# Project Sea Shell

*Due: November 1, 2016*

## 1 Introduction

Dory loves collecting shells from the bottom of the ocean, however due to her short term memory loss she's misplaced all her shells! To start replenishing her collection, Dory realizes she must make a shell of her own.

In this 2-part assignment, you will also 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).

For your shell you will have the same basic functionality as the shells you are used to working in (e.g. `bash`, `csh`, `zsh`), meaning it will read commands from the user and execute those commands. The shell will also provide a few built-in commands, such as `cd`, as well as some basic features such as file redirection and (eventually) job control.

This assignment serves as an exercise in C string parsing and an introduction to *system calls*.

## 2 Assignment

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.

Additionally, the command line may contain some special characters which will correspond to file redirection. The shell 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.

Install the project stencil by running

```
cs033_install shell_1
```

### 2.1 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 excutables if they've changed since the last `make`. Refer to the Makefile lab handout if you need help.

Moving forward in CS033, we will, whenever reasonable, not be providing a Makefile, so it is a good idea to get in the practice of writing your own.

## 2.2 The File System

Crucial to understanding how your shell will work is a working knowledge of the UNIX Virtual File System model.

In the VFS model, there is a root file system denoted as */*, and zero or more mounted file systems which reside at mount points, like */dev*. All file systems expose an internal structure of directories and files. Within the root file system there might be subdirectories such as */bin*, */home*, */home/jcarberr*, and */home/jcarberr/src*. There are also files within these directories, like *sh.c* and *README*. Mounted file systems behave just like root file systems, except that names of files within the file system are prefixed with the mount point.

The effect of all this is to abstract the particular way of accessing a file (the on-disk structure) from the fact that a file exists. In fact, some file systems might have no on-disk structure at all, and simply provide names that behave like files for other purposes. For instance, files in */proc* are not really stored anywhere; they simply provide a file interface to kernel data structures.

## 2.3 Files, File Descriptors, Terminal I/O

- `int open(const char *pathname, int flags, mode_t mode)`
- `int close(int fd)`
- `ssize_t read(int fd, void *buf, size_t count)`
- `ssize_t write(int fd, const void *buf, size_t count)`

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 and output using file descriptors and system calls instead of the high-level abstraction.

### 2.3.1 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 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[1] 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.

#### 2.3.2 `open()`

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

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.

#### 2.3.3 `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.

#### 2.3.4 `read()`

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

`read()` reads up to `count` bytes from the given file descriptor 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.

`read()` blocks waiting 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.

[1] 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.

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*. In this case, **your shell should exit**.

#### 2.3.5 `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. It returns the number of bytes successfully written, or $-1$ on an error.

### 2.4 Executing a Program

- `pid_t fork(void)`
- `int execv(const char *filename, char *const argv[])`
- `pid_t wait(int *status)`
- `which`

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.

Since your shell knows nothing of the behavior of any process but its own, 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. This allows the parent and child processes to branch off and execute different tasks, which happens to be exactly the behavior needed here.

To actually execute a process, use the system call `execv()`, which begins the execution of a new program. Since doing so replaces the entire process image with that of the new program, this function never returns if it is successful. `execv()` takes as arguments `filename`, which is the path to the program to be executed, and a null-terminated[2] argument vector `argv`. The final path component of `filename` and `argv[0]` should be the same. Note that this 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;
```

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

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

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

    /* we won't get here unless execv failed */
    char msg_buf[128];
    if (errno == ENOENT) {
        int num_chars = sprintf(msg_buf, "sh: command not found: %s\n", argv[0]);
        if (write(STDERR_FILENO, msg_buf, num_chars) < 0) {
            /* handle a write error */
        }
    } else {
        int num_chars = sprintf(msg_buf, "sh: execution of %s failed: %s\n",
                                argv[0], strerror(errno));
        if (write(STDERR_FILENO, msg_buf, num_chars) < 0) {
            /* handle a write error */
        }
    }
    exit(1);
}
/* parent process continues to run code out here */
```

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 - if you pass `wait` the null pointer (`0`) then it will not store any data. Type `man 2 wait` into a terminal for further information.

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)!

## 2.5 Built-In Shell Commands

In addition to supporting the spawning of external programs, your shell will support a few built-in commands. When a built-in command is input, 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 <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.

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.

### 2.5.1 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);
```

## 2.6 Prompt Format

While the contents of your shell's prompt are up to you, we ask that you 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 `write(STDOUT_FILENO, "33sh> \n", 7);`, you would replace it with the following:

```
#ifdef PROMPT
if (write(STDOUT_FILENO, "33sh> \n", 7) < 0) {
    /* handle a write error */
}
#endif
```

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.

Any other writes to standard output from your shell should also be enclosed with the `#ifdef PROMPT` and `#endif` directives.

## 2.7 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.

### 2.7.1 Redirecting a File Descriptor

An open file descriptor is associated with a single file, and can't readily be "switched" to another, which poses a problem when trying to redirect input or output, which are already associated with file descriptors. However, there is 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 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 corresponds to file descriptor 1 by default). Nothing special must be done with the descriptor once it's been closed and re-opened.

The system call `dup2()`[3] encapsulates several of these steps.

