CMSC 216 Project #5 Fall 2010

A Simple Shell **Due: November 29, 2010, 6:00PM**

This project will have you write a simple shell, much like the one you have been using for all your work on the Linux Grace machines. You will use your knowledge of file operations and process control to effect file redirection, backgrounding, and program pipelines.

1 Procedure

1.1 Obtain the project files

We have supplied several files for your use in this project, in addition to the usual `.submit` file: (a) `parser.c`, which defines a function to parse a command line into a C structure that is easier to use in your program; (b) `parser_test.c`, a program that demonstrates the use of the `parse_line()` function; (c) `mysh.c`, a very simple shell program that only recognizes two commands: `exit` and `cd`; and (d) `execute.c`, which defines the function `execute_command_line()`. In addition, we also supply the header files `parser.h` and `execute.h`. All these files are contained in `~/216public/project5/project5.tgz`; you should copy this file to your home directory and extract the files in a similar manner to the procedure you followed in previous projects.

Your code for this project should be contained in your `~/216/project5` subdirectory, which is created by extracting files from the tarball you copied.

1.2 Create a `Makefile`

As for Project #3, you must create a `Makefile` that we will use to build your shell implementation. Section 2.1 lists the targets you are required to implement, as well as other requirements for your `Makefile`.

1.3 Implement and test the shell

You must implement your shell program by supplying the necessary code in `execute.c`. As you implement various features of the shell, you can test them by either interactively running `mysh` while typing in commands, or you can create text files with one command per line, and use redirection to feed them as input to your shell.

2 Specifications

2.1 `Makefile`

Your `Makefile` should be set up so that all programs are built using separate compilation (i.e., source files are turned into object files, which are then linked together in a separate step). All object files should be built using the `-ansi` and `-Wall` flags to `gcc`.

You must have the following targets in your `Makefile`:

1. `all`: make all executables
2. `clean`: delete all object files and executables
3. `parser.o`, `execute.o`, and `mysh.o`, the object files for the parsing code and shell code
4. `mysh`: the executable created by linking the parsing object file with the shell object files

We will use the `Makefile` you provide to build the public test executables on the submit server; if there is no `Makefile`, your shell program will not be built, and you will not receive credit for **any** tests.

2.2 The Shell

The shell you are to implement is a very basic shell, which does not have all of the functionality of the shell you use on a daily basis. It does, however, support three kinds of file redirection (input, truncated output, and appended output), backgrounding, and the pipelining of several programs into a single command.

To aid in writing your code, we have provided you with a function, `parse_line()`, which populates a structure to hold (a) string vectors for the individual commands that are part of the command line; (b) filenames for files that are to be redirected; and (c) several integer flags to indicate what kind of redirection is taking place, whether backgrounding is occurring, and how many programs are in the pipeline. All strings held by this function are pointers to various places in the string parameter that is passed in as an argument. That string parameter **is modified** by `parse_line()` so that it doesn't have to allocate space for copies of parts of the string, so callers must take that into account after calling the function. The prototype for this function is in `parser.h`.

You are given the beginnings of a shell program in `mysh.c` – your task is to implement the function called in the `main()` function, `execute_command_line()`, which is defined in `execute.c`. Once the full `mysh` program is linked together from the three object files, `mysh` should function as a normal shell (with some limitations as described below).

The syntax for the features we want you to implement is very similar to the shell syntax you should be familiar with from your experience working with the Unix environment. These are the specific features your shell must provide:

1. **File redirection**: by using the <, >, and >> tokens, a user should be able to redirect standard input and output in the same manner as is done in the `tcsh` and `bash` shells (i.e., redirect standard input, redirect standard output with truncation, and redirect standard output by appending, respectively).

   If a file is created by output redirection, the file should be created readable and writable for all users. The proper permission mask for doing that is defined in the `execute.h` file.

   In the case of a multi-program pipeline, file input redirection must be applied to the first program (before the first pipe character), and file output redirection be applied to the final program.

   Also, unlike the `tcsh` and `bash` shells, if both input and output are redirected for the same program, the request for input redirection must occur before the output redirection. This means this command, which is legal in `bash`, is not legal with `mysh`:

   ```
   cat - otherfile.in > file.out < file.in
   ```

   Instead, it must be written this way:

   ```
   cat - otherfile.in < file.in > file.out
   ```

2. **Backgrounding**: if a command line ends with an ampersand ("`&`"), then the shell should execute the program(s) and immediately return to the shell prompt, not waiting for the backgrounded program/pipeline to terminate. The backgrounded program/pipeline should continue to print its output to standard output/error as if it were not backgrounded. If standard input is not redirected from a file, then the process should have its standard input closed before exec'ing (this is different from normal shell behavior; however, normal shells provide foregrounding capabilities, which we are not requiring in this project). This means that some programs will have errors when run in the background, but that is the user's fault for not handling input properly, not yours.

