ECEC-353 Systems Programming
Project 1 – Writing a Basic Shell

START THIS NOW -- YOU WILL NEED THE FULL TIME

The Big Idea

In this assignment you will be developing a user shell (i.e. command line interface) similar to `bash`. Upon completion, your shell must be able to perform the following operations:

- Display the current working directory within the prompt (before the dollar sign):

```
/home/jshack/pssh$
```

- Run a single command with optional input and output redirection. Command line arguments must be supported. For example:

```
/home/jshack/pssh$ ./my_prog arg1 arg2 arg3
/home/jshack/pssh$ ls -l > directory_contents.txt
/home/jshack/pssh$ grep -n "test" < input.txt > output.txt
```

- Run multiple pipelined commands with optional input and output redirection. Naturally, command line arguments to programs must still be supported. For example:

```
/home/jshack/pssh$ ./my_prog arg1 arg2 arg3 | wc -l > out.txt
/home/jshack/pssh$ ls -lh | awk '{print $9 " is " $5}'
/home/jshack/pssh$ ps aux | grep bash | grep -v grep | awk '{print $2}'
```

- Implement the builtin command '`exit`', which will terminate the shell:

```
/home/jshack/pssh$ exit
```

- Implement the builtin command '`which`'. This command accepts 1 parameter (a program name), searches the system `PATH` for the program, and prints its full path to `stdout` if found (or simply nothing if it is not found). If a fully qualified path or relative path is supplied to an executable program, then that path should simply be printed to `stdout`. If the supplied program name is another builtin command, your shell should indicate that in a message printed to `stdout`. The behavior should be identical to `bash`'s builtin `which` command. For example:

```
/home/jshack/pssh$ which ls
/bin/ls
/home/jshack/pssh$ which lakjasdlkfjasdlkfj
/home/jshack/pssh$ which exit
exit: shell built-in command
/home/jshack/pssh$ which which
which: shell built-in command
/home/jshack/pssh$ which man
/usr/bin/man
```

The Parser

Writing a command line shell is obviously going to require that you be able to parse the text the user types at the prompt into a more meaningful structure that is easier for you (the developer) to handle. Since text processing is not the major focus of this course (hint: it's actually systems programming), I have provided you with a parser as well as a basic skeleton for your shell program. I call this shell program skeleton the **Pretty Simple SHell (pssh)**. The main input loop looks like this:

```
...
#define DEBUG_PARSE 0
...
    while (1) {
        cmdline = readline(build_prompt());
        if (!cmdline)       /* EOF (ex: ctrl-d) */
            exit(EXIT_SUCCESS);

        P = parse_cmdline(cmdline);
        if (!P)
            goto next;

        if (P->invalid_syntax) {
            printf("pssh: invalid syntax\n");
            goto next;
        }

#if DEBUG_PARSE
        parse_debug(P);
#endif
        execute_tasks(P);

    next:
        parse_destroy(&P);
        free(cmdline);
    }
...
```

As you can see, our command line shell reads input from the user one line at a time via `readline()`, which returns a `char *` pointing to memory allocated on the heap containing a `NULL` terminated string of what the user typed in before hitting Enter. This is passed to the provided `parse_cmdline()` API function I have provided to you. The implementation can be found in `parse.c` and the API function declarations can be found in `parse.h`. The `parse_cmdline()` API returns a `Parse *`, which points to heap memory. The `Parse` structure is defined in `parse.h` as follows:

```
typedef struct {
    Task *tasks;         /* ordered list of tasks to pipe */
    int   ntasks;        /* # of tasks in the parse */

    char *infile;        /* filename of 'infile'  */
    char *outfile;       /* filename of 'outfile' */

    int background;      /* run process in background? */
    int invalid_syntax;  /* parse failed */
} Parse;
```

as is the `Task` structure:

```
typedef struct {
    char *cmd;
    char **argv;   /* NULL terminated array of strings */
} Task;
```

As you can see, the `Parse` data structure contains all of the parsed information from a command line with following anatomy:

```
command_1 [< infile] [| command_n]* [> outfile] [&]
```

where:

- Items in brackets [ ] are optional
- Items in starred brackets [ ]* are optional but can be repeated
- Non-bracketed items are required

In other words, I have done all the "hard work" for you. Your job will simply to be to take a `Parse` structure and implement all of the process creation and management logic generally provided by a shell. In order to make it obvious how to work with a `Parse` structure, I have provided the `parse_debug()` API function, which simply prints the contents of a `Parse` to the screen. **Your job, for this project, largely involves writing logic for the provided `execute_tasks()` function in `pssh.c`.** I highly recommend you start by simply getting the execution of single commands working.

