Debian Administration

# Session 2

Sunday, December 7, 2013

## Part I: Working on the Command Line

*Candidates should be able to interact with shells and commands using the command line. The Objective assumes the bash shell.*

### Key Knowledge Areas

* Use single shell commands and one line command sequences to perform basic tasks on the command line.
* Use and modify the shell environment including defining, referencing and exporting environment variables.
* Use and edit command history.
* Invoke commands inside and outside the defined path.

### Overview

A basic way to interact with a computer system is to use the command line. The shell interprets the instructions typed in at the keyboard. The shell prompt (ending with $ or # for user root) indicates that it is ready for user input.

The shell is also a programming environment, which can be used to perform automated tasks. Shell programs are called scripts.

|  |  |
| --- | --- |
| **Most Common Shells** | |
| The Bourne shell | /bin/sh |
| The Bourne again shell | /bin/bash |
| The Korn shell | /bin/ksh |
| The C shell | /bin/csh |
| Tom's C shell | /bin/tcsh |

Since the bash shell is one of the most widely used shells in the Linux world the LPI concentrates mainly on this shell.

***The interactive shell***

Shell commands are often of the form

**command [options] {arguments}.**

The the bash shell uses the echo command to print text to the screen.

|  |
| --- |
| $ echo “this is a short line” |

* **Full/Relative path**

The shell interprets the first ¨word¨ of any string given on the command line as a command. If the string is a full or relative path to an executable then the executable is started. If the first word has no ¨/¨ characters, then the shell will scan directories defined in the PATH variable and attempt to run the first command matching the string.

For example if the PATH variable only contains the directories **/bin** and **/usr/bin** then the command **xeyes** won't be found since it is stored in **/usr/X11R6/bin/xeyes** so the full path needs to be used

|  |
| --- |
| $ /usr/X11R6/bin/xeyes |

An alternative to typing the full path to an executable is to use a **relative** path. For example, if the user is in the directory where the **xeyes** program is stored then one can type

|  |
| --- |
| $ ./xeyes |

The '.' in the above command means the current working directory. Since you are already in the current directory the command could also simply be written as: (only if “.” is in your search path)

|  |
| --- |
| $ xeyes |

***Shell Variables***

Shell variables are similar to variables used in any computing language. Variable names are limited to alphanumeric characters. For example CREDIT=300 simply assigns the value 300 to the variable named CREDIT.

|  |  |
| --- | --- |
| 1. initialise a variable: | Variable-Name=value (no spaces !!) |
| 2. reference a variable: | $Variable-Name |

|  |
| --- |
| *CREDIT=300*  *echo $CREDIT* |

The value of a variable can be removed with the **unset** command.

**Export, Set and Env**

There are two types of variables: local and exported.

Local variables will be accessible only to the current shell. On the other hand, exported variables are accessible by both the shell and any child process started from that shell.

The commands **set** and **env** are used to list defined variables

|  |  |
| --- | --- |
| **The set and env commands** | |
| **set** | Lists all variables |
| **env** | Lists all exported variables |

A global variable is global in the sense that any child process can reference it. The example below exports the variable credit and then checks to see if it has been exported as expected.

|  |
| --- |
| *export CREDIT* |
| *env | grep CREDIT* |

*List of common predefined variables*

| **PREDEFINED VARIABLES** | **MEANING** |
| --- | --- |
| DISPLAY | Used by X to identify where to run a client application |
| HISTFILE | Path to the user's .bash\_history file |
| HOME | The path to the user's home |
| LOGNAME | The name used by the user to log in |
| PATH | List of directories searched by the shell for programs to be executed when a command is entered without a path. |
| PWD | The current working directory |
| SHELL | The shell used (bash in most Linux distributions) |
| TERM | The current terminal emulation |

**Special variables**

The next few variables are related to process management.

|  |
| --- |
| $! represents the PID value of the last child process  $$ represents the PID of the running shell  $? is 0 if the last command was executed successfully and non-zero otherwise |

**Metacharacters and Quotes**

Metacharacters are characters that have special meaning for the shell. They are mainly used for file globbing, that is to match several files or directory names using a minimum of letters. The input (<), output (>) and pipe (|) characters are also special characters as well as the dollar ($) sign used for variables. We will not list them here but note that these characters are seldom used to name regular files.

***Wildcards***

● The **\*** wildcard can replace any number of characters.

|  |
| --- |
| $ ls /usr/bin/b\* |

lists all programs starting with a 'b'

● The ? wildcard replaces any one character.

|  |
| --- |
| $ ls /usr/bin/?b\* |

lists all programs having a 'b' as the second letter

● [ ] is used to define a range of values.

|  |
| --- |
| $ ls a[0-9]  $ ls [!Aa]\* |

First line lists all files starting with an 'a' and have a digit in second position.

