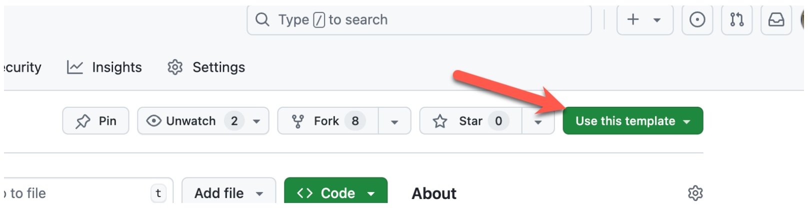
P2

Start Assignment

- Due Feb 25 by 11:59pm
- Points 100
- Submitting a website url

Overview

- Fork (**use this template**) the [starter repository (https://github.com/shanep/makefile-project-starter)](https://github.com/shanep/makefile-project-starter) into your own GitHub account
- Complete the lab as detailed below



In this project, we will be implementing a simple shell that can start background process using the examples that were presented in chapter 5.

Learning Outcomes

- 1.2 Use system library code
- 1.3 Use system documentation
- 1.4 Apply computer science theory and software development fundamentals to produce computing-based solutions.
- 2.2 Explore the system call interface

Tasks

Task 1 - Prepare your repository

The starter repository is a bare bones template that you will need to update with the starter code below.

- [lab.h (https://boisestatecanvas.instructure.com/courses/38319/files/18513441?wrap=1)](https://boisestatecanvas.instructure.com/courses/38319/files/18513441?wrap=1) **(https://boisestatecanvas.instructure.com/courses/38319/files/18513441/download?download_frd=1)**
- [test-lab.c (https://boisestatecanvas.instructure.com/courses/38319/files/18513442?wrap=1)](https://boisestatecanvas.instructure.com/courses/38319/files/18513442?wrap=1) **(https://boisestatecanvas.instructure.com/courses/38319/files/18513442/download?download_frd=1)**

- **main.c (https://boisestatecanvas.instructure.com/courses/38319/files/18537218?wrap=1) (https://boisestatecanvas.instructure.com/courses/38319/files/18537218/download?download_frd=1)**

The testing code in test-lab.c is **NOT** comprehensive. So if you pass all the tests it does not mean your shell is 100% correct. You will need to add additional tests to get better coverage.

Task 2 - Print Version

Let's start off simple and just have our shell print off its version. When the shell is started with a command line argument -v, it prints out the version of the project and then quits, you will need to leverage the header file `lab.h` for the major and minor version. Parse the command line arguments with **getop (https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html)**. The program should exit after printing the version.

Task 3 - User Input

While we could use a function like **scanf (https://cplusplus.com/reference/cstdio/scanf/)** to get input from the user a much more robust way would be to leverage the **GNU Readline library (https://tiswww.case.edu/php/chet/readline/rltop.html)**. The GNU Readline library allows a program to control the input line and adds a bunch of cool functions that allow the user to edit the line, use TAB key for filename completion, and the use of up arrow, down arrow, left arrow and right arrow keys to access the history of commands typed in by the user. You will need to include the header files `readline/readline.h` and `readline/history.h` to use the readline functions.

You need to install both the readline header files and development libraries for the above code to compile and link. All the correct libraries are installed on the lab machines. On Redhat based machines development packages end in devel. So to get all the readline development packages you would need to install **readline-devel** as well as **readline**. On ubuntu the development files are named **readline-dev**.

The readline documentation is a good starting point on how to use the readline library. Pay close attention to memory ownership. Remember that C does not have a garbage collector.

- **Readline Docs (https://tiswww.cwru.edu/php/chet/readline/readline.html#index-readline)**
- **History Docs (https://tiswww.cwru.edu/php/chet/readline/history.html)**

Task 4 - Custom Prompt

The default prompt for the shell may be anything you like. However the shell checks for an environment variable `MY_PROMPT`. If the environment variable is set, then it uses the value as the prompt. The environment variable can be set inline `MY_PROMPT="foo>" ./myprogram` so you can quickly test your program.

- Use the system call **getenv (https://man7.org/linux/man-pages/man3/getenv.3.html)** to retrieve environment variables

Task 5 - Built in Commands

Now that we can get input from users lets add in some built in commands. These commands need to be handled by the shell itself, you should not create a new process to handle these commands so it is good to implement these **before** you add in the fork/exec code in a future task.

