ICS 53, Fall 2022

Assignment 2: A Shell

A shell is a program that allows a user to send commands to the operating system (OS) and allows the OS to respond to the user by printing output to the screen. The shell allows a simple character-oriented interface in which the user types a string of characters (terminated by pressing Enter(\n)) and the OS responds by printing lines of characters back to the screen. The goal of assignment 2 is to implement a simple shell.

Typical Shell Interaction

The shell executes the following basic steps in a loop.

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

```
prompt >
```

2. The shell gets a command from a user and shows the results of commands to the screen.
   - Assumption
     - The user command is terminated with a <ENTER> character ('\n'))
     - Format of command :
       - COMMAND [arg1] [arg2] ... [argn] (if COMMAND requires args).
       - Each argument is separated by **space** or **tab** characters

   eg1. The ls command is used to list files or directories.(Note: This is an example, you do not need to implement the "ls" command.)

```
prompt > ls
hello.c hello testprog.c testprog
```

   eg2.

```
prompt > cd ./assign2
prompt >
```

3. When the shell gets a "`quit`" command, the shell is terminated.

Commands that your shell must support

There are two types of commands,

- **general commands** - These commands are the names of any compiled executable in the local directory. For example, if you want to execute an executable called `hello` you would enter the following.

```
prompt > hello
```

- **built-in commands** which are performed directly by the shell.

Your shell will support six built-in commands: **jobs, bg, fg, kill, cd, pwd, and quit**.

Stages of the assignment

This assignment consists of five different parts and the correct execution of each part has its own score. We will execute and grade these parts sequentially, so you must retest the previous parts after implementing each new part to ensure everything works. We suggest that you make backups of your code after completing each part so that you can recover the last working version in case of problems.

Stage 1: Accepting Basic Commands

Write code that supports **cd, pwd.** and **quit** commands.

- `cd <name of the directory>`: Change the current working directory of the shell. If the given directory name is ".." then the working directory should go up on level of the directory tree.
- `pwd`: Print the name of the working directory.
- `quit`: Ends the shell process.

Your code should print the prompt "`prompt >`" and accept a command from the user. This code should have the ability to extract arguments if the command accepts arguments. This is the case with the `cd` command which accepts the directory name as an argument. This should iterate until the user enters `quit`.

`cd`, `pwd` and `quit` are the built-in commands and they should be executed directly by the shell process.

Stage 2: Local executables

Write code that supports **foreground jobs**. A foreground job is one that blocks the shell process, causing it to wait until the foreground job is complete. While a foreground job is executing, the shell cannot accept commands. Some jobs complete very quickly (like printing hello on the screen) but some jobs may take a long time to complete execution and would block the shell as long as they are running. Only one job can run in the foreground at a time.
A foreground job is started when the user simply enters the name of the executable program at the prompt. For example, the following command executes a program called "hello" in the foreground.

```
prompt > hello
```

2.1. Write a function that executes the executable program in a child process which is spawned by the shell process using the fork() function in C.

2.2. Ensure that all processes forked by the shell process must be properly reaped after the child process is terminated. For a foreground job, the shell should explicitly call wait() or waitpid() after it forks the new child process.

2.3 If a foreground job is running and the user types **ctrl-C** into the shell, then the foreground job should be **terminated**. You should accomplish this by creating a handler for the SIGINT signal in the shell. The SIGINT signal is received by the shell when a user types ctrl-C and the default behavior would be to terminate the shell. Instead of terminating the shell, the child process running the foreground job should be terminated. Your handler for the SIGINT signal should send a SIGINT signal to the foreground child process using the kill() system call.

