CS 367 Project #3 - Fall 2019:
Shell

Due: December 4th, 2019 11:59pm

This is to be an individual effort. No partners.
No late work allowed after 48 hours; each day late automatically uses up one of your tokens.

1. Introduction

For this assignment, you are going to use C to implement a simple shell program called GMSH (GMU Shell). Once running, GMSH would be able to accept/execute commands from user and perform basic job management. This assignment will help you to get familiar with the principles of process management and job control in a Unix-like operating system. Our lectures on processes, signals, and Unix-IO as well as Textbook Ch8 (in particular 8.4 and 8.5) and 10.3 will provide good references to this project.

2. Project Overview

A typical shell program receives line-by-line command input by the user from a terminal. The shell would support a set of built-in commands which will be executed directly by the shell. If the user input is not a built-in command, however, the shell will interpret the input as the name of a program to be executed, along with arguments to be passed to it. In that case, the shell will fork a new child process and execute the program in the context of the child.

A shell program typically also provides job control. Normally, a user command (built-in or not) is executed as a foreground job, which means the shell will wait for the command to complete before reading the next command. But if the user command ends with an ampersand '&', the command will be started in the background and the shell will return to the prompt immediately to accept the next command. Some built-in commands are usually provided by the shell for the user to view the list of background jobs or to switch a job between background and foreground.

For this assignment, your shell implementation should be able to perform the following:

- Accept a single line of command from user;
- Execute a built-in command (detailed list of supported commands below);
- Load and run the user specified program with the provided arguments;
- Perform basic job control;
- Support basic file redirection.

We will describe each of them in more details with some examples below.

2.0 Use of the Logging Functions

In order to keep the output format consistent, you must call provided logging functions at the appropriate times to generate the right output from your shell program. **The generated output will also be used for grading.** The files `logging.c` and `logging.h` provide the functions for you to call. Most of the log functions require you to provide a process ID (`pid`) and the relevant

command line (`cmd`) to make the call. We will explain more details and specify how to use them below.

2.1 Parsing User Commands

Once start, the shell would print a prompt and wait for the user to input commands. Each line from the user is considered as a command. **Logging Requirements:**

- You must call `log_prompt()` to print out the prompt.

Each command must follow these (simplified) rules:

- Every item in the line must be separated by one or more spaces;
- It must start with either a built-in command or a name of the program to be executed with the full path of that program;
- Optionally, it could include any arguments that the user wants to supply;
- Optionally, it might specify file redirection options but at most one file for input and at most one file for output;
- Optionally, it might end with "&" to indicate the command should start a background job.

Your shell program needs to be able to parse the input line, identify the command and arguments, check whether any file needs to be prepared, and determine whether this job is a background job. **Assumptions:** You can assume that all user inputs are valid command lines (no need for format checking in your program). You can also assume that the maximum number of characters per command line is `100` and the maximum number of arguments per line is `50`.

Remember, **fgets** will keep the '\n' character in the buffer after reading from the user. If you do not take care of this, you will get command errors with exec. *(eg. "/usr/bin/ls\n" is not the same as "/usr/bin/ls" when the system tries to find the command).*

When parsing the user input, make use of **strtok** to handle the tokenizing of the inputs for you. All parts of the user input will be separated by a space delimiter. As you are tokenizing the input, think about the order of items that may be present. Your parser will need to handle the options for built-in commands, non-builtin commands with and without arguments, and each type of redirection. Remember, a command may use both redirection in and out in the same command!

Examples:

Command Line	Program/ command	Args	File Redirection	Background/ Foreground
`help`	`help`	`N/A`	`N/A`	`Foreground`
`/bin/sleep 200`	`/bin/sleep`	`200`	`N/A`	`Foreground`
`/bin/sleep 200 &`	`/bin/sleep`	`200`	`N/A`	`Background`
`/bin/ls -l .. &`	`/bin/ls`	`-l ..`	`N/A`	`Background`
`/bin/ls -l > ls.txt &`	`/bin/ls`	`-l`	`ls.txt (for input)`	`Background`
`/bin/wc < ls.txt > wc.txt`	`/bin/wc`	`N/A`	`ls.txt (for input)` `wc.txt (for output)`	`Foreground`

2.2 Non-Built-in Commands

If the user command is not a shell built-in command, it must be interpreted as a program to be loaded in and executed by the shell on behalf of the user. It can be either a foreground or background job. When your shell starts a foreground job, it must wait for the job to complete before showing the prompt to user and accepting the next command. When a background job gets started, however, the shell does not need to wait and can immediately accept the next command.

