Linux Essentials – Module 2

EXAMINING FILES

MODULE OBJECTIVE

- Understand the directory structure, navigate between directories, view file contents

LESSON OBJECTIVES

- Define “**UNIX File System**”, “**File Hierarchy**” and “**file system**”
- Use the commands to navigate the file systems and to list files and directories
- Examine text file contents with the commands “**head**”, “**tail**”, “**cat**”, “**grep**”, “**more**” and “**file**”

OVERVIEW

What is a **File System**? A file system is the layout of files on the hard disk. It let's UNIX know which parts of the disk are already used, and which aren't. (It's like an index to a large filing system or a card catalog to a library.) The way this organization is structured is called a **File Hierarchy**. There are several key functions performed by UNIX file systems:

Efficiently use the space available on your hard drive.

Catalog all the files on your hard drive so that retrieval is fast and reliable.

Provide methods for performing basic file operations, such as delete, rename, copy, and move.

Provide a data structure that allows the computer to boot off the file system.

What is a **file**? That depends on how you look at it. In UNIX, everything is a file. Files you are familiar with are text, data, and programs, but your hard drive is also a file, your serial and parallel ports, even the bit bucket.

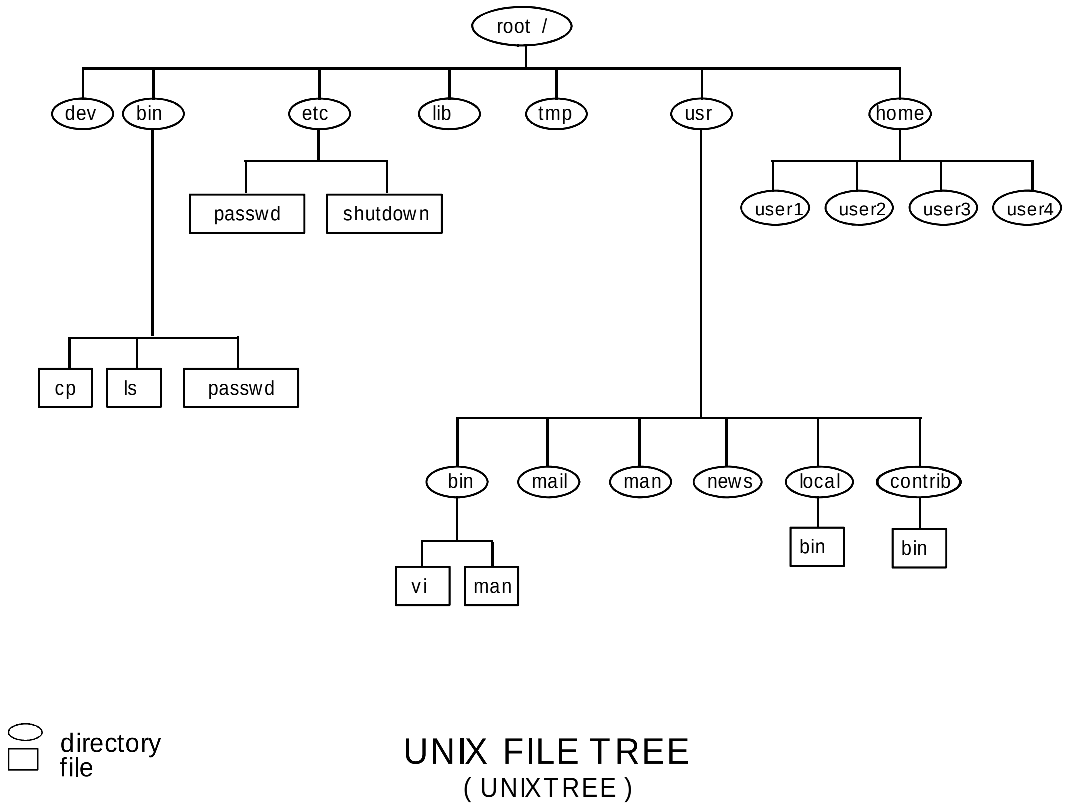
The OS views the file as nothing more that a sequence of bytes on the disk.

A program may view the file as the text for a word processing document or as the records of a database.

You probably view a file as a collection of data or a program that can be called by name.

DIRECTORY STRUCTURE

Linux/UNIX use a common directory structure, with a main directory (root) and then several subdirectories. As opposed to WINDOWS(DOS)There are two important differences. First, while DOS uses the backslash "\", Unix uses a forward slash "/". Also, while DOS differentiates disk (hard, floppy, or virtual) by letters (e.g., A: , B:, C:, etc.), Unix does not. Instead, Unix deals in "file systems." This is really transparent to the user. The end result is that all Unix pathnames merely begin with a leading / and do not have a drive letter designator to prefix it.



UNIX FILE TREE
(UNIXTREE)

FILE SYSTEM HIERARCHY

The Linux File System (or File Hierarchy) includes file systems, directories, and sub-directories. The Linux File System starts with a top level of “/” (also called root), then branches with directories and sub-directories (normally shown as an inverted tree).
A Linux file system resides on a storage device (hard disk or a partition of a hard disk).

Every Linux/UNIX system must have at least one file system for root and typically have at least a few other file systems.The top level of the Linux/UNIX file structure (/) is known as the root directory and always has a certain set of sub-directories, including sbin, dev, etc, lib, tmp, and home.

Directories are hierarchically organized. That is a directory has a parent directory “above” and may also have subdirectories “below” (child directories). Similarly, each subdirectory can contain other files and also can have more subdirectories. Because they are hierarchically organized, directories provide a logical way to organize files. With the help of directories, you can organize your files into manageable, logically related groups. For example, if you have several files for each of several different projects, you can create a directory for each project and store all the files for each project in the appropriate directory.

The following is a brief description of the above files:

/dev	Holds the device drivers. Device drivers allow the terminal to display data, the keyboard to enter data, and the disk to store data. Although UNIX looks at these drivers as ordinary files, no data is stored in these files. Rather they act as a pathway to the device.
/bin, /sbin	Holds the executable (C-compiled) programs necessary for basic Linux system operation and file manipulation. In early UNIX the bin directory held all executable binaries. As more and more binaries were developed, the bin directory was split into multiple parts (/bin, /usr/bin, /usr/local/bin, etc).
/etc	Holds the system configuration files and system administration commands. System administration can be complex. Managing user accounts, file systems, security, device drivers, hardware configuration, and more.
/lib	A central storage place for function and procedural libraries. These specific executables are included with specific programs, allowing features and capabilities otherwise unavailable. The idea is that if programs want to include certain features, they can just reference that utility in the UNIX library.

