Learning About UNIX-GNU/Linux

Module 3: Working Effectively

The *stderr* Output Stream

- Most utilities by default direct their output to the `stdout` stream
- If not given an argument most utilities read from the `stdin` stream.
- As discussed in Module 2, here the `who` command directs its output to `wc` with a pipe:

```
[you@faraday you]$ who | wc -l
11
[you@faraday you]$ _
```

- The `who` command does not recognise a `-x` flag:

```
[you@faraday you]$ who -x
who: invalid option -- x
Try `who --help' for more information.
[you@faraday you]$ who -x | wc -l
who: invalid option -- x
Try `who --help' for more information.
      0
[you@faraday you]$ _
```

  - When the illegal form of `who` is piped to `wc`, the 2 line error message is displayed.
  - The next line is the number of lines given `wc -l`, which was zero since `who` did not produce any regular output but only generated an end of file.
  - Clearly, the error message was not piped to `wc -l` since we can see it.
  - Error messages are output to a special output stream called `stderr`.

- There are numbers associated with the three input/output streams:

Stream	Number
stdin	0
stdout	1
stderr	2

- Just as the greater-than symbol > directs `stdout` to a file, you can direct `stderr` to a file with 2>:

```
[you@faraday you]$ who -x 2> errfile | wc -l
      0
[you@faraday you]$ cat errfile
who: invalid option -- x
Try `who --help' for more information.
[you@faraday you]$ _
```

- A special incantation merges `stderr` with `stdout`:

```
[you@faraday you]$ who -x 2>&1 errfile | wc -l
      2
[you@faraday you]$ _
```

  - This turns out to be amazingly useful. Old timers were really happy when this was added to the shell.

Top

Controlling Jobs With the Shell

- To run a command that takes a long time to finish without tying up your terminal window, put it in the *background* by ending the command with an ampersand **&**. The command will give you some information about the program you are running and give you back your shell prompt.

```
[you@faraday you]$ long_running_command &
[1] 12345
[you@faraday you]$ _
```

  - This is also the way to invoke a program, such as `netscape` or `emacs` in an X-window environment, that use their own window.
    - X-windows is discussed in Module 4.
  - The number in the square brackets is the *job number* assigned by the shell
  - The other number is the *process identification number* (`pid`) assigned by the kernel.
  - If the program produces output, you will want to direct it to a file with >
  - If the program requires input, you will want to prepare it in a file and feed it to the program with <
- You may see the jobs that are running with the `jobs` command:

```
[you@faraday you]$ jobs
[1]+ Running long_running_command &
[you@faraday you]$ _
```

  - You can list all your processes with `ps` as another way of finding the jobs you are running:

```
[you@faraday you]$ ps
   PID TTY          TIME CMD
12340 pts/0    00:00:00 bash
12345 pts/0    00:01:23 long_running_command
13589 pts/0    00:00:00 ps
[you@faraday you]$ _
```

- The first column is the *process identification number* assigned by the kernel.
- The second column is the device ("teletype") that invoked the command.
- The third column is the cpu time used by the process in hours:minutes:seconds.
- The fourth column is the name of the command.
- `ps` labels its columns by default. This makes it a *chatterbox* compared to many UNIX/Linux programs, such as `who`.

- You can kill a background job with `kill %n` where `n` it the job number assigned by the shell.

```
[you@faraday you]$ kill %1
[you@faraday you]$ _
```

  - You can also kill the job by giving `kill` the process identification number *pid* assigned by the kernel

```
[you@faraday you]$ kill 12345
[you@faraday you]$ _
```

  - If you kill your login shell you will be logged out.
- Say you execute a command, such as `long_running_command`, in the foreground. Then you do not get your shell prompt back.

```
[you@faraday you]$ long_running_command
_
```

  If you then wish to place it in the *background*, you first stop the job with `Ctrl-Z` and then place it in the background with `bg`

```
^z
[2]+ Stopped long_running_command
[you@faraday you]$ bg
[2]+ long_running_command &
[you@faraday you]$ _
```

- When you log out, by default all jobs that you are running will be terminated. The `nohup` command makes the process immune to being killed when you "hang up" i.e. log out:

```
[you@faraday you]$ nohup other_long_running_command &
[3] 12571
[you@faraday you]$ _
```

Top

Shell Variables, Functions and Aliases

- A few parts ot this section has some discussion that only applies to the `bash` shell.
  - In most of those cases, `tcsh` has the same functionality with slightly different syntax.
- The shell maintains a list of variables, functions and aliases that have been defined.
  - The variable `$TERM` identifies the type of terminal being used
  - Its value may be seen with `echo`

```
[you@faraday you]$ echo $TERM
xterm
[you@faraday you]$ _
```

- By convention, variables names are all upper-case, although the convention is not required.

