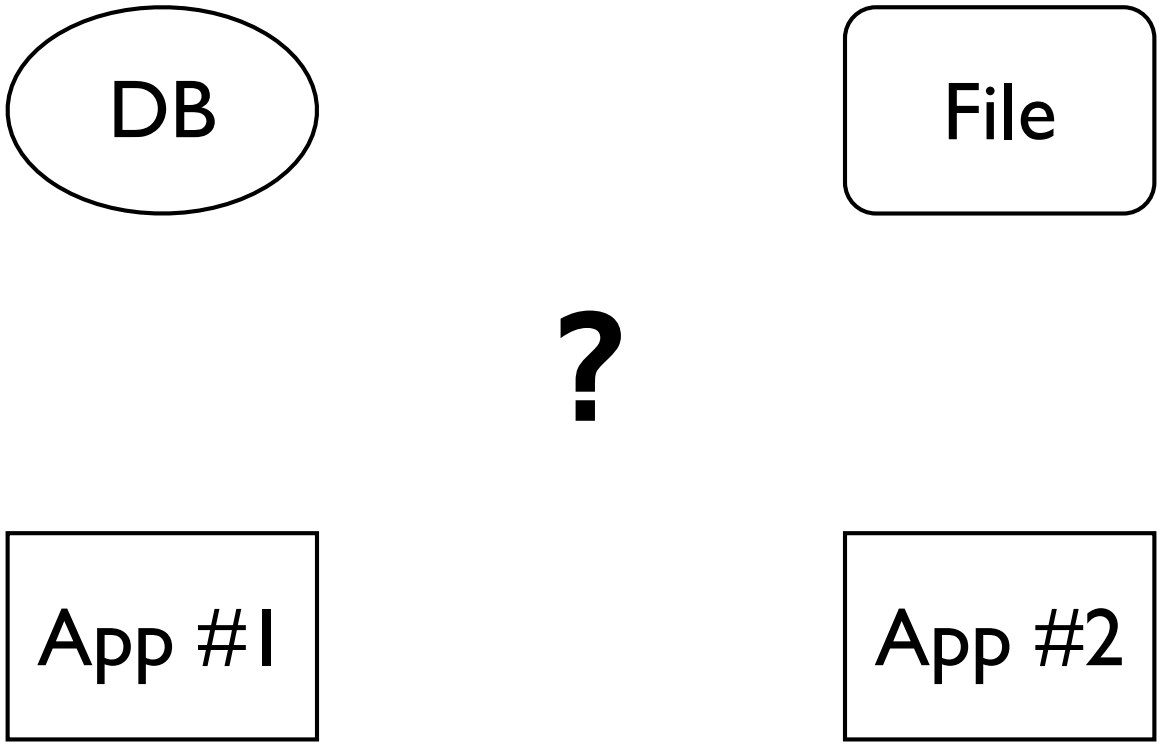
Day 11: System Interaction

Suggested reading: *Learning Perl* (6th Ed.)

