Project C-Shell

Due: October 28, 2019 at 11:59pm

1 Introduction **3**

2 Assignment Outline **3**
2.1 Minimum Requirements for Shell 2 4

3 Getting started **4**
3.1 Makefile 4
3.2 Roadmap 5

4 Parsing the Command Line **6**
4.1 Invalid Command-Line Input 7

5 Executing Shell Commands **7**
5.1 Built-In Shell Commands 7
5.2 Executing a Program 8
5.3 Files, File Descriptors, Terminal I/O 10
5.4 Prompt Format 12
5.5 Input and Output Redirection 13

6 Use of Library Functions **14**
6.1 Error Handling 15

7 Support **15**
7.1 Demo 16
7.2 Tester 16
7.3 Clean Up 17

8 GDB Tips for C Shell **17**
8.1 Following Child Processes 17
8.2 Examining Memory in GDB 18

9 Grading **18**

10 Handing In **19**

1 Introduction

It's almost Halloween, and you're taking some costume inspiration from your favorite tongue-twister: "she sells seashells by the seashore". Alas, once you arrive at the beach, you realise that you accidentally brought along your computer instead of your seashell-collecting gear. But you would hate for this to be a wasted trip. Maybe some C shells would do the trick…?

In this 2-part assignment, you will be writing your own "C" shell. A shell is typically used to allow users to run other programs in a friendly environment, often offering features such as command history and job control. Shells are also interpreters, running programs written using the shell's language (shell scripts).

This assignment serves as an exercise in C string parsing and an introduction to system calls. (You won't be implementing the system calls themselves, just calling them correctly. If you're interested in learning how syscalls are implemented, consider taking CS 167!)

2 Assignment Outline

Your task is as follows: your shell must display a prompt and wait until the user types in a line of input. It must then do some text parsing on the input and take the appropriate action. For example, some input is passed on to built-in shell commands, while other inputs specify external programs to be executed by your shell.

The built-in commands you will need to implement are:

- `cd <dir>` : changes the current working directory.
- `ln <src> <dest>` : makes a hard link to a file.
- `rm <file>` : remove something from a directory.
- `exit` : quit the shell.

Typically your shell should read from STDIN and write to STDOUT. However, the command line may contain input/output redirection tokens, in which case it must set up the appropriate files to deal with this.

As you know, users are far from perfect; a large part of this assignment will be supporting good error-checking while running your shell.

When in doubt about how your shell should behave given a particular input, consult the demo discussed in section 7.1.

Install the project stencil by running

```
cs0330_install shell_1
```

We ~highly~ recommend going through the following checklist in order to get the most out of your C Shell experience and **using gdb** if/when you run into issues (check out the GDB tips section!):

1. Ensure Makefile is up and running
2. Read & parse input as recommended in Parsing the Command Line
3. Implement built-in commands
4. Handle child processes (get fork & execv working)
5. Handle input/output redirection
6. Implement error handling for syscalls and bad user input

2.1 Minimum Requirements for Shell 2

Shell 1 is a project unto itself, but you will use most of your code for it again in next week's project, Shell 2.

For Shell 2, it is imperative that your code for Shell 1 can successfully **`fork`** and **`execv`** new child processes based on command-line input and the built-in **`cd`** command is functioning. Shell 2 does *not* require correct implementations of the built-in shell commands **`ln`**, **`rm`**, or **`exit`** or input/output redirection. In other words, you must **complete at least the first 2 tasks on the checklist (and also the `cd` built-in)** in the previous section before proceeding to Shell 2.

Here are some things to be aware of:

- A baseline Shell 1 implementation will *not* be released for work on Shell 2. All of the code you write for these two assignments will be your own.
- Late days used on Shell 1 will *not* also apply to Shell 2.