/var	Contains files that may dynamically change size, ie. mail and syslog files that are constantly changing in size. Contains the user scratch space /var/tmp.
/home	This contains one sub-directory for each user account. Every user on the UNIX system should have his or her own account. You have complete control of your account and are responsible for files and sub-directories created in your account. Normally, home directories contain only personal files and programs, **not** applications or data accessed by other users. This is because of the permission settings on home directories.
/tmp	This is a scratch space for the UNIX system. User scratch space is located in /var/tmp.

FILE SYSTEM SPACE

Before a file system can be recognized and accessed by Linux, it must be mounted to a mount point (a directory name). This is usually done automatically at system boot time and is controlled by your System Administrator.

It is important for users to realize that file systems reside on storage devices that contain limited space. If you are going to download large files, develop new programs or retrieve files stored on tape, you have a responsibility to monitor the system and prevent any action that would run the system out of space.

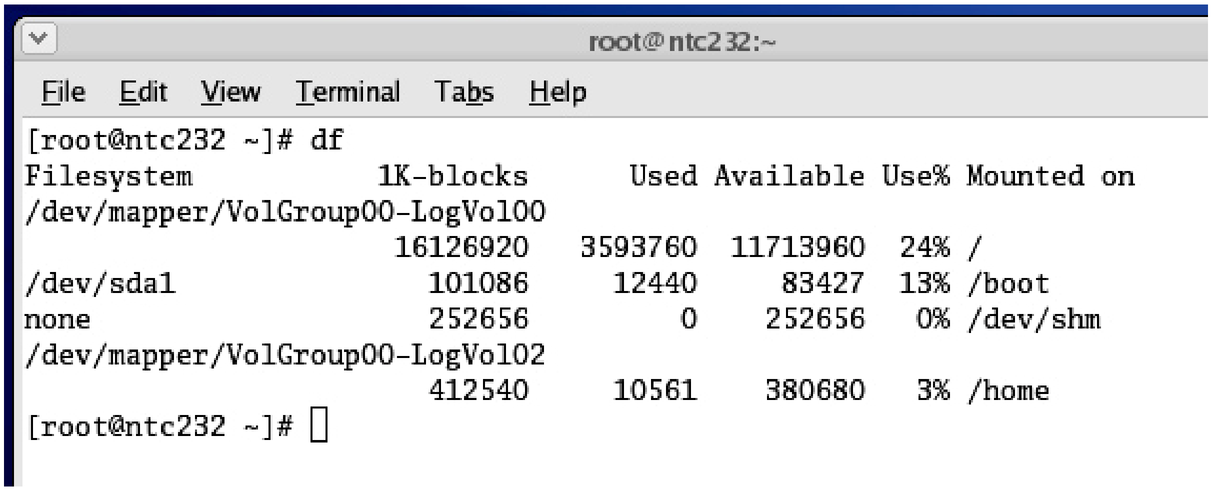
Using the command "**df**" will display the file systems that are presently mounted, size in kbytes, used space, available space, percent used, and the mount point. This display will vary from system to system depending on what size harddrives are in use, what version of Linux is being run, etc…

$ df This command will display the file systems presently mounted.

```
Filesystem        1k-blocks     used    available   %used   Mounted on

  /dev/sda2         8064304    2885344   4769304     38%     /
  /dev/sda1          101089      64982     80888     16%     /boot
  /dev/sda5          101089       6973     88897      8%     /home
  none               250432          0    250432      0%    /dev/shm
```

-or-



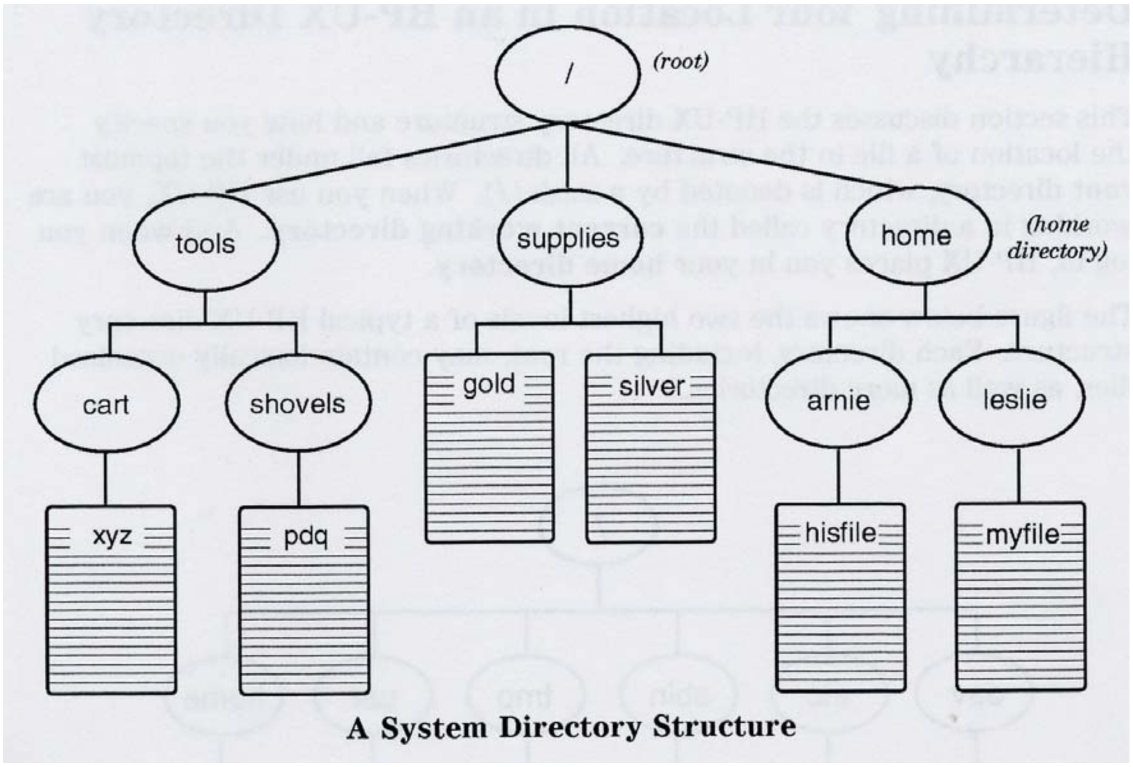
```
[root@ntc232 ~]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup00-LogVol00
                      16126920   3593760  11713960  24% /
/dev/sda1               101086     12440     83427  13% /boot
none                    252656         0    252656   0% /dev/shm
/dev/mapper/VolGroup00-LogVol02
                        412540     10561    380680   3% /home
[root@ntc232 ~]# 
```

It is important to note that the mount point of a file system (the directory name) is located on the *root* file system. You must make sure that the file system you need is listed in the output of the command "df" or you will be accessing the root file system.

