Learning About UNIX-GNU/Linux

Module 4: Working Even More Effectively

Quoth the Shell

- The two quoting mechanisms in the shell are apostrophes `' '` and quote marks `" "`.
  - Apostrophes do not expand any enclosed shell variables.

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

    - Apostrophes also protect any other characters expanded by the shell such as an asterisk **`*`**.
  - Quotes do expand shell variables.

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

    - This can be used to create variables out of other variables.

```
[you@faraday you]$ somevariable='hi sailor'
[you@faraday you]$ othervariable="$somevariable, want to party?"
[you@faraday you]$ echo $othervariable
hi sailor, want to party?
[you@faraday you]$ _
```

- Graves (pronounced *grahvs)* or backticks, ` `, execute any commands placed between them, in place.

```
[you@faraday you]$ echo `who | wc -l` users are logged in.
11 users are logged in.
[you@faraday you]$ _
```

Top

Shell Control Flow

- Here we discuss `bash` exclusively.
  - Even users who prefer `tcsh` as their interactive shell prefer `bash` for shell scripts.
    - Well … most such users.
- The `exit` command immediately exits the shell.
  - Giving the command to your login shell logs you out.
  - In a shell script it causes the script to exit.

```
[you@faraday you]$ more s1
#!/bin/bash

echo "I am in the script, about to call exit."
exit
echo "I am past the exit call in the script."

[you@faraday you]$ ./s1
I am in the script, about to call exit.
[you@faraday you]$ _
```

- The `for` loop allows an operation to be repeated.

```
[you@faraday you]$ for i in 1 2 3 hi sailor
> do
> echo $i
> done
1
2
3
hi
sailor
[you@faraday you]$ _
```

  - Recall that > is the secondary shell prompt. The shell has realised that more input is needed after the first line is typed, and prompts you for it.

- After you type `done` the shell knows the command is completed and executes it.
- In a shell script you would not include the > symbols.

```
[you@faraday you]$ more s2
#!/bin/bash
for i in 1 2 3 hi sailor
do
      echo $i
done
[you@faraday you]$ ./s2
1
2
3
hi
sailor
[you@faraday you]$ _
```

  - The `echo` command has been indented. This is only to make the script more readable.
  - Such indentation is very important for readability in complicated shell scripts.
- `for` can be combined with graves to get each word in a file.

```
[you@faraday you]$ more some_file
Twas brillig and
the slithy toves
[you@faraday you]$ for i in `cat some_file`
> do
> echo $i
> done
Twas
brillig
and
the
slithy
toves
[you@faraday you]$ _
```

- The `case` statement allows you to execute commands based on the value of some variable.

```
[you@faraday you]$ more s3
#!/bin/bash

for i in 1 2 3 hi sailor
do
      case $i in
               1) echo "Found a one";;
               2) echo "Found a two";;
               3) echo "Found a three";;
            'hi') echo "Found a salutation";;
        'sailor') echo "Found a seafaring man";;
      esac
done
[you@faraday you]$ ./s3
Found a one
Found a two
Found a three
Found a salutation
Found a seafaring man
[you@faraday you]$ _
```

  - Each possibility ends in double semi-colons.
  - The case ends with `esac`, which is case spelled backwards.
- The shell variable **$?** is the exit status for the previously executed command.
  - If a command succeeds it returns 0
  - If a command fails it returns 1.
    - The definitions of success and failure are given in the man page for the command.
  - For `grep` a return of 0 indicates at least one match, and a 1 indicates no matches.

```
[you@faraday you]$ grep 'Bozo the Clown' /etc/passwd
[you@faraday you]$ echo $?
1
[you@faraday you]$ grep you /etc/passwd
you:9X0efgCCSLyec:109:100:Your name:/home/you:/bin/bash
[you@faraday you]$ echo $?
0
[you@faraday you]$ $_
```

- The special file `/dev/null` is a "bit bucket" to which any output can be directed.
  - Anything sent to `/dev/null` is lost forever.
  - It can be used to suppress unwanted output from a command.

```
[you@faraday you]$ grep you /etc/passwd > /dev/null
[you@faraday you]$ _
```

  - `grep` also has an option to not output any lines but only to return a status.
- The `if` structure executes its contents if a *condition* is fulfilled.
  - If the *condition* is a command, it tests `$?`.

```
[you@faraday you]$ more s4
#!/bin/bash

if who | grep you > /dev/null
then
        echo user you is logged in
fi

[you@faraday you]$ ./s4
user you is logged in
[you@faraday you]$ _
```

- In a pipeline, $? is the status of the last command.
- The `if` is terminated with `fi`, which is *if* spelled backwards.

