UNIX commands PART 2

- By Meenakshi

For more guides on software Testing visit: http://www.softwaretestinghelp.com

This article will guide you on some advanced Unix commands. Also comprehensive guide is provided for each command for its use. The topics covered here includes advanced Unix commands with examples, Difference between symbolic and hard links and Unix variables guide.

The `kill`

is a command used to send simple messages to processes running on the system. By default, the message sent is the "termination" signal, which requests that the process exit. But *kill* is something of a misnomer; the signal sent may have nothing to do with process killing. The `kill` command is a wrapper around the **`kill()`** system call, which sends signals to processes or process groups on the system, referenced by their numeric process IDs (PIDs) or process group IDs (PGIDs). `kill` is always provided as a standalone utility, but most shells have built-in `kill` commands that may slightly differ from it.

Kill – l

The above command gives the list of signals

There are many different signals that can be sent (see signal for a full list), although the signals that users are generally most interested in are SIGTERM and SIGKILL. The default signal sent is SIGTERM. Programs that handle this signal can do useful cleanup operations (such as saving configuration information to a file) before quitting. However, many programs do not implement a special handler for this signal, and so a default signal handler is called instead. Other times, even a process that has a special handler has gone awry in a way that prevents it from properly handling the signal.

All signals except for SIGKILL and SIGSTOP can be "intercepted" by the process, meaning that a special function can be called when the program receives those signals. However, SIGKILL and SIGSTOP are only seen by the host system's kernel, providing reliable ways of controlling the execution of processes. SIGKILL kills the process, and SIGSTOP pauses it until a SIGCONT is received.

Unix provides security mechanisms to prevent unauthorized users from killing other processes. Essentially, for a process to send a signal to another, the owner of the signaling process must be the same as the owner of the receiving process or be the superuser.

The available signals all have different names, and are mapped to certain numbers. It is important to note that the specific mapping between numbers and signals can vary between Unix implementations. SIGTERM is often numbered 15 while SIGKILL is often numbered 9.

Examples

A process can be sent a SIGTERM signal in three ways (the process ID is '1234' in this case):

```
kill 1234
```

```
kill -TERM 1234
```

```
kill -15 1234
```

The process can be sent a SIGKILL signal in two ways:

```
kill -KILL 1234
```

```
kill -9 1234
```

The `ln`

command is used on Unix-like systems to create links between files. Links allow more than one location to refer to the same information.

There are two types of links, both of which are created by `ln`:

symbolic links, which refer to a symbolic path indicating the abstract location of another file,

a **symbolic link** (often shortened to *symlink*) is a special type of file that serves as a reference to another file. Unlike a hard link, a symbolic link does not point directly to data, **but merely contains a symbolic path which is used to identify a hard link** (or another symbolic link**). Thus, when a symbolic link is removed, the file to which it pointed is not affected**. **In contrast, the removal of a hard link will result in the removal of the file, if it is the last hard link to that file**. As a result, symbolic links can be used to refer to files on other mounted file systems. A symbolic link whose target does not exist is known as an orphan.

Symbolic links are transparent, which means that their implementation is invisible to applications. When opened, read, or written, a symbolic link will automatically cause the action to be redirected to its target. However, functions to detect symbolic links are available, so applications may find and manipulate them if needed.

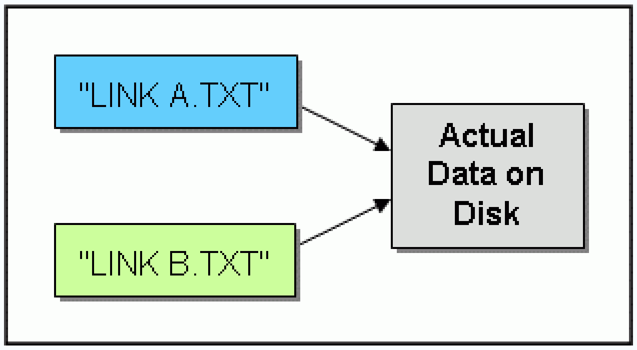
Careful attention must be given to maintenance of symbolic links. Unlike hard links, if the target of a symbolic link is removed, the data is lost and all links to it become orphans. Conversely, removing a symbolic link has no effect on its target.

hard links, which refer to the specific location of physical data.

a **hard link** is a reference, or pointer, to physical data on a storage volume. On most file systems, all named files are hard links. The name associated with the file is simply a label that refers the operating system to the actual data. As such, more than one name can be associated with the same data. Though called by different names, any changes made will affect the actual data, regardless of how the file is called at a later time. Hard links can only refer to data that exists on the same file system.

On Unix, hard links can be created with the link() system call.

The process of *unlinking* disassociates a name from the data on the volume. The data is still accessible as long as at least one link that points to it still exists. When the last link is removed, the space is considered free. A process ambiguously called *undeleting* allows the recreation of links to data that is no longer associated with a name. However, this process is not available on all systems and is often not reliable.



In the figure above, there are two hard links named "LINK A.TXT" and "LINK B.TXT". They have both been linked - that is, made to point - to the same physical data.

If the filename "LINK A.TXT" is opened in an editor, modified and saved, then those changes will be visible even if the filename "LINK B.TXT" is opened for viewing since both filenames point to the same data. The same is true if the file were opened as "LINK B.TXT" - or any other name associated with the data.

Additional links can also be created to the physical data. The user need only specify the name of an existing link; the operating system will resolve the location of the actual data section.

