Midterm Review - Topics

- Correcting Mistakes on the Command Line
- Retrieving Your Last Command Line Entry
- Aborting a Running Program
- Getting Help with Unix Commands
- Quoting and Escaping
- `cd` - Change Directory
- `pwd` - Print the Current Directory
- `cat` - Print the Contents of a File
- `rm` - Delete a File
- `mkdir` - Create a Directory
- `rmdir` - Delete a Directory
- `cp` - Copy Files
- `mv` - Move a File or Directory
- `echo` - Print Text to the Terminal
- `hostname` - Print the Name of Your Host Machine

Midterm Review - Topics

- Pagers - View a File One Screen at a Time
- Pathname Completion
- `grep` - Finding Strings inside Files
- `head` - View the Top of a File
- `tail` - View the Bottom of a File
- `sort` - Print a File in Sorted Order
- `diff` - Differences between Files
- `file` - See the File Type
- Pipes - Stringing Programs Together
- `date` - Get the Date and Time
- `which` - Finding a Program File
- `whereis` - Finding Files Used by a Program
- `locate` - Search for Any File
- `who` - See Users Logged On
- `finger` - Get information on Users
- The Hierarchical Filesystem

Midterm Review - Topics

- Unix Files and Directories
- Filenames
- Case Sensitivity
- Filename Extensions
- Current Directory
- Your Home Directory
- Navigating the Hierarchical File Systems
- Hidden Filenames
- The . and .. Directory Entries
- Pathnames
- Absolute Pathnames
- Tilde ~ in Pathnames
- Relative Pathnames
  - Relative Pathnames in Your Current Directory
  - Relative Pathnames in a Subdirectory
  - Relative Pathnames above the Current Directory
  - Relative Pathnames Neither above Nor below the Current Directory

Midterm Review - Topics

- Access Permissions
- Viewing Access Permissions
- `chmod`
- Using `chmod` with Numeric Arguments
- The root Account
- Directory Access Permissions
- Links
  - The Two Types of Links
  - `ln`
  - Removing a Link
- The Monitor and Keyboard as Files
- The Keyboard and Screen as Standard Input and Standard Output
- Redirection
  - Redirecting Standard Output
  - Redirecting Standard Input
  - Redirecting Standard Output Can Destroy a File
  - Adding Output to an Existing File
- `/dev/null`

Correcting Mistakes on the Command Line

- You can correct mistakes on the Unix command line using the following ***Control*** key combinations
  - **Control A** - Move the text insertion point to the beginning of the command line
  - **Control E** - Move the text insertion point to the end of the command line
  - **Control K** - Removes everything from the current text insertion point to the end of the line
  - **Control U** - Removes everything from the current text insertion point to the beginning of the line

Retrieving Your Last Command Line Entry

- To retrieve the previous command, hit the up arrow key ↑
- You can do this several times to go back to any previous command
- The down arrow key ↓ takes you in the opposite direction

Aborting a Running Program

- You can abort a running program using Control-C

Getting Help with Unix Commands

- Many Unix utilities have a **`help`** option
- The **`help`** option provides some basic information about the command
- Some commands use **`-h`** as the help option; others use **`--help`**
- For more information, use **`man`** or **`info`**
- Follow **`man`** or **`info`** with the name of the command

Special Characters in Unix

- Some characters have special meaning in Unix
- They are

  `& : | * ? ' " [ ] ( ) $ < > { } # / \ ! ~`

- **Whitespace** characters are special too
- They are
  - Space
  - Tab
  - Newline

Special Characters in Unix

- **`Space`** and **`Tab`** separate commands, options, and arguments on the command line
- The **`Enter`** key creates a newline character on a PC, while on a Mac it is the **`Return`** key
- When the **shell** sees a newline, it executes the commands on the command line

Quoting and Escaping

- You can turn off the special meaning of a character by preceding it with a backslash ****
- You can also turn off special meanings by enclosing a string in *quotes*
- You can continue a command onto the next line by using a backslash just before hitting Enter
- The backslash turns off the special meaning of the newline character

cd - Change Directory

- To change your directory use **cd**
- If you run **cd** without an argument, it will take you to your home directory
- To go to the directory above your current directory, use

  cd ..

pwd - Print the Current Directory

- **pwd** will print you your current location in the filesystem
- **pwd** usually takes no arguments
- However, if you travelled to a directory by way of a symbolic link, then you can use the **-P** option to get the true path.

`ls` - List the Contents of a Directory

- **`ls`** lists the contents of a directory
- To see the contents of directory **`work`**, run

  `ls work`

- When you use **`ls`** without an argument it lists the contents of the current directory
- For more information run **`ls`** with the **`-l`** (long) option
- **`ls -a`** displays the "invisible" files whose name begins with a **`.`**

cat - Print the Contents of a File

- To display the contents of a file, use **cat**

```
$ cat foo.txt
foo
bar
foobar
```

- Use **cat -n** to print a number for each line of the file

```
$ cat foo.txt
    1  foo
    2  bar
    3  foobar
```

`rm` - Delete a File

- To remove a file, use **`rm`**
- To remove all files in a directory, use **`rm *`**
- Be very careful when you use this construction
- There is **no** file recovery mechanism in Unix
- **`rm`** will not remove a directory unless you use the **`-rf`** options
- This construction is also very dangerous

Directories

- ***mkdir* - Create a Directory**
  - You create a directory using ***mkdir***
- ***rmdir* - Delete a Directory**
  - ***rmdir*** is used to remove a directory
  - ***rmdir*** will not work unless the directory is empty

Files

- ***cp*** **- Copy Files**
  - ***cp*** copies files or directories
  - To copy a directory and all its files and sub-directories use ***cp*** with the -r option
- ***mv*** **- Move a File or Directory**
  - Use the ***mv*** command to move a file or directory from one place to another
  - To rename a file or directory, you also use ***mv***

echo - Print Text to the Terminal

- ***echo*** prints text to the terminal

```
$ echo Hello
Hello
```

- You can use ***echo*** to print the value of a system variable if your precede the variable name with a **$**

```
$ echo $SHELL
/bin/bash
```

hostname - Print the Name of Your Host Machine

- ***hostname*** prints the network name of the machine that you have logged on to

```
$ hostname
vm75
```

- When used with the -i option ***hostname*** will print the IP address of the machine

```
hostname -i
192.168.106.240
```

Pagers – View a File One Screen at a Time

- Pagers are programs that display the contents of a file, one screenful at a time
- The two pagers that Unix supplies are ***`more`*** and ***`less`***
  - Hitting the **`Space`** bar advances to the next screen
  - Hitting the **`Enter`** or **`Return`** key takes you down one line
- ***`less`***, just to be confusing, has more features than ***`more`***

Pathname Completion

- When typing a long file name, it is easy to make a mistake
- To make life easier, Unix provides a feature called **pathname completion**
- You type a few characters, then hit the Tab key
- Unix will supply the rest if there is only one file or directory that matches
  - If there is more than one match Unix will supply as much as it can and then beep. If there is no match, it will also beep
  - If you don't get a complete match, hitting Tab twice will display a list of all possible matches
- This only works in the *Bash* shell

grep - Finding Strings inside Files

- ***grep*** is used to find all lines in a file that contain a certain **string**
- ***grep*** takes two arguments
  - The string you are searching for
  - The file or files in which to search
- ***grep*** has the following format

```
grep  STRING  FILE  [FILE ...]
```

grep - Finding Strings inside Files

- To run ***grep*** on the files in a directory use the -r (**r**ecursive) option
- ***grep -r*** will search through all files in a directory and in all subdirectories
- ***grep***, like Unix, is case sensitive
  - It thinks of "foo" and "FOO" as two different strings
  - To have ***grep*** ignore case, run it with the **-i** option
- To have ***grep*** find all lines that **do not** match a string run it with the -v option

More File Viewing...

