Introduction to the Unix Shell

Mike Nolta, SciNet

July 11, 2016

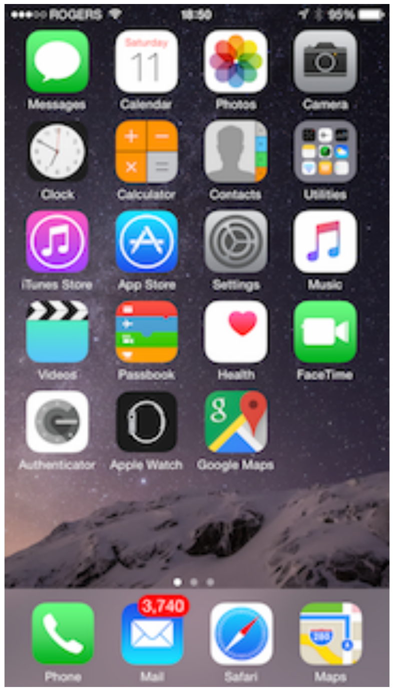
What is a shell?

A shell is a meta-program.

It's a program to run other programs.

All general purpose computers have a shell of some sort.

Graphical "shell"

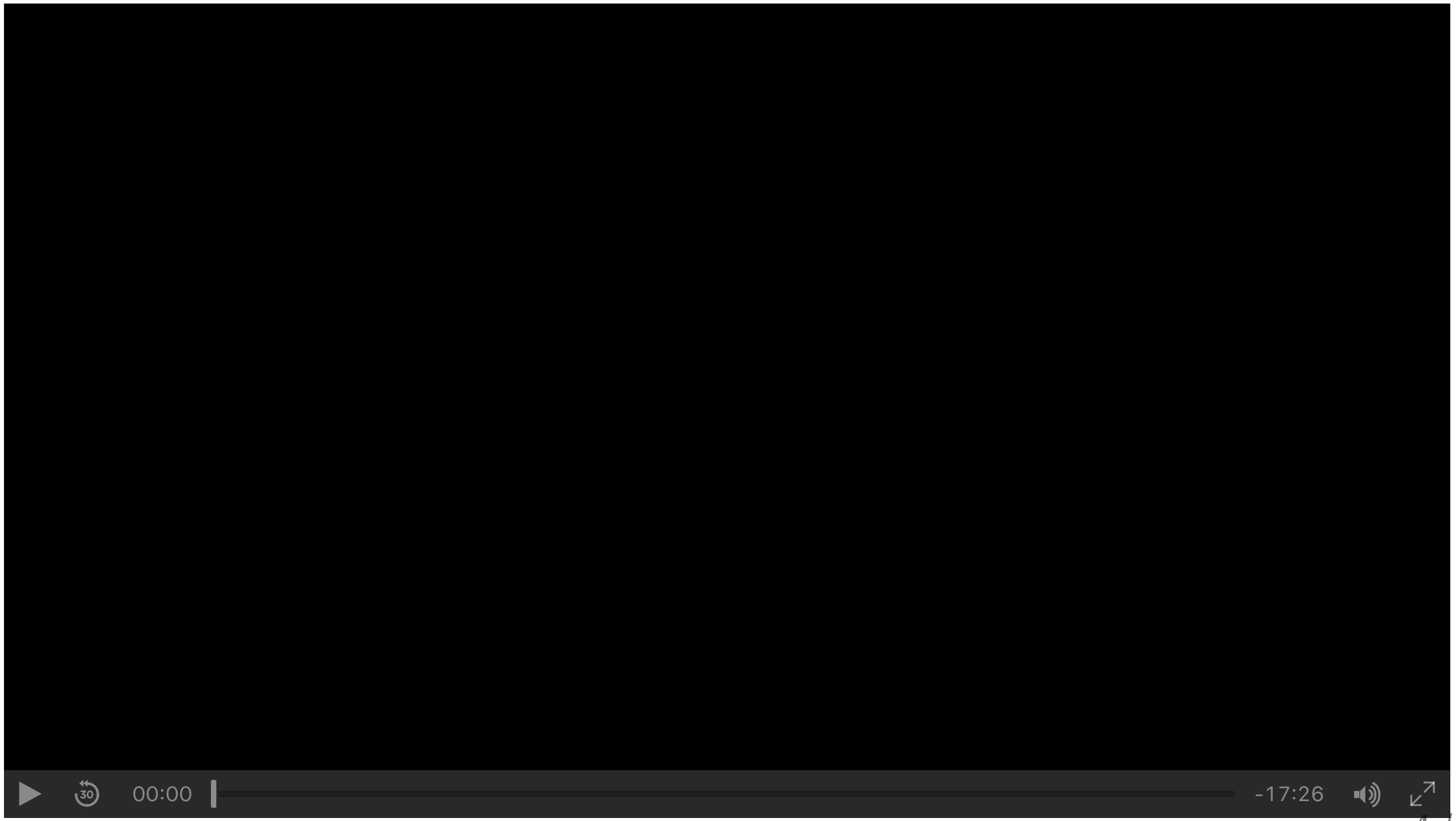


For example, the main screen of a smartphone can be considered a kind of shell. It lists the available programs, which you tap to start.

