Shell Commands

Table of Contents

General Commands

Basic Output Manipulation

Some basic commands and their most common arguments you should integrate into your workflow (and ones that will be used without warning from here on out). These are typically used as something to pipe output into to improve your experience:

- `wc [-cmlw]` : Output statistics about the number of bytes (-c), characters (-m), lines (-l), and/or words (-w) of the input stream or file(s).
- `head -n N` : Output only the first N lines of the input text.
- `tail -n N` : Output only the last N lines of the input text.
- `more` : Paging utility that lets you scroll through text a screenful at a time instead of having it be outputted all at once to the console. Like a read-only editor, it also supports interactive commands reminiscent of Vim keybindings and features like regex search.
- `less` : The direct upgrade to more, it also supports backwards navigation and more inclusive support for keybindings.

Path & the Shell

If the name of a program does not contain any slashes, then the shell automatically looks through directories listed in the PATH environment variable to find the program name. This is why when you want to run an executable in the current directory, you have to use the ./executable syntax:

`PATH=$PATH:.` appends the current directory to path so you do not have to use the `./` notation

Command Arguments

- Within shell languages, the simplest commands look like:

```
word0, word1, ..., wordn # word0 is the name of the program and word1, ...,
                         # wordn are the arguments to that program
```

- Normal (positional) arguments, like a.
- Options (aka ***flags***) -ejh which is also equivalent to -e -j -h. Jamming them together is a thing you can do on Unix, but as we know from Assignment 2, this behavior becomes tricky when there are options that take arguments.
- Options that don't take any arguments are sometimes referred to as switches, and they set some kind of configuration simply by being present or absent (think of it like a boolean option, where presence means true and absence means false).

Running Multiple Commands

Commands in Sequence

- `;` is the sequencing operator which signifies a newline queing the following shell command after the character
- `&` is the parralellization operator which signifies that the command after that character will run in parallel
- `|` is the pipeline character which pipelines the output of one command as the input to another, and the pipe is a buffer
  - the pipe really depends on the read and write priviledges of the command
  - allows you to set up a sequence of little languages that each process or transform a single stream of text output as it makes its way into some final form
- `$?` outputs the exit status of the last command

Special Characters of the Shell

```
 ~ # $ & * ( ) = [ ] \ | ; ' " < > ?

$ echo 'hello\there\n"general kenobi"'
hello here       # the \t is interpolated as an escape character
"general kenobi"

$ echo "hey there's\n\"general kenobi\""
hey there's
"general kenobi"
```

Control Structures

- if, else, then, done are not reserved
  - shell knows if it is a control structure is if it is at the start of the line or after a semicolon

If and Else

```
 cmd1 && cmd2    # run a command that executes properly, then run command 2
cmd1 || cmd2    # run command 2 if command 1 fails

if cmd1
  then cmd2
  else false
done

[ ] # takes arguments and does some of the obvious comparison functions and stuff

# Nested If Statements
if if [$a=$b]
        then cat file
        else grep x file
  then # If either then cat or grep x succeed, then go to the outer then
```

Sub Processes

```
 # Conditional Runtime Behavior
# Execute cmd1a, ignore exit, execute cmd1b and then execture cmd2 depending  on success or failure

{cmd1a; cmd1b;} && cmd2   # same process
(cmd1a; cmd1b;) && cmd2   # sub process
```

- the curly braces indicate the the shell is running the within the same process but changes the shells directory
- the paranthesis indicates that it changes to a different directory and does all work within a subprocess
  - safer but slower option

Looping

```
 for i in {1..100}
  do
    echo $i
done
```

Globbing

- glob is a shell expression like a regular expression for pattern matching done by the shell
  - ordinary strings match themselves, and you can concatenate the blogs like regular expressions similar to regular expressions
- The shell will expand strings containing these special characters to every string that matches the pattern, separated by whitespaces
- This process is done by the shell and not the programs typically associated with the shell that provide the abstraction to

the kernel/os

```
 * (regex .*)         # any number of characters (does not match leading . as it messes with the direct
? (regex *)          # any single character
[ ]                  # matches a single character within a set, very similar to regEX but negation is
{pdf,jpeg}           # - match multiple literals
[!]                  # complement of a character set if ! is the first character
\                    # escape character
```

`grep xyz *.c *-h-*` : matches all of the -h- expressions

Phases of the Shell

1. Variable Expansion

- **Variable expansion** expands a shell variable name to the actual value of the variable.
- `echo $x` where `(x=$cheeks)` will expand `$x` to `cheeks`
- `$#` expands to the number of arguments `$1 $2 $3`
- `echo ${x}y` outputs the value of x, immediately followed by the char 'y'
- `${x-default}` outputs the value of x if x is empty string, otherwise expands to `default`
- `${x:-default}` outputs the value of x if x is nonempty, otherwise expands to `default`
- `{x+set}` expands to set if x is set otherwise expands to empty string
- `${x-nonempty}` expands to the string nonempty if x is nonempty
- `${x?}` expands to x if x is set otherwise throws and error
- `${PATH?}` expands to path if the path is set, otherwise throw an error
- `${x=default}` sets x to default if x is not set
- `set a b c xyz` sets a list of arguments
  - `echo $4` -> output: xyz
- `unset xyz` will uninitialized a variable
- `export x` makes the variable visible to subcommands and subprocesses

Every process in the system has a number. `$$` expands to the process ID of the shell itself.

- `$!` the last process that your shell started in the background
- `$&` makes the shell run the process in the background
- `kill -9` is a kill signal that cannot be ignored
- `$PATH` expands to path
- `$HOME` expands to home
- `$PWD` expands to parent working directory

2. Tilde Expansion

- `~` expands the home directory when the variable calls
  - if `x = ~ashah` , then `echo $x` outputs `/home/ashah`

3. Command Substitution

- `$(abcd)` will stop the shell, run the command, takes the output, and splices into the command we ran before
  - `grep ABC $(find * -name '*c')` runs grep ABC on all files outputted with the find command

4. Arithmetic Expansion

- `$((x+5))` will expand variables to do arithmetic (slow way to do it)
    - looks up the value of x, then adds 5 to it and then expands to 32

5. Field Splitting

- the shell takes the stuff that we have already done and breaks the command that we have done into further pieces

```
 x = 'a b c *.c'
grep foo $x         # same argument
grep foo a b c *.c   # same but variable expansion
grep foo a b c *.c   # turns the space-separated word into separate arguments
```

- IFS - inter-field separators (space, newline, and tab) are argument separators
    - important for field splitting spaces for arguments

6. Filename Expansion (Globbing)

- expand all of the globs

7. Redirection

- `>f` writes output into `f`
- `>>f` appends output to `f`
- `2>&1` directs both standard output (file descriptor 1) and standard error (file descriptor 2)
- `2<&` redirect standard input 2
- `&>>word` appends standard output and standard error