Hussam Abu-Libdeh
based on slides by David Slater

September 10, 2010
Last Time
We had a brief discussion On The Origin of Species *nix systems

Today
We roll our sleeves and get our hands dirty
Unlike Windows, UNIX has a single global "root" directory / (instead of a root directory for each disk/volume).

All files and directories are case sensitive.

- `hello.txt` != `hEllO.tXt`

Directories are separated by `/` instead of `\` in Windows.

- Windows: `D:\Documents\cs2042\2009\Lecture2\`

“Hidden” files begin with “.”: `.gimp`

Let's look at directories in my root directory.
What’s Where?

- **/dev**: Hardware devices can be accessed here - usually you don't mess with this stuff.
- **/lib**: Stores libraries, along with `/usr/lib`, `/usr/local/lib`, etc.
- **/mnt**: Frequently used to mount disk drives
- **/usr**: Mostly user-installed programs and their related files
- **/etc**: System-wide settings
Programs are usually installed in one of the “binaries” directories:

- /bin: System programs
- /usr/bin: Most user programs
- /usr/local/bin: A few other user programs
Ok, but where is my stuff?

Your files can be found in your home directory, usually located at

/home/username

Your home directory can also be access using the special character

~

Which is all well and good, but how do we move around?
Many shells default to using the current path in their prompt. If not...

**Print Working Directory**

`pwd`

- Prints the full path of the current directory
- Handy on minimalist systems when you get lost
Before we try going somewhere else, let's see what is in the current directory.

**The list command**

```bash
ls [flags] [file]
```

- Lists directory contents (including subdirectories)
- Works like the dir command from DOS
- The `-l` flag lists detailed file/directory information (we’ll learn more about flags later).
Ok lets go!

**change directory**

`cd [directory name]`
- changes directory to [directory name]
- If not given a destination defaults to the user’s home directory
- takes both absolute (`cd /home/hussam/cs2042`) and relative (`cd cs2042`) paths.
Its all Relative... except when its not

Absolute path
- location of a file or folder starting at /

Relative Path
- location of a file or folder beginning at the current directory
Relative Path Shortcuts

Shortcuts:

- `~` - current user’s home directory
- `.` - the current directory (is useful I promise!)
- `..` - the parent directory of the current directory

Example

If we start in `/usr/local/src`, then
- `cd` ⇒ `/home/hussam`
- `cd .` ⇒ `/usr/local/src`
- `cd ..` ⇒ `/usr/local`
The easiest way to create an empty file is `touch`

**Using touch**

```
touch [flags] <file>
```

- Adjusts the timestamp of the specified file
- With no flags uses the current date/time
- If the file does not exist, `touch` creates it

File extensions (.exe, .txt, etc) often don’t matter in UNIX. Using `touch` to create a file results in a blank plain-text file (so you don’t need to add .txt to it).
Simple and to the point

**Make Directory**

`mkdir [flags] <directory>`

- Makes a new directory with the specified names
- Can use relative/absolute paths to make directories outside the current directory.
Unlike in window, once you delete a file (from the command line) there is no easy way to recover the file.

**Remove File**

`rm [flags] <filename>`

- Removes the file called `<filename>`
- Using wildcards (more on this later) you can remove multiple files
  - `rm *` - removes every file in the current directory
  - `rm *.jpg` - removes every `.jpg` file in the current directory
- `rm -i filename` - prompt before deletion

Hussam Abu-Libdeh based on slides by David Slater
By default, `rm` cannot remove directories. Instead we use...

**Remove Directory**

```
rmdir [flags] <directory>
```

- Removes a **empty** directory
- Throws an error if the directory is not empty.

To delete a directory and all its subdirectories, we pass `rm` the flag `-r` (for recursive)

```
rm -r /home/hussam/oldstuff
```
Copy

**cp** [flags]  <file>  <destination>

- Copies a file from one location to another
- To copy multiple files you can use wildcards (such as *)
- To copy a complete directory use **cp -r**  <src>  <dest>

**Example:**

cp  *.mp3  /Music/  -  copies all .mp3 files from the current directory to  /home/<username>/Music/
Unlike `cp`, the move command automatically recurses for directories.

**Move**

```bash
mv [flags] <source> <destination>
```

- Moves a file or directory from one place to another.
- Also used for renaming, just move from `<oldname>` to `<newname>`.
Quick Review

- `ls` - list directory contents
- `cd` - change directory
- `pwd` - print working directory
- `rm` - remove file
- `rmdir` - remove directory
- `cp` - copy file
- `mv` - move file
I mentioned that the /bin, /usr/bin, and /usr/local/bin are where most programs are installed. These three directories are always included in the UNIX system’s PATH.

**PATH**

- When you type a command into the command prompt, the Shell looks in the system’s PATH for an executable with that name.
- For instance, when you type ls, the shell looks in /bin and finds the program ls and executes it.
- To execute a program that is not in the PATH we must type the full path to the program, ex:
  - /home/user/program1
- If the program is in the current directory, we have to specify that:
  - ./program1
- See the fact that . refers to the current directory is useful!
Most commands take flags (also called options). These usually come before any targets and begin with a `-`.

