Linux Essentials – Module 4

# MORE SHELL (or the Shell on Steroids)

MODULE OBJECTIVE

- Understand various aspects of command processing in regards to formulating command input on the command line or in a script to achieve desired effect.

LESSON OBJECTIVES

- Use the command **&** to start jobs in background.
- Explain the purpose of the redirect symbols **<**, **>**, and **>>** .
- Use the shell pipe symbol **|** to construct command pipelines.
- Create command aliases
- Use shell variables
- Become familiar with simple shell scripts

## FOREGROUND and BACKGROUND PROCESSES

It’s time to see how to be a multitasking Linux user by learning how to run jobs in the background (by using the **bg** command) while you are doing something else, and how to have programs instantly go into the background (by using the **&** notation), so that you can do other things. Now to be honest in much of your live work will be done on a desktop system which supports a window-ing environment meaning you can open up multiple terminal windows on a desktop to perform multiple operations. But bear in mind that the desktop environment may not always be available. This is especially true when doing remote shell operations.

So, recalling that the O/S itself is a true multi-tasker, you can run many programs at a time even from a single shell . You are not limited to just one process. For example, if you wanted to extract information from an extremely large file, you can run those processes in the background while you are working on something else. Once a job is stopped (CTRL z), you can enter **fg** to start it up again as the program you're working with. (The **fg** command takes its name from foreground, which refers to the program that your display and keyboard are working with.) If the process will continue without any output to the screen and without any requirement for input, you can use **bg** to move it into the background, where it runs until it is done. If the program needs to write to the screen or read from the keyboard, the system will stop its execution and inform you. You can then use **fg** to bring the program into the foreground to continue running.

The following list shows the major commands used in this form of job control:

| | |
|---|---|
| ^Z (Control-Z) | typing Control-Z while the job is running will cause it to halt. At this point, you can then (1) leave it stopped, (2) send it to the background or (3) bring it back into the foreground. |
| bg | (short for "BackGround") Having stopped a job with ^Z, typing this will send it into the background. |
| fg | (short for "ForeGround") You can use this to bring a stopped or background job into the foreground. |
| jobs | This will give you a numbered list of the stopped and background jobs. |
| command & | Ending a command with an ampersand will cause it to begin executing in the background. |

Start a process in foreground, stop it, and move it to background.

This process will process files without needing any input or offering any output.

```
    $   find  /  -type  d  >  /dev/null  2>&1       (Once started press  CTRL z)
    [1]  +  Stopped                          find / -type d -print > /dev/null 2>&1
    $



    $    bg
    [1]    find  /  -type  -print  >  /dev/null  2>&1  &

    $
```

The number in the brackets is the job's control number.  The last ampersand & denotes the job is running in background.

Some systems will notify you immediately once the background job is done, but on most systems, after a background job successfully completes, it waits until you press the RETURN and get a new system prompt.

```
    $
    [1] +   Done         find  /  -type  d  >  /dev/null  2>&1  &

    $
```

If the process "BOMBS" or is **kill**ed before a successful completion, you will see the response:

```
    $
    [1]    Terminated     find  /  -type  d  >  /dev/null  2>&1  &
```

An alternative way to run a process in the background is to start it in the background from the command and the & ampersand.  This is used mostly with processes that do not need the keyboard or display.

Starting a process in the background, using the &

```
    $    find  /  -type  d  > /dev/null  2>&1  &
    [1]    2309
    $
```

As before, the number in brackets is the job number.  The 2309 is the process ID number.

Striking the RETURN will give the following:

```
    $
    [1] +   Done      find  /  -type  d  >  /dev/null  2>&1  &
```

Another option of working in foreground and background is, starting a process in background and bringing it to the foreground.

Starting a process in the background and bringing it to the foreground.

```
    $    find  /  -type  d  > /dev/null  2>&1  &
         [1]     4310
    $    fg

         find  /  -type  d  > /dev/null  2>&1

    $
```

By entering the **fg** command it will bring the process to the foreground.  Once the process is complete, it will return the prompt, just as any process running in the foreground.

## Starting a Program in the background with the Ampersand “&”

A background process is a program that runs in the background during its allocated time slices. It allows the screen and keyboard to be used to run a foreground process. However, the background process should get its input and send its output from and to some special designated points so it not does not wait for keyboard input or write text to the screen. The OS and servers run in the background, the shell runs in the foreground.

A background process is started from the shell by placing the **ampersand** “**&**” at the end of the command line. The following example demonstrates a simple “ls” command placed in background. Once the background job is started, the **job number** and **process ID** number are returned to standard out. You may need these numbers later for job control. Output from the command is also returned to standard out (somewhat defeating the purpose of placing the job in background).

```
$ ls  &
[1]    28513
$ savewho       viewlabfiles   viewlabfiles2

[1] +  Done                    ls &
$
```

.

