CS 367 Spring 2021
Project #4: Shell
Due: Friday, April 23, 2021, 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 **MASH** (**MA**son **SH**ell). Once running, MASH 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 the user, then either
  - Execute a built-in command (detailed list of supported commands below); or
  - Load and run the user specified program with the provided arguments;
- Support any number of jobs simultaneously (not limited by your system);
- Perform basic job control;
- Support basic file redirection.

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

Specifications Document: (Chapter 3 at the end has some guidance on starting your design)
This document has of a breakdown of each of the features, looking at specific details, required logging, and sample outputs. This is an **open-ended** project that will require you to make your own design choices on how to approach a solution. **Read the whole document before starting.**

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. All logging output starts with a unique header that can help us to keep track of the activities of our shell. **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 Prompt, Accepting, and Parsing User Commands

Once started, the shell would print a prompt and wait for the user to input commands. Each line from the user is considered as one command.

Logging Requirements:

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

If a command line is empty, it should be ignored by the shell. Otherwise, you need to parse the user command line into useful pieces. We provide a `parse()` function in `parse.c / parse.h`. The provided template `shell.c` has included code to call `parse()`. Feel free to use the provided `parse()` as is or implement your own parsing facility. **Check Appendix A for a detailed description of the input, output, and examples of the provided `parse()`.**

In testing, make sure you use user commands following these 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;
- Optionally, it could include any arguments that the user wants to supply;
- Optionally, one or more command line arguments may match the "$?" exit code string.
- Optionally, it might specify file redirection options;
- Optionally, it might end with "&" to indicate the command should start a background job;

For this assignment, you can make the following **assumptions**:

- All user inputs are valid command lines (no need for format checking in your program).
- The maximum number of characters per command line is `100` and the maximum number of arguments per program is `25`. Check `shell.h` for relevant constants defined for you.
- A built-in command will not specify any file redirection;
- A built-in command will not end with "&";
- A command will not specify the path (for example, you may see "ls" but not "/usr/bin/ls");
- We will use at most one file for input and at most one file for output for file redirection;

After calling `parse()`, the provided `shell.c` leaves the design and implementation up to you as an **open-ended** project for you to solve. You are encouraged to write many helper functions as well and you may add additional code in to main as needed.

A valid user command might be either a built-in command or a program to load in and execute. You must give the built-in command a higher priority in execution. For example, one of our

supported built-in command is "`help`". If there is a program that happens to share the same name "`help`", it will be shadowed by the built-in command and will not be executed.

Previous Exit Code

It is possible that one or more of the command line arguments of any command, built-in or otherwise, may be the string "`$?`". Whenever this string appears as an argument, it should be replaced by the exit code of the most foreground process which terminated normally. For example, if the most recent foreground process output an exit code of 5, then "`ls $?`" would become "`ls 5`" prior to execution. The replacement must take place prior to the execution of the command.

Assumptions:

- If there have not been any foreground processes which have terminated normally yet, then a default value of 0 may be used for the previous exit code.
- The command name itself will not be "`$?`".
- You will not need to perform a replacement on a partial match such as "$?xyz".

Logging Requirements:

- You must call `log_replace(arg,exitcode)` to print out the predefined information, once for every "`$?`" arg which is replaced. The `arg` input says which argument was replaced, while `exitcode` is the exit code which was substituted in its place.
- The `log_replace(arg,exitcode)` requirement comes before but in conjunction with any other logging requirements associated processing the line (for example, `log_start`, etc.).
- If a different logging requirement uses the command line, `cmd`, as input, it will use the original command line prior to replacement.

Implementation Hints:

- After `waitpid(pid_t pid, int *status, int options)`, you can use `WIFEXITED(status)` to check whether a child got terminated normally.
- Furthermore, if `WIFEXITED` returns true, you can use `WEXITSTATUS(status)` to extract the exit status of the child.