(However, most people would call this a launcher, not a shell.)

Text shells

The first shells were written in the early 1970s, when you talked to computers like this:



So why still use text shells?

We don't use teletypes anymore, so why isn't everything graphical?

A graphical user interface is pretty, but limited. The only thing you can do is start a program, and that's pretty much it.

As we'll see, a text interface allows you to combine simple programs together in powerful ways.

A shell is not just a *program* to launch programs, but a *language* to launch programs. We use text shells for the same reason we still write code in text -- it's more expressive.

Getting setup

- Linux:
  - Look for a Terminal application (`gnome-terminal`, `konsole`, `xterm`, …)
- OS X:
  - Open `Applications/Utilities/Terminal`
- Windows:
  - Start `MobaXterm`;
  - Go to *Settings* > *Configuration* and set "persistent HOME directory" to a permanent location.

The shell prompt

You should now see the shell prompt, awaiting input:

- Linux:

  ```
  [username@hostname ~]$
  ```

- OS X:

  ```
  hostname:~ username$
  ```

- Windows (MobaXterm):

  ```
  [username.hostname]
  ```

In these slides i'll use

```
$
```

as the shell prompt.

Running a program

Instead of tapping an icon, in a text shell you type the name of the program you want to run and hit *Enter*. Let's run the `date` program:

```
$ date
Thu  9 Jul 2015 11:55:48 EDT
```

`date` is the name of the program we want to run, which prints the current time.

Program arguments

The first word is the command, and any subsequent words are passed to the command as arguments, where the words are separated by whitespace.

The `echo` program prints the arguments you give it:

```
$ echo a b c
a b c
```

Here we've passed three arguments: "a", "b", and "c".

The `seq` program prints a sequence of integers:

```
$ seq 3
1
2
3
$ seq 4 -1 2
4
3
2
```

Getting help

Use the `man` command (short for manual) to get help

```
$ man echo

ECHO(1)                    BSD General Commands Manual                  ECHO(1)

NAME
     echo -- write arguments to the standard output

SYNOPSIS
     echo [-n] [string ...]

DESCRIPTION
     The echo utility writes any specified operands, separated by single blank (`
     characters and followed by a newline (`\n') character, to the standard output
...
```

Let's write our own `echo`

```
# python
import sys

for i,arg in enumerate(sys.argv):
    print "arg %d is %s" % (i, arg)
```

```
// print_args.c : print command line arguments
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++)
        printf("arg $d is %s\n", i, argv[i]);
    return 0;
}
```

```
$ gcc -o print_args print_args.c
$ ./print_args a b c
arg 0 is ./print_args
arg 1 is a
arg 2 is b
arg 3 is c
```

String literals

String literals are delimited by single quotes:

```
$ echo 'the cake cost $10 dollars'
the cake cost $10 dollars
```

Unlike python or C, escape sequences are not expanded:

```
$ echo 'a\nb\nc'
a\nb\nc
```

If a single-quoted string is preceded by a `$`, standard C character escape sequences are expanded:

```
$ echo $'a\nb\nc'
a
b
c
```

`\n` is a newline, `\t` is a tab, etc.

Field Splitting

Quotes prevent argument splitting, allowing you to pass spaces in a single argument:

```
$ ./print_args a b
arg 0 is ./print_args
arg 1 is a
arg 2 is b
```

```
$ ./print_args 'a b'
arg 0 is ./print_args
arg 1 is a b
```

Variables

If you want to pass the same argument to multiple commands, you can define a variable:

```
$ var='a b'
```

Valid variable names start with a letter, and consist only of letters, numbers, and underscores.

Note that there can be no spaces around the equal sign -- `var = x` is interpreted as "run the 'var' command with arguments '=' and 'x'".

To get a variable's value, prefix with a dollar sign:

```
$ echo var
var
$ echo $var
a b
```

String literals (2)

There's another form of string literal, delimited by double-quotes. It's like a single-quoted string, but interpolates variables:

```
$ var='a b'
$ echo 'the value of var is $var'       # single quotes
the value of var is $var
$ echo "the value of var is $var"       # double quotes
the value of var is a b
```

When ambiguous, surround the variable name with braces:

```
$ echo "the value of var is $vars"
the value of var is
$ echo "the value of var is ${var}s"
the value of var is a bs
```

Field Splitting (2)

Newlines and tabs can also delimit arguments, unless protected by quotes.

```
$ X=$'a\nb\nc'
```

```
$ echo $X
a b c
```

```
$ echo "$X"
a
b
c
```

In other words, command splitting happens after variable interpolation.

Where are these programs?

The which command shows you the location a program:

```
$ which echo
/bin/echo
```

```
$ which seq
/usr/bin/seq
```

But what does `/bin/echo` mean?

The Filesystem

The Unix Filesystem

Like Windows and OS X, the Unix filesystem is hierarchical, with files stored in nested directories (folders). Directories can hold both files and other directories (called subdirectories).

A *path* is a string representing a route through the filesystem, tracing a path through the directories, with the names separated by `/`.