- You can add an `else` clause to the script.

```
[you@faraday you]$ more s5
#!/bin/bash

if who | grep bozo > /dev/null
then
        echo user bozo is logged in
else
        echo bozo is not logged in
fi

[you@faraday you]$ ./s5
bozo is not logged in
[you@faraday you]$ _
```

- The shell has a `test` mechanism for `if` conditions that are not just commands.

```
[you@faraday you]$ more s6
#!/bin/bash

for i in 1 2 3 hi sailor
do
        if test $i = 1
        then
                echo "Found a one"
        fi
done
[you@faraday you]$ ./s6
Found a one
[you@faraday you]$ _
```

- There must be spaces on both sides of the equal sign =. Otherwise the shell will think that you are assigning a value of `1` to the variable `$i`.
- Another way of writing a `test` is to use square brackets.

```
[you@faraday you]$ more s6a
#!/bin/bash

for i in 1 2 3 hi sailor
do
        if [ $i = 1 ]
        then
                echo "Found a one"
        fi
done

[you@faraday you]$ ./s6a
Found a one
[you@faraday you]$ _
```

- There *must* be a space after the left bracket `[` and a space before the right bracket `]`.
- Some people feel this second form is more readable.

- More than one test can be combined with `-o` (or) and `-a` (and).

```
[you@faraday you]$ more s7
#!/bin/bash

for i in 1 2 3 hi sailor
do
        if [ $i = 1 -o $1 = 2 -o $1 = 3 ]
        then
                echo "Found a digit"
        fi
done
[you@faraday you]$ ./s7
Found a digit
Found a digit
Found a digit
[you@faraday you]$ _
```

- You can include an `elif` (for *else if*) in the `if`.

```
[you@faraday you]$ more s8
#!/bin/bash

for i in 1 2 3 hi sailor
do
        if [ $i = 1 ]
        then
                echo "Found a one"
        elif [ $i = sailor ]
        then
                echo "Found a seafaring man"
        else
                echo "Found something else"
        fi
done

[you@faraday you]$ ./s8
Found a one
Found something else
Found something else
Found something else
Found a seafaring man
[you@faraday you]$ _
```

  - The `elif` must be followed by `then`
  - The `else` is not followed by `then`

- The `test` construct includes various file operations. For example `-f` tests if a file exists.

```
[you@faraday you]$ if [ -f /etc/passwd ]
> then
> echo the password file exists
> fi
the password file exists
[you@faraday you]$ _
```

  - An exclamation mark `!` negates the sense of the test.

```
[you@faraday you]$ if [ ! -f /etc/YouHaveBeenHacked ]
> then
> echo You have not been hacked
> fi
You have not been hacked
[you@faraday you]$ _
```

    - There must be spaces around the `!` sign.
    - For a test in which we are using an equals sign the following are equivalent:

```
[you@faraday you]$ if [ ! $var1 = $var2 ]
```

```
[you@faraday you]$ if [ $var1 != $var2 ]
```

- Here are some common file testing commands:

Test	Means
`-f`	File
`-s`	File with non-zero size
`-d`	Directory
`-r`	File is readable by the process
`-w`	File is writable by the process
`-x`	File is executable by the process

- The shell supports some limited arithmetic operations. For example `-gt` tests whether some value is greater than another value.

```
[you@faraday you]$ if [ 2 -gt 1 ]
> then
> echo 2 is greater than 1
> fi
2 is greater than 1
[you@faraday you]$ _
```

  - Here are some common arithmetic comparisons:

Comparison	Means
`-gt`	Greater than
`-ge`	Greater than or equal to
`-eq`	Equal to
`-ne`	Not equal
`-le`	Less than or equal to
`-lt`	Less than

    - Note that these constructs are for arithmetic comparisons. The equals sign **=** is used for string comparisons.
  - The `expr` command evaluates its arguments numerically.

```
[you@faraday you]$ expr 1 + 2
3
[you@faraday you]$ _
```