- ***head*** **- View the Top of a File**
  - ***head*** prints the first 10 lines of any file
  - If you want a different number of lines follow ***head*** with a dash, **-** , and a number

```
head -20 foo.txt
```

- ***tail*** **- View the Bottom of a File**
  - ***tail*** prints the last 10 lines of any file
  - If you want a different number of lines follow ***tail*** with a dash, **-** , and a number

```
tail -20 foo.txt
```

sort - Print a File in Sorted Order

- ***sort*** prints a sorted list of the lines in a file to the terminal
  - ***sort*** **does not** change the file
  - ***sort*** sorts the lines of a file by the first few characters in the line
- To sort in reverse order use the -r option
- ***sort***, by default, sorts in alphabetical order
- This is a problem when the first characters on a line are numbers
  - That's because 11 will sort before 2
  - To sort in numerical order use ***sort -n***
  - To sort in reverse numerical order use ***sort -nr***

diff - Differences between Files

- ***diff*** compares two files and notes their differences
- ***diff*** was created for use with the ***patch*** utility
- Run ***diff*** with the **-y** option to get output that is easier to read

file - See the File Type

- The ***file*** utility can be used the determine the type of a file:

```
$ file *
class_notes.css:             ASCII text
common_unix_commands.html:   HTML document text
cs285L:                      directory
emacs_cheat_sheet.html:      HTML document text
index.html:                  HTML document text
it244:                       directory
tips.html:                   HTML document text
unix_cheat_sheet.html:       HTML document text
work.txt:                    ASCII text
```

Pipes – Stringing Programs Together

- A pipe takes the output of one command and feeds it into the input of another command
- Pipes allow you to chain together several Unix commands into a *single* command
  - Commands joined in this way are sometimes called **pipelines**
  - Pipes are essential to the Unix philosophy of simple tools
  - Using pipes, you can string together many commands to achieve exactly what you want

Pipes - Stringing Programs Together

- You form a pipe by placing the vertical line character | between two commands

```
$ head -5 red_sox.txt
2011-07-31  Red Sox @  White Sox    Win 5-3
2011-07-02  Red Sox @  Astros       Win 7-5
2011-07-03  Red Sox @  Astros       Win 2-1
2011-07-04  Red Sox vs Blue Jays    Loss 7-9
2011-07-05  Red Sox vs Blue Jays    Win 3-2

$ head -5 red_sox.txt | sort
2011-07-02  Red Sox @  Astros       Win 7-5
2011-07-03  Red Sox @  Astros       Win 2-1
2011-07-04  Red Sox vs Blue Jays    Loss 7-9
2011-07-05  Red Sox vs Blue Jays    Win 3-2
2011-07-31  Red Sox @  White Sox    Win 5-3
```

Pipes – Stringing Programs Together

- Notice, in the command line above, that ***sort*** does not have an argument
  - Normally, ***sort*** requires an argument that specifies the file to sort
  - But in a pipe, each command after the first takes its input from the output of the preceding commands
  - You **never** need to specify the input when using a command in a pipeline except the first command

date - Get the Date and Time

- The Unix ***date*** command will give the ***time*** and the ***date***

```
$ date
Tue Aug 21 10:20:05 EDT 2012
```

- If we follow the command with a **+** and a string we can change the format

```
$ date +"%Y-%m-%d"
2012-08-21
```

- Use ***info*** or ***man*** to see the various formatting options ***date*** provides

which - Finding a Program File

- Unix commands are programs that are located somewhere in the filesystem
- The Unix utility ***which*** gives the location of an executable file
- To find where the executable file for ***less*** is located, we can run ***which*** like this

```
$ which less
/usr/bin/less
```

which - Finding a Program File

- ***which*** uses the **PATH** system variable to find the executable file
- We'll discuss **PATH** in a future class

whereis - Finding Files Used by a Program

- ***whereis*** is another program that can be used to locate program files
- ***whereis*** takes a different approach than ***which***
  - Every Unix or Linux system has certain standard places where it stores programs and the files they use
  - ***whereis*** searches these locations
  - It returns a list of all files associated with a program

whereis - Finding Files Used by a Program

- When we run ***whereis*** on the ***tar*** utility we get more information than ***which*** returned

```
$ whereis tar
tar: /bin/tar /usr/include/tar.h /usr/share/man/man1/tar.1.gz
```

- The *first* entry is the executable file
- The *second* is a header file
- The *third* is the file that ***man*** used to provide information about ***tar***

locate - Search for Any File

- ***which*** and ***whereis*** only work on programs
- ***locate*** can be used to find any file
- You don't need to know the full name of a file to use ***locate***
- ***locate*** will search on a partial file name

```
$ locate foot
/etc/update-motd.d/99-footer
/usr/share/doc/java-common/debian-java-faq/footnotes.html
...
```

locate - Search for Any File

```
...
/usr/share/emacs/23.3/lisp/mail/footnote.elc
/usr/share/emacs/23.3/lisp/org/org-footnote.elc
/usr/share/libparse-debianchangelog-perl/footer.tmpl
/usr/share/xml-core/catalog.footer
/usr/src/linux-headers-3.0.0-12/arch/arm/mach-footbridge
/usr/src/linux-headers-3.0.0-12/arch/arm/mach-footbridge/Kconfig
/usr/src/linux-headers-3.0.0-12/arch/arm/mach-
                               footbridge/Makefile
/usr/src/linux-headers-3.0.0-12/arch/arm/mach-
                               footbridge/Makefile.boot
/usr/src/linux-headers-3.0.0-12/arch/arm/mach-footbridge/include
...
```

locate - Search for Any File

```
/usr/src/linux-headers-3.0.0-12/arch/arm/mach-
                        footbridge/include/mach
/usr/src/linux-headers-3.0.0-12/arch/arm/mach-
                        footbridge/include/mach/debug-macro.S
...
```

- ***locate*** does not actually search the file system itself
  - That would take too long
  - Instead, it uses a database of all files on the system
    - This database is created by another program ***updatedb***
    - ***updatedb*** is usually run automatically in the background to update the database

who - See Users Logged On

- ***who*** prints a list of all users currently on the machine

```
$ who
ghoffmn  pts/0    2012-08-12 13:41 (dsl092-066-
                                    161.bos1.dsl.speakeasy.net)
rouilj   pts/1    2012-08-12 04:25 (pool-74-104-161-
                                    40.bstnma.fios.verizon.net)
eb       pts/2    2012-08-12 08:19 (pool-96-237-251-
                                    11.bstnma.fios.verizon.net)
```

