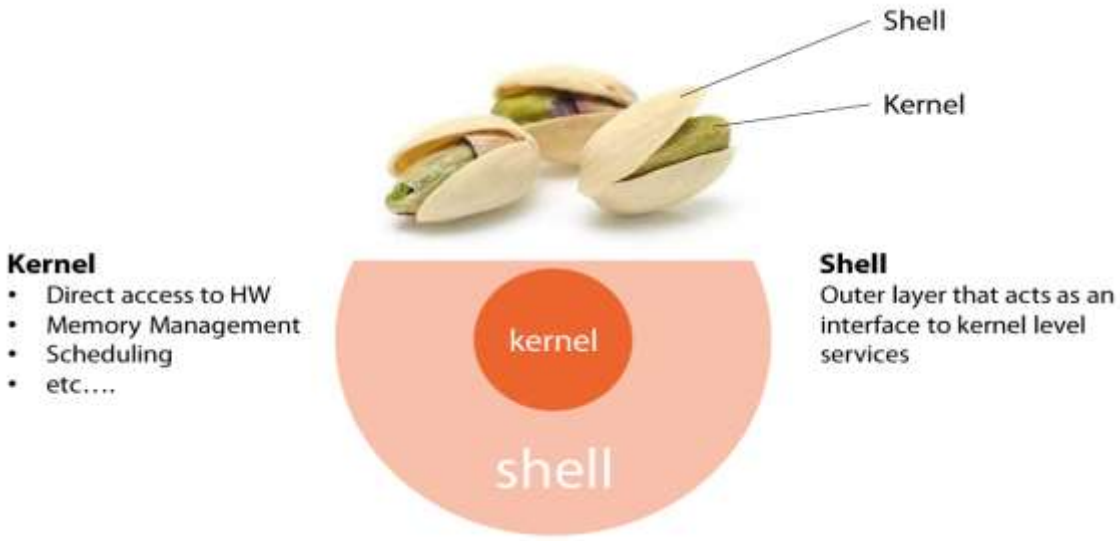
# SHELL PRIMER

## Shell Primer

- At the heart of any Linux system is the kernel. In fact, the kernel isn't just central to Linux, the kernel actually is Linux.
- Anyway, the kernel is the code that interfaces directly with the hardware, does all the memory management, decides who gets time on the CPUs, process management, all of that is the kernel.
- Wrapped around the kernel there in our picture is the shell, an outer layer that acts as kind of an interface into kernel-level services and the likes.
- So the kernel is the arguably more important part at the center, and then the shell surrounds it.
- Okay, here we are at a shell prompt, and like we said it's a character-based shell, so we type in characters and get character-based responses.



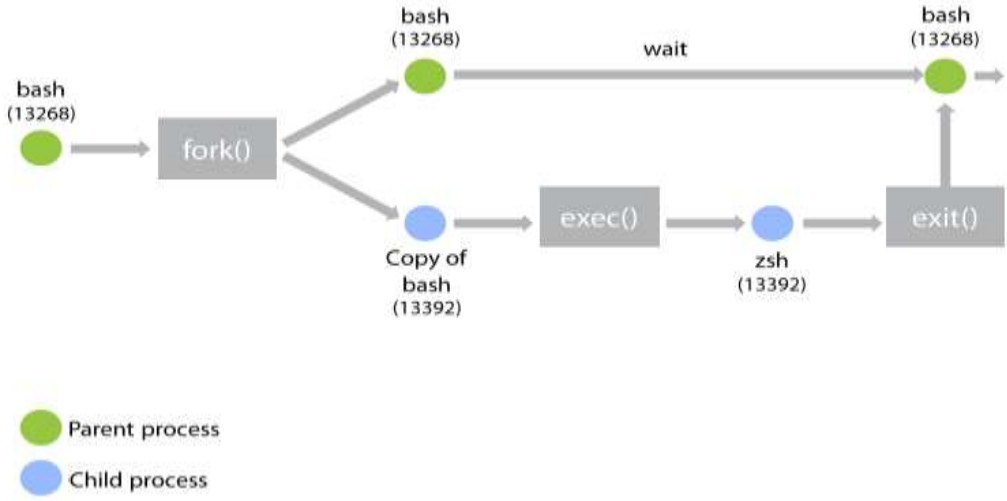

## Current Shell

- Linux supports loads of different shells, but the default, and the most popular is Bash, and Bash is an acronym for Bourne Again Shell.
- One way to find out our current shell is to use the ps command.
- To switch out of Bash and into the Z shell, all we need to do is execute the Z shell binary.

