Debian Administration

# Session 3

Monday, December 9, 2013

## Part I: Perform file management

*Candidates should be able to use Linux commands to manage files and directories.*

### Key Knowledge Areas

* Copy, move and remove files and directories individually.
* Copy multiple files and directories recursively.
* Remove files and directories recursively.
* Use simple and advanced wildcard specifications in commands.
* Using find to locate and act on files based on type, size, or time.
* Usage of tar, cpio and dd.

***Moving Around the Filesystem***

Absolute and relative paths

A directory or a file can be accessed by giving its full pathname, starting at the root (/) or its relative path, starting from the current directory.

*Absolute path*: independent of the user's current directory, starts with /

Relative path: depends on where the user is, doesn't start with /

As in any structured filesystem there are a number of utilities that can help you navigate through the system.

pwd: Gives your actual position as an absolute path.

cd: The 'change directory' command

**ls**: List the contents of a directory.

The command can take several parameters the most common of which are:

**-l** – use the long listing format,

**-a** – list all files and directories including hidden files and directories,

**-h** – show file sizes in human readable format, ie. Formatted for easy reading

**-d** – list directories only and does not list their contents.

**Finding Files and Directories**

We will describe the **find**, **which**, **whereis** and **locate** utilities.

**find**

Syntax:

find <DIRECTORY> <CRITERIA> [-exec <COMMAND> {} \;]

The *DIRECTORY* argument tells **find** where to start searching and *CRITERIA* can be a combination of serial selection criteria, including the name of a file or directory we are looking for.

*Examples:*

|  |
| --- |
| # find /usr/X11R6/bin -name ¨x\*¨.  # find / -user 502 |

The names of matching files are listed to standard outpu. Alternatively, a specific operation can be performed on each file found. For example to delete the file, or change the permission. The **find** tool has the built-in option **–exec** which allows you to do that. For example, remove all files belonging to user 502:

|  |
| --- |
| # find / -type f -user 502 –exec rm –f {} \; |

|  |  |
| --- | --- |
| ***Common criteria switches for find*** | |
| -type | specify the type of file |
| -name | name of the file (can include wildcards) |
| -user | user owner |
| -atime, ctime, mtime | access, creation and modified times (multiples of 24 hrs) |
| -amin, cmin, mmin | access, creation and modified times (multiples of 1 min) |
| -newer *FILE* | files newer than *FILE* |

**Handling directories**

*Creating a directory*

When making a directory you can set the permission mode with the **-m** option. Another useful option is **-p** which creates all subdirectories automatically as needed.

Example:

|  |
| --- |
| # mkdir –p docs/programs/versions |

*Removing directories:*

To remove a directory use either **rmdir** or rm **-r**. **rmdir** will only remove empty directories. Specify **-f** to force the deletion of files on which you do not have write permission..

**Notice**: rm –rf /dir1/\* removes all files and subdirectories leaving dir1 empty

rm –rf /dir1/ removes all files and subdirectories including dir1

Using cp and mv

**cp**

Syntax:

**cp [options] *file1 file2***

**cp [options] *files directory***

It is important to notice that **cp** *file1 file2* makes a new copy of *file1* and leaves *file1* unchanged. You can also copy several files to a directory, using a list or wildcards. The following table lists the most used options.

|  |  |
| --- | --- |
| ***Most common options for cp*** | |
| -d | do not follow symbolic link (when used with -R) |
| -f | force |
| -i | interactive, prompt before overwrite |
| -p | preserve file attributes |
| -r | recursively copy directories |

**Note**: cp –r /mydir/\* /dir2/ will copy all files and subdirectories omitting mydir

cp –r /mydir/ /dir2/ will copy all files and subdirectories including mydir

**mv**

Syntax:

**mv [options] *oldname newname***

**mv [options] source destination**

**mv [options] source directory**

The **mv** command can both *move* and *rename* files and directories. If *oldname* is a file and *newname* is a directory then the file *oldname* is moved to that directory.

If the source and destination are on the same filesystem, then the file isn't copied but the the link is simply moved to the new location. Most common options are **-f** force overwrite and **-i** query interactively.

touch and dd

**touch**

Another way of creating or modifying a file is to use **touch**.

Syntax: touch {options} *file(s)*

If *file* doesn't exist it is created. You can also update the access time of a file to the current time using the **-a** option, **-m** changes the modification time and **-r** is used to apply the time attributes of another file.