    - `expr` supports the following mathematical operations: `+ - * / %`
      - % is the *modulus*
      - For `*` (multiplication) beware of the shell expanding it. Putting it inside apostrophes `'*'` is always is always a good idea.
    - You can use `expr` in shell scripts to do some limited math.

```
[you@faraday you]$ more s9
#!/bin/bash

i=1
j=`expr $i + 2`
echo "Two greater than $i is $j"

[you@faraday you]$ ./s9
Two greater than 1 is 3
[you@faraday you]$ _
```

    - Note the two graves in the above script.
- The `while` construct runs until its given *condition* is false.

```
[you@faraday you]$ more s10
#!/bin/bash

i=1
while [ $i -lt 5 ]
do
        echo "$i is less than 5"
        i=`expr $i + 1`
done

[you@faraday you]$ ./s10
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
[you@faraday you]$ _
```

    - The `while` construct can run "forever" if you do not insure that at some point the condition is false.
        - You can interrupt a runaway shell script running in the foreground by typing `Ctrl-C`.
    - The `break` command breaks out of the `while`

```
[you@faraday you]$ more s11
#!/bin/bash

i=1
while [ $i -lt 5 ]
do
        echo "$i is less than 5"
        i=`expr $i + 1`
        if [ $i -eq 2 ]
        then
                break
        fi
done

[you@faraday you]$ ./s11
1 is less than 5
[you@faraday you]$ _
```

Top

Arguments to Shell Scripts

- Arguments are given the names $1, $2, etc. for the first, second and further arguments.

```
[you@faraday you]$ more s12
#!/bin/bash

echo $1
echo $2

[you@faraday you]$ ./s12 hi sailor
hi
sailor
[you@faraday you]$ _
```

  - This is the same syntax that we saw before for shell functions.
  - Arguments to shell scripts were implemented before the shell knew how to define functions.
- `shift` throws away $1, moves $2 into $1, $3 into $2, etc.

```
[you@faraday you]$ more s13
#!/bin/bash

echo $1
shift
echo $1

[you@faraday you]$ ./s13 hi sailor
hi
sailor
[you@faraday you]$ _
```

- $# is the number of arguments given to the script.

```
[you@faraday you]$ more s14
#!/bin/bash

echo $#

[you@faraday you]$ ./s14 hi sailor
2
[you@faraday you]$ _
```

  - Every call to `shift` decreases $# by one.

Top

Input to Shell Scripts

- `echo` outputs its arguments to `stdout`.
- `read` reads `stdin` into its arguments. Below, we underline what you might type.

```
[you@faraday you]$ more s15
#!/bin/bash

echo -n "Talk to me: "
read ARG
echo "You typed: $ARG"

[you@faraday you]$ ./s15
Talk to me: hi sailor
You typed: hi sailor
[you@faraday you]$ _
```

  - If `read` is given multiple arguments they are filled in from the input using "whitespace" as the delimiter:

```
[you@faraday you]$ more s15a
#!/bin/bash

echo -n "Talk to me: "
read ARG1 ARG2 ARG3
echo "First argument: $ARG1"
echo "Second argument: $ARG2"
echo "Third argument: $ARG3"

[you@faraday you]$ ./s15a
Talk to me: hi
First argument: hi
Second argument:
Third argument:
[you@faraday you]$ ./s15a
Talk to me: hi sailor
First argument: hi
Second argument: sailor
Third argument:
[you@faraday you]$ ./s15a
Talk to me: hi sailor, want to party?
First argument: hi
Second argument: sailor,
Third argument: want to party?
[you@faraday you]$ _
```

    - The use of "whitespace" as the delimiter is exactly the same as `awk`.