There are two kinds of paths: absolute and relative.

- absolute paths start with a `/`, and refer to a fixed location;
- relative paths do not start with a `/`, and are relative to the current working directory.

Unlike Windows, the Unix filesystem is *unified*, every file and directory descends from `/`, the root. There are no drive letters, e.g., `C:\Folder\File.txt`.

Filesystem layout

- `/` is the root of the filesystem
- `/usr` holds system software:
    - `/usr/bin` holds system programs (aka binaries)
    - `/usr/lib` holds system libraries
- `/etc` holds system configuration files.
- `/home` holds user home directories
- `/tmp` holds temporary files

Navigating the Filesystem: `pwd` & `cd`

Print current (working) directory:

```
$ pwd
/Users/nolta
```

This is my home directory (on OS X).

Change current directory with `cd`:

```
$ cd /usr
$ pwd
/usr
$ cd bin
$ pwd
/usr/bin
```

Note the `cd bin` command is interpreted relative to our current location (i.e., "change to the 'bin' subdirectory").

Navigating the Filesystem: `ls`

List the files in the current directory:

```
$ ls
Desktop     Library     Pictures    chime
Documents   Movies      Public      julia
Downloads   Music       asciinema   talks
```

List the files in a specific directory:

```
$ ls /
bin   dev  home  lib64       media  mnt  nixon  proc  run   scratch  sys  usr
boot  etc  lib   lost+found  misc   net  opt    root  sbin  srv      tmp  var
```

Creating & Removing Directories

Make a new directory:

```
$ mkdir tmp
```

Remove directory:

```
$ rmdir tmp
```

`rmdir` will fail if the directory contains any files or directories.

Removing files

```
$ rm filename
```

Adding the `-r` option lets you recursively remove a directory and all its subdirectories & files.

```
$ rm -r dir
```

Careful! There is no "trashcan", so once files are deleted they're gone for good.

Renaming Files & Directories

To rename files and directories, use the `mv` command (short for "move"):

```
$ ls
oldname
$ mv oldname newname
$ ls
newname
```

Copy a File

To copy a file, use the `cp` command (short for "copy"):

```
$ ls
original
$ cp original copy
$ ls
copy     original
```

Special directories: `.`, `..`, `~`

`.` refers to the current directory, so `ls` and `ls .` are equivalent.

`..` refers to the parent directory.

```
$ cd /usr/local/bin
$ cd ../../lib
$ pwd
/usr/lib
```

`~` refers to your home directory.

```
$ cd ~/data
$ pwd
/Users/nolta/data
```

File permissions

The Unix filesystem has basic access controls for files and directories. There are 3 basic kinds of permission: read, write, and execute. Each file & directory is owned by a *user* and a *group*.

To see this info, use `ls -l`:

```
$ ls -l notes.txt
-rw-r--r--  1 nolta  staff  2  7 Jun 15:13 notes.txt
```

File permissions (2)

Here's how to read the output of `ls -l`:

`-rw-r--r-- 1 nolta staff 2 7 Jun 15:13 notes.txt`	*user*
`-rw-r--r-- 1 nolta staff 2 7 Jun 15:13 notes.txt`	*group*
`-rw-r--r-- 1 nolta staff 2 7 Jun 15:13 notes.txt`	*file size in bytes*
`-rw-r--r-- 1 nolta staff 2 7 Jun 15:13 notes.txt`	*when last modified*
`-rw-r--r-- 1 nolta staff 2 7 Jun 15:13 notes.txt`	*user permissions*
`-rw-r--r-- 1 nolta staff 2 7 Jun 15:13 notes.txt`	*group permissions*
`-rw-r--r-- 1 nolta staff 2 7 Jun 15:13 notes.txt`	*anybody permissions*

Permission flags:

- `r--` : can read file.
- `-w-` : can modify file.
- `--x` : can execute file.

So everyone can read this file, but only `nolta` can write to it.

Directory permissions

```
$ ls -ld /bin
drwxr-xr-x  39 root  wheel  1326 16 May 16:41 /bin
```

Directory permissions flags have slightly different meanings:

- `r--` : can `ls` directory.
- `-w-` : can add, delete, and rename files in directory.
- `--x` : can `cd` into and access files in directory.

The `x` permission on a directory is a bit strange, but the upshot is that to read a file you need:

- `r` on the file itself,
- `x` on the parent directory,
- `x` on the parent's parent directory,
- etc.

Changing permissions

Default file & directory permissions are usually `rw-r--r--` and `rwxr-xr-x`.

To change permissions, use the `chmod` command (short for "change mode").

For example, to make a file executable:

```
$ chmod +x filename
```

Shield a directory from prying eyes:

```
$ ls -ld our_secret_data/
drwxr-xr-x  2 nolta  staff  68  8 Jul 18:09 our_secret_data/
$ chmod o= our_secret_data/
$ ls -ld our_secret_data/
drwxr-x---  2 nolta  staff  68  8 Jul 18:09 our_secret_data/
```

Recap

- `cp` copy file
- `cp -r` copy directory
- `ls` list contents of directory
- `mkdir` make new directory
- `mv` move file or directory
- `rm` remove file
- `rm -r` remove directory and everything it contains
- `rmdir` remove empty directory

The Environment

Environment Variables

Ok, where were we? Right,

```
$ which seq
/usr/bin/seq
```

So the `seq` program is located in the `/usr/bin` directory. When the shell sees `seq`, it runs `/usr/bin/seq`.

