CS 240 Programming in C

Unix Commands

Feb 14, 2023

Source of Content

Glenn Hoffman (our IT program director) prepared a Linux guide which I recommend to the students. It can be found here:

Link:
`https://cs.umb.edu/~ghoffman/linux/unix_cs_students.html`

The content of this presentation is taken from the same link.

Basic Unix Commands

- To see the contents of your current directory, type `ls`
- To go to a new directory, use the `cd` command
- A directory is the same thing as a folder
- Whenever you go to a new directory, it's a good idea to look at the contents of that directory using the `ls` command
- To create a new directory, use `mkdir` followed by the directory name

Basic Unix Commands

- The `pwd` command will tell you your current directory
- Use `pwd` whenever you are not sure of your current directory

Basic Unix Commands

- To remove a file, use `rm`
- To remove a directory, use `rmdir`
- The directory must be empty or `rmdir` will not work
- To move a file use `mv`
- You also use `mv` to change the name of a file or directory
- To see the contents of a file use `cat`

Your Home Directory

- Whenever you log in to a Unix host, you will always find yourself in your home directory
- This a directory that belongs to your Unix account only
- You have full control of permissions within this directory

Your Home Directory

- If you use cd with no arguments, it will take you to your home directory: `cd`
- Use `pwd` to see your current directory: `pwd`
- Example output: `/home/aaditya`

Navigating the Hierarchical File Systems

Any file or directory in the filesystem will be one of four positions relative to your current directory:

- It can be inside your current directory
- It can be below your current directory
- It can be above your current directory
- It can be off to the side of your current directory. In this last case, you must go up before you can go down to reach this file.

Hidden Filenames

A file whose filename begins with a period (.) is a "hidden" or "invisible" file.

- `ls` does not display these files unless you use the `-a` option
- You have already encountered such files in the `.forward` and `.plan` files
- Two special hidden files `.` and `..` appear in every directory:

```
$ ls -a
... hw3 work
```

The . and .. Directory Entries

Every directory has at least two entries: `.` and `..`

- When a new directory is created, these are the first two entries
- `.` stands for the current directory
- `..` stands for the parent directory of your current directory
- `..` is the directory immediately above your current location:

```
$ pwd
/home/aaditya
$ cd ..
$ pwd
/home
```

The . and .. Directory Entries (contd.)

. (dot) is most often used in two circumstances:

- To run a program in your current directory
- To move or copy a file to your current directory

```
$ ls
cs240 notes.txt work work2
$ cd work
$ ls
bletch.txt foo.txt
$ cp ../notes.txt .
$ ls
bletch.txt foo.txt notes.txt
```

Pathnames

- Every file has a **pathname** which is used to access the file
- A pathname has two components: the name of the file and a path to reach the file
- The name of the file is always at the end of a pathname
- What comes before the filename is the path to the directory that holds the file
- Within a pathname, a slash (/) to the right of a name indicates that the name refers to a directory

Pathnames: Absolute vs. Relative

- There are two types of pathnames: **absolute** and **relative**
- An **absolute pathname** specifies the entire path from the root directory to the file or directory
- A **relative pathname** specifies the path from the current directory to the file or directory
- A relative pathname can start with either the name of a directory or with **.** (for the current directory) or **..** (for the parent directory)

Absolute Pathnames

- The top of the filesystem is called the root
- The root is represented by a single slash character (/)
- It can stand alone or appear as the first character before a directory name
- An absolute path is a list of directories, starting with the root, and ending with the directory that contains the file
- When you add the filename to the end of an absolute path, you have an absolute pathname

Absolute Pathnames Contd.

- For example, the absolute pathname of the .bash_profile file in my home directory is /home/aaditya/.bash_profile
- This means that my home directory, "aaditya", is under the directory named "home", which, in turn, is under root (/)
- The advantage of an absolute pathname is that it can be used from any part of the filesystem
- It does not depend on your current directory
- The disadvantage is that absolute pathnames tend to be long
- It is easy to make a mistake typing an absolute pathname
- An absolute pathname always begins with either a slash (/) or a tilde (~)

Tilde (~) in Pathnames

- There is one form of an absolute path that is very short
- This is the tilde character (~)
- Tilde stands for your home directory
- This means you can use tilde (~) anywhere you would normally use a path to your home directory:
  - $ pwd
  - /home/aaditya/cs240
  - $ cd ~
  - $ pwd
  - /home/aaditya
- When you put a tilde in front of a Unix ID, it stands for the home directory of that account:
  - $ cd ~aaditya
  - $ pwd
  - /home/aaditya

Relative Pathnames

- Absolute pathnames are useful because you can use them anywhere
- But they are long and easy to mistype
- For most purposes, it is easier to use relative pathnames
- In a relative pathname, the path starts from your current directory
- In an absolute pathname, the path starts from the root (/)
- While all absolute pathnames start with a slash (/) or a tilde (~), relative pathnames never do
- The absence of a slash (/) or a tilde (~) from a pathname, means it is a relative pathname
- As far as Unix is concerned it makes no difference whether you use and absolute or relative pathname
- Relative pathnames are more convenient and are most often used

Relative Pathnames

- When the file is in your current directory
- When the file is in a subdirectory of your current directory
- When the file is in a directory that is an ancestor of your current directory
- When the file is in a directory that is neither an ancestor or descendant of your current directory

Relative Pathnames in Your Current Directory

Using a relative pathname with a file or directory inside your current directory is easy

- The relative pathname simply consists of the name of the file or directory
- The "path" part of the relative pathname is empty because what you want to access is already inside your current directory:

```
$ ls
notes.txt cs240 work work2
$ cat notes.txt
some text
some text in line2
```

Notice that all I need to access the file is the file name. It's the same for directories.