- ***who*** also provides information about each user's login session

who - See Users Logged On

- ***who am i*** will show the user who is logged into a terminal

```
$ who am i
ghoffmn  pts/0      2012-08-12 13:41 (dsl092-066-
                    161.bos1.dsl.speakeasy.net)
```

finger - Get information on Users

- ***finger*** provides information about a user account

```
$ finger ghoffmn
Login: ghoffmn                          Name: Glenn Hoffman
Directory: /home/ghoffmn                Shell: /bin/bash
On since Wed Sep 17 16:09 (EDT) on pts/1 from dsl092-066-
                                          161.bos1.dsl.speakeasy.net
   1 second idle
Mail forwarded to glennhoffman@mac.com
Mail last read Thu Sep  4 15:12 2014 (EDT)
Plan:
Office:          McCormack M-3-607                 Fall 2014
Office Hours:    Tuesday & Thursday, 10:00 - 12:00 PM and by appointment
Classes:
    IT 341-2  Introduction to System Administration  TuTh  12:30-1:45
                                          S3-148  (IT Lab)
    IT 244-1  Introduction to Linux/Unix             TuTh   2:00-3:15
                                          S3-028  (Web Lab)
...
```

finger - Get information on Users

- ***finger***, like ***mv***, has two functions
  - When used without an argument, ***finger*** shows every user currently logged in

```
$ finger
Login     Name                 Tty      Idle  Login Time ...
ghoffmn   Glenn Hoffman        pts/0          Aug 18 11:13 ...
rouilj    John P. Rouillard    pts/1    4:34  Aug 18 06:44 ...
ubuntu    Ubuntu Dummy        *tty1      14d  Aug  4 04:53 ...
```

finger - Get information on Users

- You can also use a last or first name with ***finger***

```
$ finger Hoffman
Login: ghoffmn                          Name: Glenn Hoffman
Directory: /home/ghoffmn                        Shell: /bin/bash
On since Wed Sep 17 16:09 (EDT) on pts/1 from dsl092-066-
                                       161.bos1.dsl.speakeasy.net
   1 second idle
Mail forwarded to glennhoffman@mac.com
...

Login: it244gh                          Name: Dummy for Glenn Hoffman
Directory: /home/it244gh                        Shell: /users/nologin
Never logged in.
Mail forwarded to glennhoffman@mac.com
...
```

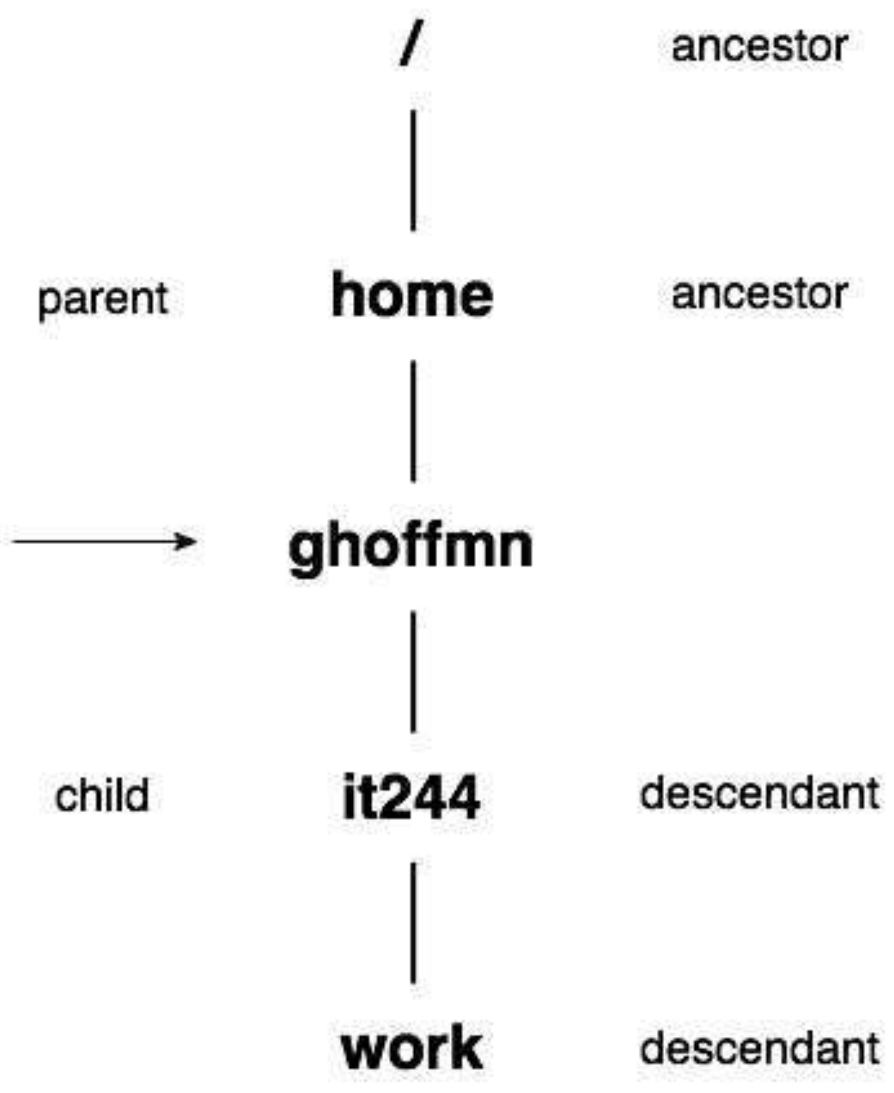
The Hierarchical Filesystem

- Unix uses a **hierarchical filesystem**
- This means there is one directory at the top, called the **root directory**
  - The root directory is indicated by a simple slash character **/**
  - All other directories are contained within the root directory or one of its many subdirectories
- This structure is called a tree because it looks like a tree turned upside down

The Hierarchical Filesystem

- A hierarchical filesystem also resembles a family tree
- So, we often use terms that describe family members when talking about directories:
  - The directory up one level from your current directory is called the **parent directory**
  - All directories above the current directory are called **ancestors**
  - All directories inside the current directories are called **child directories**
  - All directories below the current directory can be called **descendants**
  - All directories and files within the same parent directory are called **siblings**

For example...