But how did the shell know to run `/usr/bin/seq`?

There's a special environment variable called `PATH` that lists which directories the shell searches for programs.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin
```

So when you type `seq⏎`, the shell first looks in `/opt/local/bin`, then `/opt/local/sbin`, then `/usr/local/bin`, then `/usr/bin`, and then stops because `/usr/bin/seq` exists.

Changing PATH

PATH is a colon-separated list of directories. To add a directory to the PATH:

```
PATH="$PATH:/path/to/dir"
```

For example, to always look in the current directory for binaries:

```
PATH="$PATH:."
```

Other interesting environment variables

- `HOME`: user home directory

  ```
  $ echo $HOME
  /Users/nolta
  ```

- `PWD`: current working directory (same as `pwd` command)
- `HOSTNAME`: this computer's network hostname
- `PS1`: prompt string

  ```
  $ PS1="yo: "
  yo: echo a b c
  a b c
  ```

The `env` command prints out the current list of exported environment variables.

"Global variables"

Environment variables like `PATH` are like global variables -- they're a way of passing data to programs without going through the command line.

```
$ FOOBAR=jkfds
$ python -c 'import os; print os.environ["FOOBAR"]'
...
KeyError: 'FOOBAR'
```

"Global variables"

Environment variables like `PATH` are like global variables -- they're a way of passing data to programs without going through the command line.

```
$ FOOBAR=jkfds
$ python -c 'import os; print os.environ["FOOBAR"]'
...
KeyError: 'FOOBAR'
```

```
$ export FOOBAR=jfkdls
$ python -c 'import os; print os.environ["FOOBAR"]'
jfkdls
```

"Global variables"

Environment variables like `PATH` are like global variables -- they're a way of passing data to programs without going through the command line.

```
$ FOOBAR=jkfds
$ python -c 'import os; print os.environ["FOOBAR"]'
...
KeyError: 'FOOBAR'
```

```
$ export FOOBAR=jfkdls
$ python -c 'import os; print os.environ["FOOBAR"]'
jfkdls
```

Once a variable is exported, it stays exported:

```
$ FOOBAR=lslsls
$ python -c 'import os; print os.environ["FOOBAR"]'
lslsls
```

Wildcards (aka Globbing)

File globbing

Say you have a directory with thousands of files, but only a few files have the extension `.html`. You can list only those files using the `*` pattern:

```
$ ls *.html
index.html
intro_to_unix_shell.html
```

Before the command is run, the shell expands `*.html` into a list of filenames ending in `.html`, and then passes that list as arguments to `ls`.

Quoting protects arguments from globbing:

```
$ ls "*.html"
ls: *.html: No such file or directory
```

Globbing patterns

- `*` matches any sequence of zero-or-more characters
- `?` matches any single character
- `[...]` matches any single character in `...`
- `[!...]` matches any single character not in `...`
- any other character just matches itself

Globbing examples

- `*.pdf` matches any string ending in `.pdf`
- `???` matches any 3-character string
- `[a-z]*` matches any string starting with a lowercase letter
- `[!a-z]*` matches any string not starting with a lowercase letter
- `[A-Za-z0-9_]` matches any letter (upper or lower), digit, or underscore

Standard I/O & Redirection

Store Output in a Variable: `$(...)`

Use `$(...)` to capture the output of a command:

```
$ onetwothree=$(seq 3)
$ echo $onetwothree
1 2 3
$ echo "$onetwothree"
1
2
3
```

Store Output in a File: `>`

Often you want to save the output of a command to a file, not dump it to the screen:

```
$ seq 4 -1 2
4
3
2
```

This is accomplished via the `>` operator:

```
$ ls
$ seq 4 -1 2 > output
$ ls
output
$ cat output
4
3
2
```

Here the output of the `seq 3` command has been redirected to the new file `output`. The `cat` program dumps files to the screen.

Append Output to a File: `>>`

`>` clobbers the file:

```
$ echo a > output
$ cat output
a
$ echo b > output
$ cat output
b
```

Use `>>` to append to a file:

```
$ echo a > output
$ echo b >> output
$ cat output
a
b
```

Redirect Input from a File: `<`

Similarly, you can redirect the input of a command to come from a file, not the terminal, with `<`:

```
$ seq 4 -1 2 > output
$ cat output
4
3
2
$ sort < output
2
3
4
```

The `sort` command sorts the lines of its input.

Pipelines: `|`

This pattern is so common

```
$ cmd1 > tmpfile
$ cmd2 < tmpfile
```

that there's a special pipeline syntax to connect the output of one command to the input of another command:

```
$ seq 4 -1 2
4
3
2
$ seq 4 -1 2 | sort
2
3
4
```

Pipelines can be chained:

```
$ cmd1 | cmd2 | cmd3 | cmd4 | ...
```

Here Documents: `<<`

Here documents are file literals passed to stdin:

```
$ sort <<EOF
> pear
> apple
> tophat
> EOF
apple
pear
tophat
```

`EOF` is an arbitrary string. It marks the beginning and end of the input.

Here Documents: `<<<`

`<<<` lets you pass strings to stdin:

Example -- let:

```
$ X=$'pear\napple\ntophat'
```

So instead of

```
$ echo "$X" | sort
apple
pear
tophat
```

you can write

```
$ sort <<<"$X"
apple
pear
tophat
```

Behind the Scenes

Don't worry if you don't understand this material with the pink background -- it's optional!

Files

A **file** is a source and/or sink of bytes, which supports the following API:

- `int fd = open(filename, mode)`
- `read(fd, buf, n)`, read at most *n* bytes from *fd* into *buf*;
- `write(fd, buf, n)`, write *n* bytes from *buf* to *fd*;
- `close(fd)`

This is a powerful abstraction. Files are typically thought of, e.g., a PDF on a disk drive, but they can be anything you can read and/or write to. For example, a network connection can be thought of as a file.

Standard I/O

Each process starts life with 3 open files:

- standard input (aka stdin)
- standard output (aka stdout)
- standard error (aka stderr)

Their file ids are 0, 1, 2 respectively.

Not surprisingly, the process reads from stdin, writes to stdout, and writes errors to stderr.

Redirection

For a shell, stdin/stdout/stderr are typically the same file, a terminal:

```
$ tty
/dev/ttys009
```

```
$ echo a b c > /dev/ttys009
a b c
```

When the shell runs a program, by default the program inherits the same stdin, stdout, and stderr as the shell.

However, you can redirect I/O, i.e., read/write from different files.

Under the hood

`> filename` is the same as `1> filename`, i.e., "redirect the output of file descriptor 1 to filename".

`n> filename` gets translated into something like

```
/* C code */
close(n);
open(filename, O_WRONLY|O_CREATE);
```

Because `open` is called immediately after `close`, it's guaranteed to return the same file descriptor (in this case, `n`).

This is done *before* the program is run.

Stupid redirection tricks

```
// write_to_3.c : write a short message to file descriptor 3

#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    const char *s = "this is file descriptor 3";
    if (write(3, s, strlen(s)) == -1)
        fprintf(stderr, "error: couldn't write to file descriptor 3\n");
    return 0;
}
```

```
$ ./write_to_3
error: couldn't write to file descriptor 3
```

```
$ ./write_to_3 3>output
$ cat output
this is file descriptor 3
```

Redirecting output to another stream: `>&`

```
$ ls *.nope
ls: *.nope: No such file or directory
$ ls *.nope >output
ls: *.nope: No such file or directory
```

The error message is still printed to the terminal because it's written to stderr, not stdout.

To redirect stderr, use `2>`:

```
$ ls *.nope >output 2>err
```

What if you want to direct both stdout & stderr to the same file?

```
$ ls *.nope >output 2>&1
$ cat output
ls: *.nope: No such file or directory
```

Redirecting output to another stream: `>&`

`n>filename` turns into

```
/* C code */
close(n);
open(filename, WRITE);
```

`n>&m` turns into

```
/* C code */
close(n);
dup(m);
```

`dup` duplicates the file descriptor, and like `open` it reuses the last closed file descriptor.

Redirecting output to another stream: `>&`

Redirections are processed in order from left to right.

`>filename 2>&1`

```
/* C code */
close(1);
open(filename, WRITE);
close(2);
dup(1);
```

```
0    /dev/tty   -->  /dev/tty   -->   /dev/tty
1    /dev/tty   -->  filename   -->   filename
2    /dev/tty   -->  /dev/tty   -->   filename
```

Redirecting output to another stream: `>&`

Redirections are processed in order from left to right.

`2>&1 >filename`

```
/* C code */
close(2);
dup(1);
close(1);
open(filename, WRITE);
```

```
0    /dev/tty   -->   /dev/tty   -->    /dev/tty
1    /dev/tty   -->   /dev/tty   -->    filename
2    /dev/tty   -->   /dev/tty   -->    /dev/tty
```

Writing Scripts

Our first script

If you find yourself executing the same series of commands frequently, you'll eventually want to collect them into a file called a script.

```
$ echo 'echo "hello, world!"' > script1
```

To run the script:

```
$ bash script1
hello, world!
```

Our first script (2)

A script is just a text file, with each line a separate command.

Blank lines are ignored, along with everything after the comment character `#`.

```
$ echo > script2
$ echo 'echo "hello, world!"' >> script2
$ echo '# this is a comment' >> script2
$ cat script2

echo "hello, world!"
# this is a comment
```

```
$ bash script2
hello, world!
```

Editing files

Building a script using `echo` is tedious for long scripts, so we need a text editor.

MobaXterm has an editor, under *Tools* > *MobaTextEditor*.

Otherwise, the `nano` text editor is good for beginners.

To create a new text file, enter `nano` to start the program, edit your text, and then press `Control-X`. It will ask you enter the filename, and then quit.

To edit an existing text file, enter `nano filename`.

Passing Arguments to a Script

`$0` is the name of the script, `$1` is the first argument, `$2` is the second argument, etc.

