CIS 18A
Introduction to Linux / Unix

The Shell

De Anza College
Instructor: Clare Nguyen

Topics

- Redirection
- Pipe and tee
- Multiple commands and grouping
- Quoting rules
- Command substitution
- Job control
- Alias
- Variables and environment variables
- Saving environment customization
- Command history
- Different shells

Working with the Shell

- The shell is the interface between you and the Linux system. So far you've observed the shell working in a cycle to support you:
  1. Print the prompt and wait for your command.
  2. Parse your command into the utility name, options, arguments.
  3. Search for a utility that matches the name.
  4. If no match, print an error message and return to step 1.
  5. If match, start a process to run the utility, pass to the utility the options and arguments, expand argument names if there are wildcards.
  6. Wait for utility to finish and end the process.
  7. Print any error message or output from the utility, if any is sent back from the utility.
  8. Return to step 1
- Now we learn how to use some more advanced features of the shell

Input and Output

- When you give input to a command, you type in from the keyboard. *Standard input* for the shell is from the keyboard.
- When the command has an output, the shell prints it to the screen. *Standard output* for the shell is to the screen.
- When the shell or the command has an error message, the shell prints it to the screen. *Standard error* for the shell is to the screen.
- It is possible to tell the shell to get input from some place other than the keyboard.
- It is also possible to tell the shell to send output and error to some place other than the screen. When this happens, you won't see any output or any error on screen.

Redirection (1 of 2)

- Redirection tells the shell that input, output, and/or error is not from the keyboard or the screen, but from a text file.
- Redirecting input tells the shell to get input from a file.
  Redirecting output tells the shell that output goes to a file.
  Redirecting error tells the shell that error goes to a file.
- To redirect input: command < filename
  - command means the full command line, with options and arguments if necessary.
  - filename can have a path, it is the input to the command.
- To redirect output (overwrite): command > filename
  To redirect output (append): command >> filename
  - command means the full command line, with options and arguments if necessary.
  - filename can have a path, it stores all output from the command.
  - If filename doesn't exist, a new file is created.
  - If filename exists, the overwrite option will overwrite the existing file, and the append option will append the output to the file.

Redirection (2 of 2)

- To redirect error (overwrite): command 2> filename
  To redirect error (append): command 2>> filename
  - command means the full command line, with options and arguments if necessary.
  - filename can have a path, it stores error messages from the command.
  - There is no space between the 2 and the > or >>
  - If filename doesn't exist, a new file is created.
  - If filename exists, the overwrite option will overwrite the existing file, and the append option will append to the file.
- A combination of redirection of input, output, and error can be done: command < input_file > output_file 2> err_file
  where the order of redirecting input, output, error is not important.
- To have output and error redirected to the same file:
  command > output_file 2>&1

  In OS tradition, the input is assigned the value 0, output is assigned the value 1, and error is assigned the value 2.
  So 2>&1 means the error (2) is redirected (>) to the address (&) of output (1)

Pipe

- Instead of getting input / output from / to a file, you can also ask the shell to get the input/output from / to another command. This is done through a pipe with the symbol: |
- Format: command1 | command2 | command3
  - command1, command2, command3 are different commands with options and arguments, if necessary.
  - The output of command1 does not get printed on screen, instead it is sent to command2 and becomes the input of command2, therefore command2 does not need an input argument.
  - Likewise, the output of command2 becomes the input of command3 so command3 does not need an input argument.
- Rules for piping:
  - The command before the pipe symbol | needs to have output.
  - The command after the pipe symbol | needs to accept input.
  - You can pipe as many commands together as needed, as long as there are command output that can be used as input to another command.

tee

- So far we can tell the shell to send output to a file *or* to another command.
- It is also possible to tell the shell to send the output to *both* a file *and* another command, by using a tee.
- It is called a tee because the shell essentially makes 2 copies of the output and pipes both through a **T** intersection. At the T intersection, one copy goes to the file and the other copy goes to another command.
- Format: command1 | tee filename | command2
  - command1, command2 are different commands with options and arguments, if necessary.
  - The filename comes after the tee.
  - The output of command1 is piped | into the tee, the first copy goes to filename, and the second copy goes to another pipe | to become the input to command2. command2 does not need an input argument.
  - If there is no piping into command2, then the second copy goes to standard output by default:
    command1 | tee filename