## Linux Process Theory

- So we started out at the Bash shell and we switched to the Z shell, and then back again. So anytime we switch shells under the hood, the current shell process forks itself and starts a new child process.

- This new child process then goes on to become the process that runs the new shell. So we start out here on the left with our Bash process running. We've given it a PID, a Process ID of 13268.
- Now then, when we invoke the Z shell from the Bash prompt by typing zsh, Bash initiates a fork system call. What this fork does is it creates a second process, so we end up with Bash here, but also a copy of the Bash process down here.




- But see how we've given this copy of Bash its own PID, its own PID 13392? This fork system call creates a relationship between Bash on the top here, and the copy of Bash at the bottom. In the relationship Bash is the parent and the copy is the child.
- Then, under normal circumstances, Bash continues in the wait state and the copy of Bash then performs one of the exec system calls. What this exec does is it changes the child process from being pretty much just a clone of its parent, to being its own running program. So, in our instance, the exec loads the program code for the Z shell onto the child process so that our child, down here, effectively becomes the Z shell.
- So we end up with two processes, Bash up here in the wait state. Remember, we've just initiated the Z Shell so we're not interacting with the Bash shell anymore. So that's just up here waiting. Down here, we're now interacting with the Z shell, so the Z shell continues to run until we type exit, at which point the Z shell then performs an exit system call, which passes its exit code back to its parent, Bash. And Bash then comes out of the wait state, and we can interact with it again.

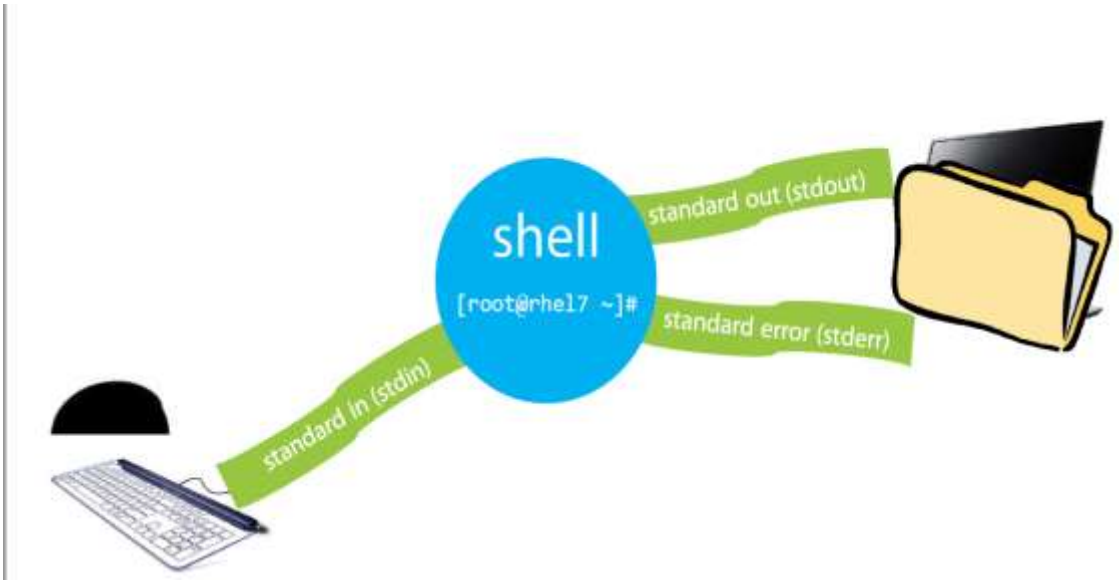
## Virtual Consoles

- Whether we stood physically in front of a physical Linux server, or if we're looking at the console of a virtual machine, Linux gives us a number of virtual consoles that we can connect to. So when we boot a Linux system, when the boot process is done and dusted, we end up with a UNIX login prompt, like this right here.