The second line lists all files that don't start with and 'a' or an 'A'

● {string1,string2}**;** although not a file naming wildcard, it can be used to generate a list of names that have a common stem.

|  |
| --- |
| $ mkdir {mon, tues, wednes} day |

**Quotes and escape codes**

The special meaning of metacharacters can be cancelled by *escape characters*, which are also metacharacters.

The backslash (\)is called the **escape character** and cancels the meaning of the following character, forcing the shell to interpret it literally.

The single quotes (' ')cancel the meaning of all metacharacters except the backslash.

The double quotes(" ") are the weakest quotes but cancel most of the special meaning of the enclosed characters except the pipe (|), the backslash (**\)** and a variable ($var).

**The Back Tick**

Back quotes **``** will execute a command enclosed and substitute the output back on the command line. The next example defines the variable TIME using the **date** command.

|  |
| --- |
| $ TIME="Today's date is `date +%a:%d:%b`”  echo $TIME  Today's date is Sun:15:Jul |

Another way of executing commands (similar to the back ticks) is to use **$()**. This will execute the enclosed command and treat it as a variable.

|  |
| --- |
| $ TIME=$(date) |

***The Command History***

To view the list of previously typed commands you can use the **bash** built-in command history.

|  |
| --- |
| $ history   1. ls 2. grep 500 /etc/passwd |

This has listed all the cached commands as well as the commands saved in **~/.bash\_history**. When a user exits the shell cached commands are saved to **~/.bash\_history**.

You can recall commands by using the Up-arrow and Down-arrow on your keyboard. There are also emacs key bindings that enable you to execute and even edit these lines.

|  |  |
| --- | --- |
| **Emacs Key Bindings for Editing the Command History** | |
| Ctrl+P | Previous line (same as Up-arrow) |
| Ctrl+n | Next line (same as Down-arrow) |
| Ctrl+b | Go back one character on the line (same as Left-Arrow) |
| Ctrl+f | Go forward one character on the line (Same as Right-Arrow) |
| Ctrl+a | Go to the beginning of the line (Same as <Home>) |
| Ctrl+e | Go to the end of the line (Same as <End>) |

The **bang** (**!)** key can be used to rerun a command.

Example

|  |
| --- |
| !x executes the latest command in the history list starting with an 'x'  !2 runs command number 2 from the **history** output  !-2 runs the command before last  !! runs the last command  ^string1^string2 run previous command and replace string1 by string2 |

**Other Commands**

**Aliases**

You can create aliases for commands needing many arguments. The format to create an alias is

|  |
| --- |
| $ alias myprog='command [options]{arguments}' |

By typing alias alone at the command line you will get a list of currently defined aliases.

**Command completion**

By pressing **TAB,** the shell will complete the commands you have started typing in.

*Compound commands*

|  |  |
| --- | --- |
| command1; command2; command3 | The three commands are run in sequence regardless of the success of the previous command |
| command1 && command2 && command3 | Each command will execute only if the previous exit code is 0 (success) |
| command1 || comand2 || command3 | The next command will execute only if the previous exit code is not 0 (failure) |

**The ''exec” command**

This command is not a binary but rather is part of the shell. It is used to start other commands. Ordinarily if a command is executed, a sub-process is started. If the exec command is used to initiate the new program, it reoccupies the process used to start it. It replaces the current shell (in a script or the interactive shell).

When the new command terminates, control is not passed back to the calling shell, but returns to the process that called the shell used to make the exec call.

|  |
| --- |
| $ echo $$  414  $ bash  $ echo $$  455  $ echo hello  hello  $ echo $$  455  $ exec echo hello  hello  $ echo $$  414 |

The above shows control falling back to the second shell (process 455) after a straight forward echo and the first shell (process 414) using an exec.

***Manpages and the whatis database***

| **The manpages are organised in specific topics** | |
| --- | --- |
| NAME | the name of the item followed by a short one line description. |
| SYNOPSYS | the syntax for the command |
| DESCRIPTION | a longer description |
| OPTIONS | a review of all possible options and their function |
| FILES | files that are related to the current item (configuration files etc) |
| SEE ALSO | other manpages related to the current topic |

These are the main topic sections one can expect to find in a manpage.

The **whatis** database stores the NAME section of all the manpages on the system. This is updated regularly through a daily **cron**. The **whatis** database has the following two entries:

|  |
| --- |
| **name(key) – one line description** |

The syntax for **whatis** is:

whatis <string>

The output is the full NAME section of the manpages where *string* matched *named(key)*

One can also use the mancommand to query the whatis database. The syntax is

**man -k <string>**

This command is similar to **apropos**. Unlike whatis this will query both the “name” and the “one line description” entries of the database. If the string matches a word in any of these fields the above query will return the full *NAME* section.

