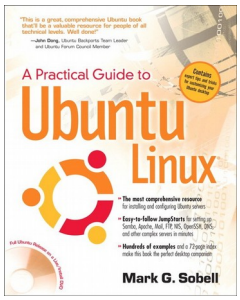
CIS 214
Unix and Linux Fundamentals



This version of the study guide references the 3rd edition of the textbook

A Practical Guide to Ubuntu Linux / Chapter 9 – The BASH Shell

- Introduction to the BASH Shell
    - The shell is where the incredible power of Unix opens up for you, once you understand it.
    - We will only study the BASH shell.
    - Please make sure you understand the following parts of chapter 9, listed here in sequence. Skip the other parts as indicated in the outline below.
- Shell Types
    - Know the names of the most important shells other than the BASH shell (which you do need to know in more detail as it is the default for most systems):
        - Bourne Shell (sh)
        - C Shell (csh)
        - Korn Shell (ksh)
        - Bourne Again Shell (bash)
- Startup Files for the BASH Shell (to initialize the shell environment)
    - First, for the ***login*** shell (the one opened at first login or with a terminal emulator application):
        - **/etc/profile**
          this is the first script executed - system wide
        - **~/.bash_profile**, **~/.bash_login**, **~/.profile**
          the first one found, in the sequence above, is executed from the user's home directory (note that your Ubuntu variants use ~/.profile by default)

    - Second, for ***interactive***, non-login shells (the subshells you run below the login shell):

        - **~/.bashrc**
          is executed by each interactive subshell launched from a login shell (note that Ubuntu variants also call this script from ~/.profile by default so most of the local configuration is done in this file – all in one place – for both login and interactive shells after an Ubuntu or LinuxMint install)

    - Third, for ***non-interactive***, non-login shells:

        - **BASH_ENV**, **ENV**
          when you run a script, utility or other application, the first variable found in the sequence above (in the invoking shell's environment) is used to determine the name of the file to use to initialize the non-interactive, non-login subshell's environment (in which the command will be executed)

- Source Command or Dot (**.**) Command

    - A dot (.) by itself is actually a command (when delimited by white space) – it loads and runs the script file (which is the following argument) in the current shell, not a subshell, and thus the script can affect the local environment variables – as scripts usually run in a subshell, they cannot normally affect the environment of the parent shell
    - No need to worry about Table 9-1 on page 297, for now

- I/O Channels or Standard I/O Local File Handle Numbers

    - **0** - standard in **cin** in C++ **System.in** in Java
    - **1** - standard out **cout** in C++ **System.out** in Java
    - **2** - standard error **cerr** in C++ **System.err** in Java

- I/O Redirection

    Table 9-2 on page 299 is poorly written, just understand the following about shell redirection:

    - Understand how to use the <0, 1>, and 2> redirect symbols with the standard I/O channels to redirect standard I/O
    - Understand how to use the &1 and &2 “duplicate” file handle symbols with the standard I/O channels to merge both standard out and standard err into the same I/O stream without them conflicting with each other (bottom of page 298):

    - **not** command 1> file 2> file
    - **but** command 1> file 2>**&1**

- Running Shell Scripts

  Using the **chmod** command and **PATH** environment variable:

  - understand how scripts must be made executable and findable (by the shell) to be run as commands
  - know how to run scripts that are not executable
  - know how to compensate for missing directories in the PATH
  - know how the #! comment is used within a script to invoke the proper shell for the language in which the script was written...

    - **chmod a+x script**
      makes the file “script” executable (adds eXecutable property for All users)

    - **./script**
      runs a script in the working directory even if that directory isn't on the PATH (notice a relative path “./” is specified in front of the file “script”

    - **#!/bin/bash**
      at the beginning of the very first line always runs the script containing it in a BASH shell no matter what the current shell actually is that launches the script (this is called the “hash bang” line).

  Normally, the shell calls fork to create a subshell and then calls exec to replace the subshell with an executable binary. Since the script isn't a compiled binary, the exec fails before the script runs. The hash bang avoids the failing exec.