3 Getting started

In this section we'll outline the order in which you should tackle the different parts of this project. **Make sure each section is working before moving on to the next one.** One way to guarantee a VeryBadTime™ is to write the whole project in one go. ***Run and test incrementally.***

3.1 Makefile

For this assignment, we have given you an outline for a Makefile that you can use to compile your program. This outline consists of a list of flags that we will use to compile your shell when grading. You are responsible for handling any warnings that these flags produce. The Makefile also includes the target names that we would like you to use, but it does not include rules for any targets, and running **`make`** with this stencil Makefile will not compile your shell.

It is up to you to create a working Makefile for this assignment with what you learned from this week's lab. **We will be grading the Makefile you write on this assignment**, mainly for functionality and conciseness. Use variables wherever necessary to reduce repetition, and specify dependencies correctly so make knows to recompile executables if they've changed since the last make. Refer to the Makefile lab handout if you need help. **Please make sure not to use SHELL as a variable name in your Makefile.**

If you decide to create additional C files, you should make sure your Makefile correctly compiles them into a binary.

3.2 Roadmap

Now that you've got your Makefile working, here's one way to start tackling the rest of Shell 1:

1. **Parse the Command Line:**
    a. While how you parse user input is up to you, we have provided an example algorithm in Parsing the Command Line.
    b. Note that you MUST use the `read` system call to get user input (no **`scanf`**).
    c. Ensure that the program can handle incorrect input, with some potential cases to keep in mind:
        i. User can enter nothing and hit the return key (your terminal should simply ignore the input and start a new line in the terminal, as Linux systems do).
        ii. User should enter a supported command if there are any non-whitespace text.
        iii. User can enter redirection token at ANY point in a command (see 4 for details on redirection).
        iv. User cannot use any more than one of the same redirect in one line (i.e. one input, one output).
        v. User can enter any amount of whitespace between tokens
2. **Implementing `cd`, `rm`, `ln`, and `exit`:**
    a. For these, take a look at the `chdir`, `link`, and `unlink` commands (use `man`, or `man` pages online).
3. **Handle Child Processes:**
    a. To handle any other process, you will need to create a new process using the `fork` system command.
        i. This will create a new child process that is an exact replica of the parent process.
        ii. Make sure that the parent process waits for the child to complete (i.e. using the `wait` function)

   b. From there, you can use the `execv` function. This will replace this newly created child process with the program of your choosing.
      i. The two arguments to execv will be the full path to the program you wish to execute and an array of arguments for that program.
         1. To get the full path to a program, you may want to run:
            `which <command>`
4. **Handle Input/Output Redirects:**
   a. File redirection allows your shell to feed input to a user program from a file and direct its output into another file.
   b. There are three redirect symbols that your program will need to handle: `<` (redirect input), `>`, and `>>` (redirect output).
   c. Keep in mind that redirects are not guaranteed to appear in any particular order. A redirect can appear after or before a command.
5. **System Call Error Checking:**
   a. Almost all system calls return a value, which usually are used to determine if the system call was successful or not. You can use these values to do error checking on said calls.
      i. See the `man` pages for the respective commands for details on return values.

4 Parsing the Command Line

A significant part of your implementation will most likely be the command line parsing. Redirection symbols may appear anywhere on the command line, and the file name appears as the next word after the redirection symbol. **The way you parse your commands is completely up to you**, but the most straightforward approach is to use **`strtok()`**.

Symbols and words can be separated by any amount and combination of spaces and tabs, which can also appear before the first token and after the last.

When tokenizing the command line, the first word (if it is not a redirection symbol) will be the command, and each subsequent word will be an argument to the command. You will want to remove all redirection tokens and store the remaining ones in an array of **`char *`** (e.g. **`char *tokens[] = ... ;`**) for use with `execv`. Make sure that this array terminates with a null character for it to work with `execv`.

No input should ever crash your shell. You do not need to special case quotes (in most shells quotes would group several words into a single argument that contains white space). Make sure to check section 6 for the list of allowed built-in C functions you can use.

4.1 Invalid Command-Line Input

Be very careful to check for error conditions at all stages of command line parsing. When grading, we will run additional tests to check your code specifically for error handling on bad user input and will be deducting points if error messages aren't printed. **Refer to our demo if you are unsure what is valid vs. invalid input**.