Relative Pathnames in a Subdirectory

Things get a little more complicated when you are dealing with a file in a subdirectory. Here, you must list every directory between your current directory and the file you want:

- Use a slash (/) to separate the names of each directory.
- There is nothing before the name of the first directory in the path.

For example:

```
$ ls work
foo.txt  notes.txt
$ cat work/notes.txt
some text
some text in line2
```

Relative Pathnames above the Current Directory

- When the file or directory is above the current directory you can't list the directory names.
- Instead, you have to use the special .. entry in each directory.
- Use one .. for each directory up the chain of directories in the path with a slash (../) between each ..

```
$ pwd
/home/cs240/work
$ ls ../..
cs240 jharris mphamman
```

Relative Pathnames Contd.

- What if the file is neither above nor below?
- Here you have to go up to a common ancestor directory and then go down to the directory that holds the file.
- The path starts with one or more ..
- You keep going up until you get to a directory that is an ancestor of your current directory and the file you are trying to reach.
- Once you get to the common ancestor, you go down to the directory that holds the file:

```
$ ls ../../../courses/cs240/s23/aaditya/
```

Shell Variables

- A variable is a name that has been given a value.
- Shell variables are variables that can be used in the shell.
- To get the value of a variable put a dollar sign ($) in front of its name:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/b
```

- Some variables are set and maintained by the shell itself.
- They are called keyword shell variables or just keyword variables.
- Most of these variables are defined in /etc/profile.
- Many of these keyword variables can be changed by the user.
- Other variables are created by the user.
- They are called user-created variables.
- The environment in which a variable can be used is called the scope.
- Shell variables have two scopes: Local and Global.

Local Variables

- Local variables only exist in the shell in which they are defined.
- To create a local variable, use the following format: VARIABLE_NAME=VALUE.
- There cannot be any spaces on either side of the equal sign when setting bash variables.

```
$ foo=Foo
$ echo $foo
Foo
```

- Variables are local unless you explicitly make them global.
- If the value assigned to a variable has spaces or tabs, you must quote it:

```
$ hello="Hello there"
$ echo $hello
Hello there
```

- Local variables only exist in the shell in which they are created.

Global Variables

- Global variables defined in one shell maintain their values in all subshells created by that shell.
- Global variables are defined in bash by preceding the variable definition with the keyword export:

```
$ echo $foo
Foo
$ export foo=FOO
$ echo $foo
FOO
```

- Usually, global variables are declared in a startup file like .bash_profile or .bashrc.
- The env command, when used without an argument, displays the values of global variables:

```
$ env
```

Keyword Shell Variables

- Keyword shell variables, or keyword variables, have special meanings to the shell.
- They have short, mnemonic names.
- When you start a shell, that shell inherits several keyword variables from the shell that created it.
- By convention, the names of keyword variables are always capitalized.
- Most keyword variables can be changed by the user.
- This is normally done in the startup file .bash_profile.

User-created Variables

- User-created variables are any variables you create.
- By convention, the names of user-created variables are lowercase:

```
$ foo=Foo
$ echo $foo
Foo
```

- User-created variables can be either local or global in scope.

The PATH System Variable

- When you run a program by entering a pathname, such as /usr/bin/php, the shell has no difficulty finding the executable file.
- The shell checks a system variable called PATH to find the correct file.
- PATH contains a list of directories to search for an executable file.
- The shell searches each of these directories in turn until it finds an executable file with the name of the command.
- PATH always has a default value which is created when the system is installed. Here is the default value on cs240:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin ...
```

The PATH System Variable Contd.

- The absolute pathname of each directory is separated from the next by a colon (:).
- If the shell reaches the end of the directory listings in PATH without finding the command, it will display an error message.
- If the shell finds an executable file but you do not have to execute privileges, it will tell you this in an error message.
- You can modify the PATH variable in your own Unix environment.

Access Permissions

- All Unix files and directories have access permissions.
- The access permissions allow the owner of a file or directory to decide who gets to do what with that file or directory.
- By default, the owner of a file or directory is the user account that created it.
- Every file, directory, or device on a Unix filesystem has three types of permissions:
  - Read
  - Write
  - Execute
- If you have read permission on a file you can look at the data in the file.
- You can run `cat`, `more` or `less` on these files.
- If you only have read permission, you cannot change a file.
- To change a file you need write permission.
- To run a program you must have the execute permission.

Access Permissions Contd.

- Each of the three permissions is set either on or off to three classes of users:
  - The owner
  - The group
  - Every other Unix account
- Every file or directory has an owner.
- A group is a collection of Unix accounts.
- A group can only be set up by a system administrator.
- Every file or directory is assigned to a group.
- The last class of users is any account that is not the owner or a member of the group.
- Unix calls this class of users "other".

Viewing Permissions

To view the permissions of a file or directory use `ls -l`:

- The character in the first column indicates the type of file.
  - A dash (-) means an ordinary file.
  - The letter "d" indicates a directory.
  - The letter "l" indicates a link.
- The next three characters indicate the permission of the owner.
  - "r" means the owner has read permission.
  - "w" means the owner has the write (change) permission.
  - "x" means the owner has the execute (run) permission.
  - "-" means the owner does not have the permission that would normally appear in this column.

Viewing Permissions Contd.

- The next three characters indicate the permissions of the group.
- The last three characters are the permission of all other accounts.
- After the permissions is a number that indicates the number of links to the file or directory.
- The following column is the owner of the file or directory.
- Next, you will find the group assigned to the file or directory.
- Following this is the size of the file in bytes.
- Next is the date and time the file or directory was created or last modified.
- The last column is the name of the file or directory.