/
ancestor
parent
home
ancestor
ghoffmn
child
it244
descendant
work
descendant

Unix Files and Directories

- **Files** are sequential arrangements of data on disk
- There are several types of files
  - Program files
  - Text files
  - Data files
  - Configuration files
- For the user, directories are simply containers that hold files

Unix Files and Directories

- Unix tends to treat everything is sees as a file
  - Unix even considers devices, such as printers, as files
  - Directories are files too, as far as Unix is concerned
- You cannot run ***cat***, ***more***, or ***less*** on a directory
- The information that directory files contain can only be accessed by system programs and system calls

Filenames

- When you ask Unix for a file you must give it two pieces of information
  - The name of the file
  - The location of the file in the hierarchical file system
- Every file has a filename
  - The maximum number of characters permitted in a filename varies from one Unix to another
  - Most Unix flavors allow file names of up to 255 characters
- It is best to keep filenames short because this makes typing and remembering them easier

Filenames

- Never use a space in a file or directory name
  - This is a **bad idea**
  - Use an underscore, _ , instead of a space in file names
- To avoid problems, only use the following characters in file names:

```
Uppercase letters (A-Z)        Underscore _
Lowercase letters (a-z)        Dash -
Digits (0-9)                   Period .
```

- You cannot have two files with the same name in the same directory

Case Sensitivity

- Unix is **case sensitive**
  - This means that "Foo", "foo" and "FOO" are three different things as far as Unix is concerned
  - Unix utility and program names are always lowercase
- Some operating systems do not distinguish between UPPERCASE and lowercase characters
  - Windows is such a system
  - Make life easy for yourself
  - Use only lowercase characters in Unix filenames

Filename Extensions

- Extensions are strings of characters that appear at the end of the filename after a period
  - Extensions are **not** recognized by the Unix filesystem
  - As far as Unix is concerned they are just legal characters that are part of the filename
- Some Unix programs expect their files to have certain extensions
  - For example, the C compiler, ***gcc*** expects the filenames of source code files to end in **.c**
  - The Java compiler, ***javac*** expects Java source files to have **.java** at the end of the filename
- These extensions are required by the program not by Unix

Current Directory

- The way a Unix command works depends, somewhat, on your environment
- One of the most important parts of your environment is your **current directory**
- The ***pwd*** (**p**rint **w**orking **d**irectory) command will always tell you your current directory
- If a command expects a directory as an argument, then you can usually omit it and the program will assume you mean you current directory
- For example, ***ls*** used with no arguments will list the contents of your current directory

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
- If you use ***cd*** with no arguments, it will take you to your home directory

```
$ cd

$ pwd
/home/ghoffmn
```

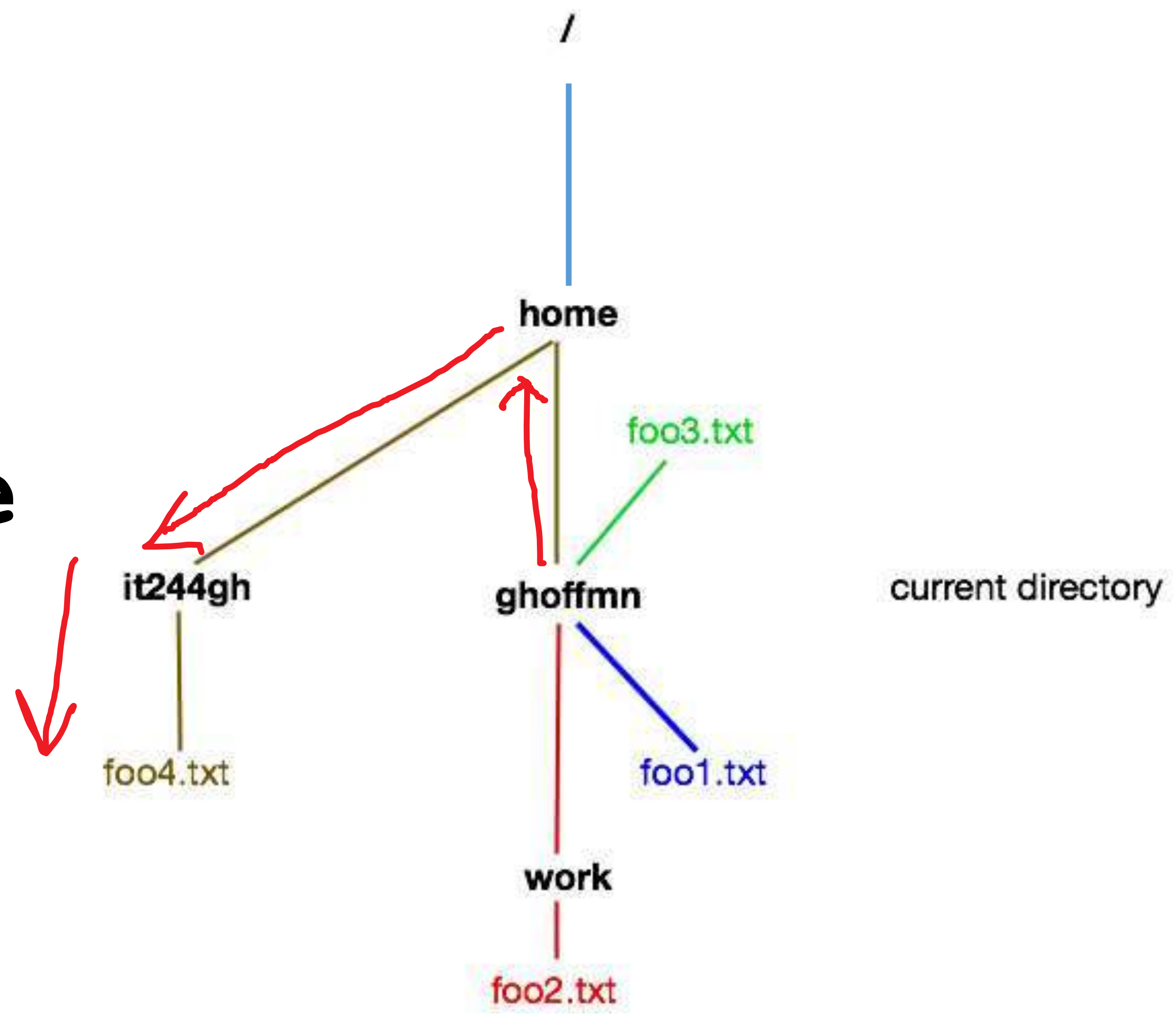
Your Home Directory

- Your home directory contains a number of hidden files which customize your environment like `.forward`
- On most Unix systems, home directories are found inside the `/home` directory
- On a Mac, home directories appear inside `/Users`
- ***The name of your home directory is the same as your Unix username***

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

Off to the side...