- **One Option**
  - `ls -l`

- **Two Options**
  - `ls -l -Q`
  - `ls -lQ`

- **Two Options**

- **Applies options left to right**
  - `rm -fi file` ⇒ prompts
  - `rm -if file` ⇒ does not prompt
How do I know how some fancy new command works?

**The `man` command**

`man <command_name>`

- Brings up the manual page (manpage) for the selected command
- Unlike google results, manpages are **system-specific**
- Gives a pretty comprehensive list of all possible options/parameters
- Use `/<keyword>` to perform a keyword search in a manpage
- The n-key jumps to successive search results
There are subtle differences with options on different systems. For instance `ls -B`

- **BSD/OSX** - Force printing of non-printable characters in file names as `\xxx`, where `xxx` is the numeric value of the character in octal
- **Ubuntu** - do not list implied entries ending with `~`

This is why `man` is your best friend and `google` is your second best friend!
Unix was designed to allow multiple people to use the same machine at once. This raises some security issues - How do we keep our coworkers from reading our email, browsing our documents and changing/deleting programs and files while I’m using them?

- Access to files depends on the users account
- All accounts are presided over by the Superuser, or ”root” account
- Each user has absolute control over any files he/she owns, which can only be superseded by root.
Files can also be assigned to groups of users, allowing reading, modifications and/or execution to be restricted to a subset of users.

**Example:**

If each member of this class had an account on the same server, it would be wise to keep your assignments private (user based). However, if we had a class wiki hosted on the server, it would be advantageous to allow everyone in the class to edit it, but no one outside of the class.
Each file is assigned to a single user and a single group (usually written user:group).

For example my files belong to hussam:users, and roots files belong to root:root.

Generally it takes root permission to change file ownership as a regular user can’t take ownership of someone else’s files and can’t pass ownership of their files to another user or a group they don’t belong to.

To see what groups you belong to type groups.
We can use `ls -l` to tell us about ownership and permissions of files.

- `ls -l` - lists files and directories in the long format.

Example

```
-rw-r--r-- 1 hussam users 3775 2009-08-17 15:52 index.html
```
Cracking the Format

-\texttt{rwxrwxrwx}

- User’s Permissions
- Group’s Permissions
- Other’s permissions

\texttt{R = Read, W = Write, X = Execute}

Directory Permissions begin with a \texttt{d} instead of a \texttt{-}

What permissions would \texttt{-rw-rw-r--} mean?
Cracking the Format

-rwxrwxrwx
- User’s Permissions
- Group’s Permissions
- Other’s permissions

R = Read, W = Write, X = Execute

Directory Permissions begin with a d instead of a -

What permissions would -rw-rw-r-- mean?

User and group can read and write the file while everyone else can just read it

Hussam Abu-Libdeh based on slides by David Slater CS2042 - Unix Tools
Normal users cannot change system files and cannot globally install programs. This is a major advantage of Unix as it greatly restricts what malicious code can do. With that in mind, how do you change the permissions of your own files?

**Change Mode**

```bash
chmod <mode> <file>
```

- Changes file/directory permissions based on `<mode>`
- The format of `<mode>` is a combination of 3 fields:
  - Who is affected - a combination of u, g, o, or a (all)
  - Whether adding or removing permissions - + or -
  - Which permissions are being added/removed - any combination of r, w, x.
Examples

- `chmod ug+rx myfile` - adds read and execute permissions for user and group.
- `chmod a-r myfile` - remove read access for everyone
- `chmod ugo-rwx myfile` - removes all permissions from myfile
Changing Ownership

If you want to change the group a file you have ownership of belongs to you use the following

**Change Group**

```
chgrp group <target>
```

- Changes the group ownership of file `<target>`

If you have root access and you want to change who owns a file you use

**Change Ownership**

```
chown user:group <target>
```

- changes ownership of file `<target>`
- group is optional
- use the flag ”-R” to do a recursive change to a directory and the files within
Most commands (for which it makes sense) have a recursive option. This is used to act on every file in every subdirectory of the target.

- Usually `-r` or `-R` option (check manpage)

**Example:**

```
chmod -R o-w ~/Documents/
```

- Removes write privileges for other uses for every file and every directory in `~/Documents/`
Types of files

There are two main types of files. The first is plain text files.

**Text Files**

Plain text files are written in a human-readable format. They are frequently used for:

- Documentation
- Application settings
- Source code
- Logs
- Anything someone might want to read via a terminal

- Like something you would create in notepad
- Editable using many existing editors
Binary Files

Binaries

Binary files are written in machine code.

- Not human readable (at least without using hex editors)
- Commonly used for executables, libraries, media files, zips, pdfs, etc
- To create need some sort of binary-outputting program
Just like in windows, we can create links to files and directories. There are two types of links, hard links and symbolic links.