Assumptions: Your shell does NOT need to support a program that need exclusive access to terminal (e.g. vi) or that reads from terminal (e.g. wc without a file input).

- Non-Built-In Commands that will be used in testing (with and without arguments, with and without redirection): `/bin/sleep, /bin/ls, /bin/pwd`
- Non-Built-In Commands that will be used in testing (only with redirection in from a file!): `/bin/wc` (eg. `/bin/wc < file.txt`)

Logging Requirements:

- For background jobs, you must call `log_start_bg(pid, cmd)` to report the start of the job.
- If the program cannot be loaded and executed, you must call `log_command_error(cmd)` to report the error.

Implementation Hints:

- You can use `fork()` to create a new child process;
- You can use either `execl()` or `execv()` to load a program and execute it in a process.

Even though `execl` or `execv` do not normally return, they **will** return with a `-1` value if there are any errors, such as the path or command not being found. Use the man pages for the command you wish to use to see what it will output on errors and think about how to handle them.

Example Runs:

```
GMSH>> /bin/ls ./test -l
total 8
-rw-r--r--. 1 yzhong itefacstaff   32 Nov 17 23:36 temp.txt
-rw-r--r--. 1 yzhong itefacstaff 2512 Nov 17 23:37 trivial.txt
Foreground Job 16952:/bin/ls ./test -l Terminated Normally
GMSH>> /bin/ls ./test -l &
Background Job 17767:/bin/ls ./test -l & Started
GMSH>> total 8
-rw-r--r--. 1 yzhong itefacstaff   32 Nov 17 23:36 temp.txt
-rw-r--r--. 1 yzhong itefacstaff 2512 Nov 17 23:37 trivial.txt
Background Job 17767:/bin/ls ./test -l & Terminated Normally
```

2.3 Basic Built-in Shell Commands

Your shell program must support the following built-in commands:

- `help`: when called, your shell should print on terminal a short description of the shell including a list of built-in commands and their usage. **Logging Requirements:**
  - You must call `log_help()` to print out the predefined information.
- `quit`: when called, your shell should terminate. **Logging Requirements:**
  - You must call `log_quit()` to print out the predefined information.
  - You will then need to exit your shell program.

- **`fg/bg/jobs/kill`**: These are described in the Job Control section.

Example Runs:

```
GMSH>> help
Welcome to GMSH (GMU Shell)!
Built-in Commands: fg, bg, jobs, kill, quit, help.
       kill -SIGNAL PID
       fg [JOBID]
       bg JOBID
GMSH>> quit
Thanks for using GMSH! Good-bye!
zeus-2:~/
```

2.4 Job Control and Relevant Built-in Commands

Your shell might have multiple jobs running concurrently: 0 or 1 foreground job; 0 or more background jobs. Your shell needs to maintain the record for the foreground job (if there is any) and a ***list of background jobs*** that are not terminated. Simple job control tasks include:

- On start, every background job gets assigned a non-negative integer job ID: if there is no other non-terminated background job, it gets job ID 1; otherwise, it takes the next integer that is higher than all existing job IDs.
- On start, a foreground job gets a dummy job ID 0. If a foreground job is switched to background, it will be assigned a non-negative job ID, following the same rules as above.
- A job can be switched between background and foreground (see details below). Regardless of the switching, once assigned a job ID, a job keeps the same job ID until it is terminated.
- You will need to update your record if there is any status change (process stopped, terminated, continued, etc) or if there is any switching between foreground and background jobs.
- Some details need to be included in your record for each job, including the assigned job ID, the process ID, the execution status of the job, and the initial command line that starts the job. You can assume that the execution state is either "**`Running`**" or "**`Stopped`**",

Some additional built-in commands related to job control must be supported.