- Command Line Separators or Extenders

  - Multiple commands can be placed on one line (terminated by the Enter key) if they are separated by one of these special shell characters:

    - **;**
      to run multiple commands entered on one line in left to right sequence

    - |
      to run multiple commands specified on one line, all in parallel, and pipe the output from the command on the left to the command on the right

    - **&**
      to run one (or more) command(s) in the background (in parallel)

    - \
      placed at the end of a line to continue it on another without invoking any of the commands already entered (this escapes the Enter key)

- Command Line Grouping
    - This is not optional (as indicated in the textbook). See page 306:
        - **( )**
          used to organize multiple commands into command groups - commands in command groups run in the same shell and share the same standard input, standard output and standard error channels

      If one of the commands in a sequence doesn't read from standard in (as is the case with commands like ls or echo) then commands in front of it that may be sending data to commands after it (via a pipe) will have the pipe broken (as the intervening command won't pass the data along). Placing the sequence in a command group avoids this problem because the shared shell will pass the pipe to whichever is the first in the sequence that does read from standard input.

    - For example, though this command could be easily rewritten to avoid the problem (in this case), it's easy to see that:
        - **this will fail**: ls | echo “Here comes the data:” | cat
        - **this works**: ls | (echo “Here comes the data:” ; cat)

- Job Control
    - These commands and interrupt keys (signaling the keyboard monitor) control the multiprocessing of jobs managed by your shell:
        - **jobs**
          list local jobs
        - **^C**
          to abort the foreground job
        - **^Z**
          to suspend the foreground job (and leave it ***stopped*** in the background)
        - **bg**
          restart a suspended job in the background – changing it to ***running*** in the background
        - **fg**
          move a job in the background to the foreground, restarting it if necessary

- Directory Stack
    - These commands manage a list of directories on a directory stack, making it easy to retrace your navigation of the file system:
        - **dirs**
          to list the directories on the directory stack
        - **pushd**
          to change to a new directory and push the previous one onto the stack
        - **popd**
          to change to the last directory on the stack while removing it from the stack
- Shell Variables
    - Understand how to create, remove and change the values in shell variables:
        - **var=*value***
          remember, there can be no embedded white-space anywhere in the above syntax (it is usually best to place double quotes around the value)
        - v**ar="*value with embedded spaces*"**
          double-quotes allows embedded white-space in the value, but never allow spaces around the equal (=) operator
        - **echo $var**
          echo the contents of the variable in a standardized format (any sequences of white-space, including multiple spaces and/or tabs, are normalized to a single space, each)
        - **echo "$var"**
          echo the contents of the variable in the original format (the original white-space sequences are retained)
        - **echo '$var'**
          echo \$var
          both of the above echo the name of the variable (including the $ sign), literally, as a string of characters, not the contents of the indicated variable – single quotes are skipped over by the shell (left alone) and the backslash escapes the following character (avoiding its normal meaning)
        - **unset var**
          removes the shell variable

- Shell Variable Attributes
    - Understand how to change the properties of shell variables:
        - **declare -r var**
          readonly var
          both of the above make var read-only
        - **declare -x VAR**
          export VAR
          both of the above will export var to any new subshells launched from this shell – note that by Unix shell naming standards, all exported variables should be declared with all upper case letters (i.e., VAR, not var)
- Keyword Shell Variables
    - Know the meaning and use of the following keyword shell variables
        - HOME contains the logged-in user's home directory path
        - PATH contains the directory paths searched by the shell
        - PS1 contains the initial command prompt
        - PS2 contains the continuation prompt (for multi-line commands)
        - IFS Inter-Field Separator – this defaults to space, tab and newline (whitespace) but can be modified to temporarily switch to comma separated fields, etc.
- **Skip** the following sections of the textbook, for now (pp. 326 to 356, except p. 353):
    - **Special Characters** (except for those you already know)
    - **Processes**
    - **Executing a Command**
    - **History** (very useful, but won't be on any exam)
    - **Readline Library**
    - **Aliases**
    - **Functions**
    - **Controlling bash: Features and Options**
    - **Processing the Command Line**
- But **do** consider Shell Features and Options (page 353)
    - set -o option i.e, to enable the option
    - set +o option i.e, to disable the option
    - Some options you should know: (**history**, **noclobber**)

- Processing the Command Line – skip ahead to read only the subsections on:
    - Brace Expansion i.e., ls file_{one,two,three}.txt
    - Parameter and Variable Expansion i.e., $var
    - Arithmetic Expansion i.e., $((arithmetic-expression))
    - Command Substitution i.e., $(command)