Example

<counter.c>

```
#include <stdio.h>
#include <unistd.h>

int main() {
    unsigned int i = 0;
    while(1)
    {
        printf("Counter: %d\n", i);
        i++;
        sleep(1);
    }
    return 0;
}
```

If you execute counter on your prompt, it prints the counter each second. When you type Ctrl-C (^C), the foreground job is terminated and then the shell can get a new command from a user.

```
prompt > counter
counter: 0
counter: 0
counter: 0
counter: 0
counter: 0
^Cprompt >
```

Stage 3: Background process

Write code that supports background jobs. When you start a program and enter an ampersand (&) symbol at the end of the command line, the program becomes a background job. In this case, the shell process is not blocked while the job is executing. immediately after a background program is started, the shell will print a new prompt and the user can enter new commands. This is an example of starting a job in the background.

```
prompt > hello &
```

The ampersand (&) symbol is a unique token, separated from the text before it by one or more space characters. For this assignment, you can assume that space separation will always exist.

3.1 Ensure that your program can execute a job in the background when an & is added to the end of the command line.

3.2 Your code must guarantee that all child processes forked by the shell process must be properly reaped after the child process is terminated. For a background job, the shell should not call wait() or waitpid() because this would cause it to block until the child process is complete. Instead, you should create a handler for the SIGCHLD signal which calls wait() or waitpid(). The SIGCHLD signal is received by the shell process whenever one of its child processes terminates. By using the SIGCHLD handler to reap the child process, the shell process does not need to block while the job executes.

Stage 4: Job Control

Each job can be moved between one of the three job states, Foreground/Running, Background/Running, and Stopped. Each job can also be forcibly terminated by the user.

- If the user enters ctrl-C into the shell while a foreground job is executing, the job is terminated.
- If the user enters the kill built-in command then the indicated job is terminated, whether it is in the state Background/Running or Stopped.
- If the user enters ctrl-Z into the shell while a foreground job is executing, the foreground job moves to the Stopped state. You can assume that built-in commands are never stopped.
- If a user enters fg <job_id|pid> into the shell and the job indicated by <job_id|pid> is currently in the Stopped state or the Background/Running state, then it is moved to the Foreground/Running state. In order to move a process from the Stopped state to the Foreground/Running state, the process must be restarted by sending it a SIGCONT signal using the kill() system call.
- If a user enters bg <job_id|pid> into the shell and the job indicated by <job_id|pid> is currently in the Stopped state, then it is moved to the Background/Running state. In order to move a process from the Stopped state to the Background/Running state, the process must be restarted by sending it a SIGCONT signal using the kill() system call.

4.1 Make a struct to record the information about each job. You need to assign a job id and a pid when you make a new process. Your struct must contain these numbers as well as the state of the job (Foreground or Background).

4.2 Your code should accept the command **jobs** which should list the running and stopped background jobs. Status can be “Running” (if it is in the “Background/Running” state) and “Stopped”. The format is shown here.

```
[<job_id>] (<pid>) <status> <command_line>
prompt > jobs
[1] (30522) Running hello &
[2] (30527) Stopped sleep
```

4.3 Write a signal handler that stops a foreground job. A job is stopped if it is not currently executing but is not terminated, so it can be restarted later. A job which is stopped must be automatically placed in the background so that it does not cause the shell to be blocked. You can assume that a built-in command will never be stopped.

If a foreground job is running and the user types **ctrl-Z** into the shell, then the foreground job should be **stopped**. You should accomplish this by creating a handler for the SIGTSTP signal in the shell. The SIGTSTP signal is received by the shell when a user types ctrl-Z and the default behavior would be to stop the shell process. Instead of stopping the shell, the child process running the foreground job should be stopped. Your handler for the SIGTSTP signal should send a SIGTSTP signal to the foreground child process using the kill() system call.

4.4 Your code should accept the command **fg <job_id|pid>** which changes the state of a job currently in the Stopped state or the Background/Running state to the Foreground/Running state. A user may refer to the job using either its job_id or pid. In case the job_id is used, the JID must be preceded by the "%" character.

- JID case

```
prompt > fg %1
```

- PID case

```
prompt > fg 30522
```

4.4 Your code should accept the command **bg <job_id|pid>** that changes the state of a job currently in the Stopped state to the Background/Running state.

