ICS 53, Spring 2022

Assignment 2: A Simple Shell

A shell is a program which 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.

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]. (each argument is separated by space or tab characters)

```
prompt> ls
```

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

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

4. Some commands may accept arguments. These arguments must be provided after the command, on the same line as the command. All arguments will be separated by 1 or more whitespace characters, either space or tab characters. An example of a command which accepts arguments is shown below where a program called “foo” is invoked and the program takes three command-line arguments.

```
prompt> foo 1 2 3
```

Types of commands that your shell must support

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. Your shell will support six built-in commands: jobs, bg, fg, kill, cd and quit. Your shell must also support general commands.

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. Be sure to reap all terminated child processes.

Job Control

A job is a program started interactively from the shell. Each job is assigned a sequential job ID (JID). Because a job is executed within a process, each job has an associated process ID (PID). Each job can be in one of three states:

1. **Foreground/Running**: A foreground job is one which 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 execute 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`

   You can assume that any built-in command will be executed in the foreground.

2. **Background/Running**: 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, so the shell can be used 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 a background job.
    - `prompt> hello &`

   In this example, 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 that space separation will always exist.

3. **Stopped**: A job is stopped if it is not currently executing but it 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.

Built-In Commands

- **jobs**: 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
```

- **fg <job_id|pid>**: Change 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.

```
prompt> fg %1

prompt> fg 30522
```

- **bg <job_id|pid>**: Change the state of a job currently in the Stopped state to the Background/Running state.
- **kill <job_id|pid>**: Terminate a job by sending it a SIGINT signal using the kill() system call. Be sure to reap a terminated process.
- **cd <name of the directory>**: Change the current working directory of the shell.
- **quit**: Ends the shell process.

Processes and Jobs

Built-in commands are executed in the shell process, but general commands are executed in a child process which is forked by the shell.

All processes forked by the shell process must be properly reaped after the child process is terminated. How reaping should be performed depends on whether the job is a foreground or background job. If the job is a foreground job then the shell should explicitly call wait() or waitpid() after it forks the new child process. If the job is a background job then 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.

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.

If a foreground job is running and the user types ctrl-Z into the shell, then the foreground job should be stopped (moved to the Stopped state). 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.

Summary of Job State Changes

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.

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:

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

Your shell should also support the command to append to a file. It is virtually the same as standard output redirection except 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> ls >> 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_TRUNC, mode);
  dup2(outFileID, STDOUT_FILENO);
  ```

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 4.8.5. The only compiler switches which may be used are -o (to change the name of the executable), -std=c99 (to use C99 features), and -std=c11 (to use C11 features).

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 4.8.5) 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}.

Example 1

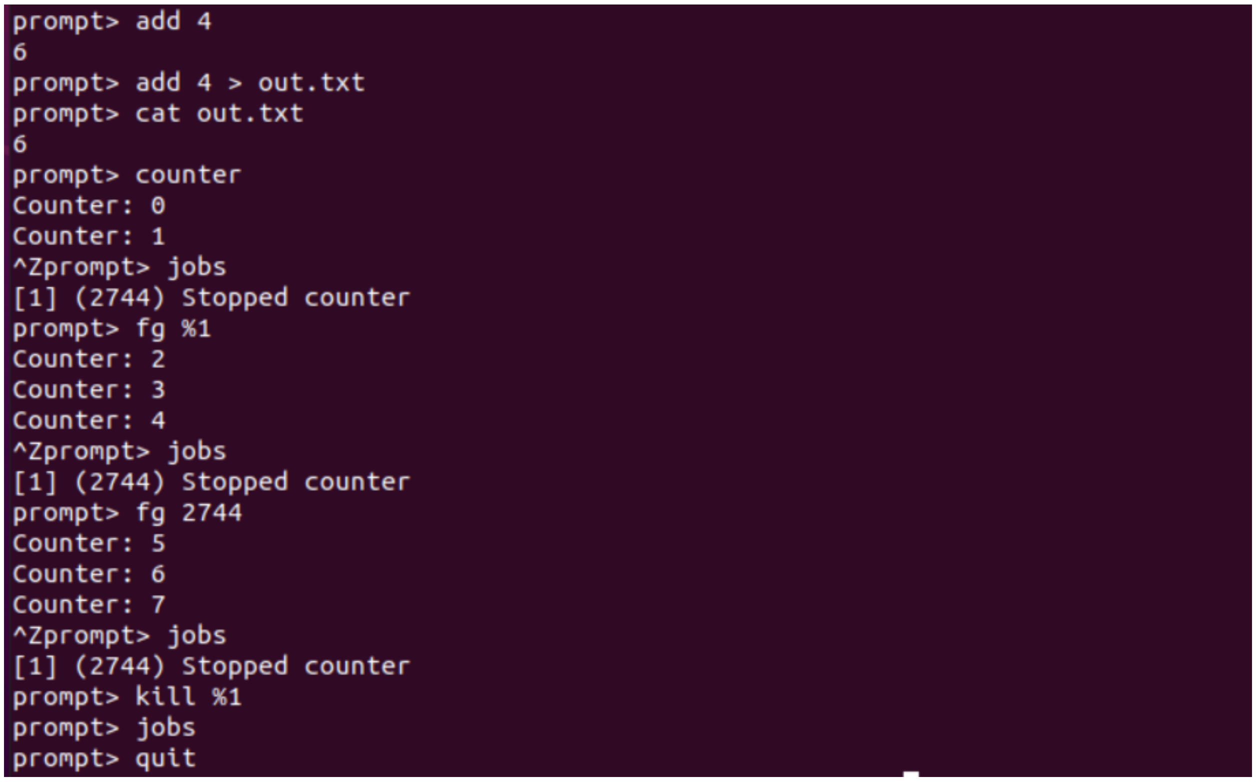
1. Create 2 programs, add.c and counter.c, and compile them.

```
< add.c >
int main(int argc, char * argv[]){
   int n = atoi(argv[1]);
   printf("%d \n", n+2);  // print n+2
   return 0;
}
<counter.c>
int main() {
   unsigned int i = 0;
   while(1)
   {
      printf("Counter: %d\n", i);
      i++;
      sleep(1);
   }
   return 0;
}
```

```
$gcc add.c -o add
```

```
$gcc counter.c -o counter
```