Example: (the matching string has been highlighted)

|  |
| --- |
| *whatis* ***lilo*** |
| **lilo** (8) - install boot loader |
| **lilo**.conf [lilo] (5) - configuration file for lilo |

|  |
| --- |
| *man -k* ***lilo*** |
| grubby (8) - command line tool for configuring grub, **lilo**, and e**lilo** |
| **lilo** (8) - install boot loader |
| **lilo**.conf [lilo] (5) - configuration file for **lilo** |

The filesystem hierachy standard, a recommended layout for Linux filesystems, recommends manpages to be kept in **/usr/share/man**. However additional locations can be searched using the MANPATH environment variable set in **/etc/man.config**. Each directory is further divided into subdirectories corresponding to manpage sections.

| ***Manpage Sections*** | |
| --- | --- |
| Section 1 | Information on executables |
| Section 2 | System calls, e.g mkdir(2) |
| Section 3 | Library calls, e.g stdio(3) |
| Section 4 | Devices (files in /dev) |
| Section 5 | Configuration files and formats |
| Section 6 | Games |
| Section 7 | Macro packages |
| Section 8 | Administration commands |
| Section 9 | Kernel routines |

Sometimes manpages with the same name are present in more than one section.

To access a specific section *N* one has to enter:

**man *N* command**

Examples:

| $ man mkdir  $ man 2 mkdir |
| --- |

|  |
| --- |
| $ man crontab  $ man 5 crontab |

**file**

file is used to try and detect what type a particular file is.

For example

|  |
| --- |
| $ file picture.png  picture.png: PNG image, 179 x 179, 8-bit/color RGBA, non-interlaced |

The utility will identify files that have been incorrectly named so if picture.png had been named readme.txt the command “file readme.txt” would still identify the file as a png file.

**uname**

The uname command prints information relating to the kernel version, machine name, processor type and node name. It is most commonly used to identify which version of the kernel a machine is running.

|  |
| --- |
| $ uname -r |

Prints the currently running kernel's version number.

**pwd**

This is a command which simply prints out the current working directory for the shell.

Used files, terms and utilities:

* bash
* echo
* env
* exec
* export
* pwd
* set
* unset
* man
* uname
* history

## Part II: Process text streams using filters

*Candidates should be able to apply filters to text streams.*

### Key Knowledge Areas

* Send text files and output streams through text utility filters to modify the output using standard UNIX commands found in the GNU textutils package.

***Text Processing Utilities***

Linux has a rich assortment of utilities and tools for processing and manipulating text files. In this section we cover some of them.

**cat** - cat is short for concatenate and is a Linux command used to write the contents of a file to standard output. Cat is usually used in combination with other command to perform manipulation of the file or if you wish to quickly get an idea of the contents of a file. The simplest format of the command Is:

|  |
| --- |
| # cat /etc/aliases |

Cat can take several parameters; the most commonly used being -n and -b which output line numbers on all lines and non-empty lines only respectively.

**head** and **tail** - The utilities headand tail are often used to examine log files. By default they output 10 lines of text. Here are the main usages.

List 20 first lines of **/var/log/messages**:

|  |
| --- |
| # head -n 20 /var/log/messages  # head -20 /var/log/messages |

List 20 last lines of **/etc/aliases**:

|  |
| --- |
| # tail -20 /etc/aliases |

The **tail** utility has an added option that allows one to list the end of a text starting at a given line.

List text starting at line 25 in **/var/log/messages**:

|  |
| --- |
| # tail +25 /etc/log/messages |

Finally **tail** can continuously read a file using the **-f** option. This is most useful when you are examining live log files for example.

**wc** -The **wc** utility counts the number of *bytes*, *words*, and *lines* in files. Several options allow you to control **wc**'s output.

*Options for* ***wc***

|  |  |
| --- | --- |
| **-l** | count number of lines |
| **-w** | count number of words |
| **-c or -m** | count number of bytes or characters |

**nl -** The nl utility has the same output as **cat -b**

Number all lines including blanks

|  |
| --- |
| # nl -ba /etc/lilo.conf |

Number only lines with text

|  |
| --- |
| # nl -bt /etc/lilo.conf |

**expand/unexpand** - The expand command is used to replace TABs with spaces. One can also use **unexpand** for the reverse operations.