Since the shell is controlled by a user, it is possible to receive bizarre input. For example, your shell should be able to handle all these cases (as well as many others):

```
33sh> /bin/cat < foo < gub
ERROR - Can't have two input redirects on one line.
33sh> /bin/cat <
ERROR - No redirection file specified.
33sh> > gub
ERROR - No command.
33sh> < bar /bin/cat
OK - Redirection can appear anywhere in the input.
33sh> [TAB]/bin/ls <[TAB] foo
OK - Any amount of whitespace is acceptable.
33sh> /bin/bug -p1 -p2 foobar
OK - Make sure parameters are parsed correctly.
```

You will not be held responsible if your input buffer is not big enough to handle user input. Use a large buffer length (e.g. 1024 bytes) and assume that the user will not enter more than that many characters.

You may assume that redirection characters are surrounded by whitespace.

5 Executing Shell Commands

5.1 Built-In Shell Commands

In addition to supporting the spawning of external programs, your shell will support a few built-in commands. When a user enters a built-in command into your shell, your shell should make the necessary system calls to handle the request and return control back to the user. The following is a list of the built-in commands your shell should provide.

- `cd <Bash command file paths for command>` : changes the current working directory.

- `ln <src> <dest>` : makes a hard link to a file.
- `rm <file>` : removes something from a directory.
- `exit` : quits the shell.

Note that we are only looking for the basic behavior of these commands. You do not need to implement flags to these commands such as **`rm -r`** or **`ln -s`**. You also do not need to support multiple arguments to **`rm`**, multiple commands on a single line, or shortcut arguments such as **`rm *`** or **`cd ~`.** Your shell should print out a descriptive error message if the user enters a malformed command. **For best performance with the testing script, your error message should match the one produced in the demo**, although you will not be penalized if these differ slightly.

5.1.1 UNIX System Calls for Built-In Functions

To implement the built-in commands, you will need to understand the functionality of several UNIX system calls. You can read the manual for these commands by running the shell command “**`man 2 <syscall>`**”. It is highly recommended that you read all the man pages for these syscalls before starting to implement built-in commands.

```
int chdir(const char *path);
int link(const char *existing, const char *new);
int unlink(const char *path);
```

Please note that redirecting a built-in command to its bash file is **not** a valid implementation. You must implement the functionality yourselves, using the syscalls mentioned above.

5.2 Executing a Program

When a UNIX process executes another program, the process replaces itself with the new program entirely. As a result, in order to continue running, your shell must defer the task of executing another program to another process. Below is a list of system calls, functions, and shell commands useful to this project, related to executing a program:

5.2.1 `fork()`

```
pid_t fork(void)
```

First, you’ll need to create a new process. This must be done using the system call **`fork()`**, which creates a new “child” process which is an exact replica of the “parent” (the process which executed the system call). This child process begins execution at the point where the call to **`fork()`** returns. **`fork()`** returns 0 to the child process, and the child’s process ID (abbreviated pid) to the parent.

5.2.2 execv()

```
int execv(const char *filename, char *const argv[])
```

To actually execute a program, use the library function **`execv()`**. Because **`execv()`** replaces the entire process image with that of the new program, this function never returns if it is successful. Its arguments include **`filename,`** the full path to the program to be executed, and **`argv`**, a null-terminated[1] argument vector. Note that **`argv[0]`** MUST be the binary name (the final path component of **`filepath`**), NOT the full path to the program (which means you will have to do some processing in constructing **`argv[0]`** from **`filename`**).

As an example, the shell command **`/bin/echo 'Hello world!'`** would have an argv that looks like this:

```
char *argv[4];
argv[0] = "echo";
argv[1] = "Hello";
argv[2] = "world!";
argv[3] = NULL;
```

See the **`which`** section to figure out how to get the full path to the program.