Now we have compiled executables, “add” and “counter”. ^Z in below == ctrl-Z



```
prompt> add 4
6
prompt> add 4 > out.txt
prompt> cat out.txt
6
prompt> counter
Counter: 0
Counter: 1
^Zprompt> jobs
[1] (2744) Stopped counter
prompt> fg %1
Counter: 2
Counter: 3
Counter: 4
^Zprompt> jobs
[1] (2744) Stopped counter
prompt> fg 2744
Counter: 5
Counter: 6
Counter: 7
^Zprompt> jobs
[1] (2744) Stopped counter
prompt> kill %1
prompt> jobs
prompt> quit
```

Example 2

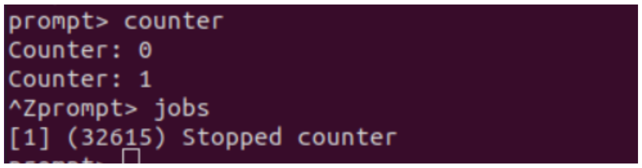
Prepare two terminals on the same server.

[Terminal1]
ssh <your_id>@openlab.ics.uci.edu
You will see your server name e.g. <your_id>@circinus-14
circinus-14 is your current server name in this example.
[Terminal2]
ssh <your_id>@<server_name>.ics.uci.edu
e.g. ssh <your_id>@circinus-14.ics.uci.edu
In this way you can access to the same server using two terminals.

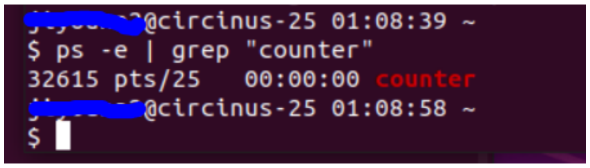
1. [Terminal1] Run your shell
2. [Terminal1] prompt> counter
3. [Terminal1] ctrl-z
4. [Terminal1] jobs
   [1] (<a_pid>) Stopped counter
5. [Terminal2] ps -e | grep “counter”
   <a_pid> pts/0   00:00:01 counter
6. [Terminal1] kill <a_pid>

7. [Terminal1] jobs
   No output
8. [Terminal2] ps -e | grep “counter”
   No output

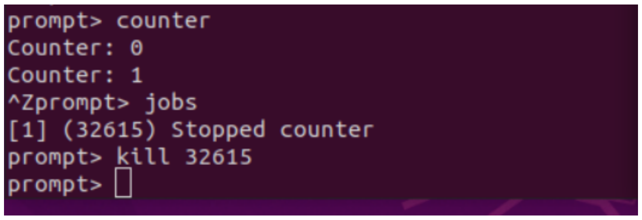
< Terminal 1 >   <Terminal 2 >



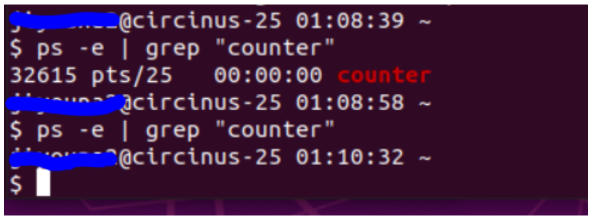
```
prompt> counter
Counter: 0
Counter: 1
^Zprompt> jobs
[1] (32615) Stopped counter
```



```
[redacted]@circinus-25 01:08:39 ~
$ ps -e | grep "counter"
32615 pts/25    00:00:00 counter
[redacted]@circinus-25 01:08:58 ~
$ 
```



```
prompt> counter
Counter: 0
Counter: 1
^Zprompt> jobs
[1] (32615) Stopped counter
prompt> kill 32615
prompt> 
```



```
[redacted]@circinus-25 01:08:39 ~
$ ps -e | grep "counter"
32615 pts/25    00:00:00 counter
[redacted]@circinus-25 01:08:58 ~
$ ps -e | grep "counter"
[redacted]@circinus-25 01:10:32 ~
$ 
```

If you can still see “<a_pid> pts/0   00:00:01 counter” after kill,
it means that you did not properly terminate a child process.

References

“Understanding the job control commands in Linux – bg, fg and CTRL+Z“, Understanding the job control commands in Linux – bg, fg and CTRL+Z”
https://www.thegeekdiary.com/understanding-the-job-control-commands-in-linux-bg-fg-and-ctrlz/