Operating Systems	Programming Project 1
CSC3320 Fall 2022	Due Date: Monday, Sept. 26, 2022

> Please solve the problem(s) alone. Remember to test your solution thoroughly. Code that does not work correctly will loose credit. ***Code that does not compile will receive only 5 points of credit.*** Please provide a print out of your source code and answers to any accompanying questions at the **start** of class. Please also submit a digital copy, **before** the start of class, via Classroom. The digital copy of your code should be a zip file containing the project folder named with your last name followed by the project number. For example, *kissel1.zip* would be my zip file for project 1. Good luck!

In this project you will gain experience with Linux system calls by constructing a new Linux shell. Your shell will be called msh (Merrimack Shell). Recall that a shell is a command line interface (CLI) that allows you to interact with the operating system. The assignment will be broken down into phases to help you organize your thoughts. To help you get started I have uploaded starter code to classroom.

Notice you have been given **two(2) weeks** to complete this project. This project can **not** be completed in one night. You should complete the phases below in order. Each phase will consists of a few days work (see recommend completion date).

Phase I – Basic Shell (*35 points*)

Recommended Completion Date: Friday, Sept. 21

In the first phase of this project you will implement the majority of the shell. You must construct the basic command line interpreter (it's the classic REPL algorithm we discussed in class). Your shell (i.e., the main process) should enter a tight loop that prompts the user for a command and then executes that command by `fork(2)`-ing the main process and having the child `exec(3)` the command. The parent should use `wait(2)` to wait for the completion of the child process before continuing.

There is one exception to the program flow above. If the user enters the command `quit` or `exit`, the main process should not call `fork(2)` and instead just exit the program successfully.

An example interaction with my solution to Phase I is below. Please make your output look *exactly* like mine. In other words, please make your prompt be `msh>`. You can just call `get_command` in `shell.c` to achieve this task.

```
msh> make clobber
rm -f shell.o history.o
```

```
rm -f shell
msh> ls
head  history.cpp  Makefile  shell.cpp
msh>
msh>
msh> quit
```

Notice that I may hit enter as many times as I want with out causing the shell to do any work.

As mentioned previously, you will need the `fork(2)` and `exec(3)` system calls. Recall that `exec(3)` is a family of system of calls. I would recommend you use the `execvp(3)` system call. To do use this you will need to construct an array of C strings (character arrays terminated with a `NULL` character). This special array you construct is sometimes referred to as the `argv` array. The first element in the array is the name of the command and the subsequent elements are are the options to the command. To help you with converting a C string to this `argv` array, I have provided you with the the function `build_argument_array` in `shell.c`.

The manual pages (man pages) will provide some assistance with `fork(2)` and `execvp(3)`.

Phase II – Backgrounding Processes (*30 points*)

Recommended Completion Date: Thursday, Sept. 23

In this phase your are going to extend your shell with the ability to launch what is called a background process. To launch a background process the user appends and ampersand(&) to the end of the command

A background process is run in a child process just like in Phase I, except with a background process the shell won't wait for the process. Won't this make zombies you ask? Why yes it will if you don't keep track of the outstanding process and make sure they are all finished before you exit your program. Recall that when you `fork(2)` a process `fork(2)` returns, to the parent process, a non-zero number. This non-zero number is actually the PID of the newly constructed child process. After the parent process has `fork(2)`-ed its child process you should add this new PID to a linked list. You can use the `joblist` data structure I have provided for you to keep track of the active PIDs.

You must make some additional modifications as well. You must:

- Modify the parent code, following the `fork(2)` call, to add the PID to the list of outstanding PIDs if the command is to be backgrounded. If it is not a background command you should wait for the process to complete. There is a pitfall here, when you call `wait(2)` the call will return once one of the children has completed. The return value will tell you which one. Please make sure you wait for the correct PID. If a different PID completes you should remove that PID from the list of PIDs.

- Modify your code that handles `quit` and `exit` to make sure that all the processes in the list have completed (i.e., the list is empty). If the list is not empty, you should call `wait(2)` until it is empty. Note that `wait(2)` returns -1 and sets `errno` to `ECHILD` if there are no children of the process left. If this occurs, you can just empty the list and exit the program. The `errno` variable is a special global variable used to pass errors around Unix based systems. To use this global just include `errno.h` in your program.

The functions that are provided by a the `joblist` data structure can be found in `head/joblist.h`. In order to understand what each function does, you should read the header comments in `joblist.c`.

Phase III – Add Built-In Commands (*20 points*)

Recommended Completion Date: Monday, Sept. 27

In this phase you will add some additional built-in commands. Built-in commands are commands that are part of the shell program itself. We are going to add command line history to our shell. To help you with this process I have provided you with a finite buffer class. I would recommend you use it to manage your command line history.