- You assign a new variable name with:

```
[you@faraday you]$ myvariable='hi sailor'
[you@faraday you]$ echo $myvariable
hi sailor
[you@faraday you]$ _
```

  - Note that when defining the variable, the dollar sign `$` is not part of the name.
  - There are no spaces around the equal sign `=`
  - For `tcsh` the equivalent command is: `set myvariable='hi sailor'`

- The primary shell prompt is defined by a variable `$PS1`

```
[you@faraday you]$ echo $PS1
[\u@\h \W]\$
[you@faraday you]$ _
```

  - The variables names `\u` etc. are certainly obscure.
  - The variable names are documented in the man page for `bash`.
  - You may change the prompt to anything that you wish:

```
[you@faraday you]$ PS1='hi sailor: '
hi sailor: _
```

- The shell is just another program. You can invoke a shell from the shell.
  - Here we spawn a shell from the login shell. It is not totally obvious that you are no longer running your login shell. But if you exit the spawned shell you get back to your login shell:

```
[you@faraday you]$ bash
[you@faraday you]$ exit
[you@faraday you]$ _
```

    - You can spawn any shell from any other shell in a similar way:

```
[you@faraday you]$ tcsh
[you@faraday ~]$ exit
[you@faraday you]$ _
```

  - By default, variables are defined only for the current shell:

```
[you@faraday you]$ myvariable='hi sailor'
[you@faraday you]$ echo $myvariable
hi sailor
[you@faraday you]$ bash
[you@faraday you]$ echo $myvariable

[you@faraday you]$ exit
[you@faraday you]$ echo $myvaraible
hi sailor
[you@faraday you]$ _
```

- You can tell the shell to export a variable to all sub-shells and processes with `export`:

```
[you@faraday you]$ other_variable='bye sailor'
[you@faraday you]$ export other_variable
[you@faraday you]$ echo $other_variable
bye sailor
[you@faraday you]$ bash
[you@faraday you]$ echo $other_variable
bye sailor
[you@faraday you]$ exit
[you@faraday you]$ echo $other_varaible
bye sailor
[you@faraday you]$ _
```

  - For `tcsh` to define a variable that will be known to all sub-shells, define it with: `setenv other_variable 'bye sailor'`
    - Note there is no equal sign in the above.

- The shell variable `$PATH` defines the path used to find commands to be executed:

```
[you@faraday you]$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:.
[you@faraday you]$ _
```

  - The directories are searched in the order in which they are listed.
  - The fields are separated by colons `:` just like the `/etc/passwd` file. This is common in UNIX/Linux.
  - The directory indicated by a dot `.` stands for the present working directory.
    - Some environments do not include the present working directory in the path by default.
    - If it is included, it should *always* be last to prevent a "trojan horse" attack.
  - You can append a directory to the path with:

```
[you@faraday you]$ PATH=$PATH:/some/directory
[you@faraday you]$ _
```

  - The `which` command identifies where a program is located:

```
[you@faraday you]$ which netscape
/usr/bin/netscape
[you@faraday you]$ _
```

- You can also define functions. For example, the following sequence of commands is executed often by all users:

```
[you@faraday you]$ cd some_directory
[you@faraday you]$ ls
```

  - You can execute the same sequence in a single line by separating the commands with a semi-colon `;`

```
[you@faraday you]$ cd some_directory; ls
```

  - You can define a function `chd` that rolls these two commands into one:

```
[you@faraday you]$ function chd()
> { cd $1; ls; }
[you@faraday you]$ chd some_directory
```

    - `$1` refers to the first argument given to the function.
    - If you invoke `chd` with no arguments, it will `cd` to your home directory and list its contents.
    - A function can have multiple arguments, named as `$1`, `$2` etc.
  - `tcsh` does not have functions.

