CS441 Project 3

The Shell

CS 441/541 – Fall 2020

Project Available		Sep. 18
Component	**Points**	**Due Date (at 11:59 pm)**
Group selection		Oct. 1
Demo timeslot		TBD
Parsing	5	
Foreground	3	
Background	3	
Batch mode	2	
`jobs`	3	
`history`	2	
`wait`	3	
`fg`	3	
`exit`	3	
File redirection	3	
Style & Organization	5	
Documentation	5	
Tests	5	
Demo	5	
Total	50	Oct. 18
Submission	Groups of 2 allowed	PR and Demo

Objectives

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

There are three specific objectives to this assignment:

- To further familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

Packaging & handing in your project

A template project is available on BitBucket that includes data structures and some suggestions for functions. You are not required to use the provided suggestions if you do not find it useful, but you must use the template repository and maintain its file structure for your project.

In order to form a group, use the appropriate tools in Canvas to create your group. If you have any trouble, let the instructor know.

You will submit your code as a pull request (PR) and set me (ssfoley) as a reviewer. In order to do this, be sure to follow our usual workflow for the class:

- **Fork** the template code.
- Give me **write permissions** on the repo. If you are working with a partner, give your teammate write access now, too.

- Create a **branch** called `dev` where you will do your work.
- Clone your dev branch and do the work.
- When you are ready to turn it in, create a **pull request from dev to master** on your fork. Please **make me a reviewer on the PR** so I can easily find your work.

Evaluation of your shell

In order to grade your shell, you will sign up for a demo timeslot to demo your code for the instructor. Both partners must attend the demo. The demo will consist of brief discussion of what you did (or didn't) get working (this should also be documented in the README), followed by the execution of some tests on the class server (via the instructor's machine), and finally, inspection of the code itself. You may bring written notes to the demo to help you remember any particular items about the state of the code and your approach. The code that was turned in to Canvas will be available on the server for grading during the demo, you do not need to bring your own code or laptop.

Deliverables

◯ Code written in a **consistent style** according to the **Style Requirements** handout.
◯ Code should be written with a defensive programming mindset and efficient algorithms should be utilized.
◯ Properly parses the jobs into the binary and argument set
◯ Supports interactive mode
  ◯ Continue processing until `exit` command or end-of-file marker (CTRL-D)
  ◯ Display shell job summary before exiting
◯ Supports batch mode
  ◯ Does not display the prompt
  ◯ Able to process multiple batch files
  ◯ Processes each batch file specified in the order they appear
◯ Supports multiple jobs specified on a single command line
◯ Supports multiple jobs specified on a single command line with different job separators
  (For example, a mix of `;` and `&` separators between jobs).
◯ Supports sequential jobs (with and without the explicit [`;`])
◯ Supports background jobs (with the [`&`])
◯ Builtin commands and executables identified properly and correct job totals displayed.
◯ Builtin command `exit` waits for all background jobs to finish before terminating the program
◯ Builtin command `wait` waits for all background jobs to finish
◯ Builtin command `fg` waits for specified or default background job to finish.
◯ If waiting for background jobs at exit, then a message is printed indicating the number of jobs being waited on.
◯ Builtin command `history` identified properly and works.
◯ Builtin command `jobs` properly displays the state of background jobs
◯ Binaries are executed with the correct arguments
◯ Large numbers of jobs supported
  (Tip: Do not hardcode any limit on the number of jobs your shell can process)
◯ Redirection of standard input and standard output identified properly and works.

Project 3: UNIX Shell – (50 Points)

In this assignment, you will implement a command line interpreter (CLI) or, as it is more commonly known, a shell. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Unix. If you don't know what shell you are running, it's probably bash. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command exit, at which point it exits. That's it!

For reading lines of input, you should use `getline()`. This allows you to obtain arbitrarily long input lines with ease. Generally, the shell will be run in interactive mode, where the user types a command (one at a time) and the shell acts on it. However, your shell will also support batch mode, in which the shell is given an input file of commands; in this case, the shell should not read user input (from `stdin`) but rather from this file to get the commands to execute.

In either mode, if you hit the end-of-file marker (EOF), you should call `exit(0)` and exit gracefully.