For example, the "/home" file system in our first display was only 3% full and could easily more data. However, if the "/home" *file system* was **not** mounted (ie. did not show up in the output of the df command), the "/home" *directory* would still exist. Writing the data into the "/home" directory on the root file system could exceed the available space on the root file system (/) and cause the entire system to be unavailable to other users.

UNIX File System or File Hierarchy

These two phrases are interchangeable, they refer to the overall layout of individual filesystems, directories, and files. These directories usually contain more directories; thus, the typical **home directory** develops a branching tree structure. At the top of the inverted tree structure is the **root directory**, represented in path names as "/".



A System Directory Structure

The phrase "**file system**", when in lower case, will usually refer to a single file system within the File Hierarchy. File systems are created by the Operating System when it is loaded and can also be created by the System Administrator, to configure unused disk space.

Assuming /home was created as a file system, **/home** is then a directory *and* a file system. **/home/leslie** is a directory in the /home filesystem.

Specifying Files and Directories

When you log onto the system you are positioned within the file hierarchy in your in your home directory – your starting directory, which is thus your initial “current working directory” . You can see what that directory is via ‘echo $HOME’. In the course of your work however you are likely to navigate the file system and leave your home directory. Wherever you land at any given point in time that then becomes your new current working directory. This position (directory) can be listed with the pwd command at any time.

Command: **pwd** Reports the **p**resent **w**orking **d**irectory or current directory.

```
$ pwd
```

When using commands to work with files in your current working directory you can refer to them just by their file names. But when referring to directories and files outside your current working directory, you must use **path names**, which tell Linux how to get to the appropriate directory. The exception would be if the files are executables (commands, scripts, programs, …) that may exist in some directory that is included in your “search path” (The setting of your PATH variable to be discussed later).

Absolute vs. Relative path

Pathnames are pointers/directions to a filenames by using **absolute path** (starting at the top of the File Hierarchy), or **relative path** (starting at your current location in the File Hierarchy). Linux commands will accept absolute pathnames, relative pathnames, or simple filenames, as identifiers to locate files or commands.

Absolute path names specify the path to a directory or file, starting from the root directory at the top of the inverted tree structure. The root directory is represented by a slash (/). The path consists of a sequential list of directories, separated by slashes, leading to the directory or file you want to specify. The last name in the path is the directory or file you are pointing to.

ABSOLUTE pathname:

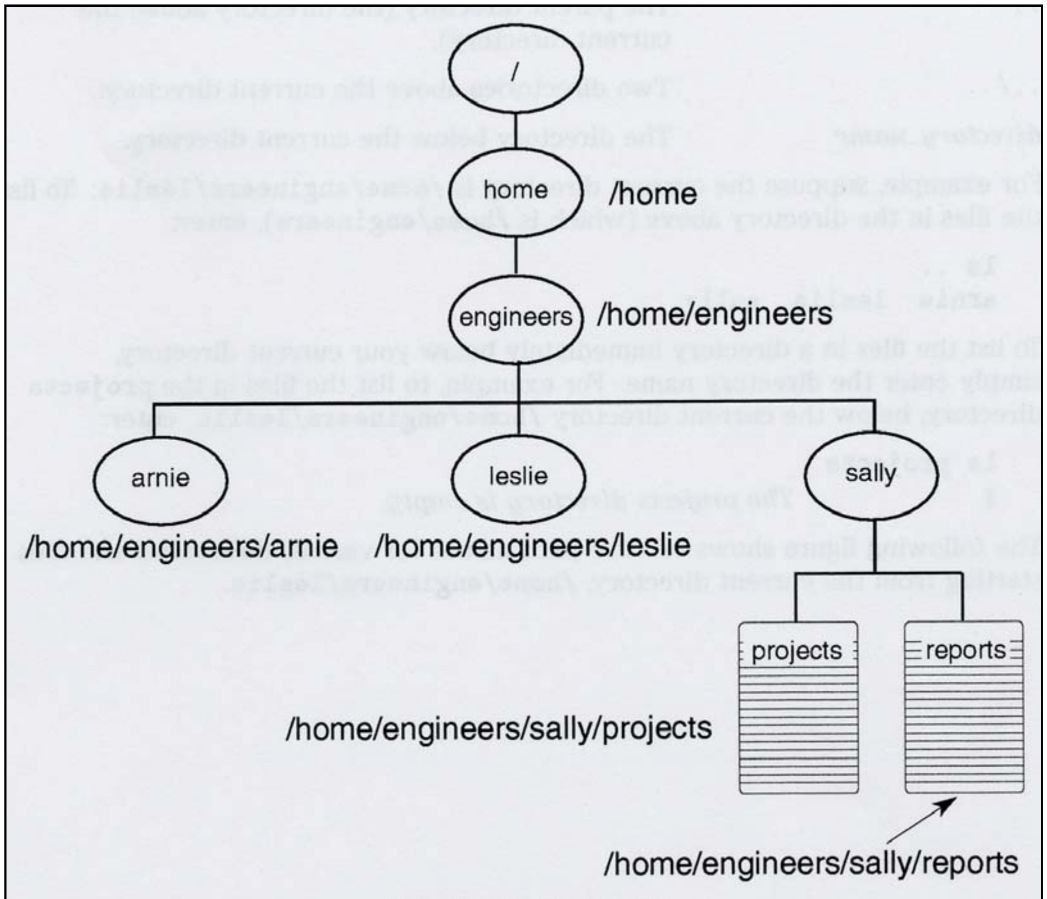
- * gives the complete path of the location of a file or directory.
- * always starts at the top of the UNIX Hierarchy (root)
- * always starts with a leading /
- * not dependant on your present location
- * always unique across the UNIX File System.

Here is an example of an absolute path, displayed with the **pwd** command:

```
$ pwd
/home/engineers/leslie
```

This specifies the location of the current directory, leslie, by starting from the root and working down.