Example Run:

```
MASH>> my_echo 20
[MALOG]Foreground Process 244082: my_echo 20 Started
my answer is 20
[MALOG]Foreground Process 244082: my_echo 20 Terminated Normally
MASH>> echo the number is $?
[MALOG]Argument replacement performed on argument 4 using exit code 20
[MALOG]Foreground Process 244083: echo the number is $? Started
the number is 20
[MALOG]Foreground Process 244083: echo the number is $? Terminated Normally
MASH>>
```

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. The shell program must fork a process to take the job for both cases. 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.

Paths

When you execute these commands using execv or execl, you will also need to know the full path of the command to satisfy its first argument. To generate this, you will need to check two different paths for each command. These are: "`./`" and "`/usr/bin/`". Both of these paths must be checked, in this order, for an entered command. A command as entered on the command line will not have any path to begin with.

For example, if the user enters the command "`ls -al`", you will try both "`./`" and "`/usr/bin/`" as the path argument to `execv` or `execl`, **in that order**. Check the error code on `execv` or `execl` to see if the path was not found before checking the next one. If neither path option leads to a valid program, then you handle it as a path error and issue the appropriate log function.

argv[0]

Since the path argument of execv or execl needs to be modified from the original command by concatentating in "`./`" or "`/usr/bin/`", you will simply keep the original command name as **argv[0]**. So, if the user inputs "`ls -al`", then your path may be either "`./ls`" or "`/usr/bin/ls`" depending on which one works, but your `argv[0]` will still need to be "`ls`", which is what the user typed in.

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` or `cat` without a file input).
- All commands will be entered without a path (the path will be added by your shell).

Logging Requirements:

- For foreground jobs, you must call `log_start(pid, LOG_FG, cmd)` to report the start of the job. Here, `LOG_FG` is a predefined constant.
- For background jobs, you must call `log_start(pid, LOG_BG, cmd)` to report the start of the job. Here, `LOG_BG` is a predefined constant.
- If the program cannot be loaded and executed, you must call `log_command_error(cmd)` to report the error then immediately exit the process with exit code 1.
- In all cases above, "`cmd`" refers to the full command line as entered at the prompt.

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 the details and think about how to handle them. Remember to check both valid paths (./ and /usr/bin) with each command before considering it an error.

Example Run (Command Error):

```
MASH>> hello
[MALOG]Foreground Process 12095: hello Started
[MALOG]Error: hello: Command Cannot Load
[MALOG]Foreground Process 12095: hello Terminated Normally
MASH>>
```

Example Run (Foreground):

Note: the actual display of "ls" depends on the contents of your working directory.

```
MASH>> ls .. -al
[MALOG]Foreground Process 11803: ls .. -al Started
total 8
drwxr-sr-x. 4 yzhong itefacstaff   30 Apr  8 15:02 .
drwxr-s---. 9 yzhong itefacstaff  139 Apr  8 14:52 ..
drwxr-sr-x. 4 yzhong itefacstaff 4096 Apr 12 14:43 code
drwxr-sr-x. 2 yzhong itefacstaff   40 Apr  8 15:02 test
[MALOG]Foreground Process 11803: ls .. -al Terminated Normally
MASH>>
```

Example Run (Background):

```
MASH>> ls .. -al &
[MALOG]Background Process 12027: ls .. -al & Started
MASH>> total 8
drwxr-sr-x. 4 yzhong itefacstaff   30 Apr  8 15:02 .
drwxr-s---. 9 yzhong itefacstaff  139 Apr  8 14:52 ..
drwxr-sr-x. 4 yzhong itefacstaff 4096 Apr 12 14:43 code
drwxr-sr-x. 2 yzhong itefacstaff   40 Apr  8 15:02 test
[MALOG]Background Process 12027: ls .. -al & Terminated Normally
MASH>>
```

2.3 Basic Built-in Shell Commands

A typical shell program supports a set of built-in command. If a built-in command is received, the shell process must execute that directly **without** forking any additional process. For this assignment, 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 Section **2.4 Job Control**.