Here is an example of forking and executing **`/bin/ls`**, with error checking:

```
if (!fork()) {
    /* now in child process */
    char *argv[] = {"ls", NULL};
    execv("/bin/ls", argv);

    /* we won't get here unless execv failed */
    perror("execv");
    /* hint: man perror */

    exit(1);
}
/* parent process continues to run code out here */
```

5.2.3 wait()

[1] An array for which argv[argc] is NULL, if argc is the number of entries in argv.

```
pid_t wait(int *status)
```

Your shell should wait for the executed command to finish before displaying a new prompt and reading further input. To do this, you can use the **`wait()`** system call, which suspends execution of the calling process until a child process changes state (such as by terminating). If the status argument to wait is non-zero, details about that change of state will be stored in the memory location addressed by status. You don't need that information in this assignment - you should pass **`wait`** the null pointer (0), which will tell it not to store any data. Type **`man 2 wait`** into a terminal for further information.

5.3 Files, File Descriptors, Terminal I/O

You have previously read from and written to files using the **`FILE`** struct and functions such as **`fopen()`** and **`fclose()`**. This struct and these functions provide a high-level abstraction for how file input and output actually works, obscuring lower-level notions such as file descriptors and system calls. In this assignment, you will be performing input using file descriptors and system calls instead of the high-level abstraction of **`fopen()`** and **`fclose()`** - though we are allowing you to use **`printf()`** and **`fprintf()`** as you normally would for output.

5.3.1 File Descriptors

At a lower level, file input and output is performed using *file descriptors*. A file descriptor is simply an integer which the operating system maps to a file location. The kernel (AKA the core of the computer's operating system) maintains a list of file descriptors and their file mappings for each process. Consequently, processes do not directly access files using **`FILE`** structs but rather through the kernel by using file descriptors and low-level system calls.

Subprocesses inherit open files and their corresponding file descriptors from their parent process. As a result, processes started from within a normal UNIX shell inherit three open files: **`stdin`**, **`stdout`**, and **`stderr`**, which are assigned file descriptors **`0`**, **`1`**, and **`2`**[2] respectively. Since your shell will be run from within the system's built-in shell, it inherits these file descriptors; processes executed within your shell will then also inherit them. As a result, whenever your shell or any process executed within it writes characters to file descriptor **`1`** (the descriptor corresponding to **`stdout`**), those characters will appear in the terminal window.

5.3.2 `open()`

```
    int open(const char *pathname, int flags, mode_t mode)
```

[2] The header file `unistd.h` defines macros `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` which correspond to those file descriptors. This is useful for making code more readable.

The **`open()`** system call opens a file for reading or writing, located at the relative (starting from the process working directory) or absolute (starting from the root directory, /) pathname, and returns a new file descriptor which maps to that file.

The other arguments for this system call are bit vectors which indicate how the file should be opened. In particular, **`flags`** indicates both status flags and access modes, allowing the user to determine the behavior of the new file descriptor. **`mode`** is used to determine the default permissions of the file if it must be created.

We recommend looking at the man pages (**`man 2 open`**) for more information.

File descriptors are opened lowest-first; that is, **`open()`** returns the lowest-numbered file descriptor available (i.e. currently not open) for the calling process. On an error, **`open()`** returns −1.

5.3.3 close()

```
int close(int fd)
```

`close()` closes an open file descriptor, which allows it to be reopened and reused later in the life of the calling process. If no other file descriptors of the calling process map to the same file, any system resources associated with that file are freed. **`close()`** returns 0 on success and −1 on error.

5.3.4 read()

```
ssize_t read(int fd, void *buf, size_t count)
```

`read()` reads up to **`count`** bytes from the given file descriptor (**`fd`**) into the buffer pointed to by **`buf`**. It returns the number of characters read and advances the file position by that many bytes, or returns −1 if an error occurred. **Check and use this return value**. It is otherwise impossible to safely use the buffer contents. Note that read **does not null terminate the buffer**.