The following figure shows the absolute path names for various directories and files in a typical directory structure:



Absolute Path Names

Relative Path Names

You can use a relative path name as a shortcut to the location of files and directories. Relative path names specify directories and files starting from your current working directory (instead of the root directory).

RELATIVE pathname:

- always starts at your current location
- will never start with a /
- unique relative to your current location
- often shorter than an absolute path

This relative path name ...	**Means**
. (Dot)	The current directory
.. (Dot, Dot)	The parent directory (the directory above the current directory).
../..	Two directories above the current directory
directory_name	The directory below the current directory

DOT "." and DOT DOT ".." DIRECTORIES

Whenever a directory is created, there are always two entries present in that directory, dot (.) and dot dot (..). The (.) is synonymous with the pathname of the present working directory and can be used in its place. The (..) is synonymous with the pathname of the parent directory and can also be used in its place.

Dot "." Directory

```
./application name
```

The dot in front of the application name is a relative address that will tell Linux to search only the current directory for this program.

Dot-Dot ".." Directory

The dot-dot can be used as a relative address for the parent of the present working directory when moving through the Linux File System. In the case of the root directory (/), there is no parent directory and the dot (.) and dot dot (..) paths both refer to the root directory (/).

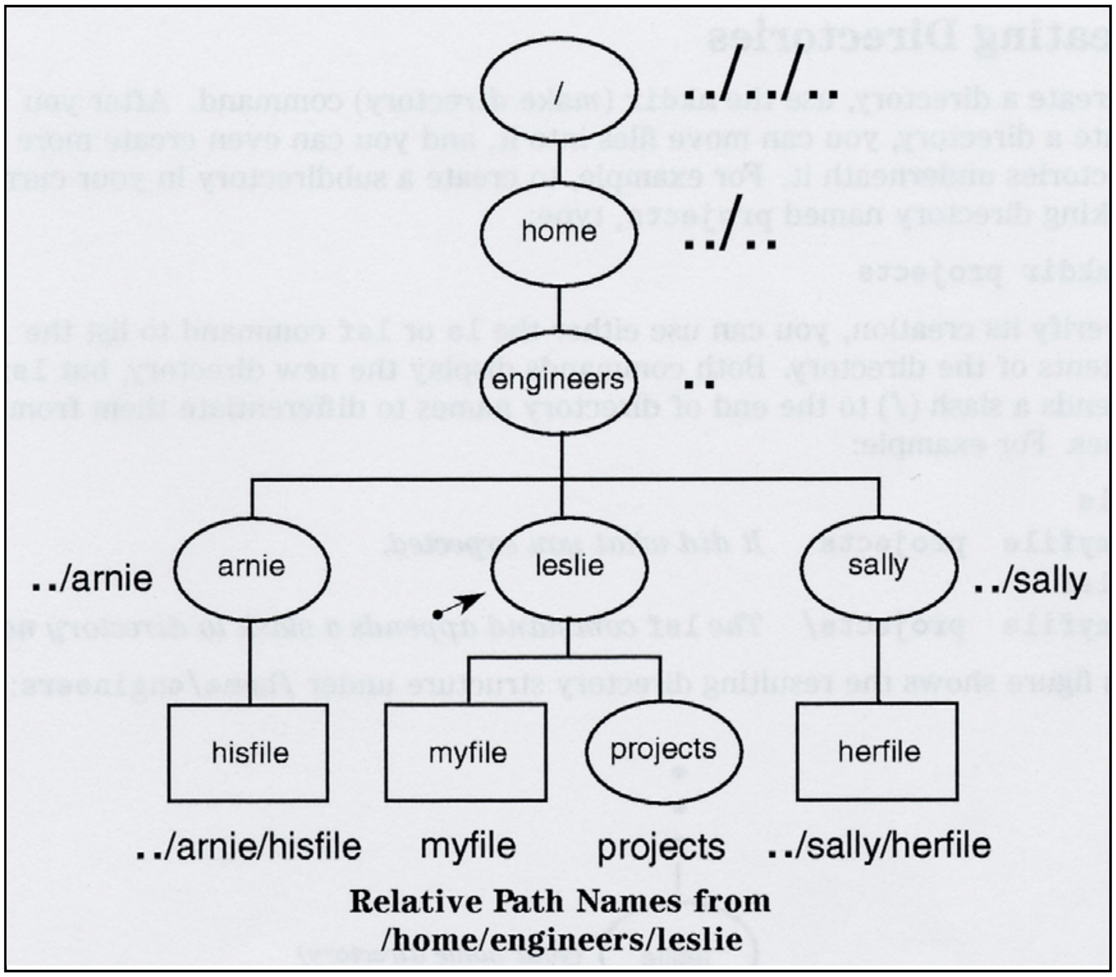
If you are in the directory /home:

..	represents / (or root)
../..	also represents /
../etc	represents /etc
../etc/file1	represents /etc/file1

If you are currently in the directory /home/user3:

..	represents /home
../..	represents /
../user2	represents /home/user2
../user2/file1	represents /home/user2/file1

The figure on the next page displays **relative path names** for various directories and files starting from the current directory,   /home/engineers/leslie.



Relative Path Names from /home/engineers/leslie

You will use absolute or relative addresses anytime you reference a file or directory. Quite commonly you will reference directories in this was as you move through the file system structure with the **cd** command (next topic).

You will also use absolute or relative references for file operations:

```
$ ls /home
$ ls ..
$ ls ../sally
$ ls ../../home
```

Or to read/run a script file

```
$ runscript.sh
$ ./runscript.sh          (same as above actually)
$ /programs/runscript.sh
$ ../../scripts/runscript.sh
```

It may appear confusing at first but as you work through the course the idea will gel.

Changing Directories

To change your present working directory, use the **cd** command.
The format of the **cd** command is:

cd directoryname

Command: **cd** **C**hange current working **d**irectory

```
$ cd /home
```

Using the absolute pathname the command changes the current directory to "home" in the "root /" directory.

```
$ cd student1
```

Using the relative pathname the command changes the current directory to the "student1" sub-directory of the present working directory.

