CS 170 - Week 1

Dani Kudrow & Jasen Hall

Your TAs

Dani Kudrow

Office hours: TR 10:00-11:00am
Phelps 3525

Email: dkudrow@cs.ucsb.edu

Jasen Hall

Office hours: R 12:00-1:00pm
F 1:00-2:00pm
Phelps 3525

Email: jasen@cs.ucsb.edu

Projects

- Discussion sections will focus on projects
- **Start early!**
  - Each project builds on the last - you don’t want to fall behind!
  - No solutions will be posted - your code **must** work!
- Prerequisites
  - C
  - Unix utilities (man, make, gdb, gcc, etc.)

Project1 - jshell

- The shell provides an interface to the operating system kernel

- Performs two tasks
  - Interpret user commands
  - Execute those commands

Parsing

- You will need to be able to identify the following symbols from an entered command line:
  - <, >, >>
    - Input and output redirection
  - |
    - Pipelining
  - &
    - Background process
- **`$ man bash`**
  - **`/REDIRECTION`**
  - **`/Pipelines`**

File Redirection

- You can replace STDIN with a file
  - $ sort < /etc/passwd
  - $ jsh < testcommands.txt

- You can replace STDOUT with a file, too
  - $ ls -l > filelist.txt
  - $ jsh < testcommands.txt > testoutput.txt
    - *Important*: if the file you are writing STDOUT to already exists, you will overwrite it. This is called clobbering.

- You can replace STDOUT with a file, but append to it
  - $ jsh < testcommands.txt >> testoutput.txt
    - This will create a new output file only if it doesn’t already exist. Otherwise, it will append to it.

Pipelining and Background

- You can connect the output of one command to the input of another using a pipe
  - $ sort < /etc/passwd | less
  - $ jsh < badcommands.txt | grep ‘command not found’

- By placing a command in the background, you allow it to run without blocking your further use of the shell
  - $ calculate_pi.py &
    [1] 1889
    …
    [1]+ Done calculate_pi.py

System Calls

- System calls allow user programs to invoke kernel procedures
  - `MOV 0x29, %rax  ;; specify dup() system call`
  - `MOV 0x1, %rbx  ;; specify argument`
  - `SYSCALL  ;; trap to operating system`

- #include <unistd.h> /* Unix system call wrappers */

- By convention:
  - On failure, return -1 and set errno (errno.h)

- Man pages
  - The C system call wrappers have man pages in section 2
    - $ man 2 open

System Calls

fork
exec
wait/waitpid
pipe
dup

System Calls - fork()

```
int child_pid = fork();          /* create a new process */

if (0 > child_pid) {
        /* Fork failed! */
}
else if (0 == child_pid) {
        /* This code will execute in the child process */
}
else {
        /* This code will execute in the parent process */
        /* retval here contains the PID of the child process */
}
```

System Calls - wait()

```
int child_pid = fork()
int status;
…

else {
    /* block until a child process terminates */
    wait(&status);

    /* don’t block, even if no child has terminated */
    waitpid(-1, &status, WNOHANG);
}
```

System Calls - exec()

```
char *args[3];
int child_pid = fork();
int status;

args[0] = “ls”; args[1] = “-l”; args[2] = NULL;

if(child_pid == 0)
        execvp(args[0], args);
else
        wait(&status);
```

System Calls - pipe()

```
int pipefd[2];  /* create our pipe array */
char bufout[] = "Pipe!\0";
char bufin[6];

pipe(pipefd);  /* create our in/out pipes */
write(pipefd[1], bufout, strlen(bufout));     /* 2nd element is the write pipe */
read(pipefd[0], bufin, strlen(bufout));       /* 1st element is the read pipe (ikr!) */

/* bufin now contains “Pipe!\0” */
```

System Calls - dup()

```
int fd, newfd;
fd = open("dupfile.txt", O_WRONLY|O_CREAT);
newfd = dup(fd);   /* dup() creates a new file descriptor
                      opened to the same file as fd */
write(newfd, "Writing to dup'ed file descriptor.\0", 256);
close(newfd);
close(fd);

/* String written to dupfile.txt using duplicated descriptor */
```

What are you (not) responsible for?

- You need to create the shell program and turn in:
  - Source code
  - Makefile

- You DO NOT need to write:
  - ls, mkdir, cd, grep
  - Anything else?

Strategy

1. Create a loop that only exits on a CTRL-D or the command “exit”
    a. How does C interpret CTRL-D?
2. Write string handling functions that can parse a command
    a. What is the command and what are it’s arguments?
    b. Is I/O being redirected, and if so to where?
    c. Is the user chaining multiple commands together with a pipe?
3. Create the process(es) to run your commands
    a. fork()? exec()? pipe()?
4. Make sure the spawned process completes
    a. Or don’t depending on how the user called the program.

Any questions?