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
- These files are used to configure your Unix environment

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
- . 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

Pathnames

- Every file has a **pathname** which is used to access the file
  - A pathname has two components
    - The name of the file
    - A **path** to reach the file
  - The path is a list of directories that you must go through to reach the file you want
  - A pathname is like an address on a letter a name and directions to get there
- The name of the file is always at the ***end*** of a pathname

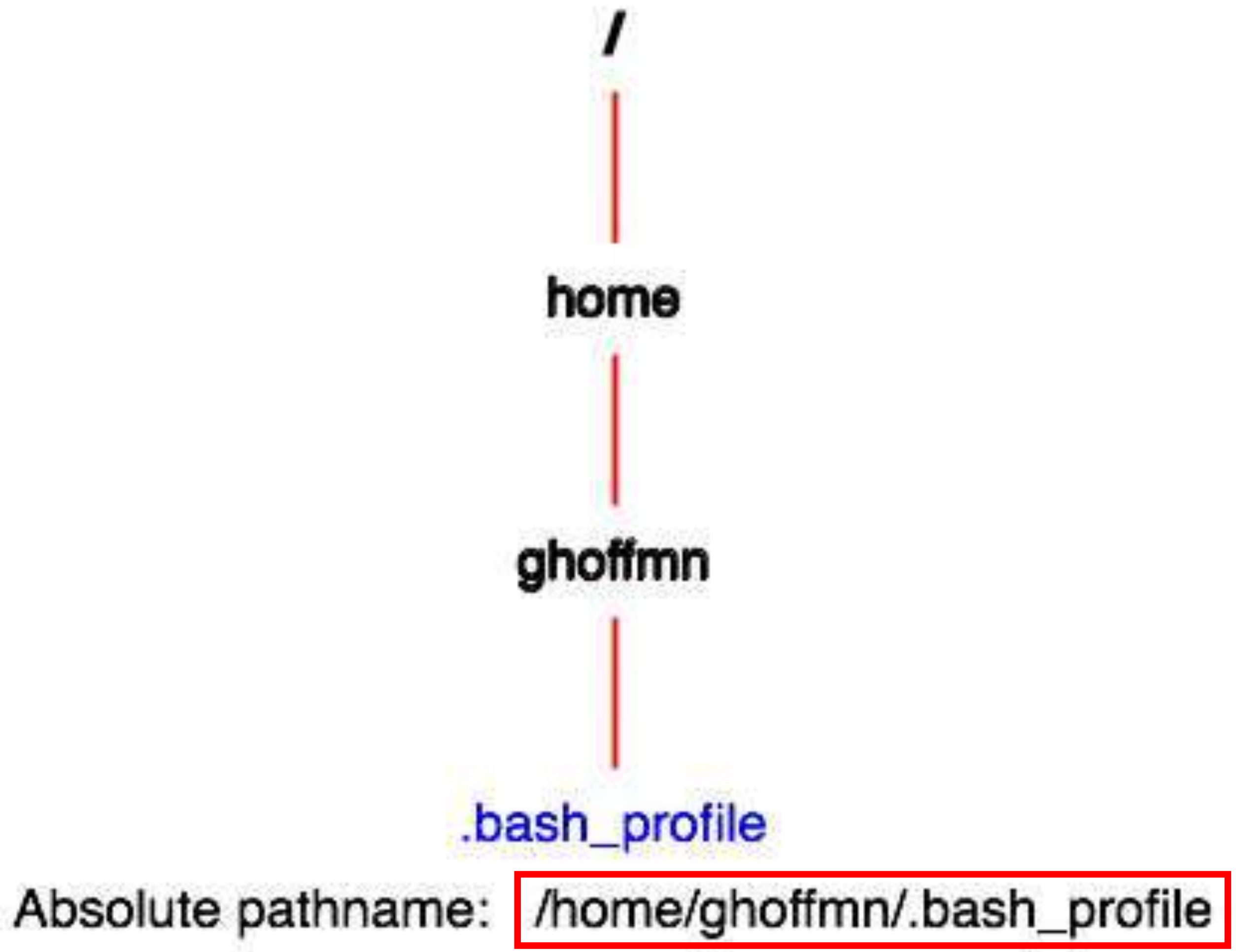
Pathnames

- When the slash **/** appears **between** names in a pathname it is used to separate a directory name from what comes after it
- When a **/** is the first character in a pathname it stands for the **root directory**
- There are two types of pathnames
  - Absolute
  - Relative

Absolute Pathnames

- The top of the filesystem is a directory called the **root directory**
  - The root directory 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 directory 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



Tilde ~ in Pathnames

- There is one form of 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
  - When you put a tilde in front of a Unix username it stands for the home directory of that account
- ~ **always means an absolute path**

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
- As far as Unix is concerned it makes no difference whether you use and absolute or relative pathname

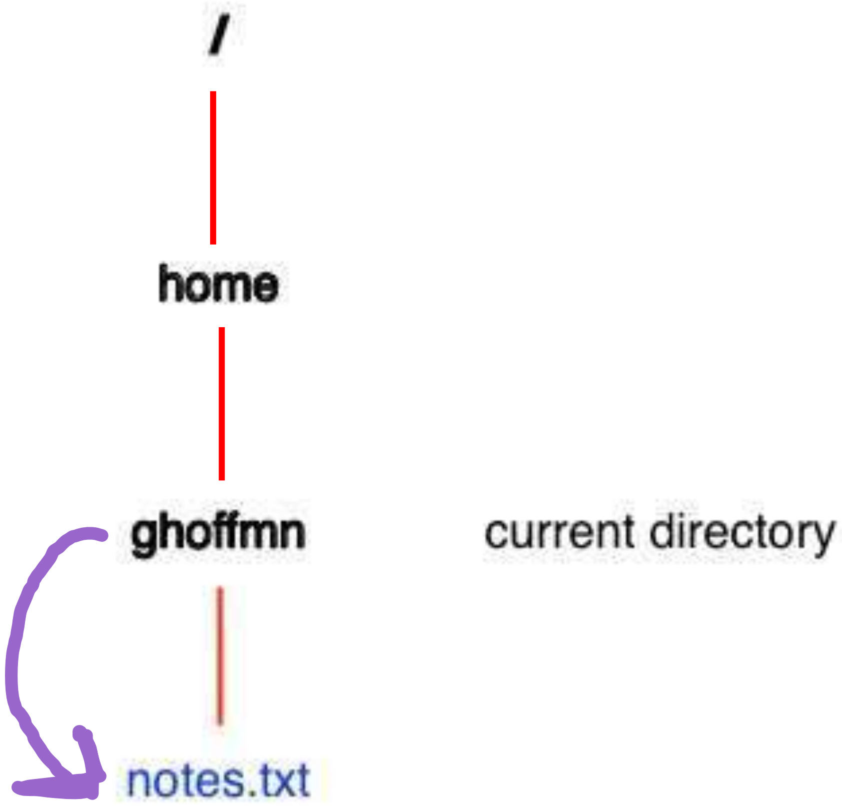
Relative Pathnames

- There are four types of relative pathnames:
  1. When the file is in your **current** directory
  2. When the file is in a **subdirectory** of your current directory
  3. When the file is in a directory that is an **ancestor** of your current directory
  4. When the file is in a directory that is **neither** an ancestor or descendant of the current directory
- A relative pathname of a file or directory inside your current directory is simply *the **name** of that file or directory*