Executing Single Commands

Naturally, this is a simple `fork()` and `exec()` problem. Get this working first. Also, remember that `exec()` is not a real function, but is rather a term used to refer to the family of `exec` functions: `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`, `execvpe()`. Read the manual page for exec to figure out which one you want to use!

```
~$ man 3 exec
```

Once your `pssh` is capable of executing simple commands with arguments, such as:

```
$ ls -l
```

you need to add support for input and output redirection using the `<` and `>` command line operators... just like in `bash`! This is actually easier than in seems. *Hint: you will need to modify the file descriptors tables.* Once you have this working, you will be able to do great things like:

```
$ ls -l > directory.txt
$ grep "some phrase" < somefile.txt
```

(Hint: `man 2 open` and `man 2 dup2`)

Great job! Next step is connecting multiple commands together with pipes!

Executing Multiple Pipelined Commands

This is actually very similar to what we did in lecture using the `dup2()` system call. In fact, it is nearly exactly the same – just far more practical. Unlike our example in lecture, you must be able to pipeline more than just two processes – you must pipeline all of the commands in the `tasks` array found in the `Parse` structure. You're going to need a `for-loop` – spend some time planning it out well before writing it!

Adding the Built-in Commands

Since it simply ends the `pssh` process, the built-in command `exit` is going to be the easiest. Implement that first. Now, the `which` command is going to be a little bit more complicated. I recommend looking at the `command_found()` function in `pssh.c` and use that as a working base. Also, be sure to read the manual page for the `access()` system call:

```
~$ man access
```

Keep in mind that, with the exception of `exit`, any built-in command should support input and output redirection. For example, the following should work:

```
$ which man > man_location.txt
$ cat man_location.txt
/usr/bin/man
```

This may sound more challenging than it actually is. Did you figure it out?

Building the Code

The provided code uses a `Makefile` to help build the code. All you need to do to build the program is run the `make` command within the source directory:

```
~/src/pssh$ ls
builtin.c  Makefile  parse.h
builtin.h  parse.c   pssh.c
~/src/pssh$ make
gcc -g -Wall -c builtin.c -o builtin.o
gcc -g -Wall -c parse.c -o parse.o
gcc -g -Wall -c pssh.c -o pssh.o
gcc  builtin.o  parse.o  pssh.o -Wall -lreadline -o pssh
~/src/pssh$ ls
builtin.c  Makefile  parse.o  pssh.o
builtin.h  parse.c   pssh
builtin.o  parse.h   pssh.c
```

and just like that, all of the necessary steps to produce the `pssh` executable will be automatically ran. If you want to delete all of the intermediate object files and the `pssh` executable, simply run `make clean` within the source directory:

```
~/src/pssh$ ls
builtin.c  Makefile  parse.o  pssh.o
builtin.h  parse.c   pssh
builtin.o  parse.h   pssh.c
~/src/pssh$ make clean
rm -f *.o
rm -f pssh
~/src/pssh$ ls
builtin.c  Makefile  parse.h
builtin.h  parse.c   pssh.c
```

Having Trouble Getting Started?

If you are having trouble getting started, remember that the main ideas here are manipulating process file descriptor tables and pipelining processes – that is, have the parent process connect the input and output of a series of sibling child processes together using pipes. You will implement this logic in `execute_tasks()` in the `pssh.c` source file. (Although, you will probably need to make small changes elsewhere in `pssh.c` and `builtin.c` to accommodate your specific approach, and that's okay!) In fact, writing a few small helper functions that each do a small, specific job well (and reliably) will probably make your code in `execute_tasks()` much easier to write and manage.

The most important thing to get right before starting is understanding the `Parse` data structure returned by `parse_cmdline()`. Start by enabling the parser's debugging output by setting `DEBUG_PARSE` to `1`, compile, and run `./pssh`. Now, try a few of the example commands given on Page 1 of this document to see the anatomy of the parse. Refer to `parse.h` while doing this.

Next, read the function `debug_parse()` in `parse.c` to see how the debug output was generated – this will show you how to work with the `Parse` structure that is passed in to `execute_tasks()`, which is where most of your code for this project will go.

Note: You do not need to modify anything in `parse.c` – that part is done. Reading it may be fun and educational, though.

Submitting Your Project

Once you have implemented all of the required features described in this document, submit your code by doing the following:

- Run `make clean` in your source directory. We must be able to build your shell from source (and we don't want your precompiled executables or intermediate object files). **If your code does not at the very least compile, you will receive a zero.**
- Create a zip file containing your code and a README.txt file.
- Name your zip file `pssh.zip`
- Upload your zip file using the Blackboard Learn submission link found on the course website.

Failure to follow these simple steps will result in your project not being graded.

Have fun with this project!