Assumptions:

- You can assume that the user will never end a built-in command with "&".
- You can assume that the user will never request file redirection with a built-in command.
- You can assume there are no nonterminated background jobs when `quit` command is given.

Example Run (Built-in quit/help):

```
MASH>> quit
[MALOG]Thanks for using MASH! Good-bye!
zeus-2:~/cs367/projects/p3/code>
```

```
MASH>> help
[MALOG]Welcome to MASH (MAson SHell)!
[MALOG]Built-in Commands: fg, bg, jobs, kill, quit, help.
[MALOG] kill SIGNAL PID
[MALOG] fg JOBID
[MALOG] bg JOBID
MASH>>
```

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 positive 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 any non-terminated job IDs.
- On start, the foreground job gets a dummy temporary job ID 0. If the foreground job is switched to background for the first time, it will be assigned a positive 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 positive job ID, a job keeps the same job ID until it is terminated.
- You will need to update your record/list of background jobs 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 (without the ending newline "\n"). 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 background 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 look up these macros with **man waitpid**

- You can use `sigaction()` to override the default signal handling and define what actions to take when a signal arrives. See **Appendix D** to get more details.
- Read **Section 2.8** on race conditions for information about situations to watch out for when updating your jobs list.

Example Runs (Built-in/jobs):

```
MASH>> jobs
[MALOG]0 Job(s)
MASH>> sleep 100 &
[MALOG]Background Process 2422: sleep 100 & Started
MASH>> sleep 10 &
[MALOG]Background Process 2431: sleep 10 & Started
MASH>> sleep 200 &
[MALOG]Background Process 2434: sleep 200 & Started
MASH>> jobs
[MALOG]3 Job(s)
[MALOG]Job 1: Process 2422: Running sleep 100 &
[MALOG]Job 2: Process 2431: Running sleep 10 &
[MALOG]Job 3: Process 2434: Running sleep 200 &
MASH>> [MALOG]Background Process 2431: sleep 10 & Terminated Normally
MASH>> jobs
[MALOG]2 Job(s)
[MALOG]Job 1: Process 2422: Running sleep 100 &
[MALOG]Job 3: Process 2434: Running sleep 200 &
MASH>> sleep 20 &
[MALOG]Background Process 2467: sleep 20 & Started
MASH>> jobs
[MALOG]3 Job(s)
[MALOG]Job 1: Process 2422: Running sleep 100 &
[MALOG]Job 3: Process 2434: Running sleep 200 &
[MALOG]Job 4: Process 2467: Running sleep 20 &
MASH>>
```

- `kill SIGNAL PID`: when called, your shell should send a signal specified by `SIGNAL` to the process with a process id matching `PID`.

  Logging Requirements:
  - You must call `log_kill(signal, pid)` to report kill command has been received and activated.

  Implementation Hints:
  - You can use `kill()` to send a signal to a particular process.

  Assumptions:
  - You can assume only these signals will be used in testing built-in command `kill`: `2(SIGINT)`, `9(SIGKILL)`, `18(SIGCONT)`, `20(SIGTSTP)`.

Example Runs (Built-in/kill):

```
MASH>> jobs
[MALOG]1 Job(s)
[MALOG]Job 1: Process 25169: Stopped sleep 100 &
MASH>> kill 18 25169
[MALOG]Built-in Command kill for Sending Signal 18 to Process 25169
MASH>> [MALOG]Background Process 25169: sleep 100 & Continued
MASH>> jobs
[MALOG]1 Job(s)
[MALOG]Job 1: Process 25169: Running sleep 100 &
MASH>> [MALOG]Background Process 25169: sleep 100 & Terminated Normally
MASH>>
```

- `fg JOBID`: when called, your shell should switch the specified background job to be foreground and then wait until it completes. If the job has been previously stopped, resume its execution **after** moving it back to foreground and the status should be "`Running`".