- **`jobs`**: When the user inputs **`jobs`** command, your shell should print on terminal the list of background jobs with the job ID, process ID, running state, and initial command line.
- **Logging Requirements:**
  - You must first call **`log_job_number(num_jobs)`** to report how many background jobs are currently alive (not terminated).
  - For each of the job, you must then call **`log_job_details(job_id, pid, state, cmd)`** to report the job ID, process ID, execution state (either "**`Running`**" or "**`Stopped`**"), and the original command line that triggered this job.
  - If there are multiple jobs, they should be printed in the ***ascending order of their job IDs***.
- **Implementation Hints:**
  - A **`SIGCHLD`** will be sent to parent process when there is a status change to a child process (stopped, terminated, continued). You can use **`waitpid()`** to specify which situations you want to monitor. You can also check the status of involved child

process using different macros (**`WIFEXITED, WIFSTOPPED, WIFSIGNALED, WIFCONTINUED`**, etc).

- You can use **`sigaction()`** to override the default signal handling and define what actions to take when a signal arrives. See the Appendix to get more details.

Example Runs:

```
GMSH>> /bin/sleep 100 &
Background Job 1168:/bin/sleep 100 & Started
GMSH>> /bin/sleep 10 &
Background Job 1184:/bin/sleep 10 & Started
GMSH>> /bin/sleep 200 &
Background Job 1189:/bin/sleep 200 & Started
GMSH>> jobs
=====3 Jobs=====
1: 1168: Running /bin/sleep 100 &
2: 1184: Running /bin/sleep 10 &
3: 1189: Running /bin/sleep 200 &
GMSH>> Background Job 1184:/bin/sleep 10 & Terminated Normally
GMSH>> jobs
=====2 Jobs=====
1: 1168: Running /bin/sleep 100 &
3: 1189: Running /bin/sleep 200 &
GMSH>> /bin/sleep 400 &
Background Job 1210:/bin/sleep 400 & Started
GMSH>> jobs
=====3 Jobs=====
1: 1168: Running /bin/sleep 100 &
3: 1189: Running /bin/sleep 200 &
4: 1210: Running /bin/sleep 400 &
```

- **`kill -SIGNAL PID`**: when called, your shell should send a signal with number **`SIGNAL`** to process with **`PID`**.
- **Logging Requirements:**
  - On error, you must call **`log_kill_error(pid, sig)`** to report the failed attempt.
- **Implementation Hints:**
  - You can use **`kill()`** to send signals to a particular process.

Example Runs:

```
GMSH>> jobs
=====1 jobs=====
1: 30388: Running /bin/sleep 200 &
GMSH>> kill -19 30388
GMSH>> Background Job 30388:/bin/sleep 200 & Stopped
GMSH>> jobs
=====1 jobs=====
1: 30388: Stopped /bin/sleep 200 &
```

- **`fg [JOBID]`**: when called, your shell should change the specified background job to be foreground and wait until it completes. If no **`JOBID`** is specified, switch the background job with the highest job ID to foreground. If the job has been previously stopped, resume its execution. No change should be made if provided **`JOBID`** is invalid.

- **Logging Requirements:**
  - You must call `log_job_fg(pid, cmd)` to report which background job has been switched to be a foreground job.
  - If there is no background jobs at all, you must call `log_no_bg_error()` to report the issue.
  - On failure of locating the specified job ID, you must call `log_fg_notfound_error(job_id)` to report the issue.
  - On failure of resuming the specified job, you must call `log_job_fg_fail(int pid, char *cmd)` to report the issue.
- **Implementation Hints:**
  - You can use `kill()` to send signals to a particular process.