Relative Pathnames in a Subdirectory

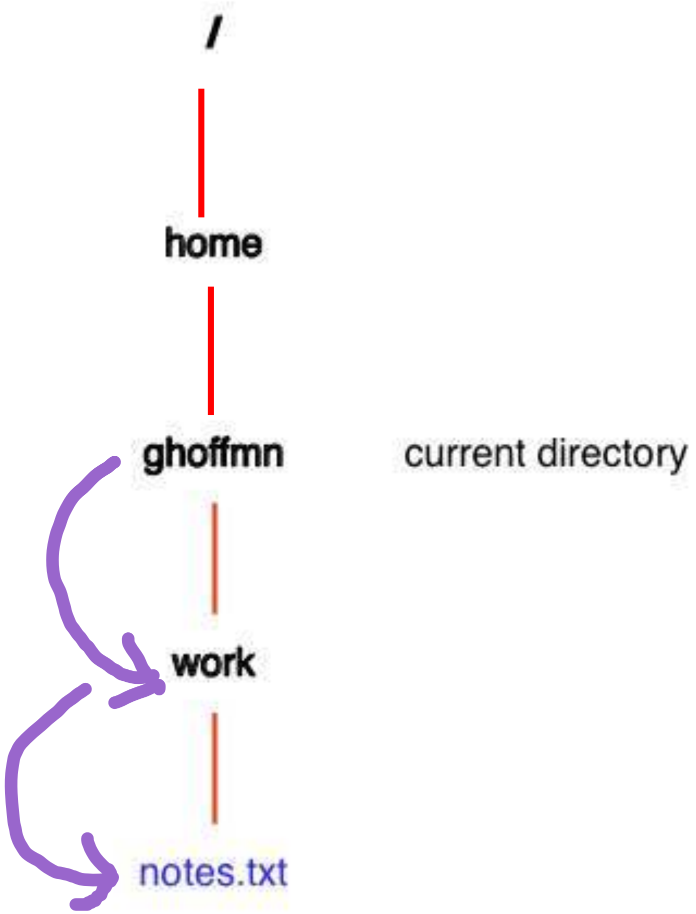
- Things get a little more complicated when you are dealing with a file in a subdirectory
- Here, you must list every directory between your current directory and the file you want
- You must use a slash **/** to separate the name of each directory from what comes after it

Current Directory



Relative pathname: notes.txt

In a Subdirectory



Relative pathname: work/notes.txt

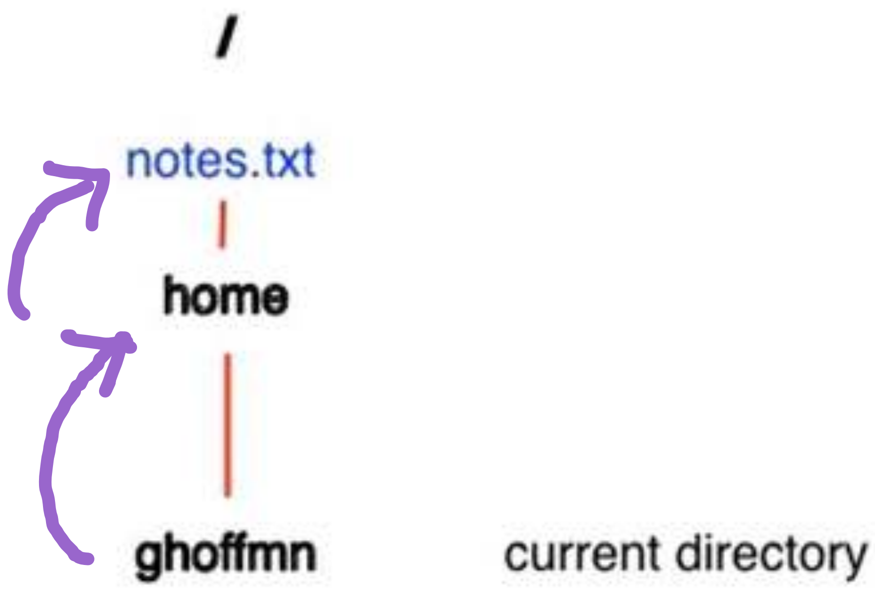
Relative Pathnames above the Current Directory

- 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 in the path
- Use a slash **/** between each **..**

Relative Pathnames neither Above nor Below the Current Directory

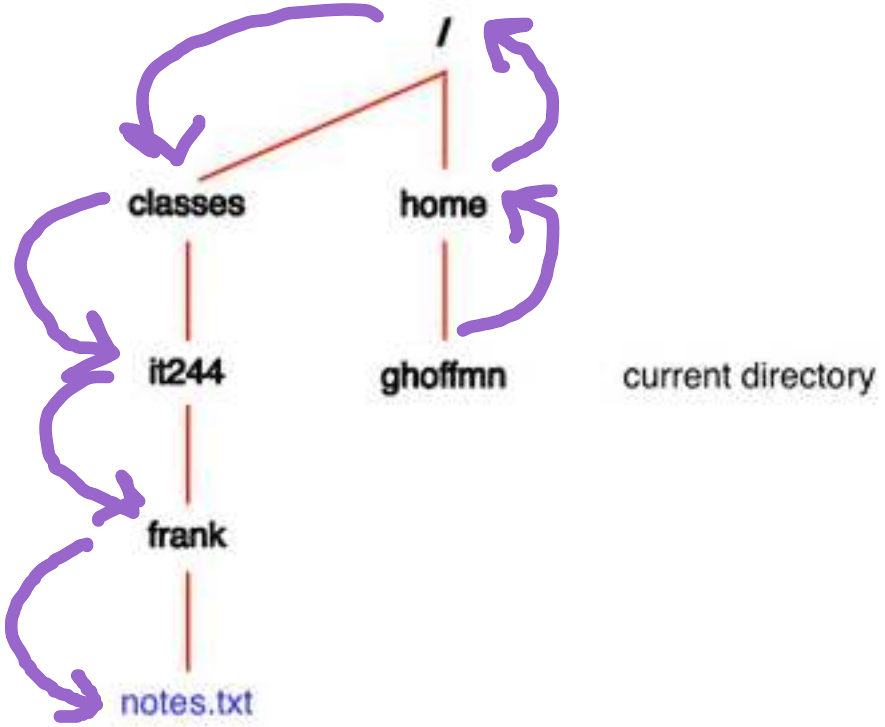
- ***What if the file is neither above nor below?***
- Here, you have to go ***up*** to a common ancestor and then ***down*** to the directory that holds what you want
- The path starts with one or more **..**
- You keep going **up** until you get to a directory that is an ancestor of both your current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

In an Ancestor Directory



Relative pathname: ../../notes.txt

Neither Above nor Below



Relative pathname: ../../classes/it244/frank/notes.txt

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 the 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

Access Permissions

- Each access permission can be either ***on*** or ***off***.
- If you have **read permission** on a file you can look at the data in the file
- If you *only* have read permission, you cannot change a file
- To change a file, you need **write permission**
- You cannot run a program or script file unless you have **execute permission** on that file

Access Permissions

- Each of the three permissions is set either **on** or **off** to three classes of users:
  - The owner
  - The group
  - Every other account
- Every file or directory has an owner
- The account that created the file is usually the **owner**

Access Permissions

- 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 everyone else any account that is not the owner or a member of the group

