Lecture 2 - Processes

January 20, 2015

Compass

Where are we?

- introduced UNIX, saw some commands
- first drill already up and available
- today: processes and commands

Back to UNIX

The building blocks of a shell interaction:

1. Commands
2. Arguments
3. Pipelines
4. Redirections

Some useful commands:

1. cat
2. ls
3. grep
4. sort
5. sed
6. man
7. less

Let's look at this last one - `man`. How do we learn about commands?

```
$ man cat
```

We can search:

```
$ man -k sort
```

What does `man` do?

```
$ gunzip -c </usr/share/man/man1/cat.1.gz |nroff -mandoc |less
```

How do I generate the syllabus?

```
$ groff -t -Tdvi -mdoc -rS12 syllabus.man >syllabus.dvi
$ dvipdfm syllabus.dvi
```

Little note: the `-t` switch makes `groff` run it through `tbl` first.

Touring Commands

cat

We saw `cat` Tuesday.

```
$ cat grades.csv
```

What does `cat` do?

- Writes files to output
- Puts files one after another
- If a file is `-`, or there are no arguments, copies its input to output

Processes have 3 I/O streams:

- standard input
- standard output
- standard error (diagnostic messages)

nl

Numbers lines:

```
$ nl grades.csv
```

head and tail

Work a lot like `cat`!

- `head` shows the first lines of a file
- `tail` shows the last lines of a file

```
$ head grades.csv
```

Look at man page.

- `-n` option to pick lines
- `tail` has more sophisticated options (on Mac)

grep

Top lines, bottom lines, now let's pick lines.

```
$ grep wk student-info.csv
```

Grep takes several arguments:

- options (optional)
- pattern (a *regular expression*, for now just text to look for)
- files to search

It handles its file arguments as follows:

- If no files, search its input
- If 1 file, search that file and print matching lines
- If ≥ 2 files, print matching lines from each (in turn), prefixed with file name.

```
$ grep wk student-info.csv grades.csv
```

cut

We have sliced lines; `cut` lets us slice columns.

```
$ cut -f1-2 -d, student-info.csv
```

We can put things together: to find the SSN for 'wsb6112':

```
$ grep wsb6112 student-info.csv |cut -d, -f2
```

rearrange lines

We can sort the lines:

```
$ sort student-info.csv
```

- By default, sorts lines *lexicographically*
- Can sort by fields, e.g. birthday:

  ```
  sort -k5 -t, student-info.csv
  ```

- Can also sort numerically, interpreting SI suffixes, reverse the sort…

On Linux, `shuf` will shuffle the lines.

editing lines

The `sed` command lets us edit lines.

```
$ sed -e 's/,/ /g' grades.csv
```

- `-e` introduces a *sed command*; `s/,/ /g` is the sed command.
- `s/,/ /g` means 'replace comma with space globally'
- quotes make it all a single argument (interpreted by shell)
- then come filenames

There are many sed commands, but `s` is by far the most frequently used.

The Shell

So we've now seen a couple things our shell does for us:

- connect programs (pipe)
- quote arguments

What does the shell do?

1. Split (*tokenize*) the input
2. Evaluate commands and pipelines
    - pass arguments to commands
    - route input and output

Most commands are programs (look at `/usr/bin`); some are *builtins*.

Writing Shell Functions

Let's write a shell function:

```
print_args()
{
    for arg in "$@"; do
        echo "argument: $arg"
    done
}
```

Splitting Arguments, Running Commands

Show off with argument function.

```
$ print_args foo bar bletch
$ print_args "foo bar" bletch
```

Quotes form a single argument (a *word*).

Variables

We can set variables in shell:

```
foo=bar
echo "$foo"
```

- `echo` prints to standard out
- `$var` includes value of the variable
- variable substitution works inside double quotes
- general rule: always quote variable expansions
  - there are some exceptions, but default should always be quoted.

Control Structures

In our function, we have `for`. `for` is one of those built-ins. It loops.

for *arg* in *words*; do
commands
done

Each time through *commands*, the variable *arg* is bound to one of the words in *words*.

```
for arg in foo bar bletch; do
    echo $arg
done
```

We have a special case in our function: `"$@"` does not expand to one word, it expands to all the command line arguments for the function, one per word.

