1 Description of the programming project

In this second programming project, we will continue developing the shell that we started in the first project. We will add to the shell the capability to redirect standard input/output and execute external programs. **All previous functionality should continue to work correctly**. In addition to all commands from the previous project, the shell should process the following commands:

1. `myecho string1 [... stringN]`: prints `string1 ... stringN` to the standard output.
2. `redirect input [output]`: redirects the standard input and output to `input` and `output`, respectively. If `output` is omitted, then only the standard input is redirected. If `input` is `-` and `output` is provided, only the standard output is redirected. If `input` is `-` and `output` is omitted, an error should be reported. See Section 2 for more details.
3. `pid [-p]`: Prints the `pid` of the process executing the shell; `pid -p` prints the `pid` of its parent process.
4. `fork`: The shell creates a child process with a `fork()` system call (this child process executes the same code as the shell) and *waits* for it to end.
5. `exec prog [arg1 arg2 ...]`: Executes program `prog` with arguments `arg1 arg2 ...` (they can be more than two). `prog` is either the path (absolute or relative) of an executable file in the file system or the name of a file in the of of the directories listed in the PATH environment var The program is excused without creating a process (that is, the shell's code is REPLACED by the new program).
6. `fg prog [arg1 arg2 ...]`: The shell executes program `prog` with arguments `arg1 arg2 ...` in the *foreground*. Executing a program in the foreground means that the shell creates a process that executes `prog` and waits until it exits. In other aspects, the execution of the program should be handled as in the `exec` command above.
7. `bg prog [arg1 arg2 ...]`: The shell executes program `prog` with arguments `arg1 arg2 ...` in the *background*. Executing a program in the background means that the shell creates a process that executes `prog` and does NOT wait until it exits. In other aspects, the execution of the program should be handled as in the `exec` command above.

Error handling The shell must not crash if a command longer than expected is given. In general, the shell must not crash if something unexpected occurs or a command cannot be performed for any reason. The shell must not fail to execute a command without reporting a reason. Instead an error message should be produced.

2 Redirection and buffer issues on the standard input/output C library

The redirection of standard input and output is done with the system call `dup2`. When redirecting the output all data written before the redirection should be written to the the terminal or file that was the standard output before the redirection. For instance, if `input.txt` contains the following commands

```
myecho Hellow world!
```

```
redirect - output2.txt
myecho Good bye.
```

and we invoke **myshell <input.txt >output.txt**, then string "**Hellow world!**" should be written to file **output.txt** while string "**Good bye.**" should be written to **output2.txt**. Depending on how the rest of the shell is implemented, a common error is that both strings get written to **output2.txt**. In particular, this may be the case if you use functions other than **write()** (e,g, **printf()**, **fwrite()**, etc.) to write to the standard output. The reason behind this error is that some functions may write data to a buffer before using **write()** to actually write the data. When calling **dup2()**, the operating system is not aware that there are pending writes and the I/O library is not aware that a redirection has happened. When the library writes the data, it writes it to the new file instead of the intended one. To solve this problem, it is necessary to force the I/O library to write all data before calling **dup2()**. This is achieved by calling **fflush(stdout)** before **dup2()**.

A similar issue may occur when the system calls **fork()** and **exec()** are performed, or when the input is redirected. Regarding the first, function **fflush(stdout)** should be called before **fork()** and **exec()** are called. Regarding the redirection of the input, if **input.txt** contains the following commands

```
myecho Hellow world!
redirect input2.txt
myecho Good bye.
```

and **input2.txt** contains

```
myecho Bye bye.
```

and we invoke **myshell <input.txt >output.txt**, then **output.txt** should contain the following

```
Hellow world!
Bye bye.
```

Note that the last command of file **input.txt** (**myecho Good bye.**) should not be executed, since after the redirection it never should be read. This will work correctly if user commands are read using the **read()** system call. However, some functions from the standard library (e.g. **fgets()**, **scanf()**, **getc()**, **getchar()**, **fgetc()** etc.) may read more than one line and store it in a buffer. When a new line is requested, the function may return some of the already read lines (or characters) from the buffer before starting reading from the new file. If any of these functions is used, the input redirection should be made using the library function **freopen()** instead of the system call **dup2()**. This function will take care of the input buffer before calling **dup2()**.

3 Details and clues

Recall that the shell must be programed in C. To implement the given functionality you may need to use some system calls and functions available in the standard library. In particular, you may be interested in looking into the functions `dup2`, `freopen`, `fflush`, `fork`, `exec`, `waitpid`,... Information about these functions may be obtained by typing `man function` into the Linux system shell.

The difference between executing in foreground and background is that in foreground the parent process waits for the child process to end using one of the wait system calls, whereas in background the parent process continues to execute concurrently with the child process. Executing in background should not be tried with programs that read from the standard input in the same session.

Recall that to create processes we use the `fork()` system call. It creates a processes that is a clone of the calling process, the only difference is the value returned by `fork()` (0 to the child process and the child's `pid` to the parent process). The `waitpid()` system call allows a process to wait for a child process to end. The following code creates a child process that executes `function1()` while

the parent executes `function2()`. When the child ends, the parent process executes `function3()`.

```
...
pid_t pid = fork();
if (pid == -1) {
    /* error handling */
}
else if (pid==0) {
    function1();
    exit(0);
}
else {
    function2();
    waitpid(pid,NULL,0);
    function3();
}
...
```

In the following code both the parent and the child execute `function3()`.

```
...
pid_t pid = fork();
if (pid == -1) {
    /* error handling */
}
else if (pid==0) {
    function1();
}
else {
    function2();
}
function3();
...
```

For a process to execute a program we use one of the system calls of the `exec` family. In particular, `execvp()` searches for the executable in the directories specified in the PATH environment variable. It only returns a value in case of error, otherwise it replaces the calling process's code. The following code executes the **`ls -l`** command using the `execl()`.

```
execl("/bin/ls","ls","-l","/usr", NULL);
function(); /* it will not be executed unless execl fails*/
```

`execvp()` works the exactly the same but with two small differences:

- it searches for executables in the PATH so, instead of specifying `"/bin/ls"` it would be enough to pass just `"ls"`
- it receives a NULL terminated array of pointers instead of a variable number of pointers to the arguments.