For example, if your home directory was /home/leslie and you ran the “cd projects” command, **pwd** would display the following:

```
$ pwd
/home/leslie
$ cd projects
$ pwd
/home/leslie/projects
```

To change into the directory new under projects:

```
$ cd new
$ pwd
/home/leslie/projects/new
```

Remember that “**..**” is the **relative path name** for the parent directory of your current working directory. So to move up one level, back to projects:

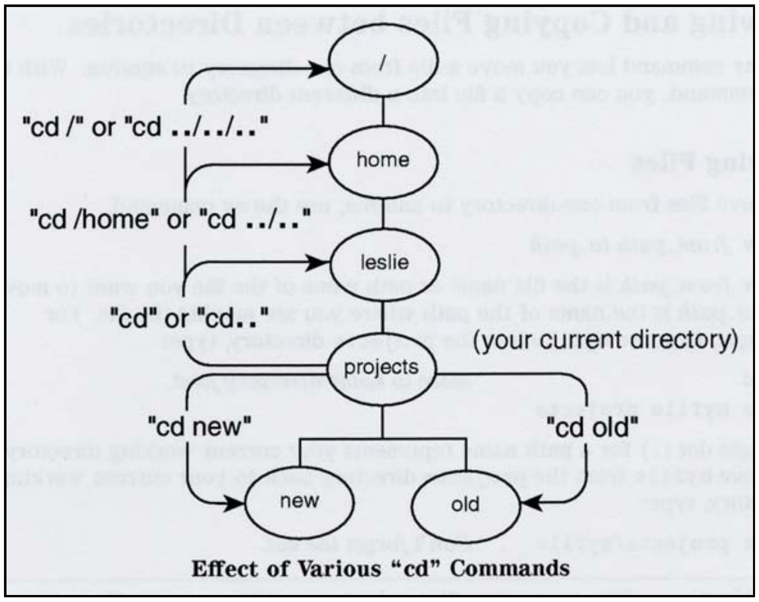
```
$ cd ..
$ pwd
/home/leslie/projects
```

To return to your home directory from anywhere in the UNIX File Hierarchy, just type “cd”.

Example:

```
$ cd
$ pwd
/home/leslie
```

The following figure illustrates how various **cd** commands change your current working directory. The example assumes you’re starting at the directory



Effect of Various “cd” Commands

/home/leslie/projects, and that your home directory is /home/leslie:

Once you have reached your destination directory using the **cd** command, you may want to know what files and sub-directories are in that directory. This can be done with the “ **ls** “ command. NOTE: Files and sub-directories can be listed from any location on the system by using the “absolute path name”, or clever relative addressing.

Listing Files and Directories

Command: **ls** **Lis**ts the files found in a directory and sub-directories.

Example: ls

```
data    data_purge    file1    file2    file3
```

Example: ls /home/student1

```
data    data_purge    file1    file2    file3
```

The ls command will list only those files that do not have a leading dot (.) as part of the file name. To list all files, including those with a leading dot, use “ls -a”.

Example: ls -a

```
.    ..    .profile    .bashrc    data    data_purge    file1    file2    file3
```

Example: ls -F

```
data/      data_purge*      file1      file2      file3
```

The addition of the -F flag places a ' / ' after directory entries, and an ' * ' after executable programs.

Example: ls -l

```
drwxr-xr-x   2   student1   class   1024   June 15   10:23   data
-rwxr-xr-x   1   student1   class     64   May 10    14:10   data_purge
-rw-rw-r--   1   student1   class     64   May 11    14:12   file1
-rw-rw-r--   1   student1   class     64   May 11    14:12   file2
-rw-rw-r--   1   student1   class     64   May 11    14:12   file3
```

In the above example, the -l flag will provide a *long listing*. This information will display the file type (most common are: d=directory, -=regular file, l=soft link), file permissions, number of hard links, owner, group, size, modification date, and filename (with a pointer to another file if it is a soft link), the details of which will come later.

COMMAND STRUCTURE

You will find that you use ls (or a variation {alias} thereof – ll) more than any other command on the system. But let’s digress a moment away from the output and consider command structure and syntax in Linux - the Linux command syntax is composed of three basic sections:

command   [- { flags }]   [arguments]

Example: ls [-l]   [/home/student1/file1]
([ ] are used for explanation only, not in actual command entry)

command	The task that you want completed.
flags	Flags (aka Switches) allows you to display the results of the command in various forms. Flags are specific to a command and most are preceded by a dash " - ". Multiple flags can be used together, as long as they do not conflict.
	Example: ls -lai /home/student1
argument	Parameters (directory or filename) to be passed to the command. Multiple arguments may be passed to a command
	Example: ls -l file1 file2 file3

We should understand the term “White Space”. This could be a <space>, <tab>, or <newline> placed in the commandline. The shell will decode the command line (ls -l file1 file2) as two separate filenames (file1 and file2). If the argument includes white space, it can be protected with single or double quotes (ls -l ‘file1 file2’). Anyway, as stated previously, simple filenames can be used with UNIX commands as a form of relative addressing. For example, “ls file1" will search the present working directory for a file called file1. Okay, you would have guessed that, but there will be times, when arguments and their position are more critical.

If you look at the **ls** command man page, you will notice the it has approximately 24+ different options that can be used to produce different display output. For our purposes, we will show the more commonly used options.

Command	ls	List directory contents
Options	-a	Do not hide entries starting with a . (dot), usually entries whose name begin with period (.) Are not listed
	-l	Use a long listing format, giving modes, number of links, owner, group, size in bytes, time last modified, and file name.
	-r	Reverse order while sorting, to get reverse (descending) collection or oldest first, as appropriate.
	-t	Sort by modification time, (latest first) before sorting alphabetically.

So for now simple examples: Suppose the current directory is **/home/engineers/leslie**. To list the files in the directory above (which is /home/engineers):

```
$ ls ..
arnie    leslie     sally
```

To list the files in a directory immediately below your current directory, simply enter the directory name. For example, to list the files in the projects directory, below the current directory   **/home/engineers/leslie** :

```
$  ls projects
$
```

>>> Use UNIX wildcards with the command “**ls**”.

File Name Shorthand: Wildcard Characters

File naming conventions are quite free in Unix. UNIX has no restriction on the length of the file name or extension. In fact, you don't need to have extensions in Unix, or you can have more than one (e.g. nexrad.schedule.fy_04). In essence, the dot (".") is really just another character in the file name. Thus, the file **text.** and **text** are not the same. You should understand that UNIX is case sensitive. The files **Text** and **text** are different filenames.

These differences in file naming conventions have important ramifications concerning the user of wild cards. UNIX does use the * (which matches any amount of characters), , and again UNIX is pretty open-ended as for rules. One example is that a legal UNIX syntax is *abc*, which would return any file with the string 'abc' in its name.