`read()` waits for input: it does not return until there is data available for reading. When reading from standard input, **`read()`** returns when the user types **`enter`** or **`CTRL-D`**. These situations can be distinguished by examining the contents of the buffer: typing **`enter`** causes a new-line character (**`\n`**) to be written at the end of the line, whereas typing **`CTRL-D`** does not cause any special character to appear in the buffer. You are allowed to assume that an input command ends with **`\n`,** as the demo does. Also, be sure to terminate your buffer with '/0' after each iteration! Otherwise, your buffer will sometimes contain garbage characters.

If a user types **`CTRL-D`** on a line by itself, **`read`** will return 0, indicating that no more data is available to be read—a condition called *end of file* (EOF). In this case, **your shell should exit**.

NOTE: *The values **`0`**, **`NULL`**, and **`\0`** are interchangeable and completely equivalent.*

5.3.5 `write()`

```
ssize_t write(int fd, const void *buf, size_t count)
```

`write()` writes up to **`count`** bytes from the buffer pointed to by **`buf`** to the given file descriptor (**`fd`**). It returns the number of bytes successfully written, or `-1` on an error.

While this is the lowest level and safest system call we can use to write to `STDOUT` and `STDERR`, you are not required to use **`write()`** for output on this assignment but may instead use **`printf()`** and **`fprintf()`** to write to `STDOUT` and `STDERR`. You will, however, have to use **`read()`** to read input.

5.3.6 `printf()`

```
int printf(const char *format, ...)
```

You're already familiar with **`printf()`** writing formatted output to `STDOUT`. The primary difference is that a file descriptor doesn't need to be specified when using **`printf()`**; its default file descriptor is `STDOUT`.

Note: if you're using **`printf()`** to write a string that doesn't end in a newline (hint: your prompt), you must use **`fflush(stdout)`** after **`printf()`** to actually write your output to the terminal.

5.4 Prompt Format

While the contents of your shell's prompt are up to you, you must implement a particular feature in order to make your shell easier to grade. Specifically, you should surround the statement that prints your prompt with the C preprocessor directives **`#ifdef PROMPT`** and **`#endif`**, which will cause the compiler to include anything in between the two directives only when the **`PROMPT`** macro is defined.

For example, if you print your prompt with the statement
`printf("33sh> ");` you would replace it with the following:

```
#ifdef PROMPT
```

```
    if (printf("33sh> ") < 0) {
        /* handle a write error */
    }
    #endif
```

Note: If you choose to use **`printf()`** to write your prompts, and not **`write()`**, there is an additional step you will have to take to get the prompt to show up in the terminal, because the prompt does not end in a newline. See the **`printf()`** section for more details.

Your Makefile should compile two different versions of your shell program: *`33sh`*, which compiles with `PROMPT` defined, and *`33noprompt`*, which compiles without `PROMPT` defined. If you do not remember how to compile your program with a macro defined, refer back to the maze solver Makefile or the Makefiles lab.

Any other user-defined writes to standard output (i.e. debugging printouts) from your shell should also be enclosed with the **`#ifdef PROMPT`** and **`#endif`** directives. Otherwise, the testing suite will not run correctly with your shell.

5.5 Input and Output Redirection

Most shells allow the user to redirect the input and output of a program, either into a file or through a *pipe* (a form of interprocess communication). For example, bash terminals allow you to send a program input from a file using <, send output from a program to a file using > or >>, and chain the output of a program to the input of another using |. **Your shell will be responsible for redirecting the input and output of a program but not for chaining multiple programs together.**

5.5.1 File Redirection

File redirection allows your shell to feed input to a user program from a file and direct its output into another file. You do not need to support redirection for built-in commands. **File redirection should be the last shell functionality you implement**.

The redirection symbols ('<', '>', and '>>') can appear anywhere within a command in any order. For instance, the command **`/bin/echo hello > output.txt`** will write the results of **`/bin/echo hello`** to a new text file **`output.txt`**, and the command **`/bin/echo > output.txt hello`** should do the same thing, even though the redirection symbol is in a different spot. You can visit the Linux's Redirection Definition page for specific examples and additional details of redirection.

- `< [path]` - Use file `[path]` as standard input (file descriptor 0).