**Link Creation**

```bash
ln [options] <target file> [link_name]
```

- Creates a link to `<target file>` at `[link_name]`, defaulting to the current directory
- The link points to the same file on the system i.e. the link is indistinguishable from the original file
- In other words both the original file and the link both point to the same underlying object
### Symbolic Link

```bash
ln -s <target_file> [link_name]
```

- Creates a symbolic link to the target file or directory
- The link file contains a string that is the pathname of the original file or directory
- In other words the symbolic link points to the other file
Symbolic Link Examples

Say on my machine I have three partitions, one where Windows is installed, one where openSuSE is installed, and a shared fat32 partition for my documents. I then have symbolic links to the mounted fat32 partition in my home directory for Documents, Music, Videos etc.

You can see what files are symbolic links by doing `ls -l`:

```bash
hussam@rumman:~$ ls -l ~ | grep >
lrwxrwxrwx 1 hussam users 29 2009-07-26 23:53 Documents -> /mnt/newhome/hussam/Documents
lrwxrwxrwx 1 hussam users 29 2009-07-22 20:08 Downloads -> /mnt/newhome/hussam/Downloads
lrwxrwxrwx 1 hussam users 29 2009-07-22 20:07 Music -> /mnt/newhome/hussam/Music
lrwxrwxrwx 1 hussam users 29 2009-07-22 20:09 Videos -> /mnt/newhome/hussam/Videos
```
The shell is designed to allow the user to interact in powerful ways with plain text files. Before we can get to the fun stuff let's cover the basics:

### Nano

**nano filename**

- Opens filename for editing
- In terminal editor
- Since you (most likely) will be sshing into UNIX machines, this editor will do fine for everything we do in this course
- Shortcuts for saving, exiting all begin with ctrl.
Reading Files

Often we only want to see what is in a file without opening it for editing.

Print a file to the screen

```
cat <filename>
```

- Prints the contents of the file to the terminal window
- `cat <filename1> <filename2>` prints the first file then the second which is what it is really for.

More

```
more <filename>
```

- Allows you to scroll through the file 1 page at a time

Less

```
less <filename>
```

- Lets you scroll up and down by pages or lines
Sometimes you only want to see the beginning of a file (maybe read a header) or the end of a file (see the last few lines of a log).

**Head and Tail**

- `head -[numlines] <filename>`
- `tail -[numlines] <filename>`
  - Prints the first/last numlines of the file
  - Default is 10 lines

**Example**

- `tail /var/log/Xorg.0.log` - Prints the last ten lines of the log file.
We have already seen a variety of ways to print text to the screen. If we just want to print a certain string, we use

**Echo echo... echo...**

`echo <text_string>`

- Prints the input string to the standard output (the terminal)
- `echo This is a string`, `echo 'This is a string'` and `echo "This is a string"` all print the same thing
- We will see why we talk about these three cases later
The Shell has a variety of built in operators to perform specific tasks.
Bash scripting is all about combining simple commands together to do more powerful things. This is accomplished using the "pipe" character.

### Piping

<command1> | <command2>

- Passes the output from command1 to input of command2
- Works for lots of programs that take input and provide output to the terminal

### Example:

```
ls -al /bin | less
```

- Allows you to scroll through the long list of programs in /bin

```
history | tail -20 | head -10
```

- Displays the 10th-19th last commands from the current session
The && Operator

<command1> && <command2>
- Immediately after command1 completes, execute command2
- command2 executes only if command1 executes successfully

Example:

mkdir photos && mv *.jpg photos/
- Creates a directory and moves all jpegs into it
Exit Codes

The command after a `&&` only executes if the first command is successful, so how does the Shell know?

- When a command exits it always sends the shell an exit code (number between 0 and 255)
- The exit code is stored in the variable `$?`
- An exit code of 0 means the command succeeded
- The man page for each command tells you precisely what exit codes can be returned

**Example:**

```
hussam@rumman:~$ ls /Documents/cs2042
2003 2004 2007 2008 2009
hussam@rumman:~$ echo $?
0
```
Being able to display the contents of a file or edit it on an editor is all well and good, but often we want to pass the output of some command (or some set of commands) to a file.

### Redirecting to a File

```markdown
<command> > <file>
```

- redirects the output of command to file
- commands normally print to stdout
- any program that prints to stdout (terminal) can have its output redirected to a file
- useful for logging output or creating/modifying files
Example:

- `echo "This is a new file" > newfile`
  - Writes the string to ./newfile

- `cat test1 test2 > test3`
  - Concatenates test1 and test2 and stores the result in test3

Appending

- `<command> >> <file>`
  - Appends the output of command to file instead of overwriting
We have already seen some special characters:

- ** - expands to everything in the current directory
- | - used to pass the output of one file to another
- && - used to run two commands sequentially
- > - used to pass output to a file

What happens if we type

```bash
echo 3+5 > 10?
```
We have already seen some special characters:

- * - expands to everything in the current directory
- | - used to pass the output of one file to another
- && - used to run two commands sequentially
- > - used to pass output to a file

What happens if we type

```
echo 3+5 > 10
```

This writes to the file 10, 3+5. If we wanted to print 3+5 > 10 we can do it a couple of ways:

- Escape the special character > : `echo 3+5 \ > 10`
- Quote the String using either single or double quotes: `echo ’3+5 > 10?’` or `echo "3+5 > 10"`
Next Time:
- Shell Expansion
- Variables
- Quoting
- And Much more!