Using `execvp()` will probably make your life easier for this assignment.

To check the state of a process, we can use `waitpid()` with the following flags.

```
waitpid(pid, &state, WNOHANG|WUNTRACED|WCONTINUED)
```

This system call provides information about the state of process `pid` in the variable `state` ONLY WHEN THE RETURNED VALUE IS `pid`. The information can be inspected with macros described in man **`man waitpid`** (`WIFEXITED`, `WIFSIGNALED`, etc.)

4 Submission

The submission must consist of a single file named exactly **`p2.tar.gz`** and should be submitted through the **`canvas`** web platform. Check in canvas the deadline for the submission. The file **`p2.tar.gz`** can be created with the command

```
tar -czvf p2.tar.gz *
```

After executing this command a file **`p2.tar.gz`** will be created containing all the files and directories in the current directory. The content of the file can be extracted with the command

```
tar -xzvf p2.tar.gz
```

All the content of **`p2.tar.gz`** will be extracted in the current directory. The file **`p2.tar.gz`** must contain a directory called **`src/`** and a file called **`Makefile`**. It may contain in addition a directory called **`test/`**. Inside the directory **`src/`** should be all the source code of your shell. Executing the command **`make`** should:

- compile your code,
- create a directory called **`bin/`**, and
- place a command called **`myshell`** inside it.

After executing the command **`make`**, executing the command **`./bin/myshell`** should execute your shell.

An example file **`p2.tar.gz`** can be download from the canvas platform. It contains the needed structure, **`Makefiles`** and a dummy **`myshell.c`** file. Inside the **`test/`** directory there is a bash script **`compile_test.sh`** that can be used to automatically test that these steps work correctly. To check that your **`p2.tar.gz`** file can be extracted and compiled correctly, copy it together with the script **`compile_test.sh`** into an empty directory; then execute the script. No error should be reported. **Projects that fail this test will yield no score.**

5 Automatic testing

The sample file **`p2.tar.gz`** posted on canvas also includes a series of scripts for automatic testing. These scripts are there to help you to check your progress. There is also a script named **`all_test.sh`** that runs all the other scripts and provides a summary of the results. You can run all the test by typing **`test/all_test.sh`** in the system shell. In case of an error on any of the tests, you can run that test individually. For instance, if test **`cdir_test.sh`**, then you can invoke **`test/cdir_test.sh`**. After invoking an individual test, you can check the files **`output.txt`**, **`error.txt`**, **`expected_output.txt`**, **`expected_error.txt`**, **`output_ls.txt`** and **`expected_ls.txt`** in the directory **`test/`** to see the difference between your **`shell`** results and the expected ones. These files are not available after running **`all_test.sh`**. These tests will also be used for evaluating your project (see Section 6 below). If you believe that your shell is producing the correct result according

to the above specification and yet the automatic test reports an error, then contact the instructor before submitting the project.

6 Evaluation

Be sure to test your code in the school **odin server** before submitting the exercise: `https://www.unomaha.edu/college-of-information-science-and-technology/about/odin.php`

Your program will be evaluated by testing a variety of commands. To receive 100 percent, your program must process the entire set of the given commands correctly. If **any of the following errors** occur, the exercise **will not yield any score**:

- The program does not compile or execute on the odin server following the instructions in Section 4. In particular, if the **`compile_test.sh`** does not run correctly as specified in that section.
- The program executes, but it produces a runtime error (like segmentation-fault, bus error...)

The project will be evaluated through (i) automatic testing and (ii) manual/visual code inspection. **Note:** If the provided automatic test for a command fails, then the code implementing that command will not be inspected. That is, if any of the following tests (provided to you in the sample file **`p2.tar.gz`** on canvas) fail, then a 20% of the score will be automatically subtracted from the maximum grade.

`redirect_test.sh`, **`fork_test.sh`**, **`exec_test.sh`**, **`fg_test.sh`** and **`bg_test.sh`**

Extra tests will be executed to determine your final grade. You are encouraged to exhaustive test your code and write your own automatic tests.

Continuous evaluation. This is the second iteration of the shell. As mentioned in the beginning of the document, **all previous functionality should continue to work correctly**. For the evaluation, this means that if any functionality that worked correctly in the first iteration (the submission of the first project) does not work in this second submission, then a 20% of the grade obtained from it in the first project would be substracted from this second project. In no case the grade of the first project will be reduced.

On the other hand, you may receive up to 75% of the grade corresponding to any of the following tests of the first project that did not work in the previous submission.

`create_test.sh`, **`delete_test.sh`**, **`hist_test.sh`** and **`list_test.sh`**

This means that you can get up to 60% of the total grade for the first project in this submission. The test **`cdir_test.sh`** is not graded in this submission and it is provided as part of the a shell that you can use as base for the second project.

The script **`test/all_test.sh`** has been updated to test for the commands of the first project. The script **`test/all_test_p2.sh`** can be used to run the commands of the second project alone. Be sure to run **`compile_test.sh`** and **`test/all_test.sh`** before submission!