- `> [path]` - Use file `[path]` as standard output (file descriptor 1). If the file does not exist, it is created; otherwise, it is truncated to zero length. (See the description of the `O_CREAT` and `O_TRUNC` flags in the `open(2)` man page.)
- `>> [path]` - Use file `[path]` as standard output. If the file does not exist, it is created; otherwise, output is appended to the end of it. (See the description of the `O_APPEND` flag in the `open(2)` man page.)

Your shell should also support error checking for input and output redirection. For example, if the shell fails to create the file to which output should be redirected, the shell must report this error and abort execution of the specified program. Additionally, it is illegal to redirect input or output twice (although it is perfectly legal to redirect input and redirect output). You can experiment with I/O redirection in the demo, which should serve as a model for the expected functionality of your shell.

5.5.2 Redirecting a File Descriptor

To make a program executed by your child process read input from or write output to a specific file, rather than use the default **`stdin`** and **`stdout`**, we have to redirect the **`stdin`** and **`stdout`** file descriptors to point to the specified input and output files. Luckily, the kernel's default behavior provides an elegant solution to this problem: when a file is opened, the kernel returns the smallest file descriptor available, regardless of its traditional association. Thus, if we close file descriptor 1 (**`stdout`**) and then open a file on disk, that file will be assigned file descriptor 1. Then, when our program writes to file descriptor 1, it will be writing to the file we've opened rather than **`stdout`** (which traditionally corresponds to file descriptor 1 by default).

For the purposes of this project, we won't be concerned with restoring the original file descriptors for **`stdout`** and **`stdin`** in the child process as it won't affect your shell. If you're interested in the technically safer (but more complex) way to redirect files, check out the **`dup()`** and **`dup2() man`** pages.

6 Use of Library Functions

You should use the **`read()`** system call to read from file descriptors **`STDIN_FILENO`** (a macro defined as 0), **`STDOUT_FILENO`** (1), and **`STDERR_FILENO`** (2), which correspond to the file streams for standard input, standard output, and standard error respectively. You should use the **`write()`** system call to write to **`STDOUT_FILENO`** or **`STDERR_FILENO`** OR the higher level non-system calls **`printf()`** (which doesn't require a specified file descriptor) and **`fprintf()`**.

You may use any syscall. Specifically, a system call is any function that can be accessed by using the shell command **`man 2 <function>`**. Do not use floating point numbers. If you have any questions about functions that you are able to use, please post your question on Piazza.

In order to avoid confusion, here is a list of **allowed** non-syscall functions. While use of some of these functions would be helpful in many implementations, **it is by no means required**.

`Memory-Related:`

`memset()`	`memmove()`	`memchr()`	`memcmp()`
`memcpy()`			

`Strings Manipulation:`

`str(n)cat()`	`tolower()`	`strtol()`	`isalnum()`
`isalpha()`	`iscntrl()`	`isdigit()`	`islower()`
`isprint()`	`ispunct()`	`isspace()`	`isxdigit()`
`str(n)cat()`	`str(n)cmp()`	`str(n)cpy()`	`strtol()`
`isgraph()`	`isupper()`	`strlen()`	`strpbrk()`
`strstr()`	`strtok()`	`str(r)chr()`	`str(c)spn()`
`toupper()`	`atoi()`		

`Error-Handling:`

`perror()`	`assert()`	`strerror()`

`Output:`

`fflush()`	`printf()`	`(v)s(n)printf()`	`fprintf()`

`Misc:`

`exit()`	`execv()`	`opendir()`	`readdir()`
`closedir()`			

6.1 Error Handling

You are responsible for dealing with errors whenever you use the allowed system calls or **`printf/fprintf()`**. Any function that returns or sets an error value should also be error checked, such as **`execv()`** and **`wait()`**. As this could get repetitive, you may want to make helper functions that will handle error-checking for you for frequently-used functions such as **`printf()`**. As with Maze, you are not required to error-check **`fprintf()`** when it is used to print an error message.

7 Support

We are providing you with a demo shell program and an automated testing program to use as you work on this project.