Example:

touch file1.txt file2.txt creates new files

touch myfile -r /etc/lilo.conf myfile gets the time attributes of lilo.conf

**dd**

This command copies a file with a changeable I/O block size. It can also be used to perform conversions (similar to **tr**). Main options are **if=** (input file) **of=** (output file) **conv=** (conversion)

The conversion switch can be: lcase ucase ascii

Example:

|  |
| --- |
| # dd if=/dev/sda1 of=/dev/sda2 |

Notice that unlike **cp** the **dd** tool will copy portions of a device and preserve the underlying filesystem. On the other hand **cp** only deals with the data and will transfer it from one filesystem to another:

***File Archiving and Compression***

Linux has several utilities for compressing and archiving files. Some of these tools have their origins in tape archiving and backup solutions and the parameters and names reflect this.

**tar**

The tar (tape archive) command is used to archive directories and optionally compress the archive. Originally the tar command was used to archive to tape but can now also archive to disk, which is its most common use. An archive is created as follows:

|  |
| --- |
| # tar – cvjf backup.tar.bz /home/user1 |

This will create a bzip compressed archive of user1's home directory. The options provided to tar are:

* c – create the archive,
* v – show verbose output during archive creation,
* j – compress the archive with bzip compression,alternatively you could stipulate z which would use gzip compression
* f – the name of the file to created, in this case backup.tar,bz

To extract the backup.tar.bz archive you would use the following command

|  |
| --- |
| # tar -xvjf backup.tar.bz |

This would extract the archive to the current directory. The command line parameters are mostly the same as the above example except for the -x (exctract) parameter which replaces -c (create) To list the contents of an archive without extracting it you would use the -t parameter:

|  |
| --- |
| # tar -tf backup.tar.bz |

**cpio**

cpio is an older archive utility that does not support compression natively. cpio stands for copy in/out. Although cpio has been largely superseded by tar it is still used in Linux. In particular the initrd image file is a cpio archive. cpio expects a list of files to archive on standard input and so is usually used in combination with the **find** or **ls** command.

|  |
| --- |
| $ ls | cpio -ov > backup.cpio  $ find / | cpio -ov > backup.cpio |

The above two commands create an archive using cpio. The v parameters tells cpio to provide verbose output during archive creation. To extract an archive you will use a command such as:

|  |
| --- |
| $ cpio -iv < backup.cpio |

This will extract the cpio archive to the current directory. One of the tricky things to remember with cpio is its parameters. With archiving we usually talk of archiving and extracting which suggest the parameters -c for creating and -x or -e for extraction. The easiest way to remember that -o is for creating and -i is for extraction is to remember cpio stands for copy in/ copy out. You will create an archive by copying out from the filesystem and extract an archive by copying in from an archive.

**gzip/gunzip**

gzip is used to compress files using Lempel-Ziv coding. As with most Linux commands it can take a plethora of parameters but is most commonly used as follows:

|  |
| --- |
| $ gzip largefile.txt |

By default gzip creates an output file with the same name as the input file but with the extension .gz added. The above command would create a compressed file with the name largefile.txt.gz. To uncompress he file you would run the command

|  |
| --- |
| $ gunzip largefile.txt.gz **or**  $ gzip -d largefile.txt.gz |

**bzip/bzip2**

bzip compresses files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding, which is considered more efficient than the Lempel-Ziv file. Its most commonly used format for compressing a file take a number from 1 – 9 as a parameter. This number is used to tell bzip2 to use the least efficient but fastest compression block size for 1 and use the most efficient but slowest compression block size for 9. If no number is specified the default value of 9 is used.

|  |
| --- |
| $ bzip2 -9 largefile.txt |

It will create the smallest file with the original file name and .bz appended. In our case the file would be largefile.txt.bz. To uncompress the file you would use

|  |
| --- |
| $ bzip2 -d largefile.txt.bz2 **or**  $ bunzip2 largefile.txt.bz2 |

Used files, terms and utilities:

* cp
* find
* mkdir
* mv
* ls
* rm
* rmdir
* touch
* tar
* cpio
* dd
* file
* gzip
* gunzip
* bzip2
* file globbing

## Part II: Streams, Pipes and Re-directs

*Candidates should be able to redirect streams and connect them in order to efficiently process textual data. Tasks include redirecting standard input, standard output and standard error, piping the output of one command to the input of another command, using the output of one command as an argument for another command and sending output to both standard output and a file.*

### Key Knowledge Areas

* Redirecting standard input, standard output and standard error.
* Pipe the output of one command to the input of another command.
* Use the output of one command as arguments to another command.
* Send output to both stdout and a file.