- So let's log in. Okay, first things first, let's see which virtual console we're currently on, and we do that by typing tty at the prompt. And we're on tty1. Let's also see who else is logged in by typing who.
- Now, let's press ALT and F2 to get a new virtual console. So here we are, on the same machine, vbrhel7-01, but on a different virtual console. And this time let's log in as root. And let's run the who command again, and sure enough, there we are as an additional user connected to tty2, virtual console 2.
- We can do the same thing with ALT and F3 to get to tty3, ALT and F4 for tty4, and so on. Now there's normally 7 of these virtual consoles, and they're all accessible by using ALT + F1 through to F7 on the keyboard

## Standard Streams

- First up, let's maybe imagine streams as simple streams of characters, like a command that we type into the shell. So this command here, who am i, and that command literally is a stream of characters that we've just input into our terminal or shell.
- We hit Enter. And the output that we get is literally another stream of characters, this time an output stream from the shell. So we've just sent an input stream to our shell a command, and we've got an output stream back from the shell. So each time we type a command into the shell, what we're basically doing is sending a stream of characters to the shell. In response to our commands, the shell pumps out a response, another stream of characters, usually to the screen, and that is our output stream.




- Now our shell has three standard streams, one for input, one for output, and one for error messages. First up, let's attach our standard input stream, often called just standard in, and we'll put that over here on the left.Now, let's add our second stream, standard output, over here. This guy's usually just called standard out. And finally standard error, or standard err, just over here.

- So that's our three standard streams, in, out, and error. These are the major communication paths that we have with our shell. We send commands to the shell via the standard in stream, and we get responses back from either the standard out stream or the standard error stream, depending of course on whether or not our command generates a normal response or an error response.
- By default, in just about every shell scenario, standard in gets connected to the keyboard, and standard out and standard error to the screen, console, terminal window. Now a second or two ago we said that standard in is usually connected to the keyboard, and standard out and error usually connected to the terminal window. Well, they don't have to be, standard in could be configured to accept input from somewhere else, maybe like a speech recognition subsystem. And likewise, standard out and standard error can be connected elsewhere, maybe the printer or a file. So streams are flexible, we can attach them to different devices.

## File Descriptors

- Each of our three standard streams is also known by a numeric character, known technically as a file descriptor. Standard in is referred to as file descriptor 0, standard out is file descriptor 1, and standard error is file descriptor 2.
- Literally, the shell just accepts input from standard in and it kicks responses out to standard out, or maybe standard error.

## Standard Out

- By default the shell will send standard out to the screen. We should be able to just hook standard out up to another device, so let's see how we hook it up to a file.
- To do that we just use the redirect operator, the greater than symbol. So we can redo our last command, but this time we add the greater than sign, and we give it a file, let's say /tmp/alertfile, hit Enter, and voila, no alert message on the screen. Standard out, it looks like, was successfully diverted to the alertfile in tmp.

## Clobbering

If the file that we're redirecting to doesn't already exist, then it'll be created automatically. We need to be really careful if the file does already exist, because if it does then redirecting standard out to it will literally clobber that file. Clobber actually being the technical term for overwriting it, destroying it, blitzing, smashing it.

## Redirection

- So a couple of times now we've mentioned that everything in UNIX, and therefore Linux, is a file. Devices attached to a Linux system show up as files in the local file system.
- So our screen shows up as a file, keyboard, mouse, disk drives, the whole shebang, they all show up as files. And we, as sysadmins, can interact with them as files.

