Command Line Syntax And Standard I/O

- Syntax of the Command Line
  - Command Options
  - `tty`
  - Parsing the Command Line
  - The `PATH` System Variable
- Running Executables
  - Running a Program in the Current Directory
  - Running the Command Entered on the Command Line
- Main Data Streams
  - Standard Input
  - Standard Output
  - Standard Error

Syntax of the Command Line

- The syntax of the command line is straightforward
  - First, comes the command
  - Then, perhaps, some options
  - Finally…some arguments
- The command is executed when you hit the **Enter** key

```
COMMAND  [OPTIONS]  [ARG_1]  [ARG_2]  ...  [ARG_N]
```

- The brackets indicate that the contents are optional

Syntax of the Command Line

- Commands vary in the number of 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*

```
cp   -r   /etc   /tmp/etc.backup
```

Command Options

- Many commands have 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
    - The options in *other Unix* programs are usually a single letter

Command Options

- When a command uses a single dash **-** before an option, you can usually combine options behind a single **-**
  - An example of this is `ls -ltr`
  - This means run `ls`
    - To get a *long* listing
    - *Sorted* by modification date and time
    - In *reverse* order
- Options using two dashes **--** cannot usually be combined
- In this case, each option must be *written separately* and preceded by two dashes

Command Options

- Sometimes, the option can have its own argument
- When this happens, the argument is usually separated from the option by spaces

```
gcc  -o  prog  prog.c
```

- Utilities that report the size of files usually do so in *bytes*
  - This works well with small files
  - But with large files, a size in bytes can be hard to read
    - 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

- **df** (**d**isk **f**ree) shows the amount of space on the various filesystems

```
$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1              1352600   1268580     15312  99% /
none                   2021964       168   2021796   1% /dev
none                   2029532         0   2029532   0% /dev/shm
none                   2029532        84   2029448   1% /var/run
none                   2029532         0   2029532   0% /var/lock
blade66:/disk/sd0g/courses/it244
                       8260768   2484096   5694048  31% /courses/it244
blade61:/disk/sd0g/home/it244gh
                       8260768   8149792     28352 100% /home/it244gh
mx1:/disk/sd1e/spool/mail
                       4129312   1350144   2737888  34% /spool/mail
blade61:/disk/sd0f/home/sd86
                       8260768   5835520   2342624  72% /home/sd86
blade61:/disk/sd0f/home/as1414
                       8260768   5835520   2342624  72% /home/as1414
```

Command Options

- When used with the **-h** option, **df** produces more readable output

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             1.3G  1.3G   15M  99% /
none                  2.0G  168K  2.0G   1% /dev
none                  2.0G     0  2.0G   0% /dev/shm
none                  2.0G   84K  2.0G   1% /var/run
none                  2.0G     0  2.0G   0% /var/lock
blade66:/disk/sd0g/courses/it244
                      7.9G  2.4G  5.5G  31% /courses/it244
blade61:/disk/sd0g/home/it244gh
                      7.9G  7.8G   28M 100% /home/it244gh
```

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

`tty`

- As you type at the command line, what you type is *collected* by a program called **`tty`**
- **`tty`** is a **device driver** that is part of the kernel
- It looks at each character as you type – and takes appropriate action
- Most of the time, **`tty`** just places the character in a buffer
- But, **`tty`** responds differently to special characters

tty

- When the character you type is the *backspace*, it erases the previous character from the buffer
- When the character is the Control-U **tty** erases the buffer from the current insertion point to the beginning of the line
- **tty** is responsible for all **command line editing**
- When **tty** sees a newline character, it passes the contents of the buffer to the shell
- *Newline* is the character you get from hitting Enter on a windows machine (or Return on a Mac)

Parsing the Command Line

- The shell takes the contents of the buffer and breaks it up into **tokens**
- Tokens are the strings of text separated by spaces or tabs
- This action is called **parsing**: Making a list of all the strings on the command line and throwing away the whitespace
- Next, the shell looks for the name of the *command* – usually, the command name is the first string on the command line
- The command can be specified by a simple filename

  `ls`
- Or… by using a **pathname** to the executable file

  `/bin/ls`

The `PATH` System Variable

- When you run a program by typing a pathname at the terminal – such as `/usr/bin/php` – the shell has no difficulty finding the executable file to run
- How can the shell know where to find an executable file if all it gets is the command name?
- Programs are executable files that can be stored anywhere in the filesystem
- So, how does the shell find the correct file?
- The shell checks a system variable called `PATH`

The `PATH` System Variable

- **`PATH`** contains a list of directories to search for an executable file whose name matches the command typed at the command line
  - 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 **`it244a`**:

```
$ echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games:/usr/local/games
```

The PATH System Variable

- 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 executable file, then it will print an *error message*
- If the shell finds an executable file but you *do not have execute permission*, then it will tell you this in an error message
- You can *modify* the **PATH** variable in your own Unix environment
- We'll see how to do this in a few classes...

Running a Program in the Current Directory

- For reasons having to do with security, you should never put the current directory, **.** in the **PATH** list
- Then, how do you run a program that is located in your current directory?
- You can do this using the following construction
  ./PROGRAM_NAME
- This will *always* work, regardless of the contents of **PATH**

Running a Program in the Current Directory

- Here is an example:

```
$ ls -l foo.sh
-rwxrwxr-x 1 ghoffmn grad 16 Oct  1 15:49
foo.sh

$ ./foo.sh
Foo to you
```

- Notice that I did not have to run **bash** to run this script
- That's because the script file has *execute* permission set

Running the Command Entered on the Command Line

- When you type a command at the command line, the **shell** has to find the right executable file – by looking through the directories listed in **`PATH`**
- If you have execute permission on the executable file, the shell will ask the kernel to start a **process** for that program
- A process is a running program
- Only the kernel can create a process, so…
    - the shell gives the kernel the pathname of the executable file, and…
    - the kernel does the rest

Running the Command Entered on the Command Line

- Each process needs resources to do its job
- One of the most important resources is *memory*
- Each process has memory allocated to it that it alone can use
- This prevents one program from interfering with another
- In addition to memory, the shell gives the process various system resources like pointers to certain files

Running the Command Entered on the Command Line

- The shell also gives the program the 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
  - The shell has no idea what options or arguments are appropriate to a given program
  - The program has the responsibility to check the options and arguments for correctness
  - If the program gets the wrong options or arguments, then it is the responsibility of the program to print an error message and take appropriate action

Running the Command Entered on the Command Line

- While the program is running, the shell waits for it to finish
  - It does this by going into an inactive state known as "sleep"
  - When the program finishes, it must tell the shell that it is done
- It does this by sending the shell an **exit status**
  - The exit status is an integer that must be 0 or greater
  - An exit status of 0 means the program finished without error
  - Any exit status greater than zero indicates an error
  - A program can issue different exit 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
```

- The exit status is 1, meaning the command failed. (Notice that I had to put a dollar sign **$** in front of the variable name to get its value.)
- When the shell gets an exit status it returns to an active state
  - It prints a prompt to the terminal
  - And it waits for the next command

Standard Input, Standard Output, and Standard Error

- Every Unix process always has access to three 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 unless specifically told to get it from a file
  - By default, standard input is the ***keyboard***

Standard Input, Standard Output, and Standard Error

- **Standard output** …
  - …is where the program prints the results if it is not told specifically where to send it
  - By default, standard output is the ***terminal***
- **Standard error** …
  - …is where the program sends error messages
  - By default, standard error is the same as standard output the ***terminal***
- Each of these "files" can be changed by the user using a Unix feature called **redirection**