Handling Arguments

- The variable `$@` has all the arguments
- The variables `$1`, `$2`, etc. have individual ones
- The `shift` built-in moves variables up

So we can rewrite our function body:

```
while [ -n "$1" ]; do
    echo "argument: $1"
    shift
done
```

What's up with `[`? It is how we write boolean expression tests.

- `[` introduces a test, `]` concludes it
- `-n` checks whether its next argument is non-empty (`-z` tests for empty)

Control Structures

We've seen two

- `for`
- `while`

also:

- `if`
- `case`

`if` and `while` are alike. They take a command (often `[`), and consider *success* to be true.

What is success? Programs have *exit codes*; by convention, 0 is success and anything else is failure.

```
if [ condition ]; then
    commands
elif [ condition ]; then
    commands
else
    commands
fi
```

Writing Scripts

A *shell script* is just a file that contains shell commands. The whole thing is interpreted as if you typed it in at the shell prompt.

Typical shell script:

```
#!/bin/sh

commands
```

The `#!/bin/sh` line at the top is a *shebang*, and it tells UNIX to run this program by using `/bin/sh` (the *interpreter*). This is also how you run Python scripts, etc.

A script should be *executable* (`chmod +x file`), so Unix can execute it.

Commands and Substitution

Where can we run commands?

- just run them as statements
- as the tests of `if`/`while`
- in *command substitutions*: `$(cmd)`, where the output is put back into the shell.

Arguments

The process gets arguments, as we saw.

- The shell parses the arguments.
- The process receives them as separate arguments.
- We can quote arguments (show this).
- Aside: this is kinda different from Windows.

UNIX has some conventions for arguments. Unfortunately, there are multiple conventions.

Traditional UNIX

- flags are single letters (`l`, `a`, `x`), or occasionally numbers
- some flags take a single argument (options)
- argument can appear as a second argument (`-t :`) or combined (`-t:`)
- multiple non-argument-taking flags can be compressed (`-la`)
- non-flag arguments are left over (whether they can be mixed in, or must appear after all flags/options, is command-specific)

The `getopt` function provides support for parsing these kinds of arguments.

Some commands allow `-` to be omitted before initial flags (`tar`, `ps`).

Of course, some commands run off and do their own thing (`dd`)

GNU

Problem: these flags are cryptic, and we're limited to about 62 options.

Solution: wordy arguments

- traditional UNIX rules in force
- each flag also has a *long form*: `--long`, `--numeric-sort`
- if long form option takes an argument, can either appear separate (`--delimiter :`) or combined with `=` (`--delimiter=:`)
- special argument `--` says 'stop parsing flags and treat the rest as leftover arguments'

Support libraries:

- `getopt_long` (on GNU/Linux, several BSDs)
- `popt` and many other libraries
- other languages: python `argparse`/`optparse`, perl `Getopt::Long`, others

This is the single most common convention.

X11

Another old convention, used by X11 programs and a number of others.

- flags have one or more characters, started with a single dash (`-leftbar`)
- options separated by spaces (`-name foobar`)
- sometimes: prefixed with `-` or `+` to say enable/disable

Also used by Java utilities, OCaml compilers, and a few other odd things.

Standard I/O

A process has 3 streams:

- *standard input* (`cin`/`stdin`) is the process's input (from the shell's terminal by default)
- *standard output* (`cout`/`stdout`) is where the process writes its ordinary output.
- *standard error* (`cerr`/`stderr`) is for the process to write diagnostic and error messages.

Let's intermingle some `cerr` use.

When we use `|` in the shell, it runs the process's standard output (but *not* its standard error) to another program. Very helpful!

We can also send output to a file with '>'.

Sometimes we want them mixed: e.g. to view errors and output in `less`. *Shell redirects* to the rescue:

```
$ ./testproc 2>&1 |less
```

The `2>&1` means 'redirect file descriptor 2 into descriptor 1'.

Each process has a *file descriptor table*: a mapping of numbers to open files. UNIX represents open files by these numbers.

- `2`: redirect FD 2 instead of the default 1
- `>`: redirect
- `&1`: redirect into another FD, namely 1, rather than a file
- And then pipe the mess through less

We can also redirect standard input, with `<`.

Redirecting streams is the essence of shell pipelines.

TODO: write some example pipelines here.