Viewing Access Permissions

- To view the access permissions of a file or directory use ***ls -l***

```
$ ls -l
total 5
-rw------- 1 it244gh libuuid 316 2011-09-20 21:32
                         dead.letter
lrwxrwxrwx 1 it244gh libuuid  34 2011-09-06 13:21 it244 ->
                         /courses/it244/s12/ghoffmn/it244gh
drwx------ 2 it244gh libuuid 512 2011-09-07 15:03 mail
drwxr-xr-x 2 it244gh libuuid 512 2011-09-25 15:48 test
-rw-r--r-- 1 it244gh libuuid  15 2011-09-20 16:18 test.txt
```

Viewing Access Permissions

- 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** (el) indicates a link
- The next 3 characters indicate the ***owner's*** permissions:
  - **r** means the owner has read permission
  - **w** means the owner has write (change) permission
  - **x** means the owner has execute (run) permission
  - **-** means the owner does not have the permission that would normally appear in this column

Viewing Access Permissions

- The next three characters give the permissions of the ***group***
- The last three characters are the permission of all ***other*** accounts
- The next column 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
- After that, you will find the ***group*** assigned to the file or directory

chmod

- When a file is created, it has certain default permissions

```
$ touch test.txt
$ ls -l test.txt
-rw-r--r-- 1 it244gh libuuid 0 2012-09-17 14:40 test.txt
```

- To change these permission, you need to use ***chmod***
- Only the owner of a file can do this
- ***chmod*** requires two arguments
  - The permissions you want to grant
  - The name of the file(s) or directory(s) to which the change will be applied

chmod

- The format for a call to ***chmod*** is

```
chmod   PERMISSIONS   FILES_OR_DIRECTORIES
```

- The permission can be specified in two ways
  - ***Symbolically***, using letters and the plus and minus signs
  - ***Numerically***, using three digits running from 0 to 7
- I will teach the *numeric* format for expressing permissions
  - You are free to read about the symbolic format in the textbook
  - I will not deduct points for using symbolic format, as long as you use it *correctly*

Using *chmod* with Numeric Arguments

- The numeric permissions format uses three digits, where each digit is a number from 0 to 7:
  - The first digit gives the permission of the owner
  - The second digit gives the permissions assigned to the group
  - The third digit gives the permissions for every other account
- How do you get three pieces of information out of one number?
- By adding powers of two.

Using *chmod* with Numeric Arguments

- Each digit is the sum of three other numbers; when constructing the number, you add
  - 4 if you want to give read permission
  - 2 if you want to give write permission
  - 1 if you want to give execute permission
- Notice that all the number are powers of two; if we write these values in binary notation
  - 100 represents 4
  - 010 represents 2
  - 001 represents 1

Using `chmod` with Numeric Arguments

- A single decimal digit from 0 to 7 is represented by 3 binary digits
- This is how we get three pieces of information out of one digit
  - For example, to give full permissions I would add
    - 4 for read permission
    - 2 for write permission
    - 1 for execute permission
  - So the total, 7, represents all three permissions

Using *chmod* with Numeric Arguments

- Try to remember this sequence

read	write	execute
4	2	1
owner	group	everyone

- Remember that you need ***three*** of these digits to specify the full permissions for a file or directory

The `root` Account

- On every Unix or Linux system, there is a special account named **root**
- **`root`** can access any file or run any program
  - **`root`** is an administrator account
  - It is used for system configuration and maintenance
- Even a system administrator should not log in as **`root`**
- Instead, he or she should use a regular Unix account and use ***`sudo`*** when running a command that needs root privileges

`The root Account`

- ***`sudo`*** allows a user to run a command that normally only root can run
  - When you run ***`sudo`***, it asks you for **your** password not the password of the root account
  - In order to run ***`sudo`***, you must be on the ***sudoers*** list, a change which only the root account can make

Directory Access Permissions

- The Unix **access permissions** work a little differently for directories than they do for files
- Read and write permissions for a directory are similar to those for a file
  - Read permission allows you to list the contents of that directory using ***ls***
  - Write permission allows you to create, delete or change the name of **entries** in that directory
    - Write permission on a **directory** does not allow you to change the **contents** of a file in that directory

Directory Access Permissions

  - Write permission on a directory **does not apply to the directory itself**
  - If you have write permission on a directory, then you can change what's inside it, but you cannot rename the directory or delete it – unless you have write permission on its **parent** directory

- *Execute* permission on a directory allows you to do two things
  - It allows you to enter that directory using ***cd***
  - It also allows you to read a file in that directory…**if** you already have read permission on that file and know the name of that file

Links

- **Links** are like shortcuts on a Windows machine or aliases on a Mac
- Links allow you to move around the filesystem using short names
- Each of you has an entry in your home directory called `it244`
- In the home directory of my test account, cs110ck, I have such a link...

Links

```
$ ls -l it244
lrwxrwxrwx 1 cs110ck faculty 34 Sep  2 13:10 it244 ->
                                   /courses/it244/f16/ckelly/cs110ck
```

- This is a link to `/courses/it244/f16/ckelly/cs110ck`
- If you ***cd*** into this location and use ***pwd***

```
$ pwd
/home/cs110ck

$ cd it244

$ pwd
/home/cs110ck/it244
```

Links

- This path reflects the route you took to get here
  - But it is **not** the real pathname of the directory
  - You can only get that information if you use ***pwd*** with the -P (note the capitalization) option

```
$ pwd
/home/cs110ck/it244

$ pwd -P
/courses/it244/f16/ckelly/cs110ck
```

The Two Types of Links

- There are two types of links
  - Hard links
  - Symbolic, or soft, links
- **Hard links** are older
  - A hard link is like a duplicate file name
  - Hard links can only point to files not directories
  - You can only have a hard link to a file if that file is on the same hard disk volume as the link

The Two Types of Links

- **Symbolic links** are much more flexible
  - You can use either an absolute or relative pathname when creating a symbolic link
  - A symbolic link can point to a file or directory anywhere in the filesystem
  - Deleting a symbolic link does not delete the file or directory it points to

`ln`

- To create a ***symbolic*** or soft link, use ***`ln`*** with the -s option
- ***`ln`*** takes two arguments, a pathname and the name for the link

```
$ pwd
/home/it244gh

$ ln -s ~ghoffmn/examples_it244 examples

$ ls -l examples
lrwxrwxrwx 1 it244gh libuuid 28 2012-09-17 17:53
           examples -> /home/ghoffmn/examples_it244
```

Removing a Link

- To delete a link, use ***rm***
- If you delete a symbolic link, it will not affect the file or directory it points to

Syntax of the Command Line

- A command typed at the command line has *this* format:

```
COMMAND [OPTIONS] [ARG1] [ARG2] ... [ARGn]
```

- The brackets indicate that the contents are optional
- Commands vary in the number of options and arguments they accept
    - Some accept ***none***
    - Others require a ***specific*** number of arguments
    - Still others accept a ***variable*** number of arguments
- Arguments must be separated by one or more spaces

