# Shell

- **Due** Oct 3 by 11:59pm

- **Points** 100

This project is to be performed alone.

## Objectives

There are three objectives to this assignment:

1. To gain yet more familiarity with programming in C and in Linux
2. To learn how processes are handled (i.e., starting and waiting for their termination).
3. To gain exposure to some of the functionality in shells
4. To understand how redirection of stdout works

## Overview

In this assignment, you will implement a **command line interpreter**, or **shell,** on top of Linux. 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 much simpler, than the one you run every day in Unix. You can find out which shell you are running by typing **echo $SHELL** at a prompt. You may then wish to look at the man pages for **sh** or the shell you are running to learn more about all of the functionality that can be present. For this project, you do not need to implement as much functionality as is in most shells. You will need to be able to handle running multiple commands simultaneously.

Your shell can be run in two modes: **interactive** and **batch**. In interactive mode, you will display a prompt (the string "mysh> ", note the space AFTER the the ">" character) to **stdout**) and the user of the shell will type in a command at the prompt. In batch mode, your shell is started by specifying a

batch file on its command line; the batch file contains the list of commands (each on its own line) that should be executed. In batch mode, you should **not** display a prompt. In batch mode you should echo each line you read from the batch file back to the user (stdout) before executing it; this will help you when you debug your shells (and us when we test your programs). In both interactive and batch mode, your shell terminates when it sees the **exit** command on a line or reaches the end of the input stream (i.e., the end of the batch file or the user types 'Ctrl-D').

Jobs may be executed in either the **foreground** or the **background**. When a job is run in the foreground, your shell waits until that job completes before it proceeds and displays the next prompt. When a job is run in the background (as denoted with the '&' character as the last non-whitespace character on the line), your shell starts the job, but then immediately returns and displays the next prompt with the background job still running.

Each job that is executed by your shell should be given its own **unique job id** (jid). Your shell must assign each new command (whether it completes successfully or not) the next integer, starting with jid 0. Empty lines and built-in shell commands (such as 'jobs' and 'wait' described below) should not advance the jid count.

For example, given this sequence of input:

```
  mysh> /bin/ls mysh> mysh> /bin/ls   -l mysh>   output & mysh> nonexis
tentjob  mysh> output    -o 10 & mysh>   output -o 20   &
```

The jobs are given jids as follows:

| | |
|---|---|
| /bin/ls | 0 |
| /bin/ls -l | 1 |
| output & | 2 |
| nonexistentjob | 3 |
| output -o 10 & | 4 |
| output -o 20 & | 5 |

Users are able to perform very limited job control with your shell. First, users are able to find out which of their jobs are still running by using the shell built-in command 'jobs' . When your shell sees the 'jobs' command, it is to print to **standard output** the jid of each of the currently running jobs (i.e., the background jobs that have not yet finished) followed by a colon, :, and the job name and its arguments (without the '&'). Use write() for this. For your shell to

find out which jobs are still running, you may find the Unix system call waitpid useful; more details are given below. Be careful that your shell returns the current information about which jobs are really running and doesn't simply report the background jobs that the user hasn't explicitly waited for.

Given the previous sequence of input commands and assuming that all of the background jobs are still running, then, when the user types jobs, then your shell should write:

2 : output
4 : output -o 10
5 : output -o 20

Note for us to correctly test your code, **your output must match this format exactly! Remove all extra whitespace that may have appeared around command names and/or arguments.**

Second, users are able to tell the shell to wait for a particular background job to terminate by using the built-in command 'wait'. When your shell sees the 'wait' command along with the jid of a job, it waits for the specified job to complete before accepting more input. When the job completes, your shell should write the message, "JID <jid> terminated" to STDOUT; if in interactive mode, your shell should then display your prompt on the next line.

Continuing our example from above, if the user types the command 'wait 4', then your shell waits for jid 4 to terminate and then prints:

JID 4 terminated
mysh>

If the 'wait' command is used with the jid of a background job that has already completed, then your shell should immediately return and write the same message as above. Note that this case can occur even when the user was just informed that a given job is executing, if the job terminates before the user enters the 'wait' command. In other words, there is a race condition between these two events.