The wildcard notation is recognized by the various shells rather than being built into the Linux file mechanism. When a shell sees a filename containing wildcards, it translates it into a sequence of specific filenames, one after another. This process is sometimes called **filename generation** or **globbing**.

There are three notations for wildcards:

*	(splat) This character denotes any sequence of zero or more characters.
?	(question mark)This character denotes a single character.
[cset]	This denotes any single character in the set cset. Three particularly useful character sets are a-z (all lower case letters), A-Z (all capital letters), and 0-9 (the decimal digits).
[!cset]	If you put a ! in front of the character set, it ignores any characters within this set. (Some shells may not support this).

The * (splat) Wildcard

The * wildcard means “any characters, including no characters.” Suppose you have created the following files:

```
$ pwd
/home/student1
$ ls
myfile    myfile2     myfile3    unixfile     yourfile
```

To list only the file names beginning with “**myfile**”, type the following

```
$ ls
myfile    myfile2     myfile3    unixfile     yourfile
$ ls myfile*
myfile    myfile2     myfile3
```

To list file names ending in “**file**”, type the following

```
$ ls
myfile    myfile2     myfile3    unixfile     yourfile
$ ls *file
myfile   unixfile     yourfile
```

The ? (question mark) Wildcard

The **?** Wildcard means “any single character.” The following example will list only the files that start with **myfile** and end with a single additional character.

```
$ ls
myfile    myfile2     myfile3    unixfile     yourfile
$ ls myfile?
myfile2     myfile3
```

The ? wildcard character matches exactly one character. Thus, myfile didn’t show up in this listing because it didn’t have another character at the end.

Character Set	[cset]	Defines a class of characters.
	-	Defines an inclusive range
	!	Negates the defined class

NOTE: If a hyphen (-) is placed between two characters within brackets, the character class will be all characters in the ASCII sequence, from the first character to the last one inclusive.

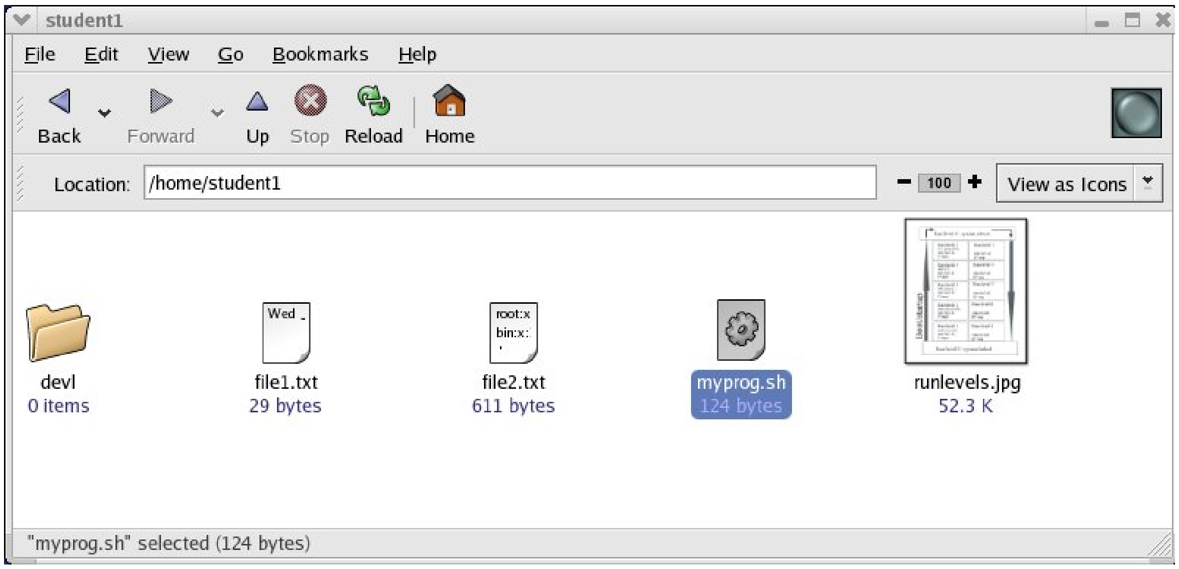
The following are some more examples of these wildcards:

The wildcard " * " will match zero or more characters, except a leading dot.

The command "ls -a" will list all files, including those with a leading dot (.)

```
$  ls *           Result:    data   data_purge    file1    file2    file3
$  ls -a          Result:     .   ..   .profile   .bashrc   data   data_purge   file1   file2  file3
$  ls .*          Result:     .   ..   .profile   .bashrc
$  ls *_pur*      Result:     data_purge
$  ls *e          Result:     data_purge
$  ls ?????       Result:     file1   file2   file3
$  ls dat?        Result:     data
$  ls data?*      Result:     data_purge
$  ls .???*       Result:     .profile   .bashrc
$  ls ?           Result:     ? not found
$  ls [ df ]*     Result:     data   data_purge  file1   file2   file3
$  ls [ df ]???   Result:     data
$  ls file[ 123 ] Result:     file1   file2   file3
$  ls *[ 1-3 ]    Result:     file1   file2   file3
$  ls [ 123 ]     Result:
```

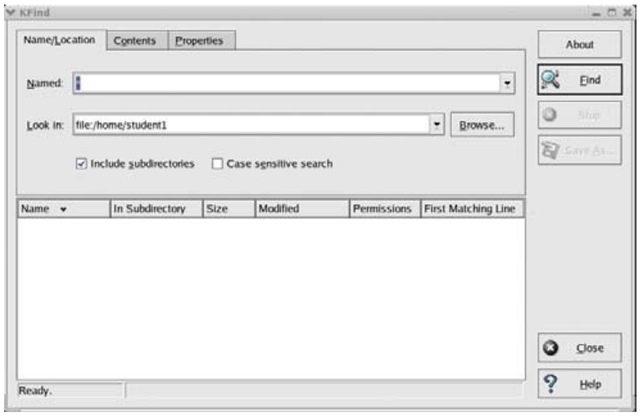
And this a good time to remind you that more often than not you will have means to perform most of your Linux tasks via a graphical interface such as Gnome desktop utilities. Using Gnome Nautilus (available from a desktop icon) for example you can navigate the file system and list out files.



