I/O and Shell Scripting

- File Descriptors
- Redirecting Standard Error
- Shell Scripts
- Making a Shell Script Executable
- Specifying Which Shell Will Run a Script
- Comments in Shell Scripts

File Descriptors

- Resources are given to each process when it is created
- Every time the shell creates a process, it gives that process connections to three "files":
  - **Standard input**
  - **Standard output**
  - **Standard error**
- Any program can open other files, besides these three standard "files"

File Descriptors

- So, how does Unix keep track these multiple files?
- It does so through **file descriptors**
- File descriptors are
  - data structures that Unix creates to handle access to files for processes
  - the abstract representation of the files that are connected to a process
  - Each file descriptor is assigned a positive number, the first of which is 0

File Descriptors

- Think of a file descriptor as an integer that *represents a file*
- Standard input, standard output, and standard error each have their own file descriptors
- So…
  - While **we** think of standard *input*, standard *output*, and standard *error*,…
  - …**Unix** thinks of the file descriptors 0, 1, and 2
- Most of the time, you do not have to worry about file descriptors – though they can appear in complex scripts.

Name	File Descriptor
Standard input	0
Standard output	1
Standard error	2

Redirecting Standard Error

- Standard error is the "file" into which error messages are usually sent
- Redirecting standard error allows a program to separate its *normal output* from its *error messages*
- To redirect standard input, we use the *less than symbol* **<** followed by a *file pathname*
- Consider the following example...

Redirecting Standard Error

```
$ ./repeat.sh < test.txt
Enter several lines
Type X on a line by itself
when done

You entered
-----------
123456789
abcdefg
987654321
hijklmnop
foo
bar
bletch
X
```

- **<** is really a shorthand for a notation using file descriptors
- When *you* type

  ./repeat.sh < test.txt

  Unix thinks of this as

  ./repeat.sh 0< test.txt
- where the **0** in front of the greater than sign is *the file descriptor for standard input*

Redirecting Standard Error

- Similarly, when *we* use output redirection

```
$ echo Hello there  >  hello.txt
```

- *Unix* thinks of this as meaning

```
$ echo Hello there  1>  hello.txt
```

- Again, the *file descriptor* precedes the redirection symbol **>**
- So how do we redirect standard error?

Redirecting Standard Error

- We can redirect standard error by placing a **2** in front of the greater than symbol **>**

```
$ ls xxxx
ls: cannot access xxxx: No such file or directory

$ ls xxxx 2> error.txt

$ cat error.txt
ls: cannot access xxxx: No such file or directory
```

- When we redirected standard error using **2>** Unix sent the error messages to the file **error.txt** , *not* to the screen

Redirecting Standard Error

- When we redirected standard error using **2>** Unix sent the error messages to the file **error.txt** not to the screen
- You can redirect **both** standard output and standard error to *the same file*
- You do this with *ampersand and greater-than* symbols together **&>**
- For example…

Redirecting Standard Error

```
$ cat foo1.txt foo2.txt
foo57.txt
foo to you
bar to everyone else
bletch to the universefoo
foo foo
bar bar bar
bletch
cat: foo57.txt: No such file
or directory

$ cat foo1.txt foo2.txt
foo57.txt &> error.txt
...
...
$ cat error.txt
foo to you
bar to everyone else
bletch to the universefoo
foo foo
bar bar bar
bletch
cat: foo57.txt: No such file
or directory
```

Shell Scripts

- A shell script is a file that contains Unix commands with their options and arguments
- You can think of a shell script as a *collection* of command line entries
- When the shell script is executed...
  - Each line of the script is run in turn...
  - ...as if you were entering them at the command line, one after the other

Shell Scripts

- A shell script can use any shell feature that is available at the command line **except** those features which are provided by ***tty*** specifically:
  - **Command line editing** (arrow keys, control key combinations)
  - Pathname completion (tab to get more of a filename)
  - The history mechanism (up arrow to recall previous command line)

Shell Scripts

- However, *other* shell features *are* available to you!
- You can use ambiguous file references in a shell script. That is, you have full use of the *metacharacters* **?** , ***** , and **[ ]**
- You can use *redirection* in a shell script, as well as *pipes*
- Unix also provides **control structures**
  - If statements
  - Loops

Shell Scripts

- *Control structures* allow you to change the path taken through the script
- We will learn more about those soon…
- The shell executes the script one line at a time, exactly as it would if you were *typing* in the line at the terminal

Making a Shell Script Executable

- You can run a shell script without using ***bash***, if you give the script both **read** and **execute** permissions
  - You need *read* permission because the shell has to read the contents of the script
  - You need *execute* permission so the script can be run without explicitly using ***bash***
- If you try to run a script without **both** permissions you will get an error

Making a Shell Script Executable

- For example…

```
$ ls -l cheer.sh
-rw-rw-r-- 1 ghoffmn grad 13 Oct 29 14:23 cheer.sh

$ cat cheer.sh
#! /bin/bash

# this file roots for the home team

echo "Let's go Red Sox!"

$ ./cheer.sh
-bash: ./cheer.sh: Permission denied
```

Making a Shell Script Executable

- Of course, you can set these permissions using ***chmod***
- Normally, you would give a shell script file **755** permissions
  - The *owner* can read, write and execute
  - The *group* and *everyone else* can read and execute

```
$ chmod 755 cheer.sh

$ ls -l cheer.sh
-rwxr-xr-x 1 ghoffmn grad 13 Oct 29 14:23 cheer.sh

$ ./cheer.sh
Go Sox!
```

Making a Shell Script Executable

- All scripts for this course **must** have `755` permissions set
- This is necessary so that I will be able to run them myself
- Moreover, it will help establish good habits
- Points will be deducted if you do not do this