```
$ echo 'echo first three args are $1 $2 $3' > script3
$ bash script3 a b c d
first three args are a b c
$ bash script3 a b
first three args are a b
```

For loops

Instead of copy and pasting similar commands,

```
echo apple
echo banana
echo durian
```

you can use a for loop:

```
fruits="apple banana durian"
for fruit in $fruits
do
    echo $fruit
done
```

Interactively, you can add semicolons for a one-liner:

```
$ for i in 1 2 3; do echo $i; done
1
2
3
```

If statements

Here's a little script to check if path is a file or directory:

```
$ cat script4
path="$1"
if [ -f $path ]; then
    echo "$path is a file"
else if [ -d $path ]; then
    echo "$path is a directory"
else
    echo "$path is neither a file nor directory"
fi
fi
```

```
$ bash script4 /bin
/bin is a directory
$ bash script4 /bin/bash
/bin/bash is a file
```

Common tests:

```
[ -f filename ]    # filename exists
[ -d directory ]   # directory exists
```

Background Processes

Background processes

The following command sleeps for 60 seconds:

```
$ sleep 60
```

During those 60 seconds, you can't use the terminal -- it blocks until the command is finished.

Adding `&` to the end tells the shell to run in the background:

```
$ sleep 60 &
[1] 72363
...
[1]+  Done                    sleep 60
```

Now you can use the terminal while it's running, and you'll see a message printed when the process finishes.

Job Control

Job control refers to managing background processes.

To list all the current background processes, use `jobs`:

```
$ sleep 50 & sleep 60 & sleep 70 &
[1] 18383
[2] 18384
[3] 18385
$ jobs
[1]   Running                 sleep 50 &
[2]-  Running                 sleep 60 &
[3]+  Running                 sleep 70 &
```

Jobs can be referred to either by their process ID or PID (e.g., `18383`) or job number (e.g., `%1`).

Killing a job

If you need to terminate a background process, use the `kill` command:

```
$ sleep 40000 &
[1] 72374
$ kill %1
[1]+  Terminated: 15          sleep 40000
```

or:

```
$ sleep 40000 &
[1] 72374
$ kill 72374
[1]+  Terminated: 15          sleep 40000
```

`wait` for background jobs

The `wait` command waits for all background jobs to finish.

```
$ sleep 50 &
$ sleep 70 &
$ wait
[1]-  Done                    sleep 50
[2]+  Done                    sleep 70
```

`wait` for background jobs (2)

`wait` isn't very useful for interactive use, but is useful in scripts. It lets you run multiple commands in parallel:

```
# this script runs 4 calculations in parallel
long_running_calculation 1 &
long_running_calculation 2 &
long_running_calculation 3 &
long_running_calculation 4 &
wait
```

Without the `wait` at the end, the script would exit right away, and the `long_running_calculation` jobs will all be killed before they finish.

Suspending Jobs

If you started a job, but forgot to add `&`, use `Control-Z` to suspend it:

```
$ ping freebsd.org
PING freebsd.org (8.8.178.110) 56(84) bytes of data.
64 bytes from wfe0.ysv.freebsd.org (8.8.178.110): icmp_seq=1 ttl=49 time=69.1 ms
64 bytes from wfe0.ysv.freebsd.org (8.8.178.110): icmp_seq=2 ttl=49 time=68.9 ms
64 bytes from wfe0.ysv.freebsd.org (8.8.178.110): icmp_seq=3 ttl=49 time=68.9 ms
^Z
[1]+  Stopped                 ping freebsd.org
```

The job is now stopped, and not executing. Use `fg` to bring it back to the foreground.

```
$ jobs
[1]+  Stopped                 ping freebsd.org
$ fg
ping freebsd.org
64 bytes from wfe0.ysv.freebsd.org (8.8.178.110): icmp_seq=4 ttl=49 time=68.8 ms
64 bytes from wfe0.ysv.freebsd.org (8.8.178.110): icmp_seq=5 ttl=49 time=68.9 ms
64 bytes from wfe0.ysv.freebsd.org (8.8.178.110): icmp_seq=6 ttl=49 time=68.6 ms
...
```

Suspending Jobs (2)

In addition to resuming a suspending job in the foreground, you can restart it as a background process with `bg`:

```
$ sleep 100000
^Z
[1]+  Stopped                 sleep 100000
$ bg
[1]+ sleep 100000 &
$ jobs
[1]+  Running                 sleep 100000 &
```

Useful Programs

The Unix Way

Instead of having a few big, complicated programs that do everything, Unix prefers lots of small, simple programs combined into pipelines.

In this section, we'll demonstrate some of the most common utility programs.

`head`, `tail`: first or last lines of file

Take first 3 lines:

```
$ seq 7 | head -n 3
1
2
3
```

Take last 3 lines:

```
$ seq 7 | tail -n 3
5
6
7
```

`head`, `tail`, continued

Drop first 2 lines:

```
$ seq 7 | tail -n+3
3
4
5
6
7
```

Drop last 2 lines (*doesn't work on OS X*):

```
$ seq 7 | head -n-2
1
2
3
4
5
```

`sort`: sort lines

Normal (lexicographic) sort:

```
$ seq 9 11 | sort
10
11
9
```

Numerical sort (`-n`):

```
$ seq 9 11 | sort -n
9
10
11
```

Reverse sort (`-r`):

```
$ seq 9 11 | sort -n -r
11
10
9
```

`uniq`: unique lines

`uniq` removes consecutive duplicate lines from a stream or file. Usually paired with sort.