"myprog.sh" selected (124 bytes)

(To get more information on Linux desktop environments it is suggested at this time to do an internet search on the terms “gnome”, and “kde” ; or get on a local system and explore the desktop screens, menus, and icons by yourself if you have not already done so.)

Our next topic is searching for files. This to can be accomplished via a GUI on most Linux systems:



find Command

Perhaps you don't know what directories to look in to list out the files you want to see, you need to *find* them – enter quite obviously the *find* command What is **find**? The find command recursively descends the directory hierarchy for each path name in *pathname_list* (that is, one or more path names) seeking files that match a Boolean *expression* (and optionally, if specific take some action with the files found).

The general format for this command is to specify the starting point for a search through the directory, followed by any actions desired.

The format of the find command is:

```
find   directory-list   [ expression ]
```

The following is a partial list of criteria that can be used to form an expression with the **find** command.

-name *filename*	True if *filename* matches pattern (metacharacters maybe used, but should be escaped "\" or quoted " ")
-type c	True if file type is c, where c is one of: f regular file d directory l link (For other file types, see **man 1 find**)
-atime [n,+n,-n]	True if file was accessed n, +n, or -n days ago. (Example: 3 = 3 days ago, +3 = 3 days or more, -3 = 3 days or less)
-ctime [n,+n,-n]	True if file was modified, permissions changed or ownership changed (inode changed) n, +n, or -n days ago. More inclusive than atime or mtime.
-mtime [n,+n,-n]	True if file was modified n, +n, or -n days ago
- user name	True if file is owned by user name (or uid)
-exec *command* {} \;	Run the UNIX *command* on each file matched by find

```
$ find  .  -type  f  -name  .profile
 ./.profile
```

The first example uses relative addressing to search the present working directory (.) and all subdirectories for a regular file with the name .profile. The result is reported as a relative address.

```
$ find   /home/student1  -type f  -name  .profile
  /home/student1/.profile
```

The second example uses absolute addressing to search the directory /home/student1 and all it's subdirectories for a file called .profile. The result is reported as an absolute address.

```
$  find   /home/student1    /home/student2    -name   “file*”
  /home/student1/file1
  /home/student2/file_from_awips
```

This example demonstrates searching multiple directories for a filename that contains a wildcard character. The quote marks ensure that the wildcard is passed to the find command for filename expansion, rather than being interpreted by the shell *before* being passed to the find command.
To search for multiple files, use the following syntax. (The -o acts as an OR function.):

```
$ find  .  -name  fileA  -o  -name  file1
```

The following command will search the directory /home/student1 (and all subdirectories) for files that have “test” as part of the filename. The wildcard “*” is used as part of the file descriptor. The results are then passed to the braces “**{ }**”, where the **-exec** command will execute the **rm** command on the contents of the braces. Note: The user will not actually see the contents of the braces, it’s a shell thing...

```
$ find   /home/student1   -name   "*test*"   -exec   rm   {}   \;
```

In the following example, the find command searches the current directory for all files that have been modified today only and then does a long list display of those found.

```
$ find   /tmp   -mtime   -1   -exec   ls   -l   {}   \;
-rw-r--r--   1 root      root         1260 Nov 29 16:37 /tmp/file1129a
-rw-r--r--   1 root      root         3780 Nov 29 16:37 /tmp/file1129b
```

Examine Text File Contents

The ability to view the contents of a file is both basic and necessary. This can be done with different commands to show different results. In fact these command line utilities have much much more flexibility than viewing files through a GUI.

Displaying the contents of text files with the following commands

File	Display the file type
more	Display file contents one screen full at a time
head	Display the first lines of a file
tail	Display the last lines of a file
cat	Display the contents, scrolling to the end of the file

Files come in many different types (text/data, scripts, compiled programs, …) When you use the 'ls' command and the filename does not give you any clues as to the contents or its format, your best friend is the ***file*** command. The command will look at the contents of a file for you and give you some indication as to its content – the type of file it is. This may be an important first step before actually looking at a file if you are unsure what the file contains, because while some file contents can be displayed cleanly, certain file types will not be friendly if you try to display them with the commands in this section. For example a data file (eg. ASCII text) can be displayed by these commands but if you try to display compiled code you terminal session can get messed up.

```
$  file  *
 levels :    ASCII text
 data :      directory
```

Viewing with the "more" command

The "**more**" command displays the text file's contents on the screen, one screenful at a time. If the file contains more lines than are on your screen, "more" pauses when the screen is full. With a longer file, press <SPACE> to continue looking at additional screens and press <Q> or <CTRL-C> when you are finished. This will return you to a command line prompt. You can also press the <ENTER> key to advance the text a single line at a time.

```
$ more .bashrc
```

Displaying the First and Last lines of a file

To see the first lines (10 lines by default) of a file, counting blank lines, use the "**head**" command.

```
$ head   .bashrc
```

To see the last ten lines (default value) of a file, use the **tail** command.

```
$ tail   .bashrc
```

Both the head and tail commands can use a numeric argument. The following displays the last 25 lines of a file.

```
$ tail  -25 .bashrc
```

Another often used option to the **tail** command is the "**-f** " option. This option allows the user to monitor the contents of a file as it updates.

```
$ tail  -f    filename.log
```

Using the **cat** command will display the contents of a text file, from beginning to end. If the content is larger than the display window, the output will scroll until the end of the file is reached. There are control commands (ctrl-s and ctrl-q) to be used to stop and restart the scrolling output, but few people are fast enough to catch the beginning of the display.

```
$ cat .bash_profile
```

But maybe, we only want to see a small portion of a file, and we are unsure where that portion may reside within a large file – hey let's *grep* it. What is **grep**? Grep is a general term for any of a family of Unix tools, including grep, egrep, and fgrep, that perform repetitive searching tasks. The tools in the grep family are very similar, and all are used for searching the contents of files for information that matches particular criteria.

You can use the **grep** ("global regular expression print") command to search for a text pattern within a file (**grep** string filename) or to display the names of files that contain a specified text pattern (**grep -l** files-to-be-checked). The **grep** command looks at each line of one or more files for a text string that matches a specified pattern. When it finds a matching text string, it displays the line in which the matching string is found. This command is useful when you want to search for information in files or directories.