To parse the input line into constituent pieces, you might want to use `strtok()` (or, if doing nested tokenization, use `strtok_r()`). Read the man page (carefully) for more details.

To execute commands, look into `fork()`, `execvp()`, and `waitpid()`. See the man pages for these functions, and also read the relevant book chapter and example program for a brief overview.

You will note that there are a variety of commands in the exec family; for this project, you must use `execvp`. You should not use the `system()` library function call to run a command. Remember that if `execvp()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified.

2.1 Interactive Mode

In interactive mode, your shell will display the prompt `mysh$ `. The users of the shell will type commands after the prompt that will be interpreted by your shell. Unless explicitly indicated by the job separator, the jobs contained in the command are executed sequentially and in the order presented. The prompt is returned when all of the jobs are completed. An example usage of the interactive shell would be (input is Highlighted):

```
[grace@CS441 ~]$ ./mysh
mysh$ date
Wed Sept 17 10:00:00 CDT 2017
mysh$ pwd
/home/grace/project2/part2
mysh$ ls tests
file1.txt  file.txt
```

2.2 Batch Mode

In batch mode, the shell is started by specifying **one or more batch files** to the shell on the command line. The batch file contains a list of commands (one per line) that should be executed. In batch mode, you **must not** display a prompt.

In the case of multiple batch files, the files should be executed sequentially in the order they are presented on the command line. If there are background jobs in multiple batch files (say the first and second) then they may be running concurrently. **The shell will need to wait** on all of those background processes to finish before exiting the shell (once all batch files have finished being processed). The job totals displayed when the shell is exiting will represent all of the jobs started by all of the batch files. See the Section 2.4 for more details.

Most batch files will not contain the `exit` command, but will terminate by reaching the end-of-file. If the shell encounters the `exit` command in a batch file it must skip the rest of that batch file, and also skip the remaining batch files that have not yet been processed. The `exit` command will start to shutdown the shell waiting for any outstanding background processes to complete before displaying the shell job statistics and terminating. See the Section 2.5.5 for more details.

An example of the batch shell interaction is below:

```
[grace@CS441 ~]$ cat tests/file1.txt
date
pwd
ls tests
[grace@CS441 ~]$ ./mysh tests/file1.txt
Wed Sept 17 10:00:00 CDT 2017
/home/grace/project2/part2
file1.txt  file.txt
------------------------------
Total number of jobs               = 3
Total number of jobs in history    = 3
Total number of jobs in background = 0
[grace@CS441 ~]$ ./mysh tests/file1.txt tests/file1.txt
Wed Sept 17 10:00:00 CDT 2017
/home/grace/project2/part2
file1.txt  file.txt
Wed Sept 17 10:00:00 CDT 2017
/home/grace/project2/part2
file1.txt  file.txt
------------------------------
Total number of jobs               = 6
Total number of jobs in history    = 6
Total number of jobs in background = 0
[grace@CS441 ~]$
```

2.3 Sequential Execution of Multiple Jobs

In both interactive and batch mode, more than one job can be specified on a single command line. To separate jobs on a command line a semi-colon [;] must be used at the end of the job. Each job is executed in sequential order, and the prompt is only shown when all of the jobs have completed in the interactive shell (no prompt is shown when in batch mode). For example:

```
mysh$ date
Wed Sept 17 10:00:00 CDT 2017
mysh$ date ;
Wed Sept 17 10:00:00 CDT 2017
mysh$ date ; pwd ; ls tests
Wed Sept 17 10:00:00 CDT 2017
/home/grace/project2/part2
file1.txt  file.txt
mysh$
```

2.4 Concurrent Execution of Multiple Jobs

In both interactive and batch mode, a job can be run "in the background." This means that a job can be processing while the user is entering more commands or other jobs are running. Background processing

can also be thought of as concurrent processing since more than one process is active at the same time.

Jobs that wish to be run in the background are followed by an ampersand [`&`] instead of a semicolon [`;`]. **Any executable** can be run in the background, not just those represented in the examples. A user can check on the state of background jobs using the `jobs` built-in command. The `jobs` command will display the state of any jobs running the background. Backgrounded jobs can be in one of two states visible to the user: *Running* or *Done*. When there are no jobs running in the background then nothing is printed. Backgrounded jobs that have completed will be marked as *Done* and can be cleaned up only after the user calls the `jobs` command or at `exit`.