Continuing from our example, if Job 2 already finished and the user types the command 'wait '2, then your shell returns immediately and prints:

JID 2 terminated
mysh>

Finally, if the 'wait' command is used with an invalid jid (either one corresponding to a foreground job or a jid that has not been yet assigned) then your shell should immediately return and print the message, "Invalid JID <jid>".

In our example, if the user types the command 'wait 20', then your shell returns immediately and prints (to STDERR):

Invalid JID 20
mysh>

Note that **exit**, **jobs**, and **wait** are all built-in shell commands. They are not to be executed and cannot be placed in the background (you should just ignore the '&' if it is specified). These commands are **not** assigned jids.

## Redirection

Many times, a shell user prefers to send the output of a program to a file rather than to the screen. Usually, a shell provides this nice feature with the `>` character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature, but with a slight twist (explained below).

For example, if a user types `ls -la /tmp > output`, nothing should be printed on the screen. Instead, the standard output of the `ls` program should be rerouted to the file `output`. In addition, the standard error output of the program should be rerouted to the file `output` (note that this part of the specification is a little different than how redirection is specified in most shells).

If the `output` file exists before you run your program, you should simply overwrite it (after truncating it, which sets the file's size to zero bytes).

The exact format of redirection is: a command (along with its arguments, if present) followed by the redirection symbol, followed by a filename. Multiple redirection operators or multiple files to the right of the redirection sign are errors.

Do not worry about redirection for built-in commands (e.g., we will not test what happens when you type "jobs > file").

## Program Specifications

Your C program must be invoked exactly as follows:

```
mysh [batchFile]
```

The command line arguments to your shell are to be interpreted as follows.

- batchFile: an **optional** argument. If present, your shell will read each line of the batchFile for commands to be executed. If not present, your shell

will run in interactive mode by printing a prompt to the user at stdout and reading the command from stdin.

For example, if you run your program as

```
mysh file1.txt
```

then your program will read commands from **file1.txt** until it sees the **exit** command.

Defensive programming is an important concept in operating systems: an OS can't simply fail when it encounters an error; it must check all parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinately, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print an understandable error message and either continue processing or exit, depending upon the situation.

You should consider the following situations as errors; in each case, your shell should print a message using write to **STDERR_FILENO** and exit gracefully with a return code of 1:

- An incorrect number of command line arguments to your shell program. Print exactly `Usage: mysh [batchFile]` (with no extra spaces)
- The batch file does not exist or cannot be opened. Print exacty `Error: Cannot open file foo`(assuming the file was named `foo`).

For the following situation, you should print a message (using write()) to the user (STDERR_FILENO) and **continue** processing:

- A command does not exist or cannot be executed. Print exactly `job: Command not found` (assuming the command was named `job`).

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 512 characters).

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

- An empty command line.
- Multiple white spaces on a command line.
- White space before or after the '&' character.
- Batch file ends without **exit** command or user types 'Ctrl-D' as command in interactive mode.

- Additional flags or arguments appear along with built-in command (e.g., `jobs hello`); do not treat as a built-in command and create a new process with the specified arguments
- The '&' character appears after a built-in command (e.g., `exit &`); ignore the '&' character and execute normally.
- If the '&' character appears in the middle of a line, then the job should not be placed in the background; instead, the '&' character is treated as one of the job arguments.

All of these requirements will be tested!

For simplicity, you can assume that the number of background jobs that will ever be running simultaneously will be 32 or fewer. This assumption may help you more easily allocate data structures in your shell. Of course, there may be many more than 32 background jobs that are started over the lifetime of your shell (the number of background jobs run within your shell is unlimited) .

# C Hints

This project is not as hard as it may seem at first reading; in fact, the code you write will be much, much smaller than this specification. Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. Your finished programs will probably be under 200 lines, including comments. If you find that you are writing a lot of code, it probably means that you are doing something wrong and should take a break from hacking and instead think about what you are trying to do.

Your shell is basically a loop: it repeatedly prints a prompt (if in interactive mode), parses the input, executes the command specified on that line of input, and waits for the command to finish, if it is in the foreground. This is repeated until the user types "exit" or ends their input.

You should structure your shell such that it creates a new process for each new command. There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows easy concurrency; that is, multiple commands can be started and allowed to execute simultaneously.