  Logging Requirements:
  - You must call `log_job_move(pid, LOG_FG, cmd)` to report which background process has been switched to be foreground. Here, `LOG_FG` is a predefined constant.
  - If the specified job ID is invalid (i.e. cannot be located in the list of background jobs), you must call `log_jobid_error(job_id)` to report the issue. No change should be made if `JOBID` is invalid.

  Implementation Hints:
  - You can use `kill()` to send signals to a particular process.

Example Runs (Built-in/fg):

```
MASH>> jobs
[MALOG]2 Job(s)
[MALOG]Job 1: Process 28621: Running sleep 30 &
[MALOG]Job 2: Process 28628: Running sleep 200 &
MASH>> fg 3
[MALOG]Error: Job ID 3 Not Found in Background Job List
MASH>> fg 1
[MALOG]Built-in Command fg for Process 28621: sleep 30 &
[MALOG]Foreground Process 28621: sleep 30 & Terminated Normally
MASH>>
```

```
MASH>> jobs
[MALOG]1 Job(s)
[MALOG]Job 2: Process 28628: Stopped sleep 200 &
MASH>> fg 2
[MALOG]Built-in Command fg for Process 28628: sleep 200 &
[MALOG]Foreground Process 28628: sleep 200 & Continued
[MALOG]Foreground Process 28628: sleep 200 & Terminated Normally
MASH>>
```

- `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_move(pid, LOG_BG, cmd)` to report command `bg` has been applied to which background job. Here, `LOG_BG` is a predefined constant.
  - If the specified job ID is invalid (i.e. cannot be located in the list of background jobs), you must call `log_jobid_error(job_id)` to report the issue.

Implementation Hints:

- You can use `kill()` to send signals to a particular process.

Example Runs (Built-in/bg):

```
MASH>> jobs
[MALOG]1 Job(s)
[MALOG]Job 1: Process 29005: Stopped sleep 50 &
MASH>> bg 2
[MALOG]Error: Job ID 2 Not Found in Background Job List
MASH>>
```

```
MASH>> jobs
[MALOG]1 Job(s)
[MALOG]Job 1: Process 29005: Stopped sleep 50 &
MASH>> bg 1
[MALOG]Built-in Command bg for Process 29005: sleep 50 &
MASH>> [MALOG]Background Process 29005: sleep 50 & Continued
MASH>> [MALOG]Background Process 29005: sleep 50 & Terminated Normally
MASH>>
```

```
MASH>> jobs
[MALOG]1 Job(s)
[MALOG]Job 1: Process 29393: Running sleep 30 &
MASH>> bg 1
[MALOG]Built-in Command bg for Process 29393: sleep 30 &
MASH>> jobs
[MALOG]1 Job(s)
[MALOG]Job 1: Process 29393: Running sleep 30 &
MASH>>
```

Assumptions:

- You can assume `kill`/`bg`/`fg` commands typed in by the user are always well-formatted. The only errors that you need to report are the `jobid_errors` as described above.

2.5 Keyboard Interaction

A number of 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 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 itself. The shell program will need to forward the signals to the appropriate process.

For this assignment, you 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 first pause its execution and **then** switch it to be a background job.
- If there is no foreground job when these combinations are input, they should be ignored (i.e. they should not affect the execution of the shell program or any background jobs).

Logging Requirements:

- You must call `log_ctrl_c()` to report the arrival of `SIGINT` triggered by ctrl-c.
- You must call `log_ctrl_z()` to report the arrival of `SIGTSTP` triggered by ctrl-z.

Assumptions:

- You can assume that `SIGINT` signals received by the shell process are only triggered by ctrl-c; and that `SIGTSTP` signals received by the shell process are only triggered by ctrl-z.

Implementation Hints:

- You can use `setpgid()` to change the group ID of a process.
  - Signals apply to all processes in the same group. By default, all new processes start in the same group as their parent, so changing child process to a new group is useful.
  - See **Appendix C** 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.