As an example, the following sequential command executes in 12 seconds, and returns a prompt when all three jobs have finished:

```
mysh$ sleep 4 ; sleep 5 ; sleep 3
mysh$
```

If we execute these commands in the background then they will all finish one second after each other taking about 5 seconds to complete all three jobs. The prompt is returned immediately to the user. The `jobs` command displays the job number, job state, and full command (**with the argument set**) for each background job. Once a *Done* job has been displayed with the `jobs` command then it may be removed from future listings.

```
mysh$ jobs
mysh$
mysh$ sleep 4 & /bin/sleep 5 & /bin/sleep 3 & /bin/date &
Wed Sept 17 10:00:00 CDT 2017
mysh$ jobs
[1]   Running    sleep 4
[2]   Running    /bin/sleep 5
[3]   Running    /bin/sleep 3
[4]   Done       /bin/date
mysh$ jobs
[1]   Done       sleep 4
[2]   Running    /bin/sleep 5
[3]   Done       /bin/sleep 3
mysh$ jobs
[2]   Done       /bin/sleep 5
mysh$ jobs
mysh$
```

2.5 Builtin Commands

Our shell will support the following builtin commands: `jobs`, `history`, `wait`, `fg`, `exit`. **Any other command that the user might type into our shell should be interpreted as an executable (a.k.a., binary) to launch (via** `execvp`**).**

WARNING!!! Do not try to implement your own `ls` or `sleep` or `echo` commands! Those are programs provided by the UNIX environment. Additionally, **do not** limit your shell to only accepting the binaries listed in the examples. The user may supply any binary name.

The builtin commands cannot be run in the background in our shell. The builtin commands may appear in a sequence of jobs as in the following example:

```
mysh$ sleep 4 & jobs ; sleep 3
```

2.5.1 Jobs

In both modes of execution, your shell will support the ability to display a list of the jobs that are currently in the background. For more details on how this should be displayed see section 2.4 and the examples throughout this document.

2.5.2 History

In both modes of execution, your shell will support the ability to display the full history of job commands typed into the shell. The `history` command displays all of the jobs executed by the shell (including the erroneous ones) from the earliest command to the latest command. For each job, it will display the job number and the full command (**with the argument set**). If the job was a background job then the '`&`' symbol is displayed after the argument set. The history display will include the builtin commands.

2.5.3 Wait

In both modes of execution, your shell will support the ability to wait for **all** currently backgrounded jobs to complete. If there are no backgrounded jobs, then the command returns immediately. This command requires that you properly manage your processes and keep track of your pids.

2.5.4 Foreground (fg)

In both modes of execution, your shell will support the ability to wait for a specific currently backgrounded job. If there are no backgrounded jobs, then the command returns an error. `fg` takes zero or one arguments. If there are zero arguments, then the latest backgrounded process is brought into the foreground and will not return until the job has completed. If there is an argument, it will be the job id of the job to bring into the foreground, as listed by the `jobs` command. If the job has already completed, an error message telling the user that the job has completed is produced. The `jobs` command output should treat this as the user knowing about this particular job completing, and should not display it in subsequent `jobs` output.

2.5.5 Exiting the Shell

In both interactive and batch mode, your shell terminates when it sees the `exit` command or reaches the end of the input stream (i.e., the end of the batch file or the user types 'Ctrl-D').

If a background process is still running when the shell exits then the shell should **wait** for that process to complete before exiting. In this case, your shell should print a message indicating the number of jobs the shell is waiting on when given the `exit` command.

