UNIX Bash Shell Scripting

Week 1-Session2

What we learned

- Course introduction
- Obtaining your Seneca accounts Changing passwords
- The Matrix server
- The role of an operating system
- File system in Unix
  - Basic file operation

Agenda

- File system in Unix
  - COPY/Move/Rename files & Directories
- Specifying pathnames
- File name expansion
- File Permission

Basic Commands

mv *sourcepath destinationpath*

- Used to move a file from one location to another and/or rename the file. The mv command can be used to move directories as well as files. The -i option asks for confirmation if the destination filename already exists.

cp *sourcepath destinationpath*

- Used to copy a file from one location to another. The cp command can be used to backup important files.
- The -i option asks for confirmation if the destination filename already exists.
- The -r (recursive) option allows copying of directories and their contents.

Example

- cp /home/chuck/pictures/picture.jpg /home/chuck/backup/picture.jpg
  - If you are the user **chuck**, you can abbreviate your home directory ("**/home/chuck**") using a tilde ("**~**"). For instance,
  - cp ~/pictures/picture.jpg ~/backup/picture.jpg
- cp -R ~/files ~/files-backup
  - You can use **cp** to copy entire directory structures from one place to another using the **-R** option to perform a recursive copy.

mv use for MOVE or RENAME

- The mv command has a purpose in life, and that is to move files.
  - It is a happy side effect that it can be used to move an existing file into a new file, with a new name.
  - The net effect is to rename the file, so we get what we want. But mv is not a dedicated file renaming tool.
  - mv oldfile.txt newfile.txt

Pathnames

- The concept of a pathname relates to every operating system including Unix, Linux, MS-DOS, MS-Windows, Apple-Macintosh, etc!
- A **pathname** is a fully-specified location of a unique filename within the file system
  - **Directory pathname:**
    - /home/username/UNX510/assignments
  - **File pathname:**
    - /home/username/UNX510/assignments/assn1.txt

Absolute and Relative Pathnames

- Absolute Pathname
  - An absolute pathname begins from the root, which is / (forward slash)
  - This is called absolute because it is specified the same, and locates a specific file, regardless of your current directory
  - For example: mkdir /home/someuser/UNX510
    - will create the UNX510 directory in the home directory of user someuser

Absolute and Relative Pathnames

- Relative Pathname
  - Relative path is defined as path related to the present working directory(pwd)
  - A relative pathname begins from your current directory
  - This is called relative because it is used to locate a specific file relative to your current directory
  - For example: mkdir UNX510
    - will create the UNX510 directory in your current directory!

Relative Pathnames

Rules:

- A relative pathname does NOT begin with a slash.

- Following symbols can be used:

  .. parent directory (up one directory level)

  . current directory

- Not all relative pathnames begin with . or .. !

Warning:

When using relative pathname, always make certain you know your present working directory!

Relative Pathnames

Examples:

- Change to another directory branch from parent directory:
  ```
  cd ../ipc144
  ```

- copy sample.c file from parent of your current directory to your current directory:
  ```
  cp ../sample.c .
  ```

Relative-to-Home Pathnames

- You can specify a pathname as relative-to-home by using a tilde and slash at the start, e.g.,~/UNX510/notes.html
- The tilde ~ is replaced by your home directory (typically /home/username) to make the pathname absolute.
- You can immediately place a username after the tilde to represent another user's home directory. For example: ~jane = /home/jane
- But be careful, a slash makes a big difference:

  ~/jane = /home/username/jane

Which Type of Pathname to Use?

So far, we have been given several different types of pathnames that we can use for regular files and directories:

- **Absolute pathname** (starts with /)
- **Relative pathname** (doesn’t start with / or ~)
- **Relative-to-home pathname** (starts with ~)

You can decide which pathname type is more convenient, usually to minimize typing

Creating Parent Directories

- By default, a directory cannot be created in a non- existent location – it needs a parent directory
- To create directory paths with parent directories that do not exist (using a single command) use the -p option for the mkdir command
  - mkdir -p pathname
  - eg. mkdir -p mur/dir1
    - (This would create the parent directory **mur** and then the child directory **dir1**. The **-p** means "**create any required parent directories in the path**").