3. **Piping**: if programs are separated by pipe characters ("`|`") on a command line, then the standard output of the program to the left of the pipe character is to be connected to the standard input of the program to the right of the pipe character, creating a pipeline.

   Notice that when running a pipeline in a shell such as `tcsh` or `bash`, the shell does not return to the prompt until the entire pipeline is completed (unless the pipeline is backgrounded). You will need to determine how to fork processes and then wait for them to cause similar behavior in your shell.

As you can see from the code in `mysh.c`, the shell prompts the user for a command, parses the command, and then attempts to execute the command (after taking care itself of a few built-in commands). To properly execute the command, you must use the `Command_line *` parameter to see which options are set, and perform the steps necessary to execute the command as requested.

Should you encounter any errors in executing the command, your shell **must not terminate** – children of the shell may terminate, but the code you write in `execute_command_line()` may not cause the parent (shell) process to terminate. A user should not cause the shell to die because he/she, for example, mistyped the name of a program to execute.

If any errors occur while executing a command, your shell (the parent process for all commands) must print to standard output the message "`Error executing command`" and a newline. Such errors include system call errors in your shell code or if any command exits with a non-zero status. The exit status of a pipeline of commands is defined to be the exit status of the last command in the pipeline.

3 Important Points and Hints

1. Be sure to wait for **all** child processes of your shell to complete; failing to do so will cause a collection of zombie processes to accumulate and tie up Grace system resources.
2. Also be careful that any fork loops you write terminate at some point, before a Grace system administrator has to kill them.
3. Backgrounding can be implemented using either a reaping loop in the shell, or by having the child process fork and then immediately exit, with the grandchild executing the requested program. A reaping loop requires using `waitpid()` with the `WNOHANG` option (and a `pid` of -1, as shown in the textbook – see figures 8.17 and 8.18, pages 727-728), while the use of the grandchild option requires you to take care in performing the execution and forks correctly. The choice between these methods is yours to make; you may also attempt other solutions, but they must work correctly.
4. Use the `parser_test` program to gain experience with how the command lines is parsed, and which options are set with various command syntax given as input to your shell.
5. The comments in `parser.h` may also help you to understand what the various fields of the `Command_line` struct represent.

4 Grading Criteria

Your project grade will be determined by the following formula:

Results of public tests	20%
Results of secret tests	60%
Code style grading	20%
Extra credit	5%

The public tests will be made available shortly after the project is released. You can find out your results on these tests by checking the submit server a few minutes after submitting your project. Secret tests, and their results, will not be released until after the project's late deadline has passed.

The style grading will take place after the late deadline, and will follow the same guidelines as used in Project #3.

Public tests for this project consist of input files to be read by your shell via standard input, and checked against the expected output, for example like this:

```
./mysh < public00.in | diff - public00.output
```

4.1 Extra credit

Once again, we offer extra credit to students who make an early submission that passes all public tests. You will receive 5 points of extra credit if you make a submission by 6PM on Friday, November 19, that passes all public tests for the project.

5 Submission

5.1 Deliverables

The only files we will grade are (a) your `Makefile`; and (b) `execute.c`, which contains your shell implementation. We will use our versions of all other files to build tests, so do not make any changes to other files.

5.2 Procedure

To limit the number of processes your shell implementation can create when you run it on the Grace machines, put the line `limit maxproc 20` in the `.cshrc.mine` file in your home directory.[1]

As for previous projects, executing "`submit`" in your project directory will submit your project. We once again encourage you to run public tests on Grace rather than submitting to the submit server and waiting for your submission to be evaluated; it is much faster for you to see your results if you run the tests yourself, and the submit server works much more quickly if the class makes fewer submissions.

6 Other Notes

6.1 Academic Integrity

As noted in the syllabus, any evidence of cheating will be referred to the Student Honor Council and may result in a grade of XF in this course. Submissions will be checked with an automated source code comparison tool to look for evidence of cheating.

6.2 Deadlines

Submission deadlines are strictly enforced by the submit server, and we will not extend them for events such as network outages. Extensions may be given on a case-by-case basis, but will likely only be granted in emergency cases. Therefore, you should start work on this project soon, as last-minute technical problems are not an excuse for missing either the on-time or late deadline.

[1] If you use `bash` instead of `tcsh`, the command is `ulimit -u 20`.