- You can form an *alias* to customise a command.

```
[you@faraday you]$ alias foo='pwd'
[you@faraday you]$ foo
/home/you
[you@faraday you]$ _
```

  - As with shell variables, there are no spaces around the equal sign =
  - You can alias a command to itself. For example, the `-i` flag to `rm` asks for confirmation before removing a file. You can make this the default behavior for `rm` with:

```
[you@faraday you]$ alias rm='rm -i'
[you@faraday you]$ rm some_file
rm: remove `some_file'? _
```

    - If you type **y** the file will be removed.
    - Typing anything else or just pressing `Enter` will keep `some_file` from being removed.
  - For the `tcsh` the alias command is: `alias rm 'rm -i'`
    - Note that there is no equal sign in the above.
- You can customise the behavior of your *login* shell by creating a file `.bash_profile` in your home directory and defining in it any variables, functions and aliases that you wish. Such a file might look like:

```
[you@faraday you]$ cat .bash_profile
# Local aliases

alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'

function chd() { cd $1; lc; } # define a function

export MY_CIRCUS_NAME='Bozo the Clown'

[you@faraday you]$ _
```

  - Lines that begin with a sharp sign # are comments
  - A sharp sign # that appears elsewhere in a line begins a comment that extends to the end of the line.
  - Empty lines are ignored.
  - The line:
    `export MY_CIRCUS_NAME='Bozo the Clown'`
    is a shorthand way of writing:
    `MY_CIRCUS_NAME='Bozo the Clown'; export MY_CIRCUS_NAME`
  - Above we made a distinction between your login shell and any shells that you spawn from it. If the file `~/.bashrc` exists in your home directory, it is executed for all shells *except* your login shell.
    - The `rc` in the file name is a relic of ancient IBM mainframes, which had run command files named `runcomm` or `rc` for short. UNIX/Linux configuration involves many files and directories with an `rc` in their name.
    - For `tcsh` users, the file `~/.tcshrc` is read by both login and non-login shells.
  - The system file `/etc/profile` is executed for all login shells.
  - It is often helpful to see other user's configuration files to see how to set up your own:

```
[you@faraday you]$ more ~some_guru/.bash_profile
```

Top

Shell Scripts

- For simple situations, you saw in Module 2 that the shell's history mechanism can save you much typing. For more complicated or often-repeated tasks, this may not be enough.
- You can create a text file containing shell commands to be executed. Such files are called *shell scripts*.
  - The `.bash_profile` file is a shell script.
- You alert the system to invoked a shell to run the contents of the file by beginning the file with the line:

```
#!/bin/bash
```

  - The `.bash_profile` script does not begin by invoking `/bin/bash` to run it, since we want it to be invoked by the shell that called it.
  - Other programs can be used in similar scripts. For example, programs written in Perl usually begin with:

```
#!/usr/bin/perl
```

- You can then put in any shell commands that you wish:

```
#!/bin/bash

# script to output the number of lines in the
# lines in the password file

echo -n 'The number of lines in the password file is: '
cat /etc/passwd |
wc -l
```

  - The `#!` in the *first* line is an exception to the rule that lines that begin with a sharp sign # are a comment.
  - The history of this exception is ghastly!
- You then need to make the file executable:

```
[you@faraday you]$ chmod +x file_name
```

  - `chmod` stands for *change mode*
  - There are many modes to a file or directory, all of which can be controlled with `chmod`
  - `chmod` is discussed more fully in Module 4.
- You can then execute the file:
  - If your present working directory is included in your path and no executable file with the same name appears in earlier directories in your path, then just name the file:

```
[you@faraday you]$ file_name
The number of lines in the password file is: 2786
[you@faraday you]$ _
```

  - You can *always* execute the desired file with:

```
[you@faraday you]$ ./file_name
The number of lines in the password file is: 2786
[you@faraday you]$ _
```

    - Recall that the period `.` refers to the present working directory.