Before the shell exits, the shell should print a count of the total number of jobs that were executed (excluding builtin commands), the total number of jobs in the history (including builtin commands), and the total number of jobs that were executed in the background.

```
mysh$ /bin/date
Wed Sept 17 10:00:00 CDT 2017
mysh$ /bin/pwd ; /bin/ls tests
/home/grace/project2/part2
file1.txt  file.txt
mysh$
mysh$ /bin/sleep 5 & sleep 4 & sleep 3 &
mysh$ jobs
[4]   Running    /bin/sleep 5
[5]   Running    sleep 4
[6]   Running    sleep 3
mysh$ jobs
[4]   Done       /bin/sleep 5
[5]   Done       sleep 4
```

```
[6]   Done       sleep 3
mysh$ jobs
mysh$ history
   1   /bin/date
   2   /bin/pwd
   3   /bin/ls tests
   4   /bin/sleep 5 &
   5   sleep 4 &
   6   sleep 3 &
   7   jobs
   8   jobs
   9   jobs
  10   history
mysh$ exit
------------------------------
Total number of jobs               = 6
Total number of jobs in history    = 11
Total number of jobs in background = 3
```

2.6 Redirection

The shell will support file redirection via < and >. < redirects the specified file as `stdin`; > redirects `stdout` to the specified file. For example, if you want to redirect the output of a program that normally prints to standard output, you can do the following:

```
mysh$ ./hello
Hello, World!
mysh$ ./hello > outfile
mysh$ cat outfile
Hello, World!
```

If you want to redirect a file as standard input to a program, you can do the following:

```
mysh$ ./hello_name
Hello, World!
What is your name?
Sam
Hello, Sam!
mysh$ cat infile
Sam
mysh$ ./hello_name < infile
Hello, World!
What is your name?
Hello, Sam!
```

You will want to use some or all of the system calls: dup2(), dup(), open(), close(). Note that when a process is forked, it inherits the file descriptors from its parent. These pieces of the process are stored in the reserved section of the process, and upon exec, the text and data sections are overwritten, but not the reserved section. Carefully read the documentation for these system calls and write some test programs to make sure you understand how to manage these file descriptors correctly.

If you are using the template code, you may want to modify the `job_t` data structure to contain information about whether redirection is used, and the files that are redirected.

Be sure to always check return codes, and use system defined constants (e.g., STDIN_FILENO) for managing well-known file descriptors. There should not be any extra open file descriptors, so be careful to close any file descriptors if you are not using them.

I highly recommend writing a little program that will print out the state of the various file descriptors and what files a process happens to have open to thoroughly test the functionality of your redirection.

3 Miscellaneous

3.1 Testing

You will need to perform your own testing of the software and document that in the `README.md`. Additionally, I have provided some tests that you can run your program against. To do so run the `make check` command. These will run your shell with some input, **but it will not check the output. You will need to do so manually** to make sure it meets the project specification. If they do not then look at the expected output and review your code (watch for formatting and spelling errors). This command will run all of the tests (and there are many) so make sure to scroll back and look for any tests that did not pass.

```
[grace@CS441 part2]$ make check
```

You are expected to create a set of **at least** five (5) input files for your shell each testing various aspects of the project. These tests should test all of the functionality described in this document, including file redirection and the builtin commands `fg` and `wait`. You may include helper programs if needed. These are in addition to and distinct from any test input files provided by the instructor. These files must be placed in a **tests** directory under the project directory.

These tests must be distinct from the tests provided in the `given-tests` directory. Do not add to or modify the contents of the `given-tests` directory as you changes **will not** be preserved in grading.

3.2 Useful Documentation & Hints

When in doubt about how a shell should behave consult how `bash` or `tcsh` handle the situation and model your shell after that.

For the following situations, you should print a message to the user (`stderr`) and continue processing the next command and/or file:

- A command does not exist or cannot be executed.
- A batch file does not exist or cannot be opened.

Optionally, to make coding your shell easier, you *may* print an error message and continue processing in the following situation:

- A very long command line (for this project, over 1024 characters).

Your shell should also be able to handle the following scenarios, which are *not* errors:

- An empty command line (which is ignored and a new prompt displayed).
- Multiple white spaces on a command line (also ignored).
- White space (or lack thereof) before or after the [`;`] or [`&`] characters
- Batch file ends without exit command or user types 'Ctrl-D' as command in interactive mode.

Your program must be able to handle an unlimited number of jobs. Your program should allow jobs to have a nearly unlimited number of arguments (up to the limit of characters allowed by the input).