Exit command

Include a built-in command named `exit` that terminates the shell normally. Your shell should return a status of 0 when it terminates normally and a non-zero status otherwise. Your shell should also terminate normally on receiving the end of input **EOF (https://en.wikipedia.org/wiki/End-of-file)** (Under Linux and bash, this would normally be Ctl-d for you to test your mini-shell). You are required to clean up any allocated memory before you exit.

- **exit (https://man7.org/linux/man-pages/man3/exit.3.html)**

Change Directory Command

Include a built-in command named `cd` to allow an user to change directories. You will need to use the `chdir` system call. The `cd` command without any arguments should change the working directory to the user's home directory. You must first use `getenv` and if `getenv` returns NULL your program should fall back to the system calls `getuid` and `getpwuid` to find out the home directory of the user. Make sure to print an error message if the cd command fails.

- See **chdir (https://man7.org/linux/man-pages/man2/chdir.2.html)**
- See **getenv (https://man7.org/linux/man-pages/man3/getenv.3.html)**
- See **getuid (https://man7.org/linux/man-pages/man2/getuid.2.html)**
- See **getpwuid (https://man7.org/linux/man-pages/man3/getpwuid.3p.html)**

History Command

Add a new built in command to your shell to print out a history of command entered. You should leverage the **history library (https://tiswww.cwru.edu/php/chet/readline/history.html)** library to accomplish this.

Task 6 - Create a Process

Our shell will create a new process and wait for it to complete. The shell accepts one command per line with arguments. It should accept at least `ARG_MAX` arguments to any command. You will need to use the system call `sysconf` and `_SC_ARG_MAX` to get the maximum length of arguments that the the `exec` family of functions can accept. The shell will parse each line that is entered and then attempt to execute the process using the `execvp` system call. The `execvp` system call performs a search for the command using the `PATH` environment variable. This will simplify your programming since you do not have to search for the location of the command.

For our simple shell you can assume that all command line arguments will be separated by spaces, you don't have to worry about quoted arguments. For example given the command `ls -l -a` you would parse this string as an array of size 3 with the structure of `ls` → `-l` → and `-a`. The command `ls "-l -a"` would parse out to be `ls` → `"-l` → `-a"`. If you want to write a parsing algorithm that handles quotes like bash more information is available at **the linux documentation project (https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_03.html)**.

If the user just presses the Enter key, then the shell displays another prompt. If the user types just spaces and then presses the Enter key, then the shell displays another prompt as this is also an empty command. Empty commands should not cause a segfault or memory leak.

- **execvp (https://man7.org/linux/man-pages/man3/exec.3.html)**
- **fork() (https://man7.org/linux/man-pages/man2/fork.2.html)**
- **waitpid() (https://man7.org/linux/man-pages/man2/wait.2.html)**

Task 8 - Signals

The shell should ignore the signals listed below:

c

```c
signal(SIGINT, SIG_IGN);
signal(SIGQUIT, SIG_IGN);
signal(SIGTSTP, SIG_IGN);
signal(SIGTTIN, SIG_IGN);
signal(SIGTTOU, SIG_IGN);
```

In the child process don't forget to set these signals back to default!

c

```c
/*This is the child process*/
pid_t child = getpid();
setpgid(child, child);
tcsetpgrp(sh.shell_terminal,child);
signal (SIGINT, SIG_DFL);
signal (SIGQUIT, SIG_DFL);
signal (SIGTSTP, SIG_DFL);
signal (SIGTTIN, SIG_DFL);
signal (SIGTTOU, SIG_DFL);
execvp(cmd[0], cmd);
fprintf(stderr, "exec failed\n");
```

If there is no process being executed, then the shell should just display a new prompt and ignore any input on the current line. You will need to use the tcgetpgrp and tcsetpgrp system calls to get and set the foreground process group of the controlling terminal and the signal system call to ignore and enable signals.

The glibc manual links below describe a full job control shell. You are not required to implement a job control shell to the same level of functionality. You can to use the documentation linked below as a guide but be aware you shell will probably fail to function correctly if you just copy and paste the code examples without understanding what they do. You are free to use code from the manual as long as you take the time to understand what it does and why.

- **glibc - Initializing the Shell (https://www.gnu.org/software/libc/manual/html_node/Initializing-the-Shell.html)**
- **glibc - Launching Jobs (https://www.gnu.org/software/libc/manual/html_node/Launching-Jobs.html)**
- **Signal man page (https://man7.org/linux/man-pages/man7/signal.7.html)**

Submitting

Make sure all your code is pushed to GitHub, and then submit the URL to your public repository.