## The jobs command

The command **jobs** is used to display the status of an active job, including those in background or suspended with ctrl - z.  If a job is not specified, all active jobs are displayed.  The format of the jobs command is:

```
jobs [ options ] [ job ]
```

options include:

| | |
|---|---|
| -l | list process ids in addition to normal information |
| -n | display only stopped or exited jobs |
| -p | display only the specified process group |

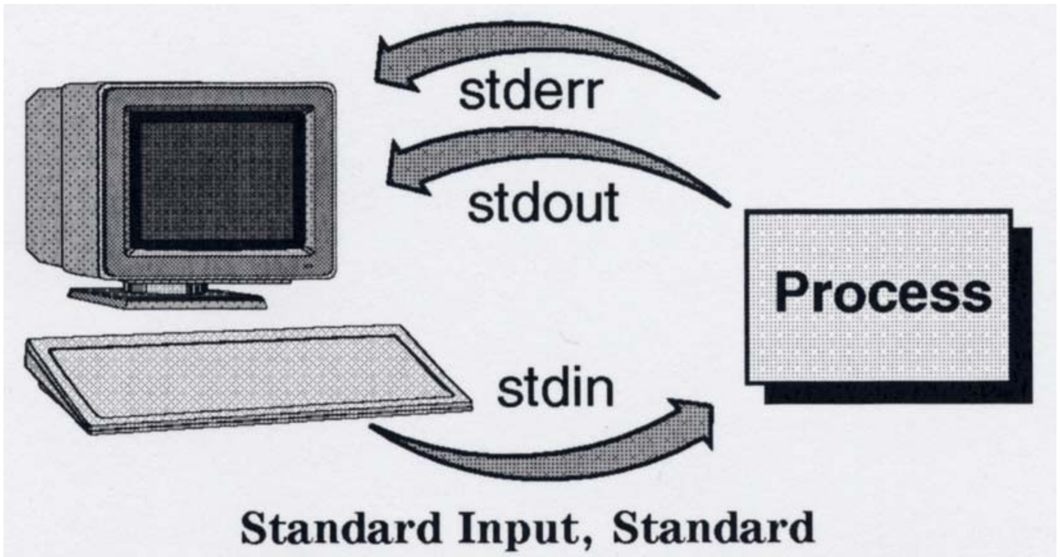
## REDIRECTION

Every program has at least three data paths associated with it: **standard input**, **standard output**, and **standard error**. Programs use these data paths to interact with you. By default, standard input (**stdin**– **fd0** – **<**) is your keyboard. The default destination for both standard output (**stdout** – **fd1** – **>** – 1>) and standard error (**stderr** – **fd2** – **2>**) is you display screen.

Programs use these paths as follows:

| | |
|---|---|
| Standard input | (fd0)The place from which the program expects to read its input. By default, processes read **stdin** from the keyboard. |
| Standard output | (fd1)The place the program writes its output. By default, processes write **stdout** to the terminal screen. |
| Standard error | (fd2)The place the program writes its error messages. By default, processes write **stderr** to the terminal screen. |

The figure below illustrates the relationship of these files to the process.



**Standard Input, Standard Output, and Standard Error**

You see that by convention, most Linux commands read from "standard input" and write to "standard output". Normally, “standard input” is the keyboard, and “standard output” is the terminal. Another file, "standard error", is used for error messages and other information about the operation of a command. Normally, “standard error” is directed to the terminal.

One of the most powerful features of Linux is that the input can come from a file as easily as it can come from the keyboard.  And the output can be saved to a file as easily as it can be displayed on your screen.

Redirecting input and output is a convenient way of selecting what files or devices a program uses.  The output of a program that is normally displayed on the screen can be sent to a printer or to a file.  Redirection does not affect the functioning of the program because the destination of output from the program is changed at the system level.  The program is unaware of the change.

The power of "file re-direction" are the special symbols used, by the shell that instruct the computer to read from a file, write to a file, or even append information to an existing file.  Each of these acts can be accomplished be placing a file redirection command in a regular command line:

| | | | |
|---|---|---|---|
| 0< | or | < | redirects standard input |
| 1> | or | > | redirects standard output |
| 1>> | or | >> | redirects standard output and appends the information to the chosen file. |
| 2> | | | redirects standard error |
| 2>&1 | | | redirects standard error to the same place as standard output |

Re-directing standard input and standard output occurs frequently in Linux commands.

## Writing Standard Output to a file

The shell lets you redirect the standard output of a process from the screen (the default) to a file.  Redirecting output lets you store the text generated by a command into a file; it's also a convenient way to select which files or devices (such as printers) a program uses.

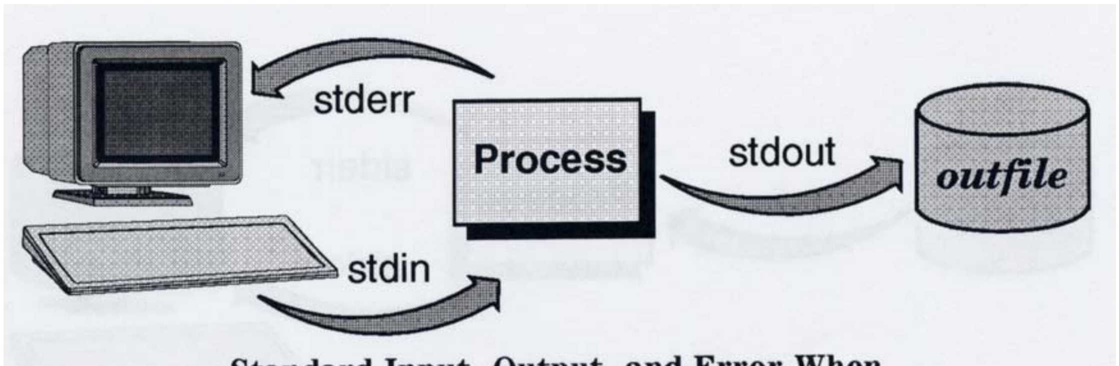
In its simplest form the command syntax is as follows (note that with redirection a command always appears to the left and the input or output file or device to the right):

*command* > *outfile*

where *command* is the command whose output is redirected, and *outfile* is the name of the file to which the process writes its standard output.  If the output file exists, its previous contents are lost.  If the file does not exist, it is created.