Command Options

- Options modify the behavior of the command
- Options are usually preceded by one or two dashes **-**
- GNU programs frequently have options that are preceded by two dashes **--**
- The options in GNU programs are usually words
- Options in other Unix programs are usually single-letter
    - When a command uses a single dash **-** before an option, you can usually combine options following the dash
    - An example of this is ***`ls -ltr`***

Command Options

- Options using two dashes **--** usually cannot be combined
    - In this case, each option must be written separately and preceded by two dashes
- Sometimes the option can have its own argument
- Utilities that report the size of files usually do so in bytes
    - Such utilities often have a -h, or --human-readable, option
    - With this option, the file size will be displayed in kilobytes, megabytes or gigabytes, as appropriate

Command Options

- Many commands display a help message when run with the **--help** option
- All GNU utilities accept this option

`tty`

- ***`tty`*** is the terminal **device driver** and is part of the kernel
- As you type each character at the command line ***`tty`*** looks at the character and takes appropriate action
  - Most of the time, ***`tty`*** just takes the character and places it in a buffer
  - It responds differently to the special editing characters

`Backspace`
`The arrow keys`
`Control-A`
`Control-E`
`Control-U`
`Control-K`

`tty`

- ***`tty`*** is where all **command line editing** takes place
- When ***`tty`*** sees a newline character, which is what you get by hitting **`Enter`** (PC) or **`Return`** (Mac), it passes the contents of the buffer to the shell

Parsing the Command Line

- The shell takes the command line and breaks it up into **tokens**
  - Tokens are the characters you that print and are separated from each other by whitespace
  - The act of breaking up text into tokens is called **parsing**
- Next, the shell looks for the name of the command
- Usually, the command name is the first string on the command line

Parsing the Command Line

- The command can be specified by a simple filename

  `ls`

- Or by using a **pathname**

  `/bin/ls`

The PATH System Variable

- To run a program, a **process** must be created
  - The shell cannot create a process; only the kernel can
  - The shell asks the kernel to start the process but in order to do that, it has to give the kernel a **pathname** for the executable file for that program
  - Most of the time, when you run a program ,you don't use a pathname; you simply use the name of the program
- To turn the name of a program into a pathname for the executable file the shell uses the **`PATH`** variable

The PATH System Variable

- **PATH** contains a list of directories to search to find the program 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 set when Unix or Linux is installed
- The absolute pathname of each directory is separated from the next by a colon **:**

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

The PATH System Variable

- 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 executable file but you do not have execute privileges, it will tell you this in an error message
- You can modify the **PATH** variable in your own Unix environment

Running a Program in the Current Directory

- For security reasons, it is never advisable to put the current directory **.** in the **PATH** list
- Then how do you run a program inside your current directory?
- You can do this using the following construction

  ./PROGRAM_NAME
- This will always work regardless of the contents of **PATH**

Running the Command Entered on the Command Line

- When the **shell** gets the command line from ***tty***, it uses **`PATH`** to find the executable file to run
- The shell then asks the kernel to start a process for that program
  - A process is a running program, and it needs resources to do its job
    - Memory
    - Access to files
    - Time in the machine CPU
  - Each process has memory allocated to it that it alone can use
  - This prevents one program from interfering with another

Running the Command Entered on the Command Line

- The shell also gives the program the list of tokens from the command line
  - The name used to call the program
  - The options used
  - The arguments used
- The shell does not check the options or arguments
- While the program is running, the shell goes into an inactive state known as "sleep"

Running the Command Entered on the Command Line

- When the program finishes, it must send an **exit status** to the shell
  - The exit status is an integer that must be 0 or greater
  - An exit status of 0 indicates that the program was able to do its work without error
  - Any exit status greater than zero indicates an error
  - A program can issue different error status values for different types of errors

Running the Command Entered on the Command Line

- You can see the exit status of the last program by looking at the value of the system variable ?

```
$ cat foo
cat: foo: No such file or directory

$ echo $?
1
```

Standard Input, Standard Output and Standard Error

- Every Unix process always has access to 3 different "files"
  - Standard Input
  - Standard Output
  - Standard Error
- Unix thinks anything it can write to or read from is a file
- **Standard input** is where the program gets input when a specific source (like a file or a device) is not specified
  - By default, standard input is the keyboard

Standard Input, Standard Output and Standard Error

- **Standard output** is where the program sends its output if a specific file or device is not mentioned
  - By default, standard output is the terminal
- **Standard error** is where the program sends error messages
  - By default, standard error goes to the same destination as standard output: ***the terminal***

The Monitor and Keyboard as Files

- Unix thinks of anything it can read from or write to as a file
- The combination of a keyboard and a monitor is called a terminal
- Unix can read what you are typing at the keyboard and it can send output to the monitor so it thinks of the terminal as a file
- The **device driver** **`tty`** handles input from the keyboard and output to the terminal

The Monitor and Keyboard as Files

- **`tty`** allows Unix to talk to the "file" that is the terminal
- When you connect to a Unix/Linux machine using ***`ssh`***, your PC is the terminal

The Keyboard and Screen as Standard Input and Standard Output

- When a command or script does not specify where input is to come from it comes from **standard input**
  - By default, standard input is keyboard
- When a command or script does not specify where output should go it goes to **standard output**
  - By default, standard output is the screen
- When a command or script does not specify where error messages should go they goes to **standard error**
  - By default, standard error also is the screen

Redirection

- **Redirection** is telling Unix to take data from or send data to a different place than usual
- Redirection is one of the features that makes Unix so flexible
    - You can take input from something other than the keyboard like a file
    - You can send output to something other than the screen like another file

Redirection

- Redirection is what makes **pipes** possible
  - When you set up a pipe you are sending the output from one program into the input of another
  - You are redirecting the *output of the first command* from the terminal to the *input of the second command*
  - This allows the next command to take its input from something other than a file

Redirecting Standard Output

- To redirect standard output use the greater than symbol **>** followed by a filename
- This tells Unix to send the output from a command to the file or device that appears after the symbol
- The format for output redirection is

```
COMMAND  [ARGUMENTS]  >  FILENAME
```

- If the file does not already exist it will be created

Redirecting Standard Input

- When we redirect standard output, we send output to something other than the screen
- When we redirect standard input, we take input from something other than the keyboard
- To do this, we use the less than symbol **<**
- Here is the format

```
COMMAND [ARGUMENTS] < FILENAME
```

Adding Output to an Existing File

- If you redirect standard output to a file that already exists, you will overwrite the contents of that file
- You will replace the original contents of the file with new data
- But Unix allows you to add to the bottom of a file
- This is called **appending**
  - The append symbol is two greater than symbols with no space in between **>>**
  - The format is

```
COMMAND   [ARGUMENTS] >>   FILENAME
```

`/dev/null`

- Sometimes a program will do something useful but produce output you don't want
- For situations like this, Unix provides `/dev/null`
  - Any output you send to `/dev/null` will disappear
  - It will never appear on the screen
  - If you redirect input to come from `/dev/null`, then the command will receive an empty string
- `/dev/null` is most useful when dealing with error messages
  - Since error message normally go to the terminal they will be mixed up with the regular output
  - Sending error messages to `/dev/null` prevents this from happening