## Standard Error

- So according to the table, standard out, we can use a combination of the file descriptor number and the greater than sign, like so. But hang on a minute, we've just been redirecting standard out by only using the greater than sign, so without specifying the file descriptor.
- And that's right, we have, because using the greater than sign on its own defaults to assuming file descriptor 1.
- Basically, to redirect standard out on its own we don't need to specify the file descriptor, but, but to redirect standard error, which we're talking about now, well, we do have to provide both the file descriptor and the greater than sign like this, but let's see what it looks like in action. Now then, in order to redirect standard error in any meaningful way we need some error text to redirect.

| Stream | Description | File descriptor | Operator |
|---|---|---|---|
| stdin | Shell standard input stream | 0 | < |
| stdout | Shell standard output stream | 1 | > |
| stderr | Shell standard error stream | 2 | > |

stderr

```
ls -l 2> /tmp/file
```

## Introduction to Variables

- The theory states, that a variable is just something that can contain a value. And that value can change, or vary, over time.

- So I suppose we could say that each of us, as individuals, has a variable associated with us called age. Right now, the value stored in my age variable is 36, but come ask me the same thing next year and that variable will contain the value 37. But in that simple example, we had two parts to the variable, the name of the variable, age, and the value of the variable, for me, 36. And we often call these kinds of constructs key value pairs, and that's generally how variables work, they're key value pairs.


- A shell, or a Linux process to be honest, also has a ton of variables associated with it. We can see that value of variables using the echo command, and then preceding the name of the variable, the key part, with a dollar sign. Echo, $BASH_VERSION shows version 4. 2. 45. The key part of the key value pair and the output that we get from the command is the value part, the actual contents of the variable.
- Maybe think of variables as buckets, or drawers, or some kind of container, something that we put stuff into. We've got a bucket for BASH_VERSION and we filled it with 4. 2. 45. And we've got a bucket for HOSTNAME, that one's got our system's hostname in it. If we want to see a list of all variables on our current shell session, well, we can use the set command.


## Introduction to Environments

- So what's an environment? Well, I've got a work environment that helps me do my job. Well, a shell environment is pretty much the same. It consists of stuff that help the shell do its job, or help us work within the shell.
- So from here on our command line, we're going to type env. Well, what we've done here is print to the screen the environment variables that make up our shell's environment. And let's pick this one here, PATH. In case you don't know, the PATH variable lists a bunch of directories that

we want the shell to search anytime we type a command, basically telling the shell where to go and find the binary program file for the commands that we type.

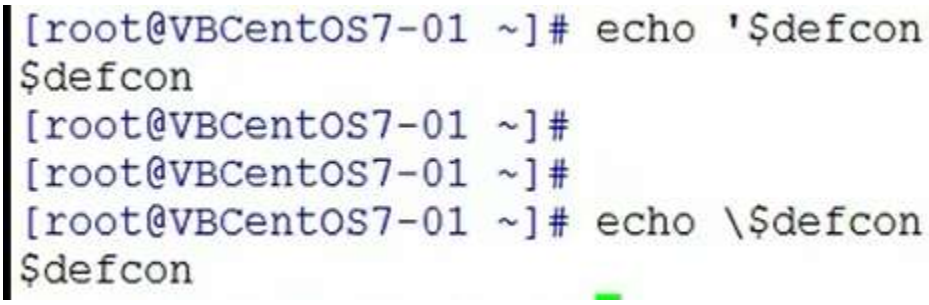
- So path is a shell construct, a variable that help us and the shell do our jobs easier.

## Shell Variables

- When it comes to the shell there are two types of variable, shell variable and environment variable. Shell variables referred to as local variables. Shell variables are created on the command line by typing the name of the variable followed by an equal sign, and then the value that we want to give the variable. And by default, that value is going to be stored as a string and the process for creating variables is called declaring and initializing.
- Let's declare a variable called defcon, for our defense condition, and initialize it with a value of 5. Now to see that variable we can go echo $defcon. If we go and fork off another shell as a subshell, that subshell won't inherit the defcon variable.
- So shell variables are local to the shell and they are not passed to other shells.