Filename Expansion

- Sometimes the user may not know the exact name of a file, or the user wants to use a command to apply to a number of files that have a similar name.
- For example, how do you copy all txt file from home directory to a /myFiles directory?

Filename Expansion

- You may have heard about “Wildcard Characters” – this is a similar concept.
  - Special characters can be used to expand a general filename and use them if they match.
- Filename expansion Symbols:
  - * (star/asterisk) – Represents zero or more of any characters.
    - **ls *.txt**
  - ? (question mark) – Represents any single character
    - **ls work?.txt**

Filename Expansion

- [ ] (character class) – Represents a single character, any of the list inside of the brackets.
  - `ls work[2-4].txt`
- Placing a ! Symbol after first square bracket means "not"). Ranges such as [a-z] or [0-3] are supported.
  - `ls work[!2-4]*.txt`

File Permission in Unix

- In every Operating system, **file** systems have methods to assign **permissions** or access rights to specific users and groups of users.
  - These **permissions** control the ability of the users to view, change, navigate, and execute the contents of the **file** system.

File Permission in Unix

- File ownership is an important component of Unix that provides a secure method for storing files. Every file in Unix has the following attributes −
  - **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
  - **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
  - **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators

- While using **ls -l** command, it displays various information related to file permission as follows
  - `$ls -l myfile`

    ```
    -rwxr-xr-- 1 shahdad users 104 Jan 2 00:10
    myfile
    ```
  - The first three characters (2-4) represent the permissions for the file's owner
  - The second group of three characters (5-7) consists of the permissions for the group to which the file belongs.
  - The last group of three characters (8-10) represents the permissions for everyone else.

File Access Modes

- The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below :
- Read
  - Grants the capability to read, i.e., view the contents of the file.
- Write
  - Grants the capability to modify, or remove the content of the file.
- Execute
  - User with execute permissions can run a file as a program.

Changing Permissions

- Using `chmod` in Symbolic Mode
  - `chmod o+wx testfile`
- Using `chmod` with Absolute Permissions
  - `chmod 755 testfile`

Using `chmod` in Symbolic Mode

- `chmod [ugoa][-+=][rwx] FILE`
- [ugoa]

  u - The file owner.

  g - The users who are members of the group.

  o - All other users.

  a - All users, identical to ugo
- [-+=]

  - Removes the specified permissions.

  + Adds specified permissions.

  = Changes the current permissions to the specified permissions. If no permissions are specified after the = symbol, all permissions from the specified user class are removed.

Examples

- `chmod a-x filename`
  - Remove the execute permission for all users
- `chmod u+x filename`
  - Add the execute permission to the file owner
- `chmod g=r filename`
  - Give the members of the group permission to read the file, but not to write and execute it
- The following commands are doing the same:
  - `chmod og= filename`
  - `chmod og-rwx filename`

Using `chmod` with Absolute Permissions

- `chmod NUMBER FILE`
- The NUMBER can be a 3 or 4-digits number
  - Each write, read, and execute permissions have the following number value:
  - r (read) = 4
  - w (write) = 2
  - x (execute) = 1
  - no permissions = 0

chmod Number

R	W	X	Octal value
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Example

- `chmod 754 testfile`
  - Owner: rwx=4+2+1=7
  - Group: r-x=4+0+1=5
  - Others: r--=4+0+0=4
- `chmod 644 filename`
  - Give the file's owner read and write permissions and only read permissions to group members and all other users

Directory Permissions

- `r` permission for a directory allows viewing of file names in the directory, but no access to the files themselves (regardless of the files' permission settings)
- `x` gives **passthrough** permission for a directory, which allows access to any files in the directory which have appropriate permissions set, but doesn't allow viewing of file names in the directory
- `x` also gives permission to `cd` to the directory, changing the `pwd`