7.1 Demo

We have provided a demo implementation of Shell 1 to show you the expected behavior of the program. This is the version with the prompt:

```
/course/cs0330/bin/cs0330_shell_1_demo
```

and this is the one without

```
/course/cs0330/bin/cs0330_noprompt_shell_1_demo
```

Make sure you create an implementation both with and without a prompt, as previously described.

You should use the demo implementation as a reference for how edge cases you think of should be handled. The demo shell differs in some respects from the **bash** you know and love. **Where they differ, emulate the demo rather than bash or another shell.** For example, the **cd** command, when run without arguments in **bash**, changes directories to the user's home directory. Since you will have no way of knowing the user's home directory location, your **cd** implementation should emulate the demo's behavior for this case.

7.2 Tester

We have provided a test suite and testing program to test your shell. There are about 40 tests in **/course/cs0330/pub/shell_1**. The tester program will run some input through your shell, and then compare the output of your shell to the expected output. Each of the tests that this script will run represents input that C Shell should handle, either printing out an appropriate error or the output of a command, depending on the test. To use this script, run

```
cs0330_shell_1_test -s 33noprompt -u /course/cs0330/pub/shell_1
```

You must run the tester with the "no prompt" version of your shell - the extra characters printed by the prompt version will cause the test suite to fail. Please also note that if your **33noprompt** executable is not in your current directory, you will need to provide the fully-qualified path to the executable.

You can also run a single test by providing **`-t /course/cs0330/pub/shell_1/<testname>`** instead of the **`-u`** option.

Each test is a directory containing an input file, and an output and error file corresponding to the expected output of **`stdout`** and **`stderr`** respectively. Note that while most tests have their output hardcoded in their **`output`** file, some have this file generated at run time by a script called **`setup`**, also in the same folder. This shouldn't matter to you while working on this project, except if you are debugging an individual test failure where it would be useful to examine the expected outputs of the test. In these cases, make sure to look at **`setup`** so that you can see exactly how the output is constructed, if it differs from the hardcoded **`output`** file. When you run a test case, the **`setup`** is run first, and then commands in **`input`** are piped into your shell (the commands will run in your shell), and then the tester checks if the output from your shell matches the content of the **`output`**.

The tester has some other options as well — run **`cs0330_shell_1_test -h`** to view.

If every test seems to be failing, your shell is likely printing extra information to **`stdout`** and/or **`stderr`**. Use the **`-v`** (verbose) option to check which. Also, each test is run with a 3-second timeout. If that seems to be happening for all of your tests, then your shell may not exit when it reaches **`EOF`** (when **`read()`** returns 0). Please make sure that this happens, since otherwise no test will pass.

The testing script provided is fairly comprehensive, but we strongly recommend you do additional testing.

7.2.1 `which`

For additional testing, you may find the `which` command helpful:

```
which <program name>
```

In order to execute a program in your shell, you will need that program's full pathname. You will not be able to use only a shortcut, as you would in a bash terminal for programs such as **`ls, cat, xpdf, gedit`**, etc. To execute these programs from your shell, you must enter **`/bin/cat, /usr/bin/xpdf, /usr/bin/gedit`**, and so on. To find the full pathname for any arbitrary program, use **`which`**.

Example usage:

```
$ which cat
/bin/cat
$ which gedit
/usr/bin/gedit
```

For more information, see the **man** page for **which**. You can even use **which** in your shell, once you have determined its full path (type **which which** in a system terminal to find its full path)!

NOTE: You do not need to implement **which** for this assignment! It is described here as a resource to find full pathnames of programs that can be executed in your shell.

7.3 Clean Up

It is important that you run **cs0330_cleanup_shell** (located in **/course/cs0330/bin**) every so often when working on your shell project. This kills any zombie processes, which if left running will eat up the computer's resources. This will be even more important when working on Shell 2.

8 GDB Tips for C Shell

As always, we recommend using GDB to help debug your project. Check out the GDB cheatsheet on the home page for more info!