To **append** the output to an existing file, use two greater-than signs (**>>**) pointing to the file to be appended on.

The figure below illustrates where **stdin**, **stdout**, and **stderr** go when output is redirected.



**Standard Input, Output, and Error When Output Is Redirected**

The following command would normally display the contents of the /etc/passwd file to the display, but with the **output** redirected to the **outfile**, the **stdout** will not be seen on the monitor. Using the more command will display the contents of the **outfile**.

```
$ cat /etc/passwd > outfile
$ more outfile
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:1:2:daemon:/sbin:/sbin/nologin
.
.
.
student1:x:501:504::/home/student1:/bin/bash
```

The following command will find any filename that contains the string “test” starting with the / directory and all directories within the slash directory. The output (**stdout**) is redirected to the **outfile**, overwriting the previous contents of **outfile**. Notice that standard error (**stderr**) was not redirected so, those results are still sent to the monitor.

```
$ find / -name test > outfile
```

The following command will **append** (**>>**) the output of the **cat** /etc/passwd to the **outfile**. Notice that there are no blank lines between the original and appended text.

```
$ cat /etc/passwd >> outfile
```

## Using Files for Standard Input

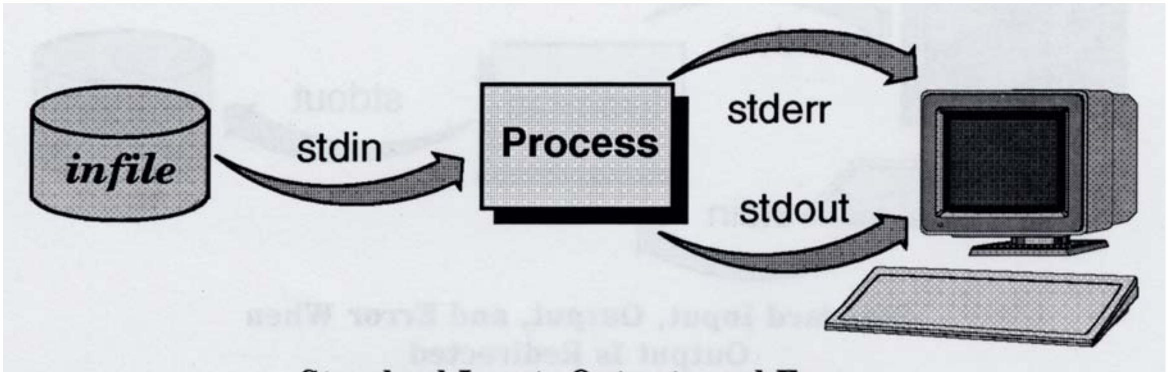
The shell lets you redirect the **standard input** of a process so that input is read from a file instead of from the keyboard. To redirect the **input** of a process, separate the command and the input file name with a **less-than sign** (**<**) directed at the command name. You can use input redirection with any command that accepts input from **stdin** (your keyboard).

In its simplest form, the command syntax is as follows:

*command* < *infile*

where *command* is the command whose input is redirected, and *infile* is the name of the file from which the process reads standard input. The file must exist from the redirection to succeed.

The figure below illustrates where **stdin**, **stdout**, and **stderr** go when input is redirected from a file.



**Standard Input, Output, and Error When Input Is Redirected**

Re-directing standard in

$ **mail student1 < /etc/passwd**

This command will send mail to student1, taking the input from the contents of the /etc/passwd file.

Re-directing standard out

The following example, **output** from the **cat /etc/passwd** command is stored in the file **savecat**. Then, the **savecat** file is used as **input** to the **wc** (Word count – count lines) command:

```
$ cat /etc/passwd > savecat
$ wc -l < savecat
```

## COMMAND PIPING

The following command will pipe the output from the **cat /etc/passwd** command to the input of the **wc** command, resulting in the line count ( **-l** option ) of the /etc/passwd file. The output of the cat command is basically redirected to the pipe then to the wc command so, you would not see the contents of /etc/passwd displayed on the monitor.

```
$ cat /etc/passwd | wc -l
12
$
```

By the use of commands known as “**filters**” (eg. grep) , the output of one command can be directed and used as the input to another command. Some of these filter commands will pass the data stream unchanged and others will adjust the data stream producing a completely different output.

Command re-direction is the process of directing either input or output to a location other than the default location. The shell pipe character ( **|** ) allows the output of one command to be piped to the input of another command. This allows the Linux user to combine simple command processes into more elaborate command lines, with the result being the same as the simple commands by themselves. Using the shell pipe character, several commands can be strung together (pipelined) to customize information from several sources.