```
$ uniq <<EOF
> a
> b
> a
> EOF
a
b
a
```

```
$ uniq <<EOF
> a
> a
> b
> EOF
a
b
```

`uniq`, continued

Add the `-c` option to get a duplicate count:

```
$ uniq -c <<EOF
> a
> a
> b
> EOF
   2 a
   1 b
```

`cut`: cut columns

```
$ echo abcdefghijklmnopqrstuvwxyz > alphabet
```

First 10 columns

```
$ cut -c-10 alphabet
abcdefghij
```

Columns 10 and greater

```
$ cut -c10- alphabet
jklmnopqrstuvwxyz
```

Columns 10-20

```
$ cut -c10-20 alphabet
jklmnopqrst
```

`cut`, continued

Columns 4-6 and 8-13

```
$ cut -c4-6,8-13 alphabet
defhijklm
```

Fields 2-4 and 6, where fields are separated by commas:

```
$ echo 'a,b,c,d,e,f,g' | cut -d',' -f2-4,6
b,c,d,f
```

Be careful, multiple delimiters are not combined:

```
$ echo 'a   b   c' | cut -d' ' -f2

$ echo 'a   b   c' | cut -d' ' -f4
b
```

`paste`: concatenate lines

Concatenate columns, separated by space

```
$ paste -d' ' <(seq 3) <(seq 3)
1 1
2 2
3 3
```

Concatenate lines, separated by comma

```
$ paste -d, <(seq 3) <(seq 4)
1,1
2,2
3,3
,4
```

"Transpose" with the `-s` option

```
$ paste -s -d, <(seq 3) <(seq 4)
1,2,3
1,2,3,4
```

`tr`: translate characters

Convert ASCII lowercase to uppercase

```
$ cat alphabet | tr 'a-z' 'A-Z'
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Convert arbitrary lowercase to uppercase

```
$ echo 'αβ' | tr '[:lower:]' '[:upper:]'
AB
```

Convert spaces to commas

```
$ echo 'a  b   c' | tr ' ' ','
a,,b,,,c
```

Convert spaces to commas, but squeeze commas together

```
$ echo 'a  b   c' | tr -s ' ' ','
a,b,c
```

`tr`, continued

Delete characters

```
$ echo 'a,b,c' | tr -d ','
abc
```

Delete all characters except

```
$ echo 'a,b,c' | tr -c -d ','
,,
```

(`-c` takes the complement of `','`)

sed: stream editor

Substitute "brown" with "red"

```
$ echo "quick brown fox" | sed -e 's/brown/red/'
quick red fox
```

Substitute "/bin" with "/usr/local/bin"

```
$ echo "/bin/bash" | sed -e 's:/bin:/usr/local/bin:'
/usr/local/bin/bash
```

Replace all occurrences of "a" with "b"

```
$ echo "a a a" | sed -e 's/a/b/'
b a a
$ echo "a a a" | sed -e 's/a/b/g'
b b b
```

awk: stream text processing

Named after the initials of its creators.

Invert column 1

```
$ seq 3 | awk '{print $1, 1/$1}'
1 1
2 0.5
3 0.333333
```

Unlike bash, where all arithmetic is integer, in awk all arithmetic is floating point.

`grep`: search for pattern

`grep` searches for lines matching a certain pattern, and prints them. It's a great tool for filtering large datasets.

```
$ echo foobar | grep oob
foobar
$ echo foobar | grep boo
$
```

The name stands for "global regular expression print". Regular expressions are out of scope for this class, but they're extremely powerful and worth learning.

For example, get the current Toronto temperature from the web:

```
$ url='http://weather.gc.ca/city/pages/on-143_metric_e.html'   # TO weather
$ curl -s $url | grep lead | grep -Eo '[0-9]+'
24
```

`scp`: Copy files between Unix systems

Use the `scp` command to copy between Unix systems where you have accounts (short for "secure copy"):

Download a file:

```
$ scp username@hostname:/path/to/filename .
filename                      100%   31KB  31.1KB/s   00:00
```

You can omit `username` if it's the same on both systems. Note the trailing `.` argument; it says put `filename` in the current directory.

Upload a file:

```
$ scp filename username@hostname:/path/to/dir/
```

If you omit `/path/to/dir/` (i.e., `scp filename hostname:`), `filename` is put in your home directory on `hostname`.

