ICS 53, Spring 2015

Lab 2: A Simple Shell

A *shell* is a mechanism with which an interactive user can send commands to the OS and by which the OS can respond to the user. The OS assumes a simple character-oriented interface in which the user types a string of characters (terminated by pressing the Enter or Return key) and the OS responds by typing lines of characters back to the screen. The character-oriented shell assumes a screen display with a fixed number of lines (say 25) and a fixed number of characters (say 80) per line.

Typical Shell Interaction

The shell executes the following basic steps in a loop.

1. The shell prints a prompt to indicate that it is waiting for instructions.

```
prompt>
```

2. The user types a command, terminated with an <ENTER> character ('\n'). All commands are of the form `COMMAND [arg1] [arg2] … [argn].`

```
prompt> ls
```

3. The shell executes the chosen command and passes the command the arguments. The command prints results to the screen. Typical printed output for an ls command is shown below.

```
hello.c hello testprog.c testprog
```

There are two types of commands, built-in commands which are performed directly by the shell, and general commands which indicate compiled programs which the shell should cause to be executed. You will support only one built-in command, quit, which ends the shell process. General commands can indicate any compiled executable. We will assume that any compiled executable used as a general command must exist in the current directory. The general command typed at the shell prompt is the name of the compiled executable, just like it would be for a normal shell. For example, to execute an executable called hello the user would type the following at the prompt:

```
prompt> hello
```

Built-in commands are to be executed directly by the shell process and general commands should be executed in a child process which is spawned by the shell process using a fork command.

General commands can be executed either in the foreground or in the background. When a user wants a command to be executed in the background, an "&" character is added to the end of the command line, before the <ENTER> character. The built-in command is always executed in the foreground. When a command is executed in the foreground, the shell process must wait for the child process to complete.

I/O redirection

Your shell must support I/O redirection. A process, when created, has three default file identifiers: **stdin**, **stdout**, and **stderr**. If the process reads from **stdin** then the data that it receives will be directed from the keyboard to the **stdin** file descriptor. Similarly, data received from **stdout** and **stderr** are mapped to the terminal display.

The user can redefine **stdin** or **stdout** whenever a command is entered. If the user provides a filename argument to the command and precedes the filename with a “less than” character "**<**” then the shell will substitute the designated file for **stdin**; this is called redirecting the input from the designated file. The user can redirect the output (for the execution of a single command) by preceding a filename with the right angular brace character, "**>**" character. For example, a command such as

```
prompt> we < main.c > program.stats
```

will create a child process to execute the “we” command. Before it launches the command, however, it will redirect **stdin** so that it reads the input stream from the file **main.c** and redirect **stdout** so that it writes the output stream to the file **program.stats**.

The book provides you with a working shell program in Figures 8.22, 8.23, and 8.24. All code, plus associated included files (csapp.h) can be downloaded from the book’s web page at http://csapp.cs.cmu.edu/2e/code.html . You can use this code as a base and modify it to fit your needs.

Handling Unusual Inputs

- We’re fairly flexible about how you handle unusual inputs but there are a few constraints.
- Input and command line format errors should not cause your shell to crash
- Your shell should be able to handle the following inputs without errors: blank lines and extra whitespaces
- Print some error if the command executable program does not exist in the local directory.