The shell lets you connect two or more processes so the standard output of one process is used as the standard input to another process. The connection that joins the processes is a **pipe**. To **pipe** the output of one process into another, you separate the commands with a vertical bar ( **|** ). The general syntax for a pipe is as follows (note that the syntax always has a <u>command</u> on both sides of the pipe):

*command1* **|** *command2*

where *command1* is the command whose **standard output** is redirected or **piped** to another command, and *command2* is the command whose **standard input** reads the previous command’s output. You can combine two or more commands into a single **pipeline**. Each successive command has its output piped as input into the next command on the command line:

command1 **|** command2 **|** ...... **|** command(n)

Command Piping

```
$ history | more
```

This pipeline will produce a listing of the user's command history and then pass it to the **more** command allowing you to view the listing one page at a time.

```
$ grep  student   /etc/passwd   |   more
```

Using the pipe symbol ( | ), the output of the grep command is sent to the input of the **more** command. The result of the **more** command is sent to the standard output. In this case, no output redirection was declared, so the output is sent to the terminal.

```
$ grep "fresh" ingredients | sort > pizza
```

As you see this pipeline has a combination of pipes and a redirection.
The **grep** command (discussed later) will extract the lines from the file "ingredients" that contain the word "fresh". The results will then be passed to the **sort** command and the lines will be sorted. The results of the **sort** command will then be redirected to the file called "pizza" and stored. (The lines will be sorted by the first character of the line; number, capital, and then lower case).

There are numerous ways that you can combine the various forms of file redirection and piping to create custom commands and to process files in various ways.

## COMMAND GROUPING

Sometimes you want to stand in the doorway and have your foot in two rooms at the same time in order to pass something from one room to another. **Parentheses** are used to execute one or more commands under a separate shell process, which connects to the first process via a pipe. In the following example there is a ton of things going on. You don't need to understand all of this but it illustrates how comples commands can become and how important it may be to get things to process in a specific sequence. This is where command grouping comes into play. Here there ar two nested command groupings.

```
 $ ( cd /usr/vue;  nice -39 tar  cf -  .  ../man | \
>  ( cd ~/tardir;   tar xf - ) > /tmp/log 2>&1  ) &
 $ ps -ef > psout; sleep 5; banner five >> psout; \
>   ps -ef >> psout; sleep 10; banner ten >> psout; ps -ef >> psout
```

## ALIASES

Linux is a computer-geeks dream for a number of reasons. In the previous section we have seen how to manipulate command lines to make complex command entries. But going a step further what if we wanted to make our own commands. In a later section we will talk about scripts as one way to do that. But there is another very simple way of creating shorthands commands – command aliases.

Bash, C, Korn, and POSIX allows the System, Superuser, or the user to define commands by new names. Aliases can be created, listed, and exported using the **alias** command and can be removed with the **unalias** command.

To display any aliases set in your environment enter:

```
$ alias
```

If you create an alias on the command line, it is only good for this login session.  When you log back in, it will be gone.  If you want it set each time you log in, you can put it in your personal startup file - **$HOME/.bashrc**, or the SA can put them in a system-wide file.

There are two other kinds of aliases  - **preset** and **tracked**. Preset are system-dependent and users cannot change them. They are created when the software is installed. Tracked aliases essentially are resolved pathnames for commonly used commands and are also created by the shell.

It is a bad practice to use personal or system-wide aliases inside shell scripts. You might not even be able to pass your programs to other users on the same machine if they are defined outside the shell script. It also makes it very difficult to read and understand the program since there are no ***man*** pages defining the aliases.

The first word of each command is replaced by the text of an alias, if an alias for this word has be defined.  An alias name consists of any number of characters excluding metacharacters, quoting characters, file expansion characters, parameter and command substitution characters, and =. The replacement string can contain any valid shell script, including the metacharacters mentioned above. Aliases can be used to redefine special commands, but cannot be used to redefine keywords.  Aliases can be created, listed, and exported using the **alias** command and can be removed with the **unalias** command.

Aliasing is performed when scripts are read, not while they are executed.   Therefore, for it to take effect, an alias must be executed before the command referring to the alias is read.  Aliases are frequently used as shorthand for full path names.

Listing the currently defined aliases.

```
# alias
history='fc -l'
stop='kill -STOP'
suspend='kill -STOP $$'
ll='ls -l'
#
```

Defining a new alias.

```
# alias dir='ls -la '
# alias
autoload='typeset -fu'
command='command '
dir='ls -la '
history='fc -l'
stop='kill -STOP'
suspend='kill -STOP $$'
ll=”ls –l”

# dir
total 416
drwxr-xr-x  28 root      root         4096 Nov 19 15:27 .
drwxr-xr-x  28 root      root         4096 Nov 19 15:27 ..
-rw-------   1 root      sys           647 Nov 17 13:30 .Xauthority
-rw-rw-r–   1 root      sys            98 Oct  5  2001 .audioCP
drwxr-xr-x  12 root      sys          1024 Nov 17 13:30 .dt
. . .
dr-xr-xr-x   2 root      root           24 Mar 20  1998 tmp_mnt
dr-xr-xr-x  25 bin       bin          1024 Jul 18 08:10 usr
drwxr-xr-x  19 bin       bin          1024 Nov 29  2000 var
#
```

The command “**dir**” is now aliased to “**ls -la**”.  Typing the command “**dir**” executes the “ls -la” command.  Notice that a **<space>** character is inserted after the option “**-la**”.  This is intentional, it means that the word following the alias is also checked for alias substitution.