More useful programs

- `chown` : change file/directory user & group ownership
- `curl`/`wget` : download URLs
- `diff` : file differences
- `du` : disk usage
- `find` : find files
- `gzip`/`gunzip` : compress/uncompress files
- `join` : join two files together along a common column
- `ln` : create symbolic links
- `ps` : list processes
- `rsync` : synchronize directory trees
- `tar` : archive a directory into a single file
- `tee` : send input to both screen and a file
- `top` : see which processes are using the most CPU or memory
- `wc` : count lines
- `zip`/`unzip` : handle ZIP files

Stupid Shell Tricks

Example: find the biggest users on SciNet

```
$ showstats -u
statistics initialized Thu Jun 25 22:39:14

        |------ Active ------|--------------- Completed --------------|
user     Jobs Procs ProcHours   Jobs   PHDed    %   FSTgt    Effic  WCAcc
user001    19  2496  58790.76    503   2.80M   6.44 -----    97.84  87.43
user002     1   152   5133.38    486   2.31M   5.32 -----     8.51  52.71
user003     1   512  12941.65   6957   2.06M   4.73 -----   159.68  55.40
user004     2    16    260.71    868   1.26M   2.90 -----    96.80  21.12
user005     4  1536  54752.64    213   1.19M   2.74 -----    98.20  46.37
user006     1   192   9098.40    164 978.8K   2.25 -----    98.58  76.00
user007     1   128   3316.02   1839 911.1K   2.10 -----     6.66   9.02
user008    33  1056  32067.62    580 879.0K   2.02 -----    98.00  97.59
user009    31   732   3118.31   3718 873.9K   2.01 -----   190.14  82.13
user010     0     0      0.00   5938 871.3K   2.00 -----    93.36  75.05
user011     0     0      0.00   2073 861.2K   1.98 -----    11.54  30.07
user012     1   520   4735.32    258 795.1K   1.83 -----     2.41  66.35
user013    20   480  17312.31   1441 712.3K   1.64 -----    31.78  52.24
user014    38  1520  30901.51    774 700.0K   1.61 -----    98.20  45.77
user015     8   320   7443.33   4419 663.1K   1.52 -----    39.80  12.38
...
```

Example: find the biggest users on SciNet (2)

First, multiply columns 7 (percent nodes used) and 9 (efficiency) together. `NF` is the number of fields per line.

```
showstats -u | awk 'NF == 16 {print $1, $7*$9}' \
```

Next, reverse numeric sort by column 2

```
| sort -r -n -k 2,2
```

Finally, just grab the top 10

```
| head -n 10
```

Example: find the biggest users on SciNet (3)

Altogether,

```
$ showstats -u | awk 'NF == 10 {print $1, $7*$9}' \
  | sort -r -n -k 2,2 | head -n 10
user003 755.286
user001 630.09
user009 382.181
user004 280.72
user005 269.068
user006 221.805
user008 197.96
user024 187.475
user010 186.72
user014 158.102
```

Example: sum fields in XML file

Say we have an XML file,

```
$ curl -O http://www.canfar.phys.uvic.ca/vospace/nodes
$ cat nodes
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="http://www.canfar.phys.uvic.ca/vospace/vosp
<vos:node xmlns:vos="http://www.ivoa.net/xml/VOSpace/v2.0" xmlns:xsi="http://www.w
  <vos:properties>
    <vos:prop uri="ivo://ivoa.net/vospace/core#ispublic" readOnly="false">true</vo
  </vos:properties>
  <vos:nodes>
    <vos:node uri="vos://cadc.nrc.ca!vospace/APASS" xsi:type="vos:ContainerNode">
      <vos:properties>
        <vos:prop uri="ivo://ivoa.net/vospace/core#length" readOnly="true">-532343
        <vos:prop uri="ivo://ivoa.net/vospace/core#date" readOnly="true">2015-07-0
        <vos:prop uri="ivo://ivoa.net/vospace/core#ispublic" readOnly="false">true
        <vos:prop uri="ivo://ivoa.net/vospace/core#creator" readOnly="false">CN=we
      </vos:properties>
      <vos:nodes />
    </vos:node>
...
```

and we'd like to sum up the "length" fields.

Example: sum fields in XML file (2)

First, filter out the "length" lines

```
grep 'core#length' nodes \
```

Next, grab just the number

```
| grep -Eo '[0-9]+' \
```

Finally, sum the entries

```
| { s=0; while read x; do s=$((s+x)); done; echo $s; }
```

All together,

```
$ grep 'core#length' nodes \
> | grep -Eo '[0-9]+' \
> | { s=0; while read x; do s=$((s+x)); done; echo $s; }
69189164087043
```

Example: sum fields in XML file (3)

You can also define a `sum` function, and use it as the last step of the pipe:

```
$ sum() {
>   local s=0
>   while read x; do
>     s=$((s+x))
>   done
>   echo $s
> }
$ grep 'core#length' nodes | grep -Eo '[0-9]+' | sum
69189164087043
```

Wrap-up

Wrap-up

The shell is abstruse, but powerful.

If you become heavily involved with computing, you could be using it for the next 40 years.

So it pays to invest the time to learn.

Thanks! Questions?