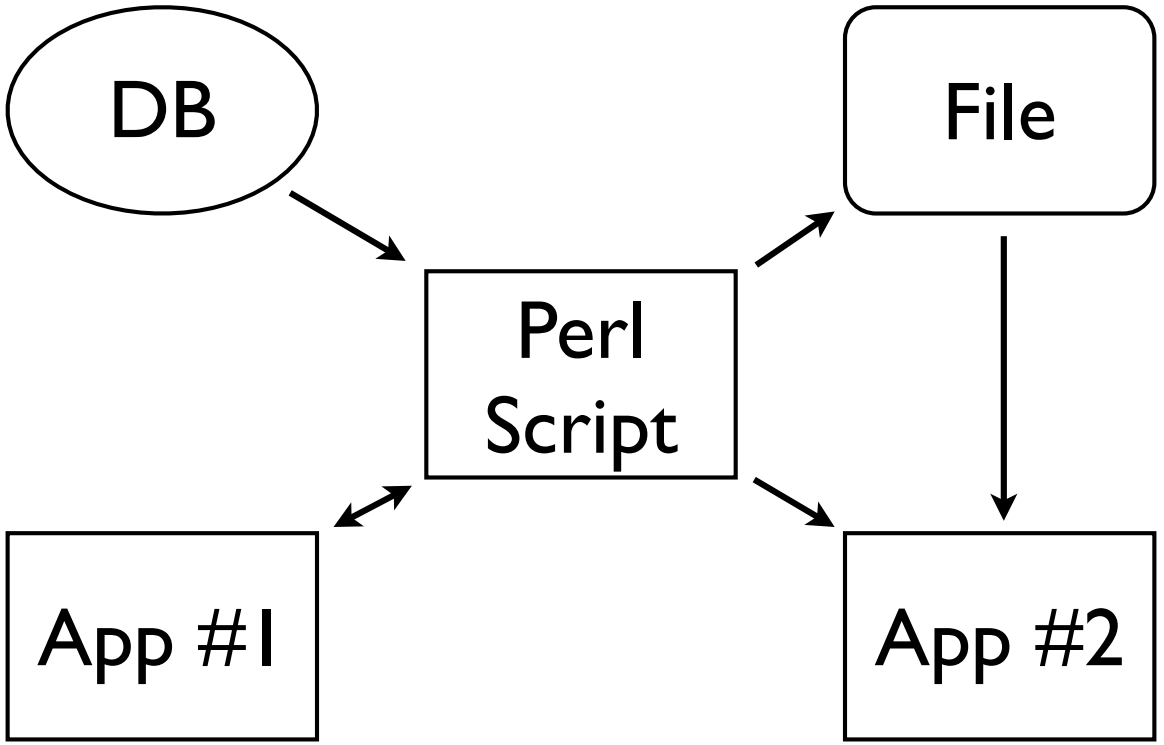
Chapter 16: Process Management

Homework Review

Problem



(One) Solution



Introducing … The Shell

The Shell

- Is a ***program*** (e.g., `/bin/sh`) – interactive or scripted
  - Another interpreted scripting language (like Perl)

```
while (1) { print '% '; my $cmd = <STDIN>; do($cmd); }
```

- Mostly, finds & runs ***other*** programs
- Has variables, control structures, functions, …

```
if [ $# -lt 3 ]; then echo 'Too few args!'; fi
```

- Allows control over input and output

```
gcc foo.c -o foo > compile.log 2> errors.log
```