**od** There are a number of tools available for this. The most common ones are od (octal dump) and hexdump.

**split** - splitting files - The split tool can split a file into smaller files using criteria such as size or number of lines. For example we can spilt */etc/passwd* into smaller files containing 5 lines each

|  |
| --- |
| # split -l 5 /etc/passwd |

This will create files called *xaa*, *xab*, *xac*, *xad* ... each file contains at least 5 lines. It is possible to give a more meaningful prefix name for the files (other than '*x'*) such as '*passwd-5.*' on the command line

|  |
| --- |
| # split -l 5 /etc/passwd passwd-5 |

This has created files identical to the ones above (*aa*, *xab*, *xac*, *xad* ...) but the names are now *passwd-5aa*, *passwd-5ab*, *passwd-5ac*, *passwd-5ad* …

**Erasing consecutive duplicate lines**

The uniq tool will send to stdout only one copy of consecutive identical lines.

Consider the following example:

|  |
| --- |
| # uniq > /tmp/UNIQUE  line 1  line 2  line 2  line 3  line 3  line 3  line 1  ^D |

The file */tmp/UNIQUE* has the following content:

|  |
| --- |
| # cat /tmp/UNIQUE  line 1  line 2  line 3  line 1 |

***NOTE:***

*From the example above we see that when using* ***uniq*** *non consecutive identical lines are still printed to STDOUT. Usually the output is sorted first so that identical lines all appear together.*

|  |
| --- |
| # sort | uniq > /tmp/UNIQUE |

**cut** The cut utilility can extract a range of characters or fields from each line of a text. The **–c** option is used to cut based on character positions.

Syntax:

**cut {range1,range2}**

Example

# cut –c5-10,15- /etc/password

The example above outputs characters 5 to 10 and 15 to end of line for each line in /etc/password. One can specify the field delimiter (a space, a commas etc ...) of a file as well as the fields to output. These options are set with the **–d** and **–f** flags respectively.

Syntax:

**{delimiter} -f {fields}**

Example:

***# cut -d: -f 1,7 --output-delimiter=" " /etc/passwd***

This outputs fields 1 and 7 of /etc/passwd delimited with a space. The default *output-delimiter* is the same as the original input delimiter. The **--output-delimiter** option allows you to change this.

**paste/join** - The easiest utility is paste**,** which concatenates two files next to each other.

Syntax:

**paste text1 text2**

With join you can further specify which fields you are considering.

Syntax:

**join -j1 {field\_num} -j2{field\_num} text1 text2 or**

**join -1 {field\_num} -2{field\_num} text1 text2**

Text is sent to stdout only if the specified fields match. Comparison is done one line at a time and as soon as no match is made the process is stopped even if more matches exist at the end of the file.

**sort** - By default, sort will arrange a text in alphabetical order. To perform a numerical sort use the **-n** option.

***Formatting output with fmt and pr***

**fmt** is a simple text formatter that reformats text into lines of a specified length.

You can modify the number of characters per line of output using **fmt**. By default fmt will concatenate lines and output 75 character lines.

***fmt*** options

| **-w** number of characters per line  **-s** split long lines but do not refill  **-u** place one space between each word and two spaces at the end of a sentence |
| --- |

Long files can be paginated to fit a given size of paper with the pr utility. Text is broken into pages of a specified length and page headers are added. One can control the page length (default is 66 lines) and page width (default 72 characters) as well as the number of columns.

**pr** can also produce multi-column output.

When outputting text to multiple columns each column will be evenly truncated across the defined page width. This means that characters are dropped unless the original text is edited to avoid this.

**tr** The tr utility translates one set of characters into another.

Example changing uppercase letters into lowercase

| tr 'A-B' 'a-b' < file.txt |
| --- |

Replacing delimiters in **/etc/passwd**:

|  |
| --- |
| # tr ':' ' ' < /etc/passwd |

**NOTE**: **tr** has only **two arguments**!.

**sed** sed stands for stream editor and is used to manipulate text stream **tr** will not read from a file, it only reads standard input. It is most commonly used to transform text input generated by other commands in bash scripts. sed is a complex tool that can take some time to master. It's most common use case is to find and replace text in an input stream. Sed's output is written to standard out, with the original file left untouched, and needs to be redirected to a file to make the changes permanent.

The command:

|  |
| --- |
| # sed ‘s/linux/Linux/g‘ readme.txt > ReadMe.txt |

will replace every occurrence of the word linux with Linux in the readme.txt file. The g at the end of the command is used to make the replacement global so sed will process the entire line and not stop at the first occurrence of the word linux. For more informaiton on sed refer to section 103.7

Used files, terms and utilities:

* cat
* cut
* expand
* fmt
* head
* od
* join
* nl
* paste
* pr
* sed
* sort
* split
* tail
* tr
* unexpand
* uniq
* wc