*nix shell intro

How does the shell run other programs?

- Your shell is a program, just like any other
- Your shell runs other programs in custom environments
- Running other programs is not magic
- Your is also a REPL (read, eval, print, loop)
- Your shell is also a scripting language

Running other Programs

1. Find the executable on disk
    a. $PATH
    b. ls vs. /bin/ls
2. Pass along any invocation arguments
    a. split on whitespace
    b. need to quote strings with whitespace (or escape the chars)
3. Record the exit status
    a. 0 == success
    b. non-zero == failure
    c. stored in $?
    d. exit status determines truthiness for conditions

Argument Expansion

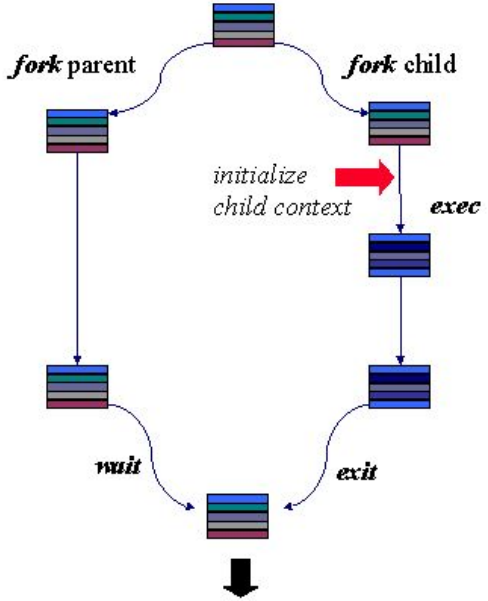
1. File globbing, expand into filepaths that match pattern
    a. `*` # as a regex: /.*/
    b. `?` # as a regex: /./
    c. `[ABCDEFG]` # as a regex: /[A-G]/
2. Command substitution, replace with output from command
    a. `` `cmd arg1 arg2` `` # deprecated syntax
    b. `$(cmd arg1 arg2)` # preferred syntax
3. Variable expansion
    a. `foo=bar` # reference variables, like to set, without $
    b. `$foo` # expand variables to their values with $
4. Brace Expansion
    a. `foo-{bar,baz}` -> `foo-bar foo-baz`
    b. `file{0..3}` -> `file0 file1 file2 file3`

which which

- type vs. which
- alias vs. function vs. builtin
- your shell vs. subshell
- sourcing

Running other programs

- fork(2): create a copy of the running process
  - execve(2): replace the running process space with an executable file
  - wait(2): suspend execution until child process terminates



int pid = fork();
Create a new process that is a clone of its parent.

exec*("program" *[, argvp, envp]*);
Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);
Exit with status, destroying the process.

int pid = wait*(&status);
Wait for exit (or other status change) of a child.

DUKE Systems & Architecture

← ilu, Matt Kelly

Input, Output, and Errors

The three always open file descriptors

1. Standard In: fd0
   - STDIN
2. Standard Out: fd1
   - STDOUT
3. Standard Error: fd2
   - STDERR

Open file descriptors are copied when a process forks and preserved in the process space after execve.

Output Redirection

1. Redirect to a file
   - `> file` # file opened for writing as fd1
   - `>> file` # file opened for writing as fd1
2. Read from a file
   - `< file` # file is opened for reading as fd0
3. Redirect STDOUT of cmd1 to STDIN of cmd2
   - `cmd1 | cmd2` # cmd2 will be the parent of cmd1

The shell will, after forking, open the appropriate file in the requested fashion, close open file descriptor and replace it with a newly created file descriptor, THEN exec

The Shell is not Magic

1. Commands are files, aliases, functions, or built-ins
2. Commands are executed with a list of arguments
3. Commands read from STDIN and write to STDOUT/STDERR
4. Commands inherit their environment, ENV and CWD