect your environment you can make a bash script!

A script is just a text file with the `.sh` extension that begins with the line `#!/bin/bash` and contains code that is isolated to its runtime only.

Example

Let's say we want to automate creating a `.tar` file with all the `.c` files or `.java` files or any other type of files we want in the directory we run it from. We can write this code in a separate file that looks like this:

```
#!/bin/bash
echo "Name of the project?"
read project_name
echo "File type to tar?"
read ext
command mkdir $project_name
command cp *.$ext $project_name/
command tar -cvf $project_name.tar $project_name/
```

The `#!/bin/bash` flag tells the terminal that you are using bash as your language and that it is located at `/bin/bash` (if this is installed in a different place you can find out by typing `type bash` into your terminal. You should use whatever it outputs after the `#!` flag). Note that variables are assigned with just their name but they are *addressed* with a prepended `$`.

You can now run this program (say we named it `myScript.sh`) with the command:

```
.  .myScript.sh
```

Also note that you must give this script execute privileges before it will be allowed to run. This can be done with the command:

```
chmod +x myScript.sh
```

This may not seem necessary (and here it isn't) since you could just add this as a function to your ~`/.bashrc` and call it directly without any issues (and it'd probably be simpler too). One advantage to this is it can significantly simplify your ~`/.bashrc` since you can still make a function that does this, except now you can just make your function call your script! This can keep your ~`/.bashrc` file from becoming too unruly.

As a practical example, you could also create a script and tie it to a keyboard shortcut! Whenever you define a keyboard shortcut you are only given a single command that is to be executed. Technically you *can* tie a bunch of commands together with `&&` but not only does this look terrible, it also doesn't allow you access to variables and conditional execution or any of the other goodies that come with a full language! You could create a keyboard shortcut that, for example, takes a screenshot for you, blurs the image and makes that image your current lock-screen. This is cool functionality that's easy to implement and would be insanely messy without scripting!

1.7 BONUS

There is *so much* more that bash can do. I encourage everyone to play around with their bash files and make their terminals a reflection of themselves and their preferences. Maybe even try out a different command language environment like zsh!

If you're not convinced yet, here's some neat stuff that might convince you how useful controlling your terminal experience can be:

- Man pages can be a pain to read so here's a great function to help add some more pizazz and get you started with customizing your terminal environment (and help you get through your system courses)!

```
man() {
  LESS_TERMCAP_md=$'\e[01;31m' \
  LESS_TERMCAP_me=$'\e[0m' \
  LESS_TERMCAP_se=$'\e[0m' \
  LESS_TERMCAP_so=$'\e[01;33;90m' \
  LESS_TERMCAP_ue=$'\e[0m' \
  LESS_TERMCAP_us=$'\e[01;32m' \
  command man "$@"
}
```

- `PROMPT_DIRTRIM=2` in your ~/`.bashrc` will limit your shown directories to 2! (eg, `.../my/dir`)
- The `PS1` variable will allow you to alter the way your terminal lets you know it's ready for input! The default is `<username>@<devicename>:<location>$`. If you want to just show the location and if you have a deep hatred for the dollar symbol for some reason, you could do this:

```
PS1='\w -> '
export PS1
```

  If you feel like playing around with this more, this is how PS1 sees some common things:

  - `\u` means your username
  - `\h` means your device name
  - `\w` from the example, means the current directory
  - Check out the `PROMPTING` section of `man bash` to see all the options!
  - `\e[5m` makes everything blink! (be careful using this, it's very stupid :P)
  - Some great examples of fun stuff you can do with your PS1 variable are given on flogisoft
- Ctrl + d will send the `exit` command and close your session.
- Ctrl + l will send the `clear` command and give you a clean prompt.

2 Tmux

Tmux is a powerful **T**erminal **mu**ltiple**x**er allowing you to arrange multiple terminal sessions in one window.