Example Runs:

```
GMSH>> /bin/sleep 10 &
Background Job 9746:/bin/sleep 10 & Started
GMSH>> jobs
=====1 Jobs=====
1: 9746: Running /bin/sleep 10 &
GMSH>> fg
Background Job 9746:/bin/sleep 10 & Moved to Foreground
Foreground Job 9746:/bin/sleep 10 & Terminated Normally
GMSH>> GMSH>> jobs
=====0 jobs=====

GMSH>> jobs
=====2 Jobs=====
1: 10862: Running /bin/sleep 100 &
2: 11693: Running /bin/sleep 300 &
GMSH>> fg 1
Background Job 10862:/bin/sleep 100 & Moved to Foreground
Foreground Job 10862:/bin/sleep 100 & Terminated Normally
GMSH>> jobs
=====1 Jobs=====
2: 11693: Running /bin/sleep 300 &
```

- `bg JOBID`: when called, your shell would resume the execution of a background job with the specified `JOBID`. No change should be made if provided `JOBID` is invalid or if the specified job is already actively running.
- **Logging Requirements:**
  - You must call `log_job_bg(pid, cmd)` to report command `bg` has been applied to which background job.
  - On failure of locating the specified job ID, you must call `log_bg_notfound_error(job_id)` to report the issue.
  - On failure of resuming the specified job, you must call `log_job_bg_fail(int pid, char *cmd)` to report the issue.
- **Implementation Hints:**
  - You can use `kill()` to send signals to a particular process.

Example Runs:

```
GMSH>> jobs
=====1 Jobs=====
1: 20979: Stopped /bin/sleep 200 &
GMSH>> bg 1
Command bg applied to 20979:/bin/sleep 200 &
GMSH>> Background Job 20979:/bin/sleep 200 & Continued
GMSH>> jobs
=====1 Jobs=====
1: 20979: Running /bin/sleep 200 &
```

Assumption: You can assume that built-in commands are always used to specify a foreground job. You can also assume there is no file redirection for built-in shell commands.

2.5 Keyboard Interaction

A number of certain keyboard combinations can trigger signals to be sent to the group of foreground jobs. Note that by default, the signal triggered by those keyboard combinations will be sent to the whole foreground process group, which by default includes both the shell program and its foreground job. Your program must change that default behavior to make sure the signal only affects the foreground job, not the shell program. **Implementation Hints:**

- You can use `setpgid()` to change the group ID of a process. See the Appendix of this document for more details.
- You can use `sigaction()` to change the default response to a particular signal;
- Your shell program can use `kill()` to send/forward the received signal to another process.

For this assignment, your only need to support two keyboard combinations:

- `Ctrl-c`: A SIGINT(2) should be sent to the foreground job to terminate its execution.
- `Ctrl-z`: A SIGTSTP(20) should be sent to the foreground job to pause its execution and switch it to be a background job.

Example Runs:

```
GMSH>> /bin/sleep 200
^CForeground Job 1340:/bin/sleep 200 Terminated by Signal
GMSH>> jobs
=====0 jobs=====
GMSH>> /bin/sleep 200
^ZForeground Job 1438:/bin/sleep 200 Stopped
GMSH>> jobs
=====1 jobs=====
1: 1438: Stopped /bin/sleep 200
GMSH>>
```

2.6 File Redirection

If the user specifies a file to be used as the input and/or output of a program, your shell needs to redirect the standard input and/or standard output of that program (process) to the specified file(s). **Logging Requirements:**

- On failure of opening the file, you must call `log_file_open_error(file_name)` to report the issue. **Note:** If there is a file open error, ignore the file redirection but you should still execute the user command (with the normal standard input and/or standard output).

Implementation Hints:

- You can use `open()` to open files;
- If a file is created by `open()`, it takes a third argument to set the reading/writing/executing permission of that file. To simplify the task, make sure to use `0600` as the third argument of `open()` if needed. It will typically set the file to be readable and writable by the owner of the file only.
- You can use `dup2()` to change the standard input/output **if** the call to open succeeded;
    - If the open fails (eg. File not found), you can skip the dup2 to keep the default.