Multiple Commands on a Command Line (1 of 2)

A. Commands that work together

- With piping, the command line contains multiple commands that work together.
- Piping can work with redirection and tee, allowing the shell to do complex coordination between the commands on the same command line.
- Examples:

  command1 < inFile | command2 | command3 > outFile

  The input of command1 is inFile, the output of command1 is the input of command2, the output of command2 is the input of command3, the output of command3 is redirected to outFile.

  command1 | tee outFile1 | command2 | tee outFile2

  The output of command1 goes to outFile1 *and* is the input to command2, the output of command2 goes to outFile2 *and* to screen.

Multiple Commands on a Command Line (2 of 2)

B. Commands that are independent from each other

- You can have multiple commands on the same command line, but the commands are *independent* from each other.
- Format: command1 ; command2 ; command3
  - command1, command2, command3 are different commands with options and arguments, if necessary.
  - The command line above runs the same way as if you type the 3 commands on 3 separate command lines.
- Most of the time, each independent command should be put on a separate command line. This causes the output of each command to immediately follow the command line, making it easier to distinguish the output.
- Putting 2 or more independent commands on the same command line is useful when:
  - the 2 commands complement each other and there aren't that many output lines combined. For example: cd; pwd
  - The independent commands all have output or error that should be redirected together into 1 file.

Command Grouping

- When multiple, independent commands need to have all their output or all their error redirected to the same file, use command grouping.
- Format: (command1; command2) > outputFile
  - command1, command2 are different commands with options and arguments, if necessary.
  - The parentheses ( ) tell the shell to group the commands together.
  - The shell runs command1 and then runs command2. The output of command1 is redirected to outputFile and the output. of command2 is automatically appended to outputFile, right after the output of command1.
  - You can group as many commands together as necessary.

Preserving Command Arguments (1 of 2)

- Command arguments typically don't have quotes around them.
- But single quotes or double quotes can be used around arguments.
- The majority of the time, quoting the arguments doesn't affect how the command runs, so quoting is not used.
- There are some cases where quoting the argument is necessary:
  1. To preserve white spaces in the argument.
     - Without quotes, extra white spaces will be ignored by the shell and will appear as a single white space.
     - You can use single or double quotes to preserve space.
  2. To preserve quotes in the argument.
     - If the argument is a text string with quotes:
       - use a pair of ' around the " to preserve the "
       - use a pair of " around the ' to preserve the '
       - use \ in front ' or " to preserve both quotes

Preserving Command Arguments (2 of 2)

3. To preserve shell special characters in an argument.
   - Shell special characters, called *metacharacters*, cause the shell to interpret the character as special instruction from you. For example, > on the command line means redirection, and not a literal > symbol.
   - Sometime an argument can contain a metacharacter, and you want to keep the literal meaning of that character.
   - To tell the shell to keep the literal meaning of metacharacters in the argument: Use \ in front of each metacharacter.
   - Metacharacters are:
     & ; | * ? ~ ! $ ^ # / \ ‘ “ ` [ ] ( ) { } < > space and newline (the enter key)

- Difference between single and double quotes:
  - In double quotes, the metacharacters $ and ! still have their special meaning, just like with no quotes.
  - In single quotes, the characters $ and ! have their literal meaning.
  - Generally, if you want all characters to keep their literal meaning, it’s safest to use single quotes around them.

Command Substitution

- Command substitution asks the shell to run a command and store the output in a string, which is then used as input to another command or to store in a variable.
- Format: command1 `command2`
  Or command1 $(command2)
  - The ` in the first format is the back quote, it is not a single quote.
  - In the second format, there is no space between the $ and the (
  - command1, command2 are commands with options and arguments, if necessary.
  - The output of command2 is stored as a string and then used as an input argument for command1.
- Another use: variable_name=`command`
  Or variable_name=$(command)
  - command can have options and arguments, if necessary.
  - The output of command is stored as a string in the variable var_name (more about variables in the Variable slides later).