Top

Printing

- Each flavor of UNIX/Linux has variations on how to do printing. We shall discuss modern Linux implementations
  - We are describing the *LPRng* software, which is available from: http://www.lprng.com/.
    - Many Linux distributions include *LPRng*.
- You may determine which printers are currently accepting requests with `lpstat -a`

```
[you@faraday you]$ lpstat -a
hp2100 accepting requests since 2002-04-27-13:19:56.892
hp4mp accepting requests since 2002-04-27-13:19:56.910
hp4050n accepting requests since 2002-04-27-13:19:56.901
hpVsi accepting requests since 2002-04-27-13:19:56.919
[you@faraday you]$ _
```

  - `lpstat` means "line printer status." I haven't seen an actual line printer in a long time.
- The shell variable `$LPDEST` determines which printer is the current print destination.

```
[you@faraday you]$ echo $LPDEST
hpVsi
[you@faraday you]$ _
```

- You may send a file to the printer with `lp` or `lpr`

```
[you@faraday you]$ lp some_file
request id is you@faraday+56
[you@faraday you]$ _
```

  - The commands `lp` and `lpr` are synonyms
  - The number (`56` in this case) is the *request id* assigned by the print spooler.
  - As a well-behaved UNIX/Linux program, if `lp` is not given an argument it prints from `stdin`.
- Many installations, like ours, use *PostScript* as the language for its printers.
  - If the print spooler is given a PostScript file, it is passed unchanged to the printer.
  - If the spooler is given a text file, it invokes a filter such as `a2ps` to convert it to PostScript.
  - For non-text files, many utilities exist to produce a PostScript file and/or convert to PostScript and send it directly to the printer.
    - For graphics files `display` and `gimp` can deal with a wide variety of formats.
- Many flavors of UNIX/Linux have a `-t` option to the `man` command to typeset the page for printing.

```
[you@faraday you]$ man -t mkdir
```

  - For many modern Linux distributions the flag produces PostScript which can then be sent to the printer:

```
[you@faraday you]$ man -t mkdir | lp
request-id is you@faraday+57
[you@faraday you]$ _
```

  - For some other flavors of UNIX/Linux, the flag typesets the page *and* sends it to the printer:

```
[you@some_machine you]$ man -t mkdir
request-id is you@some_machine+88
[you@faraday you]$ _
```

  - The man page for `man` should document how your the command works for your system.

- You can monitor the job with `lpq`

```
[you@faraday you]$ lpq
Printer: hpVsi@faraday 'Printer in Room 126'
 Queue: 1 printable job
 Server: pid 25321 active
 Unspooler: pid 25322 active
 Status: printing 'harrison@faraday+56' starting OF 'ofhp' at 13:45:36.743
 Rank   Owner/ID                  Class Job Files                 Size Time
active harrison@faraday+56         A    56  some_file              377 13:45:36
[you@faraday you]$ _
```

  - If there is more than one job spooled for the printer, they will all be listed.
- To cancel a printing job use `cancel`:

```
[you@faraday you]$ cancel -P hpVsi 56
```

  - The `-P` option is required to identify the printer.
  - The `56` is the request id assigned by the print spooler.

Top

Remote Access

- You can access most UNIX/Linux systems from anywhere on the Internet.
- `telnet` is the best-known program to log in to a UNIX/Linux box remotely
  - I strongly recommend you not use telnet
    - When you give your password, it is sent as clear text over the network. Any "packet sniffer" can grab it.
    - It is a "roach motel": full of bugs.
- The "secure shell" `ssh` encrypts the password before sending it to the UNIX/Linux machine, which then decrypts it.
  - This is much more secure than telnet.
    - For Windoze machines, the `PuTTY` program works very well. It is free and available at http://www.chiark.greenend.org.uk/~sgtatham/putty/
  - There are also secure ways to copy files from one machine to another using the same technology called `scp` and `sftp`.
    - You should consider using `scp` or `sftp` instead of the file transfer protocol program `ftp`.
  - Many system administrators very much want to remove `telnet` and `ftp` from their computers, but users object too strenuously.

Top

Regular Expressions

- *Regular expressions*, used in the name of the command `grep` for example, pervade UNIX/Linux and its utilities.
- It takes a particular kind of geek to be able to remember *all* of the sometimes very complex syntaxes of all the possible regular expressions.
- All UNIX/Linux geeks are aware of the basics of regular expressions that we will discuss here.
- The caret `^` stands for the beginning of a line.
  - All first and second year accounts on Faraday begin with the letter x. Thus to get all of these accounts out of the password file but not match the letter x anywhere other than at the beginning of each line use:

```
[you@faraday you]$ grep '^x' /etc/passwd
```

- To match a literal caret in a file precede it with a backslash:

```
[you@faraday you]$ grep '\^' somefile
```

- This rule applies to other regular expressions: precede it by a backslash to turn off its special meaning.