8.1 Following Child Processes

When debugging your code to execute programs in C Shell it may be helpful to use GDB to verify that the programs are starting correctly. It's important to note that by default, GDB won't follow child processes started with **fork()**. This means that if you set a breakpoint on a line that executes in the forked process (i.e. to make sure the arguments to **execv()** are formatted correctly), GDB won't break on that line.

However, you can change this behavior by running the GDB command **set follow-fork-mode child**. This tells GDB to debug the child process after a fork, and leave the parent process to run unimpeded. After setting this, GDB will break on breakpoints you set within a forked process. For more information, run **help set follow-fork-mode** within GDB.

8.2 Examining Memory in GDB

As you work on your command parsing logic, it may be helpful to use GDB to peek at an area of memory in your program, for instance the input buffer as you work with it to parse out the necessary tokens.

The simplest way of determining the value of a variable or expression in GDB is **[p]rint <expression>**, but sometimes you will want more control over how memory is examined. In these situations, the **x** command may be helpful.

The **x** command (short for “examine”) is used to view a portion of memory starting at a specified address using the syntax **`x/(number)(format) <address>`**. For example, if you want to examine the first 20 characters after a given address, use **`x/20c <address>`** . If instead you want to examine the first 3 strings after a given address (remember that a string continues until the null character is encountered), use **`x/3s <address>`**.

Other useful format codes include **`d`** for an integer displayed in signed decimal, **`x`** for an integer displayed in hexadecimal, and **`t`** for an integer displayed in binary.

Note that the amount of memory displayed varies depending on the size of the specified format. **`x/4c <address>`** will print the first 4 characters after the given address, examining exactly 4 bytes of memory. **`x/4d <address>`** will print the first 4 signed integers after the given address, however this will examine exactly 16 bytes of memory (assuming the machine uses 4 byte integers).

9 Grading

Your grade for the first part of the shell project will be determined by the following categories, in order of precedence:

- *Functionality*: your shell should produce correct output.
- *Code Correctness*: your Makefile should work and your code should compile without warnings. Your code should be free of memory leaks and system calls should be used correctly. You must abide by the restrictions on library functions imposed in section 4—you *will* be penalized for using disallowed functions.
- *Error checking*: your shell should perform error checking on its input and display appropriate, informative error messages when any error occurs. Error messages should be written to standard error rather than standard output. Make sure you check the return value of each system call you use and handle errors accordingly.
- *Style*: your code will be evaluated for its style. Don’t forget to run the formatter before handing in!

10 Handing In

To summarize, here is a list of features that a fully functioning shell would support:

- Continuously reads input from the user until it receives **`EOF`** (`Ctrl-D`)
- Executes programs and passes the appropriate arguments to those programs
- Supports 4 built in commands (**`cd, rm, ln, exit`**)
- Supports 3 file redirection symbols (**`<, >, >>`**), including both input and output redirection in the same line.
- Extensive error checking

Before handing in your project, be sure to run this script to reformat your code to be consistent with the style guidelines:

```
cs0330_reformat sh.c
```

You can also run this script on any other .h or .c files you have. Just be sure you don't run it on other types of files, as this formatter is only meant for .c and .h files.
To hand in the first part of your shell, run

```
cs0330_handin shell_1
```

from your project working directory. You must at a minimum hand in all of your code, the Makefile used to compile your program, and a README documenting the structure of your program, any bugs you have in your code, any extra features you added, and how to compile it.

If you wish to change your handin, you can do so by re-running the handin script.

Important note: *If you have handed in but plan to hand in again after we start grading, in addition to running the regular handin script (*`cs0330_handin shell_1`*), you must run* `cs0330_grade_me_late shell_1` *to inform us not to start grading you yet. You must run the script by the time we start grading on-time handins (11/3 at 2:00pm).*

If you do not run this script, the TAs will proceed to grade whatever you have already handed in, and you will receive a grade report with the rest of the class that is based on the code you handed in before we started grading.

If something changes, you can run the script with the `--undo` flag (before the project deadline) to tell us to grade you on-time and with the `--info` flag to check if you're currently on the list for late grading.