***Input, Output, Redirection***

UNIX processes use streams to get input, (standard input stream) send output to, (standard output stream) and a stream to send error messages to (the standard error stream). These streams can be redirected for any process. In most cases standard input (stdin) is the keyboard, and the two output descriptors, standard out (stdout) and standard error (stderr),go to the screen. Sometimes it is convenient to redirect these standard streams so that the process received input from a file and/or sends output to file.

|  |  |
| --- | --- |
| ***Numerical values for stdin, stderr and stdout*** | |
| stdin | 0 |
| stdout | 1 |
| stderr | 2 |

When redirecting or interacting with these streams we refer to them by their numerical values.

Standard Out Redirection

To redirect standard output from the screen to a file for example you use the “>” symbol.

For example

|  |
| --- |
| $ find / -iname \*.txt > textfiles.txt |

This will run the **find** utility and output the result to the textfiles*.txt* file. No output is visible on the screen.. The textfiles*.txt* file will be created first if it doesn’t exist and overwritten if not. To append to a file rather than create a new one the ‘>>’ operator can be used.

Standard Error Redirect

Standard error redirect uses the same format as for standard input redirect but you need to specify that the stderr stream and not stdout is to be redirected. This is done via placing the stderr stream id before the redirect symbol.

|  |
| --- |
| $ myapp 2> error.txt |

As per the stdout example above this will create a new file. To append to an existing file you would use “2>>” to redirect standard error.

**Redirect stdout and stderr**

To redirect stdout and stderr at the same time you would use the “&>” or “&>>” operator. This will direct both standard out and standard error to the same file.

Standard In Redirection

To have the process read input from a file rather than get its input from the keyboard you would use the “**<**' symbol as in the example below:

|  |
| --- |
| $ mysql -u root -p < createtable.sql |

Here the mysql command line interface is told to take its standard input from a file called createtable.sql rather than read input from the keyboard. This file would contain SQL statements necessary to create a table for example.

***Piped Commands***

The pipe command is used to redirect the standard output from one process to the standard input of another.

**program1 | program2**

Pipes are represented by the “|” symbol. The data stream goes from the left to the right. The next figure illustrates how the stdout for one process is redirected to the stdin for another process.

|  |
| --- |
| # ls-l | less |

**The tee Command**

***command* |** tee*FILENAME*

This command is used after a pipe and takes a filename as an argument. The standard output from the previous command is then sent to the file given as an argument but **tee** also lets the stream through to **stdout**. The **stdout** has been duplicated in this way.

**xargs**

This tool is often thought of as a companion tool to **find**. In fact **xargs** will process each line of standard output as an *argument* for another tool. We could use **xargs** to delete all files belonging to a user with:

|  |
| --- |
| $ find / -type f -user 502 | xargs rm –f |

If the list of filenames is very long, xargs will split it into pieces and invoke the rm command several times, one for each piece. This is sometimes useful if the argument list would otherwise be too long to handle.

|  |
| --- |
| $ ls |xargs rm -f |

Used files, terms and utilities:

* tee
* xargs

## Part III: Create, Monitor and Kill Processes

*Candidates should be able to perform basic process management.*

### Key Knowledge Areas

* Run jobs in the foreground and background.
* Signal a program to continue running after logout.
* Monitor active processes.
* Select and sort processes for display.
* Send signals to processes.

When the shell runs a command, it normally waits and will not prompt for further input until that command has completed. The command is said to run in the foreground.

When a program is running in the foreground it is possible to recover the shell prompt but only by interrupting the program for while. The interruption signal is Ctrl Z.

***Starting and Stopping Jobs***

A process started from a shell is also called a *job.* Once the job receives the **^Z** signal it is stopped and the shell prompt is recovered. To restart the program in the background simply type: **bg**.

Example

|  |
| --- |
| $ xclock  **xclock running in forground, shell prompt lost**  [1]+ Stopped xclock **xclock received ^Z signal**  $ bg **shell prompt recovered, issue the bg command**  [1]+ xclock & **xclock is running in the background** |

Notice the [1]+ symbol above. The integer is the process' *job number,* which it can be referred to as.