- The dollar sign **$** stands for the end of a line. Thus we could get all `tcsh` users out of the password file with:

```
[you@faraday you]$ grep 'tcsh$' /etc/passwd
```

- You can get all empty lines from a file with:

```
[you@faraday you]$ grep '^$' somefile
```

- The period **.** matches any single character. Thus to extract all the lines of a file containing exactly three characters use:

```
[you@faraday you]$ grep '^...$' somefile
```

- The asterisk ***** matches zero or more occurrences of the *preceding* character. Thus two extract all lines that contain one or more of the letter `X` in a row use:

```
[you@faraday you]$ grep 'XX*' somefile
```

- To match all lines that contain two `X` letters with anything at all between them, including nothing:

```
[you@faraday you]$ grep 'X.*X' somefile
```

- To match all lines that contain two `X` letters with one character or more between them:

```
[you@faraday you]$ grep 'X..*X' somefile
```

- Regular expressions are greedy: they always match the longest matching pattern. Thus the following matches the entire line in a file:

```
[you@faraday you]$ grep '.*' somefile
```

- Square brackets `[ ]` matches any of the characters between the brackets. Thus to extract all lines containing either `the` or `The` in a file:

```
[you@faraday you]$ grep '[Tt]he' somefile
```

- The above will also match `There`, `bother`, `their`, etc.
- Ranges can be specified. Thus `[0123456789]` is the same as `[0-9]`
- To specify all upper case letters use `[A-Z]`
- To specify all letters, upper case and lower case, use `[A-Za-z]`
- A caret *just inside* the left bracket negates the sense of the match. Thus `[^A-Z]` matches everything except an upper case letter.

Top

The Stream Editor sed

- Much of the syntax is derived from the original UNIX editor `ed`
  - I know a couple of people who still use `ed` regularly, but they are widely regarded as dinosaurs.
- `sed` is not necessarily the most used UNIX/Linus utility.
  - Here we use it to illustrate the principles of many utilities.
- The output from `sed` is always `stdout`.

- As with many UNIX/Linux utilities, it thinks of the world as made up of *lines*.
  - In common with many UNIX/Linux utilities, **q** means *quit*. Thus an equivalent to the `head` command, which quits after the first 10 lines of a file, is:

```
[you@faraday you]$ sed '10q' somefile
```

  - To get `sed` to quit whenever it encounters either `The` or `the` put the regular expression between slashes:

```
[you@faraday you]$ sed '/[Tt]he/q' somefile
```

- Above we pointed out that `ps` is verbose, in the sense that by default it labels the columns of its output:

```
[you@faraday you]$ ps
   PID TTY          TIME CMD
12340 pts/0    00:00:00 bash
12345 pts/0    00:01:23 long_running_command
13589 pts/0    00:00:00 ps
[you@faraday you]$ _
```

  - You may use `sed` to delete the column labels with:

```
[you@faraday you]$ ps | sed '1d'
12340 pts/0    00:00:00 bash
12345 pts/0    00:01:23 long_running_command
13589 pts/0    00:00:00 ps
[you@faraday you]$ _
```

    - You may similarly delete any line by giving its line number.
- You may substitute strings for other strings using `s/from/to/`. Thus to change occurrences of either `The` or `the` to `Das` in each line of the file:

```
[you@faraday you]$ sed 's/[Tt]he/Das/' somefile
```

  - By default, `sed` only replaces the first occurrence in each line. To replace all occurrences in each line add a *global* flag **g**:

```
[you@faraday you]$ sed 's/[Tt]he/Das/g' somefile
```

- `sed` by default always writes every line to `stdout`, whether or not it gets changed. The `-n` option tells the program not to output a line unless you tell it to. You tell `sed` to output a line with **p** (for "print"). Thus to *grep* lines containing either `The` or `the`:

```
[you@faraday you]$ sed -n '/[Tt]he/p' somefile
```

  - This has exactly the same effect as:

```
[you@faraday you]$ grep '[Tt]he' somefile
```