Sub-shells and the "dot" Command

- Shell scripts typically begin by invoking `bash` with: `#!/bin/bash`
  - Any variables, functions and aliases defined in the script are only defined for the shell that it invoked.
  - Any variables marked for `export` in the shell that called the script will be available to the script.
- If your interactive shell is `bash`, then if the script does not begin with the line invoking `bash` the shell will automatically invoke a `bash` sub-shell to run it:

```
[you@faraday you]$ more s16

thisvar='yabba dabba doo'
echo "Value of" '$thisvar is:' $thisvar

[you@faraday you]$ ./s16
Value of $thisvar is: yabba dabba doo
[you@faraday you]$ echo $thisvar

[you@faraday you]$ _
```

  - You can explicitly invoke `bash` to run the script:

```
[you@faraday you]$ bash s16
Value of $thisvar is: yabba dabba doo
[you@faraday you]$ echo $thisvar

[you@faraday you]$ _
```

  - If your interactive shell is, say, `tcsh` you can still invoke `bash` to run the script:

```
[you@faraday you]$ tcsh
[you@faraday ~]$ bash s16
Value of $thisvar is: yabba dabba doo
[you@faraday ~]$ exit
[you@faraday you]$ _
```

    - If you invoke the script from `tcsh` without naming `bash`, `tcsh` will spawn a `tcsh` sub-shell to run it. This usually gives errors for a `bash` script:

```
[you@faraday you]$ tcsh
[you@faraday ~]$ ./s16
thisvar=yabba dabba doo: Command not found.
thisvar: Undefined variable.
[you@faraday ~]$ exit
[you@faraday you]$ _
```

    - Beginning a `bash` script with `#!/bin/bash` insures that any reasonable shell, including `tcsh`, will invoke `bash` to run it.
- The special "dot" command (a period `.`) will invoke a `bash` script that does not begin with `#!/bin/bash` in the current shell instead of spawning a sub-shell:

```
[you@faraday you]$ more s16

thisvar='yabba dabba doo'
echo "Value of" '$thisvar is:' $thisvar

[you@faraday you]$ . ./s16
Value of $thisvar is: yabba dabba doo
[you@faraday you]$ echo $thisvar
yabba dabba doo
[you@faraday you]$ _
```

- Recall that the file `~/.bash_profile` is executed for your login shell, and `~/.bashrc` is executed for all other shells that you spawn. You can have all variables, functions and aliases defined in `~/.bashrc` also available to your login shell by including the following fragment in your `~/.bash_profile`:

```
if [ -f ~/.bashrc ]
then
    . ~/.bashrc
fi
```

Top

Files, Owners and Groups

- The `-l` flag to `ls` produces a long listing; the flag is a letter, not the number *one*.

```
[you@faraday you]$ lc
Directories:
some_directory

Files:
empty_file     some_file
[you@faraday you]$ ls -l
total 8
-rw-r-----    1 you      users           0 Apr 28 07:29 empty_file
drwxr-x---    2 you      users        4096 Apr 28 07:29 some_directory
-rw-r-----    1 you      users          34 Apr 28 07:30 some_file
[you@faraday you]$ _
```

  - The first line is the approximate total size of all the items in the listing in kilobytes.
  - For the other lines the first column is the *permissions* of the item
    - The first character is a `-` for a file and a `d` for a directory
    - The next character is a `r` if you have read permission for the item, a `-` if you do not.
    - The next character is a `w` if you have write permission for the item, a `-` otherwise.
    - The next character is an `x` if you have execute permission for the item, a `-` otherwise.
    - The next group of three characters define the read, write and execute permissions for members of the group named in the fourth column.

        - The next group of three characters define the permissions for other users that are not members of the group.
        - To enter a directory not only must the user have at least *read* permission for it, but also *execute* permission.
    - The second column is the *link count*, which we will discuss later.
    - The third column is the owner of the file or directory.
    - The fourth column is the group of the file or directory
    - The next column is the size of the item in bytes
    - Next is the time of last modification of the file.
    - The final column is the name of the item.
- You can empty the contents of an existing file with >.

```
[you@faraday you]$ ls -l non_empty_file
-rw-r-----    1 you      users        157k Apr 30 08:10 non_empty_file
[you@faraday you]$ > non_empty_file
[you@faraday you]$ ls -l non_empty_file
rw-r-----    1 you      users           0 Apr 30 08:10 non_empty_file
[you@faraday you]$ _
```

- You can create a new empty file with `touch`.

```
[you@faraday you]$ ls -l new_file
ls: new_file: No such file or directory
[you@faraday you]$ touch new_file
[you@faraday you]$ ls -l new_file
-rw-r-----    1 you      users           0 Apr 30 08:13 new_file
[you@faraday you]$ _
```

    - `touch` can also change the time stamp of a file.
    - You can also create a new empty file with a greater than symbol >:

```
[you@faraday you]$ > another_new_file
[you@faraday you]$ _
```