For each running job, you will want to track some information in a data structure. It is up to you to determine what information you need to keep and the list structure you want to use (e.g., an array or a linked list). This information will allow you to wait for the appropriate job to complete, as requested by the user.

To simplify things for you in this first assignment, we will suggest a few library routines you may want to use to make your coding easier. (Do not expect this detailed of advice for future assignments!) You are free to use these routines if you want or to disregard our suggestions.

To find information on these library routines, look at the manual pages (using the Unix command **man**). You will also find man pages useful for seeing which header files you should include.

Make sure you use the write() system call for all printing (including prompts, error messages, and job status), whether to stdout or to stderr. Why should you use write() instead of fprintf or printf? The main difference between the two is that write() performs its output immediately whereas fprintf() buffers the output temporarily in memory before flushing it. As a result, if you use fprintf() you will probably see output from your shell intermingled in unexpected ways with output from the jobs you fork(); you will fail our tests if your output is intermingled. If you decide to use fprintf() make sure you ALWAYS call fflush() immediately after the call to fprintf().

# Parsing

For reading lines of input, you may want to look at **fgets()**. To open a file and get a handle with type **FILE ***, look into **fopen()**. Be sure to check the return code of these routines for errors! (If you see an error, the routine **perror()** is useful for displaying the problem.) You may find the **strtok()** routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by ' ').

# Executing Commands

Look into fork, execvp, and waitpid.

The fork system call creates a new process. After this point, two processes will be executing within your code. You will be able to differentiate the child from the parent by looking at the return value of fork; the child sees a 0, the parent sees the pid of the child. Note that you will need to map between your jid and this pid.

You will note that there are a variety of commands in the exec family; **for this project, you must use execvp**. 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. The first argument specifies the program that should be executed, with the full path specified; this is straight-forward. The second argument, char *argv[] matches those that the program sees in its function prototype:

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

Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

/bin/foo 205 535

then argv[0] = "/bin/foo", argv[1] = "205" and argv[2] = "535" (where is string is NULL terminated). Note the list of arguments must also be terminated with a NULL pointer; that is, argv[3] = NULL. We strongly recommend that you carefully check that you are constructing this array correctly!

The waitpid system call allows the parent process to wait for one of its children. Note that it returns the pid of the completed child; again, you will need to map between your jid and this pid. You will want to investigate the different options that can be passed to waitpid so that your shell can query the OS about which jobs are still running without having to wait for a job to terminate.

## Miscellaneous Hints

Remember to get the **basic functionality** of your shell working before worrying about all of the error conditions and end cases. For example, first focus on interactive mode, and get a single command running in the foreground working (probably first command with no arguments, such as "ls"). Then, add in the functionality to work in batch mode (most of our test cases will use batch mode, so make sure this works!). Handling file redirection is probably your next step. Next, handle starting up jobs in the background; waiting for them and then listing their status should be next. Finally, make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands.

We strongly recommend that you check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls.

# Grading

To ensure that we compile your C correctly for the demo, you will need to create a simple **Makefile**; this way our scripts can just run make to compile your code with the right libraries and flags. If you don't know how to write a makefile, you might want to look at the man pages for make. Otherwise, you can start with the sample Makefile available here:

```
~cs537-1/projects/shell/Makefile
```

The name of your final executable should be `mysh` , i.e. your C program must be invoked exactly as follows:

```
% ./mysh

% ./mysh inputTestFile
```

Copy all of your .c (and .h) source files into the appropriate handin directory. Do **not** submit any .o files. Make sure that your code runs correctly on the linux machines in the 13XX labs.

The majority of your grade for this assignment will depend upon how well your implementation works. We will run your program on a suite of about 20 test cases. Be sure that you thoroughly exercise your program's capabilities on a wide range of test suites, so that you will not be unpleasantly surprised when we run our tests.

For testing your code, you will probably want to run commands that take awhile to complete. Try compiling and running the very simple C program specified below; when multiple copies are run in the background you should see the output from each process interleaved. See the code for more details.

```
~cs537-1/projects/shell/output.c
```

We will again verify that your code passes lint and valgrind tests, as in Project P1.