Example Runs (Keyboard ctrl-c / ctrl-z):

```
MASH>> sleep 200
[MALOG]Foreground Process 29883: sleep 200 Started
^C[MALOG]Keyboard Combination control-c Received
[MALOG]Foreground Process 29883: sleep 200 Terminated by Signal
MASH>>
```

```
MASH>> sleep 100
[MALOG]Foreground Process 15772: sleep 100 Started
^Z[MALOG]Keyboard Combination control-z Received
[MALOG]Foreground Process 15772: sleep 100 Stopped
MASH>> jobs
[MALOG]1 Job(s)
[MALOG]Job 1: Process 15772: Stopped sleep 100
MASH>>
```

Example Runs (Keyboard ctrl-c / ctrl-z with no Foreground Job):

```
MASH>> ^C[MALOG]Keyboard Combination control-c Received
MASH>> ^Z[MALOG]Keyboard Combination control-z Received
MASH>>
```

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. Once you log the error, immediately exit the process.

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 and exit immediately.

Assumptions:

- We will not use the same file for both input redirection and output redirection.

Example Runs (File Error):

```
MASH>> ls -l temp.txt
[MALOG]Foreground Process 24469: ls -l temp.txt Started
ls: cannot access temp.txt: No such file or directory
[MALOG]Foreground Process 24469: ls -l temp.txt Terminated Normally
MASH>> wc < temp.txt
[MALOG]Foreground Process 24557: wc < temp.txt Started
[MALOG]Error: Cannot Open File temp.txt
[MALOG]Foreground Process 24557: wc < temp.txt Terminated Normally
MASH>>
```

Example Runs (Non-bulit-in Commands w/ File Redirection):

```
MASH>> cat < hello.txt
[MALOG]Foreground Process 900: cat < hello.txt  Started
hello world!
[MALOG]Foreground Process 900: cat < hello.txt  Terminated Normally
MASH>> cat < hello.txt > copy.txt
[MALOG]Foreground Process 947: cat < hello.txt > copy.txt Started
[MALOG]Foreground Process 947: cat < hello.txt > copy.txt Terminated Normally
MASH>> cat copy.txt
[MALOG]Foreground Process 1054: cat copy.txt Started
hello world!
[MALOG]Foreground Process 1054: cat copy.txt Terminated Normally
MASH>> cat < hello.txt >> copy.txt
[MALOG]Foreground Process 1326: cat < hello.txt >> copy.txt Started
[MALOG]Foreground Process 1326: cat < hello.txt >> copy.txt Terminated Normally
MASH>> cat copy.txt
[MALOG]Foreground Process 1340: cat copy.txt Started
hello world!
hello world!
[MALOG]Foreground Process 1340: cat copy.txt Terminated Normally
MASH>>
```

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_state(pid, LOG_FG, cmd, LOG_TERM)`
	Running→Terminated (by a signal)	`log_job_state(pid, LOG_FG, cmd, LOG_TERM_SIG)`
	Running→Stopped (will soon switch to background, used for `ctrl-z`)	`log_job_state(pid, LOG_FG, cmd, LOG_STOP)`
	Stopped→Running (used for `fg`)	`log_job_state(pid, LOG_FG, cmd, LOG_CONT)`
Background	Running→Terminated	`log_job_state(pid, LOG_BG, cmd, LOG_TERM)`
	Running/Stopped→Terminated (by a signal)	`log_job_state(pid, LOG_BG, cmd, LOG_TERM_SIG)`
	Running→Stopped	`log_job_state(pid, LOG_BG, cmd, LOG_STOP)`
	Stopped→Running	`log_job_state(pid, LOG_BG, cmd, LOG_CONT)`

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()`. 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_state()`).

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 `execv()`. 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 (`project4_handout.tar`) from the same place you got this document. Once you un-tar the handout on Zeus (using `tar xvf project4_handout.tar`), you will have the following files in the `p4_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`.

- `parse.c` – This has a list of provided parsing functions that you could use to divide the user command line into useful pieces.
- `parse.h` – This has the prototypes of parsing functions implemented in `parse.c`.

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

- `hello.txt` – a simple textual file for convenient testing (of file redirection).