Foreground and Background Jobs

- When the shell starts a process to run a utility or a shell script for you, the process is also called a job.
- The default way for the shell to run a job is to run it in the foreground, which means the shell doesn’t accept any other command from you while it’s waiting for the job to finish running.
- If the job that the shell is running in the foreground takes a long time to finish, it means you wait a long time before you can enter another command. When this happens, you want to run the job in the background instead.
- When the shell runs a job in the background, it starts the process to run the job, and then immediately prints the prompt to wait for your next command. When the background job finishes, the shell will print a message on screen to let you know that the job is finished.

Running a Job in the Background

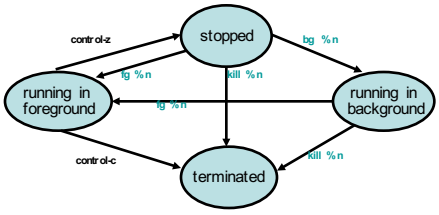
- To run a job in the background: command &
  - command can contain options and arguments and redirection.
  - if command contains options and arguments, the & must be at the end of the command line for the options and arguments to be in effect.
- Examples of multiple commands on the same command line:
  command1; command2 &
  command1 runs in foreground, command1 finishes, then command2 runs in the background.
  (command1; command2) &
  command1 runs in the background, finishes, then command2 runs in the background.
  command1 & command2 &
  command1 and command2 both run in the background, at the same time. Note that there is no ; to separate the 2 commands on the command line for this case.
  command1 | command2 | command3 &
  the whole pipe (starting with command1 and ending with command3) runs in the background.

Job Control (1 of 2)

- To see the status of a job: jobs
- jobs return the job number, command name (your request to start a job), and status of all jobs that are running by the shell.
- Status of a job:
  - running in foreground: currently running and the shell is not accepting your input.
  - running in background: currently running and the shell is waiting for your input.
  - stopped: the job is suspended and all job status are saved. When the job runs again it will continue from when it was stopped.
  - terminated: the process running the job is gone.
- If you don’t have a job running in the foreground, you can change the status of a job by using shell commands.
- If you have a job running in the foreground, you can change the status of a job by using interrupt signals.

Job Control (2 of 2)

- To switch from one job status to another:



- n is the job number. If there is only one job, the %n argument is not needed for the commands fg and bg. It is always needed for the command kill.
- The kill command can accept multiple job numbers.
- Switching the status does not change the input, output, or error destination.

alias

- Use alias to customize a command.
- 2 common reasons to customize a command:
  - change the command name to something you like.
  - have the command always run with the options you like.
- To create a command alias:
  alias aliasName='existingCommand'
  - aliasName is usually short.
  - existingCommand can contain options or arguments.
  - The single quotes around existingCommand is required if there are spaces or metacharacters in the command.
  - There is no space before or after the =
- To use the alias, type the aliasName on the command line. The existingCommand with its options and arguments will run.
- An alias created on the command line will exist only for the current session.
- To remove an alias: unalias aliasName
- To see all aliases, both the aliases you create and the system aliases: alias

Variables

- Variables are small blocks of memory you can claim, in order to store your data temporarily.
- When you create a variable you give it a name, and you access the memory by using that name. The variable name follows the same rule as a filename.
- To create a variable: variableName=data
  - data can be a number, a character, a word, or a text string. You can use quotes if there are metacharacters in the text string.
  - There is no space before or after the = sign.
- To change the data in a variable: variableName=newData
- To display the data stored in a variable: echo $variableName
  - The $ symbol tells the shell to fetch the data at the memory location with the name variableName. The $ symbol has this special meaning for the shell when it's not quoted or when it's double quoted. When the $ symbol is single quoted, it means the literal $ symbol.

Environment Variables (1 of 2)

- The shell keeps track of settings for your account. The settings that are specific to your account and affect your work session are called environment settings, and are saved in *environment variables.*
- To see all environment variables: env
- To see a single variable: echo $VAR_NAME
  - VAR_NAME is the variable name, always in uppercase.
  - the $ causes the shell to fetch the data from the variable.
- Some basic environment variables:
  - HOME: the absolute path of your home directory
  - PS1: your primary prompt
  - PS2: your secondary prompt
  - PATH: the paths of all the directories that the shell will search to find the executable that you ask it to run
  - SHELL: your default shell type

Environment Variables (2 of 2)