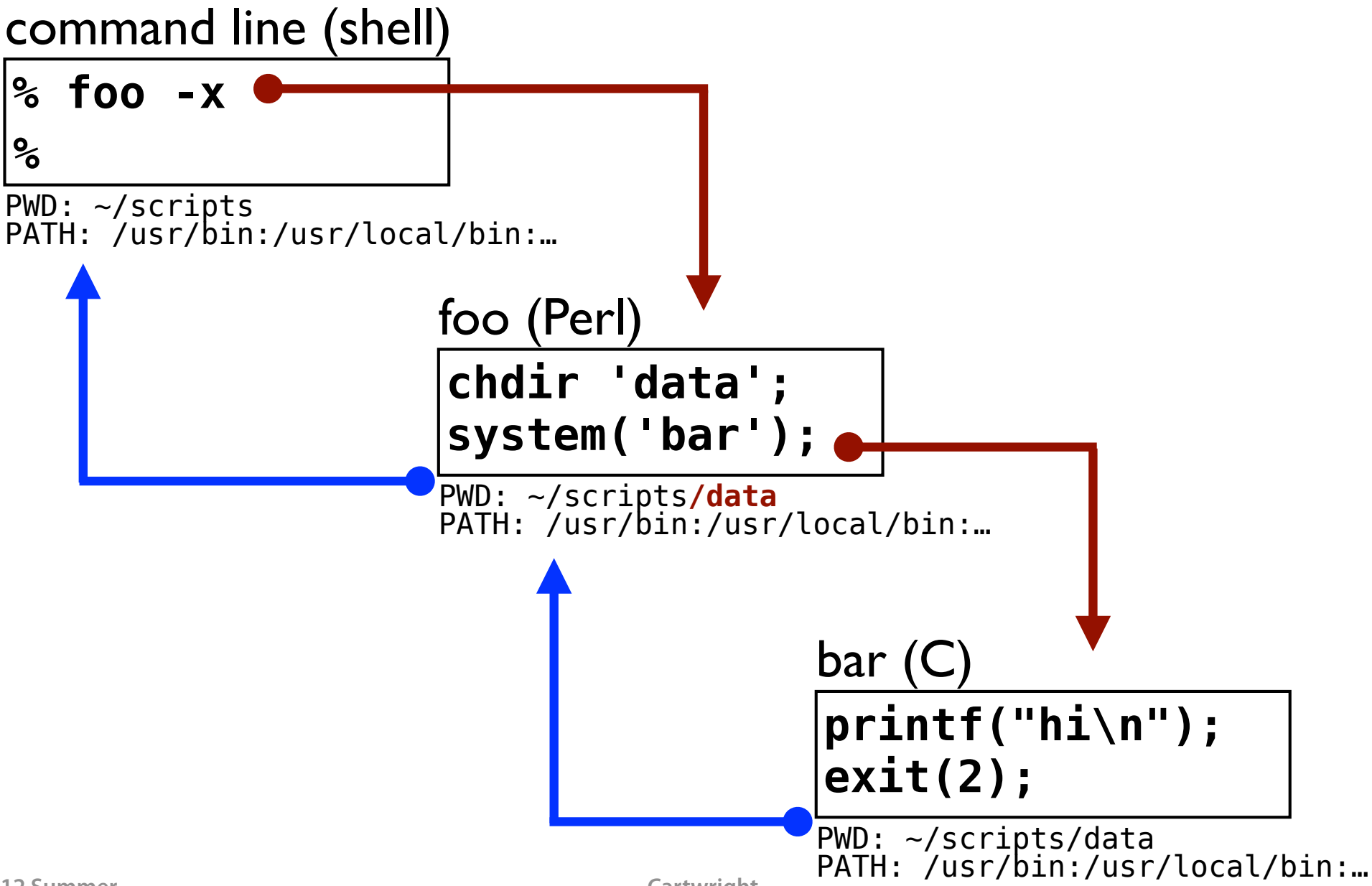
- Supports basic workflows

```
grep '^[0-9]' file.txt | sort | uniq -c
```

Common Shell Commands

Built-in	**`cd`**, **`echo`**
Manual pages	**`man`**
Show file entries	**`ls`**, **`find`**
Change file privileges	**`chown`**, **`chmod`**
Manipulate whole files	**`cp`**, **`mv`**, **`rm`**, **`ln`**, **`unlink`**
Manipulate directories	**`mkdir`**, **`rmdir`**
View files	**`cat`**, **`more`**, **`less`**, **`wc`**
Filter/change files	**`grep`**, **`sed`**, **`awk`**
Edit text	**`pico`**/**`nano`**, **`vi`**, **`emacs`**
Scripting	**`perl`**, **`python`**, **`ruby`**

Shells and Commands



The System ⇒ Your Script

How to Get Input Into Your Script

- Define (hard-code) input in script
  - \+ Easy to code
  - – Must change script to change input
- Ask user for terminal input
  - \+ Easy to code, no script changes each run
  - – Cannot automate (easily) — user must type input each run
- Read input from files
  - \+ No script changes each run, can automate
  - – Must parse file contents; separate file to manage
- Accept command-line arguments
  - \+ No script changes each run, can automate
  - – Must parse arguments; cumbersome for many or long inputs
- Read environment variables
  - \+ No script changes each run, can automate
  - – Must parse arguments; cumbersome for many or long inputs