Aliases can also be used as shorthand for full path names.

```
# alias home=/usr/local/bin
# alias
dir='ls -la '
history='fc -l'
home=/usr/local/bin
istop='kill -STOP'
suspend='kill -STOP $$'
ll='ls -l'
# dir home
total 552
drwxr-xr-x   3 bin        bin           4096 Nov  4 08:39 .
drwxrwxrwx  12 bin        bin           1024 Jul 11 10:30 ..
-rwxr-xr-x   1 root       sys           1882 Oct  9  2001 add_accounts
-rw-rw-r–   1 root       sys           2425 Nov  4 08:39 swinstall.ntc242
-rw-rw-r–   1 root       sys           2425 Nov  4 08:39 swinstall.ntc244
-rw-r-----   1 root       sys            446 Jun 21  2001 system.SAM
#
```

Removing a Previously Defined Alias

Syntax:        unalias (alias_name)

```
#unalias dir
#unalias home
#alias
history='fc -l'
stop='kill -STOP'
suspend='kill -STOP $$'
ll='ls -l'
#
```

# VARIABLES

**Variables** are simply names attached to some bit of information that is stored in a process's storage area in memory. Once the program stops running they disappear unless saved to a file. Some variables may be reset by regular users.

There are two kinds of variables – **GLOBAL (or environmental)** and **LOCAL (or shell)** .

*By convention*, **GLOBAL** variables are usually written in **UPPERCASE** characters, and **local** variables are in **lowercase.** **Environmental variables** are shell parameters that are **global** and used by your shell to create special environments for subshells and any other commands that you might invoke. These environments will be active until you logoff. These **global** ( also called **exported** ) variables can be seen and used by subshells and other subprocesses. Global variables are inherited by a **Parent** process' **Child**ren, but not by the grandparents ( a process's **Parent**) - they only move *down* the food chain.

**Shell variables** (or *local* variable) are shell parameters that are **local** to your login shell and are **NOT** passed to any subshells or subprocesses. Local variables are only known to the process in which they are defined - not to its **Parent** or **Child**ren. Once a **Child** inherits a variable, it can do whatever it wants with it; the **Parent** will never know.

## SIMPLE VARIABLE SYNTAX (bash [sh], POSIX, ksh)

```
 local_variable=value


GLOBAL_VARIABLE=value; export VARIABLE
                        OR
export GLOBAL_VARIABLE=value
```

**COMMAND SUBSTITUTION**

```
a.      $ xdate='date'

b.      $  find . –mtime -1 > findlist.$date
```

Variables are commonly used in command substitution. The above example shows us creating a list of files created in the past 24 hours and redirecting them into a file. Note the the name given to the file uses command substitution to substitute the actual date as part of the filename itself.

**COMMON GLOBAL/ENVIRONMENTAL VARIABLES**

| **VARIABLE** | **DEFAULT** | **FUNCTION** |
|---|---|---|
| **PS1** | **$** | **primary prompt** |
| **PS2** | **>** | **secondary prompt when using \nl or broken token** |
| **PATH** | varies | **search path for executables** |
| **EDITOR** | varies | **default editor in a number of commands** |
| **FCEDIT** | ed | **editor  for command history editing** |
| **HISTSIZE** | **16** | **number of past commands saved** |
| **PAGER** | more | **pause screen output in several commands** |

Environment variables and shell variables create part of the environment in which you work, such as the prompt string **PS1.** For another example - the name of your current shell program is saved in a variable called **SHELL** .

To see what your current shell is for example, enter:

```
$ echo $SHELL
/usr/bin/sh
```

**Setting user environment and shell variables**

Your login shell is determined by an entry in a local or networked 'password' file. Depending on what shell is configured as your default, at login time the initial ***sh*** login process is **EXEC**'d by the ***login*** program, Then becomes your **PARENT** login process. Before your first ***sh*** system prompt is displayed however, some scripts will be run in your honor to set up your user environment (among other things). If using the *bash* shell you might want to customize the following variables in your ***/*****.bas_profile** for example. if not already set in **/etc/profile**, or if you do not like the system defaults. Each user will have personal configuration files in their personal home directory which can override global configuration files. For example, one key variable is **PATH** which defines in which directories the shell will look for requested commands. This is often referred to as the *search path*. You may want to add the directories here for local applications that all users need.