## Expanding Variables

- Okay, let's talk about this dollar sign that we've been using with the echo command. Basically, this dollar sign tells the shell that what's coming next is a variable and not just a string. So, for example, if we just did echo defcon, see we get this string defcon echo to our standard out. But if we do the same, but precede the variable name with a dollar sign, well, the shell performs a little bit of magic called expansion. And all this is, is when we hit the Return key, the first thing the shell does is parses the command line to look for any variables indicated by a preceding dollar sign. Then when it finds them it expands them out to the values they contain, effectively replacing the name of the variable in the command with the value stored in the variable. So in our case, replacing the $defcon string with the value 5, meaning that instead of passing the string defcon, or even the variable defcon to the echo command, the shell passes the value of 5 to the echo command.
- But what if we want to echo $defcon to the screen, without the shell treating it as a variable and expanding it out? Well, a couple of options there. First off, we could enclose the value $defcon in single quotes forcing the shell to treat it as a string.
- Another way,  we tell the shell to ignore the dollar sign special meaning just this one time. And we do that by sticking a backslash right in front of the dollar sign, like this.



## Environment Variables

- We saw, that shell variables only exist in the current shell. The shell they were created in. They don't even get passed to subshells. But there are times when we need the variables that we create in one shell to be passed to other shells. In order to do that we need to make a shell variable into an environment variable, and to do that we export the shell variable with the export command.
- So export defcon, and we'll switch over to another shell, Z shell again. And echo $defcon, and see, this time our subshell knows all about our defcon variable. And that was all because we exported it as an environment variable.

## Customizing Your Shell

- We've said that a shell inherits its environment from its parent process, and that's especially true for subshells forked off from other shells. But, when our shell starts up it also runs a bunch of startup files, and these are where we can make permanent customizations.
- The basic process on a RHEL-based system is this, /etc/profile runs first. It sets a bunch of system-wide settings and it's really not recommended to make changes to this file.
- It's a much better idea to make system-wide changes to shell scripts in etc/profile. d. But changes made here are still system-wide changes remember, nothing personal yet. Next up,. bash_profile gets executed. This is a personal Bash config file. Anything we put in here will apply to us, and us alone, nobody else. We can see here that in this particular one we've got some customization to the path variable going on, adding our personal bin directory to the end of the existing search path. And we can see that this files calls. bashrc also in our home directory. Now, this guy is a great place for personalization.
- This file then calls the global /etc/bashrc file. Again, this file just configures system-wide stuff, and it's not recommended to change it. The takeaway point for us on customizing our shell is that. bashrc and. bash_profile, both of which exist in our home directories, are where we want to make personal changes.


## Shell Commands

## Executing Commands

- Okay, once we've entered our command plus any required arguments and options, we simply hit Enter, and wait for the response. Nice, but behind the scenes the shell's got a bit of work to do. One of the first things the shell does is it expands any variables.
- The shell needs to determine whether or not the command specified is a shell built in or an external binary. Most commands are external binaries, meaning that the code for the command is implemented in an external program file. For example, the vi command is a program file located in /usr/bin.

**Shell Built-in:** Code for the command included in the shell

- Faster and more efficient than an external program
- Can manipulate the shells environment
- Can bloat the shell

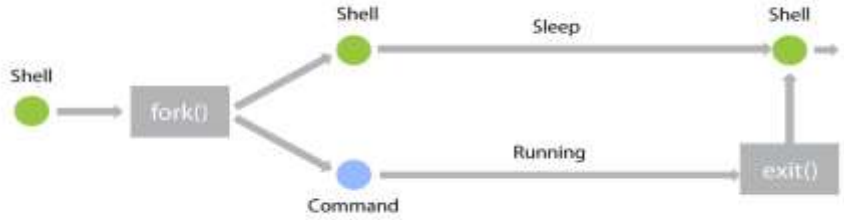
- The other way of implementing a command is as a shell built in. These commands are known natively to the shell so no external program file is required. A couple of examples include cd and set. The major advantage of a built-in is that they run faster than external binaries, basically, because there's no requirement to fork a new process off of the shell. And the fork system call carries with it its own overhead, so no fork system call equals faster to execute.
- Also, built-ins have native access to the shell's environment. What this means is they can manipulate and change variables, and the likes, that exist within the shell's environment. Great, but for every built in the shell program itself gets a little bit larger as the code for the command has to be implemented within the shell.
- Most commands are implemented as external program files, so in order to run those commands the shell needs to be able to find and execute the command's program file, which by the way is always the same name as the command itself. So the ls command is implemented as a binary program file called ls, vi as a file called vi.