Command-Line Arguments

```
% script.pl data.txt 1200 'Tim Cartwright'
```

$0	command (script) name
@ARGV	command-line arguments

```
#!/usr/bin/perl
use strict;
use warnings;

if (scalar(@ARGV) != 3) {
    die "usage: $0 PATH VALUE NAME\n";
}

my ($path, $value, $name) = @ARGV;
```

(Required) Arguments *and* Options

```
% script.pl --lo --with foo -aXb file1
```

- Use **Getopt::Long** for options (starting with **-**)
- Required arguments remain in **@ARGV**

```
use Getopt::Long;

my $lo = 0;
my $with = '';
GetOptions('lo'      => \$lo,
           'with=s' => \$with,
           # etc.);

my $file = $ARGV[0]; # do after GetOptions
```

Environment

- **%ENV** : environment variables
- Readable and writable

```
while (my ($key, $value) = each %ENV) {
    print "$key => $value\n";
}
```

```
my @path = split(':', $ENV{'PATH'});

my @new = grep(! m{/sbin}, @path);
$ENV{'PATH'} = join(':', @new);

delete $ENV{'SOME_OTHER_VAR'};
```

Your Script ⇒ The System

Running a Command

`system(…)`

- Runs the given command as a subprocess
- ***May*** create a shell to parse command
- Waits for command to finish
- Command inherits environment, etc.

```
system("gzip $output_file");
system('Rscript', 'foo.R', 1200, 42);
system("octave test-script > my_oct.log");
```

Sneaky system() Subtleties

Create a shell?

```
system('ls');                          # no
system('ls >> my_file');               # yes
system('ls', '>>', 'my_file');         # no
```

Children do not affect parent

```
system('pwd');          # => /home/cat/foo
system('cd bar');
system('pwd');          # => ???
```

Sneaky system() Subtleties II

Watch out for quoting issues

```
system("echo 'Don'\\''t say \"no\"!'");
[shell] echo 'Don'\''t say "no"!'
[output]      Don't say "no"!
```

Return Values

- **`system()`** returns exit code × 256 (**`<< 8`**)
- Exit code of **`0`** is good in shell,
- But **`0`** is *false* in Perl, so…

```
system(…) and die('fail');     # confusing
!system(…) or die('fail');     # may miss !
```

- … instead, strive for clarity:

```
system(…) == 0 or die('fail');
```

```
if (system(…) != 0) {
    die('fail');
}
```

Errors

- **`system()`** return value ≡ **`$?`** ≡ exit code **`<< 8`**
- **`$? == -1`** means it failed to execute
- **`$!`** is the system error message
- Rarely need to be this thorough:

```
if ($? == -1) {
    print "failed to execute: $!\n";
} elsif ($? & 127) {
    print "died with signal";
} else {
    print "exited " . ($? >> 8) . "\n";
}
```

Getting Output

```
my $sys_date = `date`;
```

- Like **system()** but returns standard output
- **$?** and **$!** are set in the same way

- Need standard error, too?

```
my $sys_date = `date 2>&1`;
```

- In list context, returns list of lines in output
- Backticks interpolate!

```
my @files = `find $directory`;
```

Miscellaneous

Quit script with given exit code (**0** = good)

```
exit 1;
```

Print message to standard error and exit (non-zero)

```
die 'message';
```

Print message to standard error

```
print STDERR 'message';
warn 'message';
```

When To Use

- Glue between existing commands
- More powerful shell
- Workflow management
- ***Not*** to replace Perl functions!
  - E.g., do not actually use `` `date` ``
- Your ideas?

Last 2 Slides…

Other Scripting Languages

- In command-line scripts, expect
  - Command-line arguments
  - Environment
  - Standard in, out, and error
  - System calls
  - Exit with status
  - Windows will be different…
- Embedded scripting (PHP, Lua, JavaScript, …)
  - Expect more restrictions

Homework

- Find IP addresses for image downloads
- Use external commands for:
  - Fetching an HTTP document (web page)
  - Looking up IP addresses
- Middle section will involve regular expressions