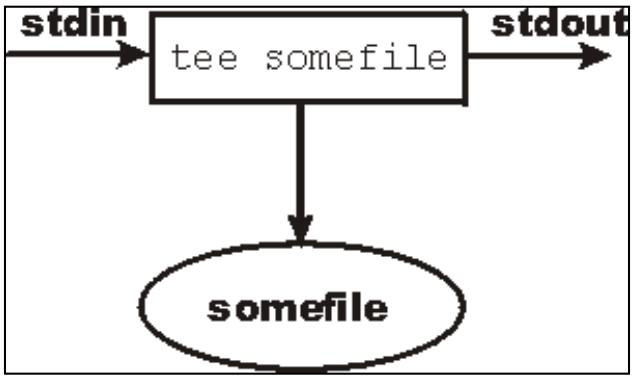
- In common with many UNIX/Linux utilities, `sed` is capable of a great number of advanced operations, which we shall not discuss here.

Top

Some Other Utilities

- The many utilities plus the ability to combine them is a major factor in making UNIX/Linux so powerful.
- Here we very briefly describe some of the most-used other utilities. The list is far from exhaustive and is only shown to give you some of the flavor of the UNIX/Linux environment.

- `sort` - investigated in Module 2.
- `grep` - discussed in Modules 2 and 3.
- `cut` - used in Module 2's Exercise 3.
- `wc` - discussed in Module 2
- `uniq` - removes any lines that are identical to the preceding line.
- `tr` - translate letters or ranges of letters into other letters or ranges of letter.
- `nl` - numbers the lines in the file
- `tee` - copies `stdin` to `stdout` unchanged, and also directs `stdin` to a file. The figure illustrates `tee`.



- `fmt` - a simple formatter for text files.
- `expand` - convert tabs to spaces.
- `awk` - a powerful "little programming language"
  - Complex enough that there is a whole book on it.
  - Named after the authors: Aho, Weinberger and Kernighan
- `perl` - a very powerful programming language
  - The learning curve to find out how to do simple things is fairly gentle.
  - The language of choice for interactive web programs, also known as *cgi scripts*.
  - Many many books are available.
  - Has largely replaced `awk` for many users.
  - Many, including the author, find its syntax baroque.
  - The name stands for "perfectly eclectic rubbish lister" among other things.

Top

Using awk Instead of cut

- In earlier Exercises you have used `cut` to cut parts out of a file.
- For ASCII text files with fields separated by one or more spaces or tabs, `awk` provides an easy alternative. Here is a text file:

```
[you@faraday you]$ cat some_file
ham      meat
spam            semi-meat
kolbassa unknown
broccoli vegetable
beer beverage
[you@faraday you]$
```

  - There are 6 spaces between `ham` and `meat`, two tabs between `spam` and `semi-meat`, and single spaces between the fields in the other three lines in the file.
  - `awk` will treat multiple instances of "whitespace" as a single field separator.
- To pick out the second field:

```
[you@faraday you]$ cat some_file |
> awk ' { print $2 } '
meat
semi-meat
unknown
vegetable
beverage
[you@faraday you]$
```

  - The fields are named `$1`, `$2`, `$3`, etc.

- You could, of course, give the file name as an argument to `awk`:

```
[you@faraday you]$ awk ' { print $2 } ' some_file
meat
semi-meat
unknown
vegetable
beverage
[you@faraday you]$
```

- It is possible to do the above operation with `sed`, but the regular expression syntax is pretty gory:

```
[you@faraday you]$ cat some_file |
> sed 's/^.*[[:space:]][[:space:]]*//'
meat
semi-meat
unknown
vegetable
beverage
[you@faraday you]$
```

- In the previous section, we mentioned the *Perl* programing language. Here is a way to have *Perl* do the same operation:

```
[you@faraday you]$ cat some_file |
> perl -lane 'print $F[1]'
meat
semi-meat
unknown
vegetable
beverage
[you@faraday you]$
```

  - *Perl* begins counting from zero. So the *second* field is `$F[1]`.
  - The options, necessary for this to work, are briefly described with:

```
[you@faraday you]$ perl --help
```


Top

Exercise 1

- Create the following shell script named `long_running_command`

```
#!/bin/bash

i=0

# Note that in the following line there
# are spaces around the square brackets
# and around the -lt.

while [ $i -lt 100000 ] # The number is 100,000
do
     # Note that 2 graves, `, and no apostrophes, '
     # are in the next line.

     i=`expr $i + 1`
done
```