- `my_pause.c` – a C program that can be used as local program to load/execute in shell. It will not terminate normally until SIGINT has been received for three times. Feel free to edit the C source code to change its behavior.

- `slow_cooker.c` – a C program that can be used as local program to load/execute in shell. It will slowly count down from 10 to 0 then terminate normally. You can specify a different starting value and/or edit the C source code to change its behavior.

- `my_echo.c` – a C program that can be used as local program to load/execute in shell. It takes an integer argument and use it as the return value / exit status. The default return value / exit status is 0. You can change the value to test exit status with shell.

To get started on this project, read through the provided code in `shell.c` and the constants and definitions in `shell.h`, `parse.h`, and `logging.h`. Make sure you understand the input/output of the provided parsing facility, in particular the structure of `argv[]` and `cmd_aux`.

You should start by creating and running a simple program with no arguments into the project as a foreground process. Once you have this in your design, add the various features, such as arguments, different paths, and redirection. From here, look at how you would handle keyboard commands (ctrl-c and ctrl-z) on this foreground process; this will add signal handling into your design.

Finally, add in background support. This will involve having to come up with a design for the job system to let you track background processes and handling all the SIGCHLD events (**stopped**/**resumed**/**terminated** child process events all send a SIGCHLD). This also requires you to design how you will handle background and foreground job tracking, moving jobs between the foreground and background, and handling suspend and resume events.

After this, make sure all of the details in each section of this document are met, such as all of the required logging is present (**this is critically important – all grading is done from these log calls)**, that you have all the cases handled, and that all features are incorporated. The more modular you make your design, the easier it will be to debug, test, and extend your code.

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 G# and name as a commented line in the beginning of your program.**

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 A: User Command Line (Format and Parsing)

The provided `parse.c` includes a `parse()` function that divides the user command into useful pieces. To use it, provide the full command line as input in `cmd_line`, and previously allocated data structures `argv` and `aux`. The `parse()` function will then populate `argv` and `aux` based on the contents of `cmd_line`.

```
void parse(char *cmd_line, char *argv[], Cmd_aux *aux);
```

- `cmd_line`: the line typed in by user **WITHOUT** the ending newline (`\n`).
- `argv`: an array of NULL terminated char pointers with the similar format requirement as the one used in `execv()`.
  - `argv[0]` should be either a built-in command, or the name of the program to be loaded and executed;
  - the remainder of `argv[]` should be the list of arguments used to run the program
  - `argv[]` must have NULL following its last argument member.
- `aux`: the pointer to a `Cmd_aux` record which is used to record the additional information extracted from `cmd_line`. The detailed definition of the struct is as below.

```
/* Cmd_aux Definition (see parse.h for details) */
typedef struct cmd_aux_struct{
  char *in_file;  /* input file name */
  char *out_file; /* output file name */
  int is_append;  /* append to the output file or not */
  int is_bg;      /* background job or not */
}Cmd_aux;
```

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 executable is `25`. Check `shell.h` for relevant constants defined for you.

Notes of Usage:

- The provided `parse.c` also has supporting functions related to command line parsing, including the initialization/free of `argv` and `cmd_aux`. Check `parse.h` for details.
- The provided `shell.c` already includes necessary steps to make the call to `parse()`.
- The provided `parse.c` also has a `debug_print_parse()` function you could use to check the return of `parse()`. It's for debugging only.

Examples of `parse()` input/output:

- **Single command w/o file redirections**

cmd_line	argv	aux			
		in_file	out_file	is_append	is_bg
"help"	{"help",NULL}	NULL	NULL	-1	0
"sleep 200"	{"sleep","200",NULL}	NULL	NULL	-1	0
"sleep 200 &"	{"sleep","200",NULL}	NULL	NULL	-1	1

- **Single command w/ file redirections**

cmd_line	argv	aux			
		in_file	out_file	is_append	is_bg
"ls -l > ls.txt &"	{"ls","-l", NULL}	NULL	"ls.txt"	0	1
"wc < ls.txt >> wc.txt"	{"wc", NULL}	"ls.txt"	"wc.txt"	1	0

Appendix B: Useful System Calls

Here we include a list of system calls that perform process control and signal handling. You might find them helpful for this assignment. The system calls are listed in alphabetic order with a short description for each. Make sure you check our textbook, lecture slides, and Linux manual pages on `zeus` to get more details if needed.