- The `chmod` command allows you to change the read/write/execute permissions of any file or directory that you own.
    - The permissions may be manipulated *symbolically* as:
      `chmod [who][do][what] filename`.
        - `[who]` is a single letter which can be:
            - `u` the user who owns the file
            - `g` member of the group
            - `o` all other users
            - `a` all users
        - `[do]` is a single symbol which can be
            - `+` turn on the permission
            - `-` turn off the permission
        - `[what]` is a single letter which can be
            - `r` read permission
            - `w` write permission
            - `x` execute permission
        - Here are some examples:

```
[you@faraday you]$ ls -l somefile
-rw-r-----    1 you      users          33 Apr 29 08:58 somefile
[you@faraday you]$ chmod o+r somefile; ls -l somefile
-rw-r--r--    1 you      users          33 Apr 29 08:58 somefile
[you@faraday you]$ chmod g+x somefile; ls -l somefile
-rw-r-xr--    1 you      users          33 Apr 29 08:58 somefile
[you@faraday you]$ chmod o-r somefile; ls -l somefile
-rw-r-x---    1 you      users          33 Apr 29 08:58 somefile
[you@faraday you]$ _
```

- The permissions may be manipulated *numerically* as:
  `chmod [d1][d2][d3] filename`

  - `[d1]`, `[d2]` and `[d3]` are each single digits controlling the permissions for the owner, group and others respectively.
  - Each digit is the `sum` of 0, 1, 2 and 4 where:
    - 0 means no permissions
    - 1 grants execute permission
    - 2 grants write permission
    - 4 grants read permission
  - Here are some examples:

```
[you@faraday you]$ ls -l somefile
-rw-r-----    1 you      users          33 Apr 29 08:58 somefile
[you@faraday you]$ chmod 644 somefile; ls -l somefile
-rw-r--r--    1 you      users          33 Apr 29 08:58 somefile
[you@faraday you]$ chmod 654 somefile; ls -l somefile
-rw-r-xr--    1 you      users          33 Apr 29 08:58 somefile
[you@faraday you]$ chmod 650 somefile; ls -l somefile
-rw-r-x---    1 you      users          33 Apr 29 08:58 somefile
[you@faraday you]$ _
```

  - Note that each possible digit corresponds to a *bit* in a binary number.

- The shell recognises the `umask` construct to control the default permissions of files and directories that you create.

  - Calling `umask` without any arguments will tell you what your current `umask` is, displayed in numerical form.

```
[you@faraday you]$ umask
026
[you@faraday you]$ _
```

    - Just as for the numerical form of `chmod` the three digits are for the owner, group and others respectively.
    - The leading zero means that the owner of a new file has all permissions.
    - The two means the members of the group have had write permissions turned off.
    - The six means others have read and write permissions turned off.
      - When you create a file, most UNIX/Linux implementations do not have the execute bit turned on for any users, including the owner.
      - When you create a directory, most UNIX/Linux implementations by default have the execute bit turned on for all users.
      - When you create a file using the *Samba* protocols from a Windoze machine, most implementations have the execute bit turned on.

- You set a `umask` numerically with:
  `umask [d1][d2][d3]`
- Calling `umask` with a `-S` flag shows the current values symbolically.

```
[you@faraday you]$ umask
026
[you@faraday you]$ umask -S
u=rwx,g=rx,o=x
[you@faraday you]$ _
```

- You may set a `umask` using the same symbolic notation as is displayed from the `umask -S` command.
- Typically a `umask` is set in your `~/.bash_profile` and/or `~/.bashrc` files.
- `umask` is not actually a separate command, but is *built-in* to the shell.

Top

The X-Window Environment

- From an X-terminal or the system console, the X-windows system provides a fully customisable windowing system.
- Ubiquitous with virtually every UNIX/Linux flavor.
- Development begun at MIT in 1984.
  - This was just after the first Macintosh and long before Windoze.
  - Released to the world in 1988. MIT formed the *X Consortium*.
  - *XFree86*, now a member of the X Consortium, is a good source for X-windows.
    - The *XFree86* web site is: http://www.xfree86.org/
- The system is very complex.
  - The same geeks who can remember all of the regular expression syntax can remember a lot of the X environment, but probably not all.
- If a file `.xsession` exists in your home directory, it is used to set up your personal environment.
  - Some environments use a file `~/.Xclients` instead of `~/.xsession`.
- A good place to start is by using somebody else's `.xsession` and customising it:

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

- One of the major complexities of X is that its nomenclature makes the documentation hard to read. When you are sitting in front of, say, an X-terminal it is called the *server*. This is the wrong name in my mind: the server to which you are logged in, such as Faraday, is the server!
- The `$DISPLAY` variable defines where X displays are sent (the so-called *server*).

```
[you@faraday you]$ echo $DISPLAY
pinncd35:0.0
[you@faraday you]$ _
```

  - When you log in, the variable is set to your login device.
  - The first part is just the name of the device.
    - For the system console it is an empty field.
  - The first `0` is the number of the so-called-server. It is almost always zero.
  - The second `0` is the *screen*, so that if the same X-terminal has multiple monitors you can direct output to the one you wish.

- If authorisations allow it, you can send an X display to another machine.
  - `xclock` generates the clock that most users set up.
  - `xclock`, and virtually all X applications, accept a `-display` option to send the output to another so-called-server.
    - Obviously X was written before the `--long_option_name` convention was established.
  - You use the `-display` option like this:

```
[you@faraday you]$ xclock -display some_machine:0.0 &
[you@faraday you]$ _
```

- A *window manager* puts the borders around windows and includes controls to exit the window, iconify it, etc.
  - The window manager should be run in the background.
  - They are fully customisable with configuration files in your home directory.
    - Each window manager does configuration differently.
  - There are many different window managers. Faraday has:
    - `mwm` - the Motif window manager.
    - `twm` - the Tab window manager (aka "Tom's window manager").
    - NCD X-terminals run a Motif-like window manager inside the terminal. When students log in to an UPSCALE NCD X-terminal we invoke this window manager for them. It is invoked with:

```
[you@faraday you]$ /usr/bin/rsh <SO_CALLED_SERVER> wm
[you@faraday you]$ _
```

      where `<SO_CALLED_SERVER>` is the `$DISPLAY` with the stuff to the right of and including the colon `:` removed.
- Further configuration uses *resources*. The system sets some of these for you. If a file `~/.Xresources` exists in your home directory it is read when you log in and its contents are merged with the system settings. As with the `.xsession` file, looking at somebody else's resource file is a good way to get started.

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

  - The program that loads the contents of the `.Xresources` file is named `xrdb`.
- The terminal program is called `xterm`. From an `xterm` window you can "pop" a new one with:

```
[you@faraday you]$ xterm &
[you@faraday you]$ _
```

- To copy and paste text:
  - "Paint" the text you wish to copy by holding down the left mouse button and moving the cursor over the text. It will be highlighted.
  - Place the cursor where you wish to paste the copied text and click the middle mouse button.
    - If you are using a two-button mouse, you can simulate the middle button by simultaneously holding down the left and right buttons.
  - Some applications use `Alt-X`, `Alt-C` and `Alt-V` as keyboard shortcuts to cut, copy, and paste respectively.
    - There are also a few applications that use the Windoze conventions of `Ctrl-X`, `Ctrl-C` and `Ctrl-V`.
- Above we saw a way to run an X program on a display other than yours. You do not want to allow this for arbitrary machines. `xhost` can disable this ability:

```
[you@faraday you]$ xhost -
access control enabled, only authorized clients can connect
[you@faraday you]$ _
```

- You can then authorise particular machines to connect with:

```
[you@faraday you]$ xhost + some_machine_name
[you@faraday you]$ _
```

- Any `xhost` command is only for your current session. You can put the commands in your `~/.xsession` to execute them on login.

- Any characters that you type on the keyboard can be captured by a snooper on the network. You can disable this by making your keyboard secure.
  - Should only be used to execute a few sensitive commands, since it disables things like the ability to move a window.
  - In the `xterm` window hold down the `Ctrl` key and then hold down the left mouse button.
  - Scroll down to `Secure Keyboard` and release the mouse button.
    - The foreground/background colors will reverse to indicate that you are running in secure mode.
  - Type the sensitive information as usual.
  - Unsecure the keyboard by again holding down the `Ctrl` key, then the left mouse button and again scroll down to the (now checked) `Secure Keyboard` item. Release the mouse button.
    - The foreground/background colors will be restored to their previous values.
- Finally, if you are on a fairly modern Linux implementation you may wish to explore the desktop environments.
  - To execute the Gnome desktop on login, in your `~/.xsession` insert the line:

```
exec gnome-session
```

    - Then log out and log back in.
  - To execute the "Kommon Desktop Environment" KDE, in your `~/.xsession` insert the line:

```
exec startkde
```

    - Log out and log back in.
  - The author of this document has begun to use Gnome at home with RedHat Linux 7.2. I found all previous versions to not be worth the trouble. Your mileage may vary.