**External Binary:** Code for the command implemented as an external binary program file

- Slower and less efficient than a shell built-in
- Keeps the shell small

- The shell needs to know where to look to find these files, and that's where the path environment variable comes into play. As we can see path is a colon-separated list of directories, and that list of directories is where we want the shell to search when looking for program files to execute. So when we hit Enter after typing a command, the shell first checks

whether or not the command is a built-in, if it is it runs the command, if it's not then the shell searches the directories listed in the path variable, in order. And the first binary it finds matching the name of the command gets executed, assuming, of course, that the user running the command has read and execute permission against the file.




- And the file it finds and executes can be either a compiled binary, or just a shell script. And the order in which we list directories in the path variable makes a difference. Once a file is found it's executed and a new process is forked off from the shell. Our shell goes into the background and sleeps while the new process executes. When the command is done, the process running it provides a return code to the shell, and the shell comes back to the foreground and enters the waiting state, waiting for us to input a command.

## Command History

- Bash keeps a history of commands that we've executed, and then, as and when we need to, we can recall those commands and execute them again without having to type them all out again. And I think the most common use of it is probably the up arrow key.
- By default, when we log out of a shell the commands that we've typed into that shell get archived off into a file called .bash_history, and that lives in our home directory.
- Bash uses environment variables to control history related settings. So if we go env and grep for HIST, we can see that we've got two currently set in our shell. HISTSIZE here set to 1000 tells Bash to save only the last 1000 commands from our current shell session when we log off. And then there's HISTCONTROL=ignoredups here, stops duplication in our history.

## Pipelining

- Remember how we've got our shell here in the middle, and then the way that we interface with the shell is through standard in, standard out, and standard error. We give the shell commands via standard in, and the shell gives us responses through standard out or standard error. But then remember, we talked about how we can redirect any of these streams, and we actually showed how that we can redirect standard out and standard error to files instead of the screen.

- The pipe symbol, what this does is tells the shell that we want to take the output of the command immediately before the pipe, and instead of sending it to the screen send it to the standard in of our next command.

## Aliases

- Aliases then are a great way for us to reduce the amount of typing we need to do. Now there are more uses for aliases, but definitely one of the major uses is to create command shortcuts.
- So to see a list of all the aliases on our system we can just type alias on its own. So only a handful or so configured. As we saw in a previous module, cp, mv, and rm, so copy, move, and remove are all aliased to interactive mode with the -i option forcing us to enter a confirmation keystroke to confirm before the actual command executes, or actually before we overwrite or delete anything.

```
[cloud_user@4360ea05821c ~]$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias rvm-restart='rvm_reload_flag=1 source '\''/usr/local/rvm/scripts/rvm'\'''
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

## Backgrounding

- So far we've talked about commands running as processes, when we've talked about the shell forking itself to run a shell command in a new process. Well, these processes are also known to the shell as jobs. But don't let all the different terminology confuse you. Typing ls to see a directory listing is giving the shell a job.
- It just so happens that, behind the scenes, the shell forks itself, creates a new process, and execs the program code of the ls binary onto the new process stack. We can see a list of current shell jobs with the jobs command.
- If you're running a command and you know it's going to take a long time, then you probably want to do something like put it in the background.
- This ampersand that we added here, at the end of the command line that tells the job to run in the background. So to run any command in the background we simply give the ampersand to the end of the command.