Startup and configuration scripts usually begin with a **. (dot)** and end with an **rc .** The initial startup files below are **EXEC**'d (<u>if they exist</u>) by the **login** program for Bash, POSIX shell, and C-shell users as configured through the /etc/passwd file shell entry, *e.g.*, the last entry in a line in /etc/passwd may be /etc/bash, before your first system prompt is displayed.

| | | |
|---|---|---|
| bash: | 1st - /etc/profile ,then the files in /etc/profile.d/ if they exist | |
| | 2nd - $HOME/.bash_profile | |
| | 3rd - $HOME/.bashrc | |
| | 4th - /etc/bashrc | |
| Posix-shell: | 1st - /etc/profile | 2nd - $HOME/.profile |
| C-shell: | 1st - /etc/csh.login | 2nd - $HOME/.cshrc |

*/etc/profile* and */etc/bashrc* are readable by ordinary users, but maintained by the SA. They are meant for running commands (once at login), and setting variables and aliases for everyone logging into the machine. *$HOME/.bashrc* (and *$HOME/.profile* or *$HOME/.cshrc)* do not necessarily have to exist, but if they do, the one appropriate for the shell used is **EXEC**'d during a login session such as ssh. Here, you can modify your shell environment for subsequent logins***, e.g.,*** the **bash** shell looks for a file called **.bashrc** in the user's home directory and if found, the values can override the global ones.

```
[root@ntc234 skel]# **cd /home/lois**
[root@ntc234 lois]# ls -la
total 32
drwx------    3 lois     lois        4096 Jul  1 15:22  .
drwxr-xr-x    4 root     root         4096 Jul  1 15:22    ..
-rw-r--r–    1 lois     lois          24 Jul  1 15:22     .bash_logout
-rw-r--r–    1 lois     lois         191 Jul  1 15:22    .bash_profile
-rw-r--r--    1 lois     lois         124 Jul  1 15:22       .bashrc
-rw-r--r–    1 lois     lois         847 Jul  1 15:22       .emacs
-rw-r--r–    1 lois     lois         120 Jul  1 15:22        .gtkrc
drwxr-xr-x    3 lois     lois        4096 Jul  1 15:22    .kde
[root@ntc234 lois]#
```

Under bash */etc/bashrc* is also read for all users. It is used to set system-wide functions and aliases. This file is not only read at login, but also each time a new shell is opened. Then *$HOME/.bashrc* is processed at login to customize the user's unique environment. This is done every time a bash shell is started. Let's look at a typical **.bashrc** file:

```
[lois@ntc234 lois]$ pwd
/home/lois
[lois@ntc234 lois]$ more .bashrc
# .bashrc

# User specific aliases and functions
alias cp='cp -i'
alias rm='rm -i'
alias mv='mv -i'

# Source global definitions
if [ -f /etc/bashrc ]; then
        .  /etc/bashrc
fi

PATH=$PATH:/usr/bin:/usr/local/bin
export PATH
[lois@ntc234 lois]$
```

In the example on the previous page user-specific aliases have been added for the **rm**, **cp,** and **mv** commands.  Adding the –i option protects against the accidental deletion of files.  The next section will source (or execute) the global definitions if they exist.  The last section will change the search path variable to add the **/usr/sbin** and **/usr/local/bin** directories.

There are some utilities that provide an editing option where the default editor is vi, *e.g.,* crontab.  If a user prefers a text editor like gedit you could make the following entry in their .bash_profile file to set the environmental variable called EDITOR

```
export EDITOR gedit
```

## SHELL PROGRAMMING (LITE)

One of the great things about Linux (and UNIX in general) is that it's made up of individual commands, "building blocks" like *cat* and *grep*, that you run from a shell prompt. Using pipes, redirection, filters, and so on, you can combine those utilities to do an incredible number of things. Shell programming lets you take the same commands you'd type at a shell prompt — and put them into a file that can execute by just typing its filename. A few minutes of shell script authoring can save a ton of time reissuing commands you use frequently. It is fairly easy to write a simple script to automate tasks or write lengthier scripts to create complex programs, which combine other programs, scripts, and commands to do just about anything you can think of.

**NOTE:** Unfortunately, the same commands on different versions of Linux/UNIX can have different options. Also the various shells, even on the same platform, have different scripting constructs for such thing as test statements or flow control. Using the man page for the shell, you are using, will describe the syntax and options available for that shell for the system you are using.

Every UNIX operating system provides at least one shell. Typically, the operating system invokes a shell when you log in. After the shell starts up, it displays a prompt and waits for you to enter commands. For example, if you were a LINUX user who wanted to know the names of the files in a directory, then you would probably type the command *ls* inside a shell. The shell will interpret the *ls* command, which means that it will figure out what command you typed ("oh, you want to execute an ls command") and then invoke the command.

**Writing a Simple Shell Program**

A shell script program can be a list of Linux/UNIX commands, saved to a file, to perform tasks repetitively, or a single complex command, used often.

The steps necessary to create a simple shell program are:

1) Create a file with shell commands using a text editor (naming a shell script is completely up to the creator). The first line *should* be the shell interpreter (eg. #!/bin/sh)

2) Using **chmod** give the file execute permission.

3) Execute the script by typing the ***filename.sh*** at the command line. If your current directory is not in your $PATH, you can execute the script with the command:

   **./filename.sh**
   (The "." dot will search the current directory. You must be in the correct current directory.)

THAT'S IT !

Well obviously there is more to shell scripting in terms of programming a script to do the things it is meant to do. Detailed programming is not going to be taught in this course. We'll leave you with this suggestion to get a good text book on the subject and then just try writing simple scripts and build your skills to the more complex. Simple scripts are quickly written.

## What can be in a Shell Script Program?

There is no set shell script form. We will try to show a form that includes the Execution shell, USAGE (how to execute the script), remarks, defining variable, and the body (the set of commands to be executed).