- `int dup2(int oldfd, int newfd);`
  - It makes `newfd` to be the copy of `oldfd`; useful for file redirection.
  - Textbook Section **10.9**
  - Manual entry: ***man dup2***

- `int execv(const char *path, char *const argv[]);`
- `int execl(const char *path, const char *arg, ...);`
  - Both are members of exec() family. They load in a new program specified by `path` and replace the current process.
  - Textbook Section **8.4**
  - Manual entry: ***man 3 exec***

- `void exit(int status);`
  - It causes normal process termination.
  - Textbook Section **8.4**
  - Manual entry: ***man 3 exit***

- `pid_t fork(void);`
  - It creates a new process by duplicating the calling process.
  - Textbook Section **8.4**
  - Manual entry: ***man fork***

- `int kill(pid_t pid, int sig);`
  - Used to send signal `sig` to a process or process group with the matching `pid`.
  - Textbook Section **8.5.2**
  - Manual entry: ***man 2 kill***

- `int open(const char *pathname, int flags);`
- `int open(const char *pathname, int flags, mode_t mode);`
  - Opens a file and returns the corresponding file descriptor.
  - Textbook Section **10.3**
  - Manual entry: ***man 2 open***

- `int setpgid(pid_t pid, pid_t pgid);`
  - It sets the group id for the running process.
  - Textbook Section **8.5.2**
  - Manual entry: ***man setpgid***

- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - Used to change the action taken by a process on receipt of a specific signal.

  - Textbook Section **8.5.5 (pp.775)**
  - Manual entry: ***man sigaction***

- `int sigaddset(sigset_t *set, int signum);`
- `int sigemptyset(sigset_t *set);`
- `int sigfillset(sigset_t *set);`
  - The group of system calls that help to set the mask used in `sigprocmask`.
  - `sigemptyset()` initializes the signal set given by `set` to empty, with all signals excluded from the `set`.
  - `sigfillset()` initializes `set` to full, including all signals.
  - `sigaddset()` adds signal `signum` into `set`.
  - Textbook Section **8.5.4**
  - Manual entry: ***man sigsetops***

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
  - Used to fetch and/or change the signal mask; useful to block/unblock signals.
  - Textbook Section **8.5.4, 8.5.6**
  - Manual entry: ***man sigprocmask***

- `Unsigned int sleep(unsigned int seconds);`
  - It makes the calling process sleep until `seconds` seconds have elapsed or a signal arrives which is not ignored.
  - Note: `sleep` measures the elapsed time by absolute difference between the start time and the current clock time, regardless whether the process has been stopped or running. This means if you suspend and resume it, it will check to see if x seconds have passed since starting.
    - So, if you use sleep 5, then ctrl-Z 1 second into the run, wait 30 seconds, and then resume it, it will see at least 5 seconds have elapsed since it started and will immediately quit, even though it only 'ran' for 1 second.
  - Textbook Section **8.4.4**
  - Manual entry: ***man 3 sleep***

- `pid_t waitpid(pid_t pid, int *status, int options);`
  - Used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
  - Textbook Section **8.4.3**
  - Manual entry: ***man waitpid***

- `ssize_t write(int fd, const void *buf, size_t count);`
  - It writes up to `count` bytes from the buffer pointed `buf` to the file referred to by the file descriptor `fd`. The standard output can be referred to as `STDOUT_FILENO`.
  - Textbook Section **10.4**
  - Manual entry: ***man 2 write***

Appendix C: 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 D: 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;

  memset(&new, 0, sizeof(new));
  new.sa_handler = sigint_handler;     /* set the handler */
  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