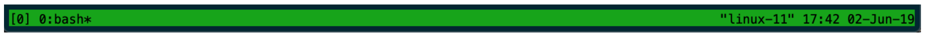
2.1 Basics

2.1.1 Sessions

To start a new session in tmux, run the following command:

```
tmux
```

When you first open a tmux session, it will look very similar to a normal terminal window, but with a status bar at the bottom of the window.



```
[0] 0:bash*                                                "linux-11" 17:42 02-Jun-19
```

This status bar means you are ready to start using tmux!

You can start tmux sessions with a specific session name to make it easier to find later.

```
tmux new -s [session name]
```

Where the $-s$ flag specifies the session's name.

Entering a session is also known as *attach*ing to it. If you only have one session running you can use the command

```
tmux attach
```

to attach to your session or, if you have more than one session running, you can specify a target session with

```
tmux attach -t [target name]
```

And similarly

```
tmux detach
```

to detach from your session.

2.1.2 Prefix

To send commands to tmux for its various functions, you need to use the prefix. The prefix is a configurable 2-key combination which by default is:

Ctrl + b

After you send the prefix key combination, the next key you press will be a command to tmux. For a list of commands see this GitHub gist. We will cover the necessities in this guide.

Since the prefix is configurable (for example, I use Ctrl + a), we will refer to is as `prefix` from now on.

For example, to detach from your current session, use `prefix` + d .

2.2 Panes and Windows

When you first open tmux, it's no different than just having one big terminal and a status bar. To take advantage of tmux's ability to run multiple terminal sessions in one place, we need to learn about panes and windows. Understanding this section will provide you the bulk of the utility of tmux in my opinion.

2.2.1 Windows

A window is similar to a tab in a web browser. You can switch back and forth between windows, but you cannot view two windows at once. Panes live in windows. In one window you can have as many panes as you want. For example, you can edit a python file in *vim* in one pane and then run it in the next pane over while still having the file open.

Basic window shortcuts:

- `prefix` + c - Create a new window
- `prefix` + & - Kill current window
- `prefix` + n - Go to next window
- `prefix` + p - Go to previous window

2.2.2 Panes

When you first open tmux, it automatically opens to a single window with one pane in it. Since this is the only pane, it takes up the entire window. There are two basic ways to split a pane in two: horizontally and vertically. Either split cuts the current pane in half.

- `prefix` + % - Vertical split
- `prefix` + " - Horizontal split
- `prefix` + x - Kill current pane

Now that we can split the panes, we need to know how to resize and rearrange them.

- `prefix` + } - Rotate pane right
- `prefix` + { - Rotate pane left
- `prefix` + o - Select next pane
- `prefix` + (up , down , left , right) - Navigate to pane in the specified direction
- `Hold prefix` + (up , down , left , right) - Resize current pane in specified direction

2.3 Tmux Configuration

If you find the default keybindings confusing (I know I do) or you want to modify the look of the status bar, you can change them with a configuration file placed in your home directory!

This file will live under $\sim /.tmux.conf$.

The following section will show you some simple changes you can make with the configuration file. These are only suggestions, feel free to use any subset of them!

2.3.1 Change default prefix

```
unbind C-b
set-option -g prefix C-a
bind-key C-a send-prefix
```

2.3.2 More intuitive pane splitting

Use `prefix` + | and `prefix` + − to split the panes more intuitively than the default shortcuts.

```
bind | split-window -h
bind - split-window -v
unbind '"'
unbind %
```

2.3.3 Change default shell

```
set-option -g default-shell /bin/zsh
```

2.3.4 Synchronize panes

```
unbind s
bind s set-window-option synchronize-panes
```

2.3.5 Use Alt for pane navigation

```
# Use Alt + <arrow key> to change panes
bind -n M-Left select-pane -L
bind -n M-Right select-pane -R
bind -n M-Up select-pane -U
bind -n M-Down select-pane -D

# Use Alt + <vim direction binding> to change panes
bind -n M-h select-pane -L
bind -n M-l select-pane -R
bind -n M-k select-pane -U
bind -n M-j select-pane -D
```