Example Runs:

```
GMSH>> /bin/ls ./test -l
total 12
-rw-r--r--. 1 yzhong itefacstaff 1209 Nov  1 19:17 driver.c
-rw-r--r--. 1 yzhong itefacstaff 5719 Nov  1 11:29 shell.c
Foreground Job 2750:/bin/ls ./test -l  Terminated Normally
GMSH>> /bin/ls ./test -l >ls.txt
Foreground Job 2929:/bin/ls ./test -l >ls.txt Terminated Normally
GMSH>> /bin/wc <ls.txt
  3  20 128
Foreground Job 3198:/bin/wc <ls.txt Terminated Normally
GMSH>> /bin/wc <ls.txt >wc.txt
Foreground Job 3728:/bin/wc <ls.txt >wc.txt Terminated Normally
GMSH>> /bin/wc <ls.txt >>wc.txt
```

./test is a local folder

```
total 12
-rw-r--r--. 1 yzhong itefacstaff 1209 Nov  1 19:17 driver.c
-rw-r--r--. 1 yzhong itefacstaff 5719 Nov  1 11:29 shell.c
```

ls.txt

```
  3  20 128
```

wc.txt after the first wc command

```
  3  20 128
  3  20 128
```

wc.txt after the second wc

2.7 Signal Handling

There are various signals that you might want to override the default handler and specify what actions to take. You will need to define signal handlers to help implementing the tasks discussed above.

Logging Requirements:

- For multiple possible status changes of a child process, you must call the corresponding logging function to report the change. All those logging functions require the process ID pid and the original command line cmd of the involved child process. They are summarized in the table below.

Foreground or Background Job	**Status Change**	**Logging Function to Call**
Foreground	Running->Terminated	`log_job_fg_term(pid, cmd)`
	Running/Stopped->Terminated (by a signal)	`log_job_fg_term_sig(pid, cmd)`
	Running->Stopped (also switch to background)	`log_job_fg_stopped(pid, cmd)`

Background	Running->Terminated	`log_job_bg_term(pid, cmd)`
	Running/Stopped->Terminated (by a signal)	`log_job_bg_term_sig(pid, cmd)`
	Running->Stopped	`log_job_bg_stopped(pid, cmd)`
	Stopped->Running	`log_job_bg_cont(pid, cmd)`

Implementation Hints:

- Signals that you need to monitor and handle include: `SIGCHLD`, `SIGINT`, and `SIGTSTP`;
- You can use `sigaction()` to register those handlers. **Note:** the textbook introduces the usage of `signal()`, which has been deprecated and replaced by `sigaction()`. Make sure you read the manual of `sigaction()` and learn how to use it first before coding. Check the Appendix of this document to get the basic idea and example usage of `sigaction()`.
- It's recommended not to use any stdio functions in a signal handler. If you need to print any debug statements to the, use `write()` with `STDOUT_FILENO` directly. You can check the provided `logging.c` for examples (e.g. `log_job_fg_term()`).

2.8 Race Conditions

You will start to experience the fun and challenges of concurrent programming in this assignment. In particular, if your design includes a global job list, be alert that race conditions might occur. The typical race is between the shell process updating the list when start or move jobs and signal handler updating the same list when a process changes its status (termination, continuation etc). For example, the following sequence is possible if no synchronization is provided:

1. The shell(parent) starts a new job (child process) and the newly created child gets to run;
2. Before the parent gets the chance to add the job into the list, the child process terminates and triggers a SIGCHLD to parent;
3. SIGCHLD handler is executed but does not see the job in list and cannot do anything;
4. After the handler completes, the shell (parent) resumes normal execution and adds the job into the list (when it is too late!).

One approach that we recommend is to block SIGCHLD signal (and other signals that might trigger the updates of the global list) before the call to `fork()` and unblock them only after the job has been added into the list. Both blocking and unblocking of signals can be implemented with `sigprocmask()`. Also notice that children inherit the blocked set of their parents, so you must unblock them in the child before calling execl() or execve(). There might be other similar situations that you need to temporarily block signals to ensure the updates to the global list are well synchronized.