    - For your convenience, we have created a version of the above script as a text file named `long.txt`. You may download it by clicking here.
        - Remember to execute: `chmod +x long.txt`
        - You may then execute: `mv long.txt long_running_command`
    - For now, the script may be treated as "magic." We will de-mystify it in the Module 4.
- Start it running in the foreground.
- Move it to the background.
- Use `ps` to verify that it is running.
    - You may see some of the commands in the shell script being executed when you use `ps`.
- Use `file` to determine what kind of file the script is.
- Kill `long_running_command`.
- Use `ps` to verify that it is no longer running.
- Start `long_running_command` again in the foreground.
- Kill it by typing an *interrupt* `Ctrl-C`.
    - Sending an interrupt to blow away some program or partial input line is a very common operation.
- Create a file `.bashrc` in your home directory if one does not already exist. Otherwise edit the existing one.
    - Have it set a variable `somevariable` to any contents that you wish.
    - Spawn a new shell from your login shell and verify that the variable is set properly.
    - Clean up the file by removing the definition of `somevariable`
    - Exit the subshell to return to your login shell.
- Create a directory in your home directory named `bin` and change into it.
    - Create a shell script in the `bin` directory with almost any name that you wish. Do not use the name of an existing command however.
    - Have it count the number of users currently logged in.
    - Verify that the script works correctly.
    - Often the same user is listed as having multiple logins. Modify the script so that it uses `awk` or `cut` to pick out the user names, `sort` the output by these names and then use `uniq` and `wc` to count the number of unique logins currently logged in.
    - Spawn a new shell.
    - Modify the `$PATH` variable so that it includes your newly created `bin` directory.
    - Change to some other directory, such as `/tmp`
    - Verify that your shell script runs just by giving its name.
    - Exit the subshell.
    - What directory are you in now?
    - What is your `$PATH`?
    - Unless you wish to keep your new shell script remove it.
        - Many users maintain their own personal `bin` of their own commands and edit `~/.bash_profile` and/or `~/.bashrc` so that the `$PATH` always includes it.

Top

Exercise 2

- Use `sed` to create a file `newpasswd` in your home directory that is a copy of `/etc/passwd` except that your login is replaced by the string `bozo`.
  - The following two ways of using `sed` are equivalent:

```
[you@faraday you]$ cat /etc/passwd |
> sed '<some_sed_commands>'
```

```
[you@faraday you]$ sed '<some_sed_commands>' /etc/passwd
```

  - Be sure that if the string matching your login appears elsewhere in the file it is not changed .
  - Be sure that if your login name is `you` that you do not change the login of a user named `you2`.
  - Use `diff` to compare the two versions of the password file.
- Recall that the fourth field in the password file is the number corresponding to the *group* (the `gid`)of the user.
  - Look in the password file to find the number of the group you are in.
  - Confirm by using `id` .
  - Look in the file `/etc/group` to discover how the `gid` is correlated with the name you see in the output from `id`.
- Create a shell script to find out how many users are in your group. Do this two different ways:
  - Use `cut`, `grep` and `wc -l` to count the number of users in your group.
  - Use only `grep` and `wc -l`. (This will be a good test of your knowledge of regular expressions.)
    - By the time your shell script is finished, it will have five different procedures in it. Use `echo` to identify which output was produced by which procedure.
      - By the end, you may need to pipe the output of your script to a pager such as `more` or `less`.
- Add to your shell script a facility to count the number of users in *each group*:
  - Use `cut` to pick out all the `gid`s, `sort` them and then use `uniq -c` to count them.
  - It will be nice to apply a further `sort -n` so that output is sorted by the number of users in each group.
    - The `sort | uniq -c | sort -n` sequence is a frequently occurring *idiom* in shell scripts.
  - Duplicate the code to count the number of users in each group, but modify it so that it sorts by the *gid* instead of the number of users in each group.
  - Duplicate *this* code and add a call to `awk` at the end to print the *gid* first followed by the number of users in that group.
    - You will want to know that when you ask `awk` to print more than one field, the fields should be separated by a comma `,`.

You, of course, know that looking at a solution is not a substitute for actually doing the exercise yourself. However, a sample script that solves the above Exercise is available here; your script may be better than this one.

Top