Common flags for the grep command are:

-i	Ignores the case of the letters in the pattern
-l	List filenames of files that match pattern only
-n	Include line numbers in output
-v	List the lines that do not match the pattern

In the following example, **grep "alias" .bashrc** will search the file .bashrc for lines containing the character string 'alias' (in this case we enclosed the string in double quotes as it's a reserved worg – a Linux command, but for most strings this not necessary. This is the simplest form of the **grep** command.

```
$ grep  "alias"  .bashrc
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
```

In the following example, **grep -l test /etc/*** will search every file in the /etc directory for the text string 'test'. The **-l** designates that only the filenames of files that contain the text string 'test' are displayed, assuming you have access permission to these files.

```
 $ grep -l test /etc/*
/etc/#update
/etc/bootptab
/etc/brc
grep:  can't open /etc/class
```

In the following example, **grep -n term ~/.profile** will search the file ~/.profile for 'term' and return the line and line number of any matches.

```
$  grep -n term ~/.profile
6 : # Set up the terminal
```

In the following example, **grep -i term ~/.profile** will search the file ~/.profile for the text string ‘term’, ignoring upper and lower case. Both ‘terminal’ and ‘TERM’ are matches for the specified pattern.

```
$ grep -i term ~/.profile
# Set up the terminal
    if [ “TERM” = “” ]
```

In the following example, **grep -i term ~/.profile | grep -v if** will search the file ~/.profile for uppercase or lowercase ‘term’ (and any combination of upper and lowercase “term”). Any matches are sent to standard out, which is piped to the standard input of the next grep command. This command discards any lines that contain the character string “if” then sends the output to standard out (in this case the terminal).

```
$ grep  -i  term  ~/.profile  |  grep -v  if
# Set up the terminal
```

LAB

Your practical exercises for this module:

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

http://www.nwstc.noaa.gov/d.train/linuxinstr.html

A couple of reminders

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 dependent upon grasping the subject matter in this course.

EXERCISE 1

The following commands will have you changing your current working directory and listing the contents of directories. (For all exercises we will present the '$' character as the only prompt; what you need to enter will be in bold; afterwhich press 'Enter' to execute the command).

1. $ **pwd**

2. $ **ls**

```
Some of the activities require the directory "front_porch" to be
present in your home directory. So at this point if it is not
present, contact Jim Kaplafka or Dave Rowell or at the NWSTC.
```

3. $ **cd /home**

After listing the contents of your own home subdirectory, the above command moved you to the /home directory using absolute addressing.

4. $ **pwd**

5. $ **ls**

Now return back to your own sub-directory and do the same maneuver using relative addressing. At this point there are a number of ways to get back to your own home sub-directory – choose any of:

6. $ **cd** (a cd with no other arguments/parameters supplied always returns you back to your home directory)

-or

$ **cd ~** (same)

-or-

$ **cd** *yourloginname* (of course substitute the literal yourloginname with whatever your login name actually is)

-or-

$ **cd $HOME** (This uses a variable that holds the value of your own home sub-directory – we'll talk about this later)

7. $ **pwd**

8. $ **cd ..** (go up one directory using relative addressing)

9. $ **pwd**

Let’s go further …

8. $ **cd /** (all the way to the top – root)

9. $ **pwd**

10. $ **ls**

11. $ **cd /tmp**

12. $ **pwd**

13. $ **cd /**

14. $ **cd tmp**

15. $ **pwd**

16. $ **cd ..**

17. $ **pwd**

Let’s go home. Choose another of:

18. $ **cd** (a cd with no other arguments/parameters supplied always returns you back to your home directory)

-or

$ **cd ~** (same)

-or-

$ **cd $HOME** (This uses a variable that holds the value of your own home sub-directory – we’ll talk about this later)

19. $ **pwd**

Now, use different options of **ls** to display file and directory information. After each command take a moment to look over all files presented

20. $ **ls –laf**

Then

21. $ **ls –laF**

Then

22. $ **ls –l**

Then

23. $ **ls –lat**

Then

23. $ **ls –latr**

Then

24. $ **ls –l ..**

Then

23. $ **ls –l /home**

EXERCISE 2 - FIND

This exercise is designed to familiarize you with use the find command and the primaries **-name**, **-exec**, **-size**, and **-mtime** to search directories for specified file names. As you go through this exercise don't be concerned with learning about the files found – the important thing is to understand what the find syntax is accomplishing.

1. $ **find .** (remember dot means current directory)
2. $ **find /home**
3. $ **find ..**
4. $ **find /home –name yourloginname** (substitute your actual login name)
5. $ **find /home –name fence**
6. $ **find /home –name tv_guide –o –name lamp**
7. $ **find /etc -name fstab**
8. $ **find /etc –name "*tab*"**
9. $ **find / -name fstab –exec ls –l {} \;**
10. $ **find . –mtime +30 –exec ls –l {} \;**

If you didn't understand some of the find option used above, remember where you can get help with commands – '**man find**'

EXERCISE 3 - The cat, more, head, tail, and grep commands

Go to the living room and let’s see whats on TV in the tv_guide (file)

1. $ **ls tv_guide**
2. $ **cat tv_guide**
3. $ **more tv_guide**
4. $ **head tv_guide**
5. $ **head -12 tv_guide**
6. $ **tail tv_guide**
7. $ **tail -12 tv_guide**
8. $ **grep Inside tv_guide**
9. $ **grep INSIDE tv_guide**
10. $ **grep inside tv_guide**
11. $ **grep –i inside tv_guide**
12. $ **grep –n Inside tv_guide**
13. $ **grep –ni Inside tv_guide**
14. $ **grep –v Inside tv_guide**
15. $ **grep ^2 tv_guide**
16. $ **grep ^20 tv_guide**
17. $ **grep –v ^2 tv_guide**
18. $ **file tv_guide**
19. $ **file ***
20. $ **file /etc**

EXERCISE 4

Okay, we'll finish with leaving you a pair of tasks to figure out on your own (don't worry if you just can't get one – but try both):

1. Use the find command to see what's cooking in the oven. Use the cat command on the contents of the oven to see if dinner is done yet.

2. From your $HOME directory, use the find command to display only the directories under your home directory. Search your home directory using relative addressing (.) and the "-type" option to the find command.

END EXERCISES

The commands used in the module are of a very basic nature. They may indeed be quite enough to get your through many if not all simple operations. But Linux has a great many tools and features, and we'd be remiss if it wasn't suggested that you look into a few more related commands, which may serve you well in later Linux work or courses…

A little further research via the 'man' pages or an internet search on the follow Linux commands would be well worth the effort: du, ln, file, diff, awk and sed

End

This is the end of this module. At this time you should proceed to module 3.