3. Getting Started

First, get the starting code (`project3_handout.tar`) from the same place you got this document. Once you un-tar the handout on Zeus (using `tar xvf project3_handout.tar`), you will have the following files in the handout directory:

- `shell.c` – **This is the only file you will be modifying (and submitting).** There are stubs in this file for the functions that will be called by the rest of the framework. Feel free to

define more functions if you like but put all of your code in this file!
`shell.h` – This has some basic definitions and includes necessary header files.

- `logging.c` – This has a list of provided logging functions that you need to call at the appropriate times.
- `logging.h` – This has the prototypes of logging functions implemented in `logging.c`.

- `sigaction_example.c` – This is the example program of sigaction() from the Appendix.

- `Makefile` – to build the assignment (and clean up).

4. Submitting & Grading

Submit this assignment electronically on Blackboard. Note that the only file that gets submitted is **`shell.c. Make sure to put your name and G# as a commented line in the beginning of your program shell.c`**

You can make multiple submissions; but we will test and grade ONLY the latest version that you submit (with the corresponding late penalty, if applicable).

Important: Make sure to submit the correct version of your file on Blackboard! Submitting the correct version late will incur a late penalty; and submitting the correct version 48 hours after the due date will not bring any credit.

Questions about the specification should be directed to the CS 367 Piazza forum.

Your grade will be determined as follows:

- **20 points** - code & comments. Be sure to document your design clearly in your code comments. This score will be based on reading your source code.
- **80 points** – correctness. We will be building your code using the `shell.c` code you submit along with our code.
    - If you program does not compile, **we cannot grade it**.
    - If your program compiles but does not run, **we cannot grade it.**
    - We will give partial credit for incomplete programs that compile and run.
    - We will use a series of programs to test your solution.

Appendix: Process Groups

Every process belongs to exactly one process group.

- `pid_t getpgrp(void); // #include <unistd.h>`
- The *getpgrp*() function shall return the process group ID of the calling process.

When a parent process creates a child process, the child process inherits the same process group from the parent.

- `int setpgid(pid_t pid, pid_t pgid); // #include <unistd.h>`
- The *setpgid()* function shall set process group ID of the calling process. In particular, a process can call `setpgid(0,0)` to create a new process group with itself in the group.

References:

- http://man7.org/linux/man-pages/man3/getpgrp.3p.html
- http://man7.org/linux/man-pages/man3/setpgid.3p.html
- Textbook Section 8.5.2 (pp.759)

Appendix: System call sigaction()

The `sigaction()` system call is used to change the action taken by a process on receipt of a specific signal. The `sigaction()` function has the same basic effect as `signal()` but provides more powerful control. It also has a more reliable behavior across UNIX versions and is recommended to be used to replace `signal()`.

- *int* **sigaction** *(int signum, const struct sigaction *action, struct sigaction *old-action) // #include <signal.h>*
    - `signum` specifies the signal and can be any valid signal except SIGKILL and SIGSTOP
    - If `action` is non-NULL, the new action for signal signum is installed from `action`. It could be the name of the signal handler.
    - If *`old-action`* is non-NULL, the previous action is saved in *`old-action`*.

Example program:

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

void sigint_handler(int sig) {   /* signal handler for SIGINT */
  write(STDOUT_FILENO, "SIGINT\n", 7);
  exit(0);
}

int main(){
  struct sigaction new;   /* sigaction structures */
  struct sigaction old;

  new.sa_handler = sigint_handler;     /* set the handler */
  new.sa_flags = 0;
  sigaction(SIGINT, &new, &old);       /* register the handler for SIGINT */

  int i=0;
  while(i<100000){                     /* this will loop for a while */
        fprintf(stderr, "%d\n", i);    /* break loop by Ctrl-c to trigger SIGINT */
        sleep(1);   i++;
  }
  return 0;
}
```

References:

- https://www.gnu.org/software/libc/manual/html_node/Sigaction-Function-Example.html
- http://man7.org/linux/man-pages/man2/sigaction.2.html