You should add the following commands:

- `history`: This command will print out the current contents of the history table or message indicating the history is empty. This command should itself be added to the history buffer.

- `!!`: This command will execute the last command the user executed. This command should not be added to the history buffer.

- `!`n: This command tells the shell to execute the n-th command in the history. If no such command exists it should display an error. This command should not be added to the history buffer. You may find the function `atoi(3)` helpful in parsing this command.

- `cd` *dir*: This command changes the directory the the location given by *dir*. This is very similar to the `cd` built-in command in BASH (e.g., `man cd`). You do *not* have to implement `cd` without a directory or `cd ~`. You will need to use the `chdir(2)` system call.

Unless otherwise noted, every command should be added to the history buffer. You will find the `history` data structure in `history.h` and `history.c` helpful in your task. Your finite buffer should be initialized to hold 5 commands in the history.

Style (*10 points*)

Please be sure to use good function decomposition and follow the department standard when coding your solution. Make sure your code is well commented I need to understand what you understand. If you have any style questions please ask.

A Clean Build (*5 points*)

To build your program you just need to type `make(1)` at the command prompt. The file `Makefile` has comments if you are curious how `Makefile`'s go together. Your code should:

- Compile clean (i.e., free of errors and warnings) on Fedora 34 using the command `make all`
- Contain no memory leaks per `valgrind`. I will be using `valgrind` with the `--leak-check=full` when checking your programs.
  - You can install `valgrind` using the command `sudo dnf install valgrind`.

Make Targets

The operations that `make(1)` takes are called the targets. The makefile I have provided you with supports three targets of interest:

1. `all` this target builds the entire project, resulting in an executable called `shell`.
2. `clean` this target removes all of the intermediate files generated by the compiler.
3. `clobber` this target does everything `clean` does as well as deleting the executable `shell`.

It should be noted that `all` is the default target and thus, you can simply write `make` and the `all` target will run.

General Reminders

- **Start early**.
- Each phase comes with a suggested completion date to keep you on track during the two weeks. You will notice that at least 24 hours of time is recommended to be used for testing and debugging your project solution.
- Complete all of the other phases before attempting the extra credit.
- You should test your shell thoroughly.

- Make sure to check your code against the assignment *before* submission.
- You are encouraged to see me for help with the project.
- You can discuss the project with each other but, the write-up should be your own.
- Don't forget about the man pages, they are as helpful as JavaDocs
- Do not modify the code in functions, unless I told you to.

Submission

Submissions in this class come in two parts:

1. **Digital Submission**: Since our projects now have multiple files that need to be graded, I ask that you submit a zip file containing the `make` file and all associated with headers and source files. This zip file **must be** named with your last name followed by the project number. Failure to follow these directions will result in a deduction of up to **5 points from your grade**.
2. **Hard Copy of Code**: A hard copy of all header and source code files you authored for a project. This hard copy should be stapled together as one packet. Failure to follow this direction will result in a deduction of up to **5 points from your grade**.

The directions above are for the purpose of ensuring a timely and correct grading of your project.

Grading

To recap the above, your grade on this assignment will be determined as follows:

- Program has good style and function decomposition (*10 points*).
- Program correctly implements phase I (*35 points*).
- Program correctly implements phase II (*30 points*).
- Program correctly implements phase III (*20 points*).
- Program compiles clean and has no memory leaks (*5 points*).
- Correctly implemented Extra credit I (*5 points*).
- Correctly implemented Extra credit II (*5 points*).

Extra Credit I (*5 points*)

You can extend your implementation of the `cd` built-in command to support `cd` without a path after. This invocation, returns the user to their home directory. You can experiment with this behavior on your VM. To implement this feature, you should you will need to read the `HOME` environment variable. You should use `apropos(1)` to determine what function (hint it will be in section 3) will allow you to read the `HOME` environment variable. For example: `cd` is equivalent to `cd /home/kisselz` when I executed the command on my machine.

Extra Credit II (*5 points*)

Once you have implemented the first extra credit, you can extend `cd` again. In this extension, you will handle `~`. When you see a `~` at the start of a path provided to `cd` you should replace `~` with the contents of the `HOME` environment variable. For example:

- `cd ~` is equivalent in behavior to `cd`
- `cd ~` is equivalent in behavior to `cd /home/kisselz` for myself on Fedora 34.
- `cd ~/public/os-examples` is equivalent in behavior to `cd /home/kisselz/public/os-examples` when I execute the command on Fedora 34.

Again, you will have to use `apropos(1)` to determine what function call will allow you to read the `HOME` environment variable.