Directory Permissions

- **`r`** and **`x`** permissions allow viewing of file names, and access to any files which have appropriate permissions set
- **`w`** and **`x`** permissions allow adding or removing of files, but don't allow viewing of file names
- **`r`** and **`w`** and **`x`** permissions allow viewing of file names, access to any files which have appropriate permissions set, and adding and removing of files

Default Permission

- The **default** permissions
  - FILES: rw-rw-rw- (666)
  - Directories: rwxrwxrwx (777)
- The **actual** permissions
  - FILES: rw-r--r--(644)
  - Directories: rwxr-xr-x (755)
- Why these two are different?
  - Because of **umask**

What is **umask**

- **`umask`** is a 3digit number which is deducted from default permission and makes the **actual** permission for the files and **directories**.
- What is the current value of **`umask?`**
  - **run → umask**
  - **It is 0022**
- Ignore the first 0, your **umask** is 022

Effect of **umask** on File/Directory

- Effect of **umask** on directories

Default Directory permission	rwxrwxrwx (777)
Minus permission removed by umask	----w--w- (022)
Effective directory permission	rwxr-xr-x (755)

- Effect of **umask** on files

Default file permission	rw-rw-rw- (666)
Minus permission removed by umask	----w--w- (022)
Effective file permission	rw-r--r-- (644)

umask

- `umask` defines default permissions for newly created files/directories, doesn't change permissions on existing files/directories
  - umask 023
- Remember that **umask** is automatically being set to its original value each time system is booted.
  - umask value is inside `/etc/profile` or `/etc/login.defs`

umask example

- umask 023

Default Directory permission	`rwxrwxrwx (777)`
Minus permission removed by umask	`----w--wx (023)`
Effective Directory permission	`rwxr-x-r-- (754)`

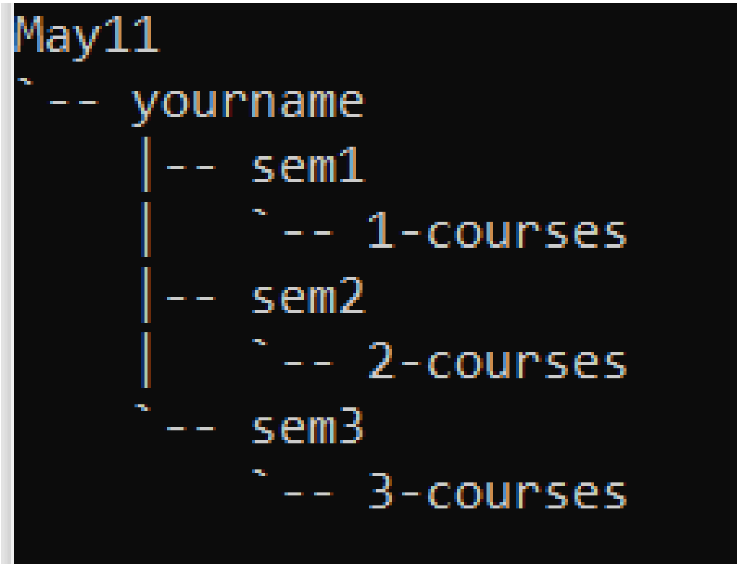
Default File Permission	`rw-rw-rw- (666)`
Minus permission removed by umask	`---r-x-wx (023)`
Effective Directory permission	`rw-r--r-- (644)`

Some other commands

- date, which, who, whoami
- Displaying contents using echo and printf

Activity2

- Create the following folder structure
  - Add three files, and list your courses in 1st, 2nd, 3rd semester
  - Change the permission of 1-courses to read only
  - Create a backup folder in May11 and make a copy all data in sem1, sem2, sem3



```
May11
`-- yourname
    |-- sem1
    |   `-- 1-courses
    |-- sem2
    |   `-- 2-courses
    `-- sem3
        `-- 3-courses
```