Specifying Which Shell Will Run a Script

- The shell is just a program that
  - *reads* what you enter at the command line and…
  - …*runs programs* for you
- It stands between you and the operating system
- When the shell runs a program for you, it normally *sleeps* until the program is finished – unless you tell the shell to run the command in the *background*

Specifying Which Shell Will Run a Script

- When the shell runs a shell script, it creates a *new* shell inside the process that will run the script
  - Normally, this sub-shell will be the same kind of shell as your login shell
  - So, if your login shell is *Bash*, a Bash sub-shell will run the script
- There are significant differences between the various shells that come with Unix and Linux

Specifying Which Shell Will Run a Script

- What if you need to run the script in a *different* shell?
  - It is always best to run a script in the *same* shell used by the programmer who wrote the script
  - Unix provides a way to specify *which shell to run* when a script is executed
- It is called the **hashbang** line or sometimes the **shebang** line. Example:

```
#! /bin/bash
```

Specifying Which Shell Will Run a Script

- That's because the first two characters on the line must be the following...
  - a hash mark (*number symbol, pound sign, etc.*): **#**
  - followed by an exclamation mark: **!**
- The exclamation mark is sometimes called “bang”
- After these two characters comes the **absolute pathname** of the shell which will run with the script

Specifying Which Shell Will Run a Script

- The pathname following **#!** must be an *absolute* pathname because you don't know *where* the user will be when the script is run
  - The hashbang line tells your current shell which shell to use to run your script
  - The hashbang line **must** be the first line in the script
- Unix looks at the first few characters of a file before running a script...

Specifying Which Shell Will Run a Script

- If it sees **#!**, then it interprets what follows as the pathname of the program that should run the script
- It is good form to *always* use a hashbang line, even when this is not necessary
- You may follow hashbang with a couple of *spaces* before the pathname
- To show you that this really works, I'm going to run the script **`shell_test_1.sh`**

Specifying Which Shell Will Run a Script

```
$ cat shell_test_1.sh
#! /bin/sh
ps -f

$ ./shell_test.sh
UID          PID  PPID  C STIME TTY          TIME CMD
ghoffmn      710   709  0 13:25 pts/1    00:00:00 -bash
ghoffmn     2741   710  0 15:35 pts/1    00:00:00 /bin/sh
./shell_test.sh
ghoffmn     2742  2741  0 15:35 pts/1    00:00:00 ps -f
```

- Here, we indicated that *this* script should be run with the **sh** shell, specifically…

Specifying Which Shell Will Run a Script

- We did this by specifying it in the *hashbang* line

```
#! /bin/sh
```

- Now, compare this with **shell_test_2.sh**

```
$ cat shell_test_2.sh
ps -f

$ ./shell_test_2.sh
UID          PID  PPID  C STIME TTY          TIME CMD
ghoffmn      710   709  0 13:25 pts/1    00:00:00 -bash
ghoffmn     2893   710  0 15:41 pts/1    00:00:00 -bash
ghoffmn     2894  2893  0 15:41 pts/1    00:00:00 ps -f
```

Specifying Which Shell Will Run a Script

- The second script has no hashbang line, so the script was run in a ***Bash*** shell
- The shell did this because I *did not tell it otherwise*
- You can leave out the hashbang line and still run a script without calling ***bash*** , but…
- …you **must** use a hashbang line for scripts written in scripting languages like *Perl* and *Python*

Specifying Which Shell Will Run a Script

- Here is a simple Python script that **does not** have hashbang line...

```
$ cat hello_1.py
print ("Hello world!")
```

- It has *read* and *execute* permissions

```
$ ls -l hello_1.py
-rwxrwxr-x 1 ghoffmn grad 21 Jun 19 17:48 hello_1.py
```

- But, when I try to run it, there is a problem...

Specifying Which Shell Will Run a Script

```
$ ./hello_1.py
Warning: unknown mime-type for "Hello world!" --
using "application/octet-stream"
Error: no such file "Hello world!"
```

- I can only run *this* script by calling the Python interpreter

```
$ python hello_1.py
Hello world!
```

- Now we'll look at the same script – but with a *hashbang* line that uses the *Python* interpreter...

Specifying Which Shell Will Run a Script

```
$ cat hello_2.py
#! /usr/bin/python

print ("Hello world!")
```

- I can run this script *directly*

```
$ ./hello_2.py
Hello world!
```

Comments in Shell Scripts

- Programs are written by people for machines
- But, programs also have to be readable for the people…
  - Who ***write*** the program
  - Who ***maintain*** the program
  - Who ***use*** the program
- To make clear what is happening inside a program, use ***comments***

Comments in Shell Scripts

- *Comments* are text which is *ignored* by whatever program is running the script – i.e., they are *only* for people to read
- Anything following a hash mark **#** is a comment – except for the hashbang line, of course! **Example:**

```
$ cat comment_test.sh
#! /bin/sh
#  demonstrates that comments do not affect the
#  way the script runs
echo Hello there

$ ./comment_test.sh
Hello there
```

Comments in Shell Scripts

- Comments are a way to *document* a program within *the text of the program itself*
- This sort of documentation is *extremely* important
  - You may create a script today and not use it for a couple of months
  - When you need to change it, you may have *forgotten* how it works
  - A few well-placed comments can save you hours of work

Comments in Shell Scripts

- It is good practice to place a comment at the top of the shell script, after the hashbang line
- This comment should say *what the script does*
- You should also comment any part of a script that does something *less than obvious*
- I'll take off points if you do not

```
$ cat bother.sh
#!/bin/bash

# keeps printing something
to the terminal until it
is killed

while [ 6 -lt 10 ]
    do
    sleep 5
    echo "Excuse me"
    done
```