The '+' sign indicates the last modified process. A '-' sign would indicate the second last modified process. One can start a process in the background by appending a **&** to the command.

|  |
| --- |
| $ xclock&  [1] 6213 |

The numbers reported here are the job numbers (in square brackets), and the process ID.

**Listing jobs**

The jobs utility lists all running processes started from the current shell. The *job number*, the job's state (running/stopped), as well as the two last modified processes, will be listed.

| ***Output for jobs*** |
| --- |
| [1]- Stopped xclock  [2] Running xman &  [3]+ Stopped xload |

**The job number**

One can conveniently stop and start a selection of jobs using the *job number*. This is achieved with the **fg** command.

*Calling job 2 to the foreground and killing job 1*

| fg 2 or  fg %2 or  fg %?xma | kill –9 %1 |
| --- | --- |

**Avoiding HUP with nohup**

There is a program called **nohup** which acts as a parent process independently from the user’s session. When a user logs off, the system sends a HUP signal to all processes owned by that process group. For example, to avoid this HUP signal a script called **bigbang** which attempts to calculate the age of the Universe should be started like this:

|  |
| --- |
| $ nohup bigbang & |

***Viewing Running Processes***

Processes have a unique Process ID the PID. This number can be used to modify a process' priority or to stop it. A process is any *running* executable. If process\_2 has been spawned by process\_1, it is called a *child* process. The spawning process\_1 is called the *parent* process.

The pstreecommand gives a good illustration of *parent* and *child* process hierarchy.

*Part of the pstree output figure below:*

| bash(1046)---xinit(1085)-+-X(1086)  `-xfwm(1094)-+-xfce(1100)---xterm(1111)---bash(1113)-+-pstree(1180)  | |-soffice.bin(1139)---soffice.bin(1152)-+  -soffice.bin(1153)  | | |-soffice.bin(1154)  | | |-soffice.bin(1155)  | | |-soffice.bin(1156)  | | `-soffice.bin(1157)  | `-xclock(1138)  |-xfgnome(1109)  |-xfpager(1108)  |-xfsound(1107)  `-xscreensaver(1098) |
| --- |

In the above figure all the process' PIDs are shown; these are clearly incremental. The most common used options are **-p** to display PIDs and **-h** to highlight a users processes only.

A more direct way to determine which processes are running is to use ps. Most users learn a favourite combination of options which work for most situations.

Here are three such options:

**ps** **ux** all processes run by the user

**ps T** processes run under the current terminal by the user

**ps aux** all processes on the system

It is recommended you read the **ps manpage** and choose your own best options!

| ***ps*** *accommodates UNIX-style and BSD-style arguments* |
| --- |
| usage: ps -[Unix98 options]  ps [BSD-style options]  ps --[GNU-style long options]  ps --help for a command summary |

| ***Summary of options*** |
| --- |
| **-a** show all processes for the current user linked to a tty (*except* the session leader)  **-e** or **-A** show all processes  **-f** gives the PPID (Parent Process ID) and the STIME (Start Time)  **-l** is similar to **-f** and displays a long list  **a** show all processes linked to a tty, including other users  **x** show all processes without a controlling tty as well |

***Sending Signals To Processes***

The **kill** command can be used to send *signals* to processes. There are 63 signals available. The default signal terminates a process and is called SIGTERM with value 15.

**kill**

*Syntax*

kill SIGNAL process\_PID

Unless you are root, you can only send signals to processes that you own.

Every process can choose whether or not to catch a signal except for the SIGKILL which is dealt with by the kernel. Most daemons use SIGHUP to mean “re-read configuration file”.

| ***Most Common Signals*** |
| --- |
| 1 or SIGHUP hangup or disconnect the process  2 or SIGINT same as Ctrl+C interrupt  3 or SIGQUIT quit  9 or SIGKILL kill the process through a kernel call  15 or SIGTERM terminate a process 'nicely'. This is the DEFAULT signal. |

One can also stop processes without knowing the process' PID using **killall**.

**killall**

*Syntax*

| killall SIGNAL process\_NAME |
| --- |

Used files, terms and utilities:

* &
* bg
* fg
* jobs
* kill
* nohup
* ps
* top
* free
* uptime
* killall