Operating Systems

myshell

Shell Intro

- A shell is a command line interpreter and is an important tool for an interactive OS
- It reads sets of commands typed at a prompt (the command line) that request one or more programs be started with input and output using a terminal, a file, or another program.
- It then issues the set of commands to the operating system to service the request or report any errors.
- You use a shell like this when time you remotely log into a department computer
- There are 2 major flavors with slight differences
  - The Bourne shell (written by Steven Bourne)
  - The C shell (written by Bill Joy)

System calls for a basic shell

```
while (TRUE) {                                     /* repeat forever */
   type_prompt( );                                 /* display prompt */
   read_command (command, parameters)              /* input from terminal */

if (fork() != 0) {                                 /* fork off child process */
   /* Parent code */
   waitpid( -1, &status, 0);                       /* wait for child to exit */
} else {
   /* Child code */
   execve (command, parameters, 0);                /* execute command */
 }
}
```

Shell Lab Overview

- The program you will write has these requirements.
  - It will be started with the command myshell [prompt]. If the optional prompt is not specified, the prompt will be ”myshell: ".
- read in the line of commands that will consist of
  - executables and arguments
  - the tokens “**<**”, “**>**”, “**>>**”, “**&**”, “**;**” and “**|**”
- You will then create all of these processes with the proper redirection of input and output.
- The shell should exit when the user types CTRL-D or exit
- You must implement “cd” as a shell builtin
- We will test it automatically, so it is important that the default prompt be as specified
  - We will stress test your implementation, so test thoroughly!

Example myshell Session

```
$ myshell
myshell: pwd
/your/source/directory
myshell: cat > f1
this is a test
Hoosiers win championship
^D
myshell: cat f1 | sort
Hoosiers win championship
this is a test
myshell: sort < f1 | head -1 | cat -n
    1 Hoosiers win championship
myshell: exit
$
```

System calls and process creation

- The shell runs a program using two core system calls: **fork()** and **execvp()**.
  - Read the man pages to see how you need to use the calls (use the command: `man 2 fork`).
- **fork()** creates an exact copy of the currently running process, and is used by the shell to spawn a new process.
- **execvp()** call is used to overwrite the currently running program with a new program, which is how the shell turns this new process into the program you want to run.

myshell Guidelines

- Input and output redirection, either to files or pipes, is done using the **dup2()** command.
- This command allows you redirect a file descriptor to point to another file descriptor.
- For this lab you will create a file descriptor using the **open()** or **pipe()** system calls and then set stdin (0) or stdout (1) to point to the new descriptors.
- Then, when you call **execvp()**, the new process will read or write from the file or pipe thinking it is the standard input or output.

myshell Guidelines

- The shell must wait until all of the previously started programs complete unless the user runs them as background processes (with &). This is done with the **wait*()** system call.
- Also, the shell will need deal with processes that become **zombies**. Zombie processes are processes that have exited, but the OS keeps them around until the parent (the shell) checks their exit status.
- You should use **wait4()** or **waitpid()** with the non-blocking option set to clean up any zombie processes before you start new processes.

myshell Guidelines

- All of these functions are system calls, which means they need to be tested to see if they fail when you use them. The man pages will tell you what the call should return on success (man 2 *function*). If there is an error you need to report it with the **perror()** function.
- Use the man pages to learn about system calls

Grading Issues

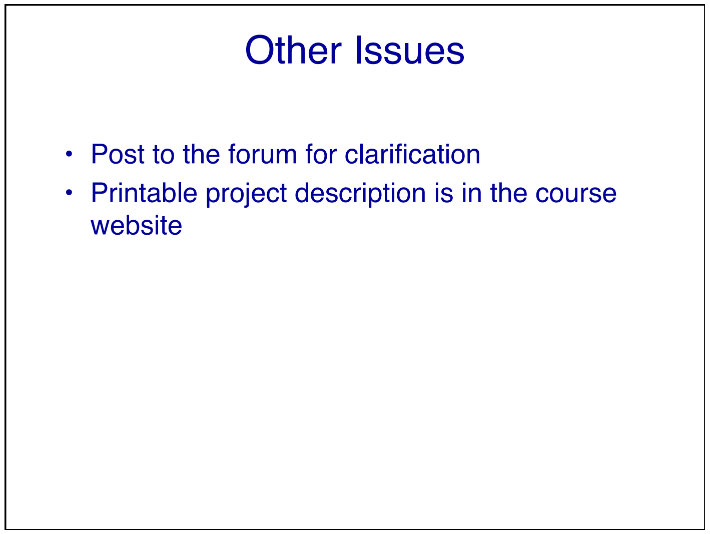
- No *illegal* input the user gives should ever cause the shell to exit. You will need to use the **exit**() call when children that have come across an error need to die.
  - However, the parent (shell) should never make that call except when it gets an 'exit' or CTRL-D.
- All errors should be handled gracefully, no matter where they occur.
  - Even if an error occurs in the middle of a long pipeline, it should be reported accurately and the shell should recover gracefully.
- This is an individual project
- Read the policies and procedures document

Grading Points

- Error messages for failed system calls need to be informative. Thus, if a file named as an output redirect cannot be opened for writing, the error message must reveal the file name and reason for the error.
  - Use the **perror**() function.
- The shell should only give a prompt when the last foreground process run completes.
  - "sleep 1 &" followed by "cat" should give a prompt when cat completes, not the sleep.

myshell Lab

- Create a directory with a Makefile
  - include a make clean target (rm *.o *~)
- Tar a cleaned up version of your directory (that includes your username)
- gzip and submit via the course web site



Other Issues

- Post to the forum for clarification
- Printable project description is in the course website