The creator of the script can make the script as simple or as complex as is necessary. An important point in shell scripting is the use of remarks (#) and white space (spaces, tabs, newline) to describe the script's purpose, execution command, and/or ease of reading the contents of the script itself. A well-documented (user-friendly) script is more readily accepted by others. When writing a script "Think of the other person". It could be someone you don't know or, it maybe "YOU" six months down the line.

One the following page is a way (not the only way) to layout a shell script…

```
#!/bin/sh        #This line, is used at the very beginning of the script (first column, first
                 # row), insuring the script is executed in a bash shell, regardless of
                 # the login shell.  The "#" beginning this line is not seen as a remark
                 # and ignored by the shell.

                 #Blank lines can be used within the script and ignored by the shell.

USAGE="usage: script_name.sh"
                #Tells how to start the script.  This line can be simple, as shown, or
                # may include various options needed to invoke this script.  Defining it
                # as a variable will allow you to call it if a user enters the wrong
                # command line.  Using this will also allow the user to "grep" for the
                # word "USAGE" to see how to invoke this script.

export VAR_NAME=value_of_var
               #This example will create a GLOBAL VARIABLE.  Variables can be
               # defined anytime, but must be defined before being called or used.

local_var=value_local_var
               #Local variables can also be defined at this time.

Body of the script.
               #This area is for the commands to be accomplished by the script.
               # These can be a list of simple commands, "Test"  syntax, "if", "case",
               # "loop" statements, and/or function calls. Functions must be defined
               # before they are called by the script.
```

An (extraordinarily) simple example:

```
#!/bin/sh
ls /home
grep “student” /etc/passwd
grep “student” /etc/group
grep “student” /etc/shadow
```

The fairly simple:

```
#!/bin/sh

#The script will provide a status of network connectivity, onto the network, to the
# gateway and verify DNS.

USAGE=”usage: chknet.sh”

DNS=231
GW=1
echo “checking network status”
hostname
if /sbin/ifconfig eth0 | grep “RUNNING”
   then
        ping –c 3 192.168.21.$GW
        ping –c 3 192.168.21.$DNS
        host ntc$DNS
        for IP in 232 233 234 235 236 237 238 239 240 241 242 243 244
             do
                 ping –c 3 192.168.21.$IP
             done
    else
         echo “Network is down”
fi
```

The above example shows the power of a few simple techniques such as using variables (e.g. DNS) , test statements (e.g. if ) and loops (e.g. for) . In just a couple of lines we can check whether the network is up (line with “RUNNING”); and the gateway and the DNS server are available (we used variables DNS and GW to make maintenance easier should the ip-addresses ever change); and check connectivity to every workstation on the network (loop all workstations addresses 232-244 to do a ping on each).

## Executing Shell Scripts

Shell scripts are simply command lines that the shell executes as a group. The command lines are entered into a file using an editor, usually vi. Generally, one command line entered per line of the file.

Once the file is created it must be made executable using the chmod command as follows:

Example:

```
$chmod +x script_name
$
```

The chmod command changes the permission on *script_name* so that it is executable.

Type in the name of the script and it will execute and display any output.

```
$script_name
$
```

If the script does not reside in directory that is part of your search path (see: echo $PATH), then you will have to use a full path or relative addressing when starting it such as (Assuming a script named myscript.sh is stored in the /home/student1 directory):

```
$ /home/student1/myscript.sh
-or-
$ cd /home/student1
$ ./myscript.sh
```

**Script Processing**

The shell will execute the shell script by reading from left to right, top to bottom, ignoring white space and # remarks. This will continue until one of five conditions occur:

1) **A system hangs** This could be caused by a number of system hardware or software conditions. Possibly the script calling for system resources not available on your system.

2) **The script hangs** This could be caused when a syntax or scripting error (a bug) occurs.

3) **Completion** The successful/unsuccessful completion of all the commands within that script. The completion of the script commands will produce a return status code (also known as return code, exit code, or status code).

4) **User Interaction** The script may call for user input from the keyboard. If the script calls for user input, it will sit and wait, for that input.

5) **Conditional Commands** If during the execution of the script, conditional commands occur (test, comparison, and/or logic constructs) breaking from the left/right/top/bottom and branching to a later part of the script will occur.

( To get more information on scripting it is well worth the time and effort to get on a Linux or UNIX system and find some scripts you can view to get a better feel for what scripting is all about; furthermore a good scripting book is worth it's weight in gold.)

## LAB

## Your practical exercises for this module:

**Again log onto the NWSTC student server (**204.227.127.133) **and practice more Linux commands.** If you need the instructions again they can be found at the following link:

**http://webdev.nwstc.noaa.gov/d.train/linuxinstr.html**

Remember:

1. You are encouraged to **EXPERIMENT** in this course and try various commands, so that you **SUCCEED** in the field and subsequent training.

2. DO NOT enter the commands robotically without trying to understand them in the process. Your success at further Linux training and actual work in the field is wholly dependant upon grasping the subject matter in this course.

## EXERCISE 1 - COMMAND RE-DIRECTION and PIPING

Using the symbol "**<**" uses a file as standard input, "**>**" redirects the standard output to another file, and "**>>**" appends the output of a command to a file's contents ( it will create a new file if the file does not exist ).