- You can change some environment variables to customize your work session.
- PS1: the default prompt displays your userID, the system name and your working directory. You can change it to display any text you like.
- PS2: the secondary prompt displays the > symbol. You can also change it to any text you like.
- HOME and SHELL: are set by system admin and you cannot change them. They are available for you to use, not to set.
- PATH: users often modify this variable to add more search paths. If you have executable files in your directories, appending these directory paths to PATH will make it possible for you to run the executable files by just typing their name on the command line. To append to PATH:
  PATH=$PATH:new_directory_path

Saving Your Setting Preferences

- So far we've learned different ways for you to customize your settings. However, because the customization is done on the command line, they only affect your current session.
- For the customization to last from session to session, you need to save them in special files.
- There are 2 hidden files in your home directory that the shell will run during initialization: .bashrc and .bash_profile. They are both text files. Any change to these files are optional.
- .bashrc: save your aliases and your messaging preference
  - Each alias is 1 line in the file: alias aliasName='command'
  - Have a line for messaging status: mesg n (or y)
- .bash_profile: save your environment variables and permission mask
  - Each environment variable change has a line: VAR=data and in the EXPORT line, add the name of the variable
  - Have a line for your default permission mask: umask value

Command History (1 of 2)

- The shell keeps track of your most recently run commands in an internal buffer.
- The size of this buffer is set by the environment variable HISTSIZE, which you can display and change.
- The default size is 1000, the last 1000 commands you typed in are saved.
- The buffer is circular, so after 1000 commands, the oldest (earliest) ones are overwritten by the newest commands.
- To display the saved commands in the buffer:
  history displays all the commands in the buffer.
  history n n is a number, displays the last n commands.
- Each command is displayed with a number that corresponds to the order that the command was run.
- Alternatively, each command in the buffer can be displayed one by one on the command line. To scroll through the buffer and display each command one by one: use the up arrow key.

Command History (2 of 2)

- To re-run the last command: !!
- To re-run any previous command, there are 2 ways:
  1. Use the up arrow key to scroll through the history buffer one command at a time.
     When you get to the command you want, hit enter to run the command.
  2. Use the history utility to see the last n commands along with their associated numbers.
     Then use: !num
     where num is the number associated with the command.
- To modify a previous command before running it:
  - Scroll through the buffer to find the command.
  - Retype any change in the command.
  - Hit enter to run the new command.
  - This is useful when you have a long command to modify.

Shell Types

- The shell is your interface to the system, but internally it is a program that runs when you successfully log in, and terminates when you log out.
- There are 4 popular types of shells that Unix/Linux users run:
  - Bourne: named after its author, it's the oldest shell that came with Unix.
  - C Shell: came from UC Berkeley when Unix was distributed to all the universities. It is not compatible with the Bourne shell.
  - Korn: named after its author, it's built upon the Bourne shell so it is similar to the Bourne shell but has more features.
  - Bash: (for Bourne Again Shell) it's the default shell for Linux. It's built upon the Bourne shell also.
- Your default shell on voyager is the bash shell because you're running Linux. The environment variable to display your default shell is SHELL.

Changing Shells (1 of 2)

- The default shell is the shell that automatically runs when you first log in.
- System admin sets the default shell for you, and you need to ask them if you want to change your default shell.
- Within a working session, you can change your shell to any type you like:
  - To change to a Bourne shell: sh
  - To change to a C shell: csh
  - To change to a Korn shell: ksh
  - To change to a bash shell: bash
  - To terminate a shell: exit
- Each time you change shell, your current shell starts a new process to run the new shell.
  If you change shell 2 times from when you log in, then you have 3 processes running: your default shell and 2 other shell processes. To completely log out, you will need to type exit 3 times. Each exit gets you back to the previous shell.

Changing Shells (2 of 2)

- To see which shell you are currently running: echo $0
- The utilities you learn in this class are standard Linux utilities that will work the same way, no matter which shell you are running. This is because the utilities are not a part of the shell, the shell only interacts with the utilities when you request them.
- When you write shell scripts, however, it is important what shell you run. This is because shell scripts use syntax that are specific to a certain shell type. For example, a script written for the C shell, which uses C shell syntax, will not run on bash.

Next stop: Filters