4.5 Your code should accept the command **kill <job_id|pid>** that terminates a job by sending it a SIGINT signal using the kill() system call. Be sure to reap a terminated process.

4.6 Complete the **quit** function to terminate all the child processes.

Stage 5: I/O redirection

Your shell must support I/O redirection.

Most command line programs that display their results do so by sending their results to standard output (display). However, before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. To redirect standard output to a file, the ">" character is used like this:

1. To redirect standard output to a file, the ">" character is used like this:

   ```
   prompt > jobs > file_list.txt
   ```

   In this example, the ls command is executed and the results are written in a file named file_list.txt. Since the output of ls was redirected to the file, no results appear on the display. Each time the command above is repeated, file_list.txt is overwritten from the beginning with the output of the command ls.

2. To redirect standard input from a file instead of the keyboard, the "<" character is used like this:

   ```
   prompt > sort < file_list.txt
   ```

In the example above, we used the sort command to process the contents of file_list.txt and the output is displayed as a standard output.

3. The results are output on the display since the standard output was not redirected. We could redirect standard output to another file like this:

```
prompt > add < number.txt > added_number.txt
```

Your shell should also support the command to append to a file. It is virtually the same as standard output redirection except for the file to which the output was redirected to isn't overwritten, instead, the incoming contents are simply appended to the end of the file. To redirect standard output and append to a file, use the ">>" string. For example:

```
prompt > jobs >> file_list.txt
```

For I/O redirection, you can assume that the commands given to your shell will contain at any given time **at most 1 input redirection and at most one output redirection** (output redirect or output append).

I/O redirection Authorization

We should add permission bits when we do I/O Redirection. Permission bits control who can read or write the file. https://www.gnu.org/software/libc/manual/html_node/Permission-Bits.html

- **Input redirection to "input.txt" (Read)**

```
mode_t mode = S_IRWXU | S_IRWXG | S_IRWXO;
inFileID = open ("input.txt", O_RDONLY, mode);
dup2(inFileID, STDIN_FILENO);
```

- **Output redirection to "out.txt" (Create or Write)**

```
outFileID = open ("out.txt", O_CREAT|O_WRONLY|O_TRUNC, mode);
dup2(outFileID, STDOUT_FILENO);
```

- **Output append to "out.txt" (Create or Append)**

```
outFileID = open("out.txt", O_CREATE|O_APPEND|O_WRONLY,
mode); dup2(outFileID, STDOUT_FILENO);
```

Here is a simple example for how to test your I/O redirection:

- Create a file "number.txt" containing an arbitrary number, and a program "add.c",then compile it. Then use I/O redirection in your shell to debug and check the result.

```
< add.c >

int main(int argc, char * argv[]){

int n = atoi(argv[1]);

print("%d \n", n+2); // print n+2

return 0;

}


prompt > add < number.txt > added_number.txt
```

Submission Instructions

Your source code should be a single c file named 'hw2.c'. Submissions will be done through Gradescope. You have already been added to the Gradescope course for ICS53. Please login to with your school (UCI) email to access it. Please remember that each C program should compile and execute properly on openlab.ics.uci.edu when it is compiled using the gcc compiler version 11.2.0. The only compiler switch which may be used is -o to change the name of the executable.

Specific directions

- Headers:
  stdio.h, string.h, unistd.h, stdlib.h, sys/stat.h, sys/types.h, sys/wait.h, ctype.h, signal.h, fcntl.h

* As long as you can compile it on openlab (gcc 11.2.0) without any additional compiler flags, you can use any headers other than those specified.

- MaxLine: 80, MaxArgc: 80, MaxJob: 5
- Use both execvp() and execv() to allow either case.
  execvp() : Linux commands {ls, cat, sort, ./hellp, ./slp}.
  execv() : Linux commands {/bin/ls, /bin/cat, /bin/sort, hello, slp}.