If one of the links is removed (ie, with the UNIX 'rm' command), then the data is still accessible under any other links that remain. If all of the links are removed and no process has the file open, then the space occupied by the data will be considered free, allowing it to be reused in the future for other files. This semantic allows for deleting open files without affecting the process that uses them - an action which is impossible on filesystems with a 1-to-1 relationship between directory entries and data.

The general syntax for the `ln` command is:

ln [-s] *target* [*name*]

The *`target`* parameter indicates the location or file to which the link should point. The optional *`name`* parameter specifies the name that link should be given. If no name is specified, the basename of the target will be used. The `ln` utility creates a hard link by default; the `-s` option indicates that a symbolic link should be created instead.

Examples

ln xyz abc

Creates a hard link called `abc` that points to the same data as the existing file `xyz`.

ln -s /usr/share/pixmaps/image.jpg

Creates a symbolic link that points to the path `/usr/share/pixmaps/image.jpg`. The link would be named `image.jpg`, because that is the basename of its target.

ln harder dir/hard

Links `dir/hard` --> `./harder` but

ln -s softer dir/soft

links `dir/soft` --> `dir/softer` which is probably unintended.

PS

The *ps* (i.e., *process status*) command is used to provide information about the currently running *processes*, including their *process identification numbers* (PIDs).

A process, also referred to as a *task*, is an *executing* (i.e., running) instance of a program. Every process is assigned a unique PID by the system.

The basic syntax of ps is

```
ps [options]
```

When ps is used without any options, it sends to *standard output*, which is the display monitor by default, four items of information for at least two processes currently on the system: the *shell* and ps. A shell is a program that provides the traditional, text-only user interface in Unix-like operating systems for issuing commands and interacting with the system, and it is *bash* by default on Linux. ps itself is a process and it *dies* (i.e., is terminated) as soon as its output is displayed.

The four items are labeled PID, TTY, TIME and CMD. TIME is the amount of CPU (central processing unit) time in minutes and seconds that the process has been running. CMD is the name of the command that launched the process.

TTY (which now stands for *terminal type* but originally stood for *teletype*) is the name of the *console* or *terminal* (i.e., combination of monitor and keyboard) that the user logged into, which can also be found by using the *tty* command. This information is generally only useful on a multi-user network.

A common and convenient way of using ps to obtain much more complete information about the processes currently on the system is to use the following:

An alternative set of options for viewing all the processes running on a system is

```
ps -ef | less
```

The *-e* option generates a list of information about every process currently running and f is for full length.

ps is most often used to obtain the PID of a malfunctioning process in order to terminate it with the *kill* command. For example, if the PID of a *frozen* or *crashed* program is found to be 1125, the following can usually terminate the process:

```
kill 1125
```

If it has not, then the more forceful *-9* option should be used, i.e.,

```
kill -9 1125
```

TRAP

Shell procedures may use the **trap** command to catch or ignore Unix operating system signals. The form of the trap command is:

```
trap 'command-list' signal-list
```

Several traps may be in effect at the same time. If multiple signals are received simultaneously, they are serviced in ascending order.

To check what traps are currently set use the trap command The `trap` shell built-in command allows you to execute a command when a signal is received by your script. It works like this:

```
trap command list-of-signals
```

If the *`command`* is a single dash (`-`), or with many shells an empty string (`""`), the listed signals are simply ignored. If the *`command`* is missing altogether, the signals listed go back to their default action.

The signals that a script commonly needs to handle include `SIGHUP (1)`, `SIGINT (2)`, `SIGQUIT (3)`, and `SIGTERM (15)`. The list can be either the names or the numbers in bash.

UNIX Variables

Variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for particular variables and if they are found will use the values stored. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

Standard UNIX variables are split into two categories, environment variables and shell variables. In broad terms, shell variables apply only to the current instance of the shell and are used to set short-term working conditions; environment variables have a farther reaching significance, and those set at login are valid for the duration of the session. By convention, environment variables have UPPER CASE and shell variables have lower case names.

Environment Variables

An example of an environment variable is the OSTYPE variable. The value of this is the current operating system you are using. Type

```
% echo  $OSTYPE
```

More examples of environment variables are

- USER (your login name)
- HOME (the path name of your home directory)
- HOST (the name of the computer you are using)
- ARCH (the architecture of the computers processor)
- DISPLAY (the name of the computer screen to display X windows)
- PRINTER (the default printer to send print jobs)
- PATH (the directories the shell should search to find a command)

Finding out the current values of these variables.

ENVIRONMENT variables are set using the **setenv** command, displayed using the **printenv** or **env** commands, and unset using the **unsetenv** command.

To show all values of these variables, type

```
% printenv | less
```

Shell Variables

An example of a shell variable is the history variable. The value of this is how many shell commands to save, allow the user to scroll back through all the commands they have previously entered. Type

```
% echo $history
```

More examples of shell variables are

- cwd (your current working directory)
- home (the path name of your home directory)
- path (the directories the shell should search to find a command)
- prompt (the text string used to prompt for interactive commands shell your login shell)

Finding out the current values of these variables.

SHELL variables are both set and displayed using the **set** command. They can be unset by using the unset command.

To show all values of these variables, type

```
% set | less
```

So what is the difference between PATH and path ?

In general, environment and shell variables that have the same name (apart from the case) are distinct and independent, except for possibly having the same initial values. There are, however, exceptions.

Each time the shell variables home, user and term are changed, the corresponding environment variables HOME, USER and TERM receive the same values. However, altering the environment variables has no effect on the corresponding shell variables.

PATH and path specify directories to search for commands and programs. Both variables always represent the same directory list, and altering either automatically causes the other to be changed.