Top

Exercise 1

- If you have not already done so, create a sub-directory of your home directory named `bin`.
- If you have not already done so, modify your `$PATH` so that includes your personal `bin`,
- In your personal `bin` directory write a shell script `myls` that when given no arguments does an `ls` command on your present working directory and exits.
  - You will want an explicit check for no arguments and an explicit call to `exit` for what follows.
  - Verify that the script works correctly when invoked from a directory other than where the script is.
- Add a part after the above section that assembles all of its arguments as a single variable, say `$FILENAMES`, and executes `ls` with the variable as its argument. Make sure that each name is separated from the others by at least one space. Check that it works as you expect.

- Add a check that a file corresponding to each argument exists. If not have it output a message and exit.

```
[you@faraday you]$ ./myls does_not_exist
does_not_exist: No such file found
[you@faraday you]$ _
```

  - This means that you can not give directory names to `myls`. You are free to allow directory names also by extending the test.
- Well-behaved UNIX/Linux commands use the name of the program at the beginning of any error message. For example:

```
[you@faraday you]$ ls does_not_exist
ls: does_not_exist: No such file or directory
[you@faraday you]$ _
```

  - This can identify where a failure occurred when a sequence of commands in a pipeline have failed.
- The shell variable `$0` is the name of the shell script. Use it in the error message generated when an argument does not correspond to an existing file.
- If you invoke `myls` with either an absolute or relative path, that will be included in `$0`. Use the `basename` program to define a variable `$PROGNAME` that includes only the name of the script file, and use `$PROGNAME` in the error message.
- Well-behaved UNIX/Linux commands like `ls` exit with status 1 upon failure. Change the `exit` statement for non-existent files in `myls` to `exit 1` to make it similarly well-behaved.
- Well-behaved programs generate error messages to `stderr`, but `echo` outputs to `stdout`. Add code to direct `echo`'s output to `stderr`.
  - This will involve an incantation similar but not identical to one discussed in the first Section of Module 3.
- Towards the top of the script, define a variable `$LS` that is defined as `'/bin/ls'`. Everywhere that the script calls `ls` substitute `$LS`
  - Setting `$LS` to the absolute path to `ls` means that any aliases for `ls` that have been defined will not be used.
- Add code so that if `$1` is `-l` (the lower case letter, not the number), `$LS` is set to `ls -l`, so it produces a long listing. Check that it works.
  - Make sure that if `-l` is the only argument, that it executes a long listing of the present working directory.
  - Make sure that any further arguments are still processed as file names for `$LS`.
- The `-m` flag to `ls` produces a short listing. Change the code that processes the `-l` flag so it processes a `-m` flag instead.
  - I have no idea what the `m` is supposed to stand for!
- Modify the script so it will accept *either* an `-l` flag or a `-m` option.
- When you have all of this script working, you can consider yourself an accomplished shell programmer!

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

Exercise 2

- Create a directory in your home directory named `yikes`.
- Try to determine its permissions with `ls -l yikes`.
- Read the man page for `ls` to learn how to view the permissions of the new directory. Verify that your new

understanding is correct.
- Make the directory readable, writable, and executable for *all* users.
- Verify that your changes worked.
- **Immediately** either change the permissions on the directory to something more reasonable or remove it entirely.
- Execute `umask` with no arguments (but possibly with a `-S` option if you wish) to verify its current value.
- Spawn a subshell of your login shell.
- In the *UPSCALE* environment we set a `umask` for student users such that other students can not read or write each other's files or directories, but users not in the student group (like me) can read them by default. Set a `umask` that achieves this functionality for you, members of your group, and other users.
- Create a new file and verify that the permissions are what you expect.
- Exit the subshell and verify the umask for your login shell is unchanged.