1. List out the contents of your home directory, but rather than having the files display on the screen redirect them to a file.

```
$ ls –la

$ ls –la > file.out

$ more file.out
```

2. Append some more info to the same file

```
$ find front_porch >> file.out

$ more file.out
```

3. Using input-redirection and the mail command, mail yourself a copy of the file created above

```
$  mail –s “Linux lab file”  your.name@noaa.gov  <  file.out
```

4. Let’s pipe

```
$ env

$ env | more

$ env | grep HOME

$ ls -la

$ ls -la | more

$ ls –la | grep “bash”
```

**5.** Altogether now

```
$ ls -l   |  grep “bash”  > bash_list

$ cat bash_list


$ file | grep “directory” > bash_list

$ cat bash_list


$ ls -l   |  grep “bash”  >> bash_list

$ cat bash_list | more

$ more bash_list


$ cd $HOME ; cd fr*/li*/k*; pwd ; ls | more ; ls .. > file.out; ls ../.. >> file.out
```

## EXERCISE 2 - aliases

1. At times you may want to create your own shortcuts to commands, especially commands that have long or complicated components. Remember however not to make an alias name the same as a command that already exists on the system. The syntax to create an alias on the commandline is:

**_alias_ alias_name=value**

a. $ alias # list preset aliases

b. $ ***alias*** **var="echo "** # Use single or double quotes if there is # WHITE SPACE in the alias string.

c. $ **var** # You should get a command not found error

d. $ ***alias*** **var**

e. $ ***var $HOME***

f. $ **who**

g. $ **whoami**

h. $ **alias who="whoami"** # here we aliases a real command. It works, # but can becomes confusing and cause you # problems.

i. $ **alias**

j. $ **who**

k. $ **\who** # You can temporarily <u>just for one operation</u>, # enter a **BACKSLASH (\)** in front of the alias # to ignore the alias and use the real ommand.

k. $ **unalias who**

l. $ **alias**

## EXERCISE 3 - COMMAND RE-DIRECTION and PIPING

1. Now let's talk variables. You may not set any in the everyday activity and enter Linux commands, but there are many already set up for you at login time via your login scripts. Furthermore the real power of Linux/UNIX is harnessed when you can write shell scripts, and in doing so variables will save you a lot of work down the road.

a. $ ***echo* $LOGNAME**

b. $ ***echo* $HOME**

c. $ ***echo* $SHELL**

c. $ ***env***

2. Set and test your own local variable

a. $ ***day*=mon** # Set a local variable in the Parent

b. $ ***echo* $day**

c. $ ***sh*** # forked another sh child process

d. $ ***echo* $day** # Why?

e. $ **day=tue** # set locally in the child process

f. $ ***echo* $day**

g. $ ***exit*** # return back to the parent process

h. $ ***echo* $day** # Is the Parent affected by the Child?

3. Let’s repeat our little experiment with a global variable

a. $ ***export* DAY=mon** # Set a global variable in Parent

b. $ ***echo* $DAY; echo $day**

c. $ ***sh*** # forked another (new) sh child process

d. $ ***print* $DAY ; print $day** # Child inherits GLOBAL variables only

h. $ ***export* DAY=tue** # set locally in the child process

i. $ ***echo* $DAY**

j $ ***exit*** # return back to the parent process

l. $ **echo $DAY ; echo $day** # Was the Parent affected by the Child?

## EXERCISE 4 – Shell Scripts

Purpose: This exercise is designed to show how to create a simple shell script and execute it.

1. Rather than using a text editor at this point we'll create a script by using the 'echo' command:

a. $ ***echo* pwd >  myprog**
b. $ ***echo* whoami >> myprog**
c. $ ***echo* 'who' >> myprog**
d. $ ***echo* 'echo $SHELL' >> myprog**
e. $ ***echo* 'echo $HOME' >> myprog**

These commands should look somewhat familiar to you. The arrows are called redirection and we'll go over that in a latter module. For now we are using them to write to a new file that will by called "myprog". Once all the above commands are entered you have a shell script – that is a file which contains shell commads. See for yourself :

f. $ cat myprog

Now it's good practice to identify which shell will act as the interpreter for the script. This is done by placing the full path to the shell (eg. *#!/usr/bin/sh* ) as the first line of the script file. It is not a requirement however, as you'll see. Your myprog script will run just fine without it. Without the shell identified in the script itself, it simply defaults to using the shell that the script is run from – meaning since you are already running a bash shell session, your script will use the bash shell to read and interpret the shell commands in the script and then pass them to the operating system.

As you see a shell script is just a text file whose contents happen to be shell commands. The one requirement we have to meet in order to process our script is to let the shell know that myprog is not just an ordinary file but rather one that can be 'executed' . We do this by giving it 'execute permission' . We'll talk all about file permissions in detail in another module, but for now …

g. $ **chmod +x myprog**

Okay – ready – let’s run this puppy …

h. $**. /myprog**
(that’s dot space slash myprog)

Yeah. But your’re probably wondering about the dot business. Again more to come, but it has to do with how the shell finds the actual command programs to run for you. Try this :

i. $ **pwd**
j. $ **echo $PATH**

Again, PATH is a variable. It hold the value of locations the shell will look for commands on the system. You may notice that it does not have the value of your current directory (pwd = present working directory). So we have to inform the shell that your script is in the current directory – and that is what the dot and slash do.

## END EXERCISES

## End

**This is the end of this module. At this time you should proceed to module 5.**