```
int dup2(int oldfd, int newfd)
```

This system call points `oldfd` and `newfd` to the same file. It first closes `newfd`, and then duplicates (hence the name of the system call) `oldfd` with `newfd` as the copy. After a successful call to `dup2()`, `oldfd` and `newfd` refer to the same file; since only one of those file descriptors is needed, the other (which should be `oldfd` for `dup2()` to have been meaningful) can be safely closed.

### 2.7.2 File Redirection

File redirection allows your shell to feed input to a user program from a file and direct its output into another file. Here is a summary from the `sh` man page.

> A command's input and output may be redirected using a special notation interpreted by the shell. (You do not need to support redirection for built-in commands.) The following may appear anywhere in a simple-command or may precede or follow a command and will not be passed on as arguments to the invoked command.

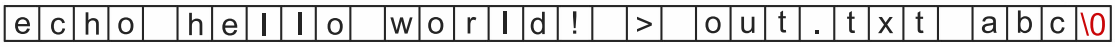
[3]There is also a system call `dup()` and even `dup3()`. Each of these system calls has a similar use with slight differences; here, `dup2()` is the most useful.

- `< <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.)

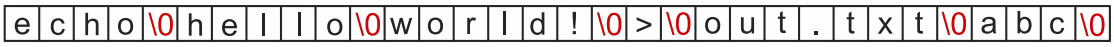
Your shell should support input and output redirection with error checking. For example, if the shell fails to create the file to which output should be redirected, the shell must report this error and abort execution of the specified program. Additionally, it is illegal to redirect input or output twice (although it is perfectly legal to redirect input and redirect output). You can experiment with I/O redirection in `bash`[4], which should serve as a model for the expected functionality of your shell.

## 3 Parsing the Command Line

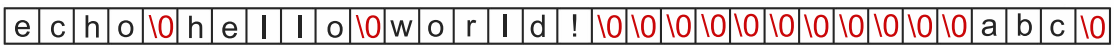
A significant part of your implementation will most likely be the command line parsing. Redirection symbols may appear anywhere on the command line, and the file name appears as the next word after the redirection symbol. One algorithm for parsing the command line is as follows:



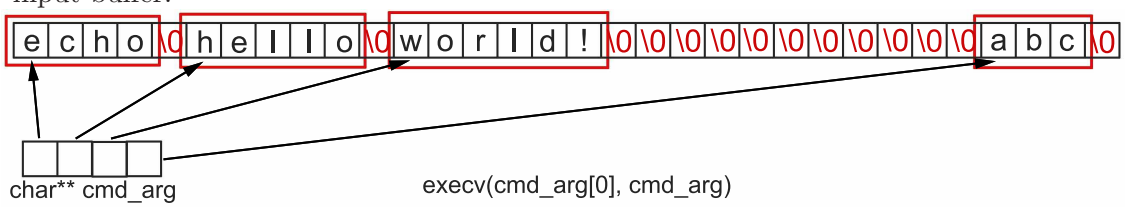

1. Split the line into words. The first word will be the command, and each subsequent word will be an argument to the command. Be sure to include the null characters so that `execv` can be given an array of `char *` and be still be able to find where each token ends.




2. Scan through the line for redirection symbols, keeping track of the input and output file names if they exist. Remove all traces of redirection from the command line (i.e. replace the relevant characters with the null character or spaces). Check for errors such as multiple redirection or missing file names (i.e. a redirection token that is not followed by a file name) at this point.




3. Construct the `char *` array so that each index points to a character array from the original input buffer.




[4] `bash` shells are the default shell of the terminals you open while logged in to the department. Other shells exist and may have different conventions regarding I/O redirection, among other things.

Symbols and words are separated by one or more spaces or tabs. Your shell must allow any number and combination of spaces and tabs when whitespace is required. Your shell must also support any number of spaces or tabs before the first token of the line, and after the last token.

Redirection characters will always be separated from arguments by spaces or tabs; they will not be immediately adjacent to the next or previous word. Special characters such as control characters should be treated just like alphanumeric characters and should not 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). Note that you are allowed to adapt and reuse any of the functions you wrote for lab 5.

### 3.1 Error Checking

Be very careful to check for error conditions at all stages of command line parsing. 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 errors (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. Note that in future assignments you will be responsible for handling similar error cases.

You may assume that redirection characters are surrounded by whitespace.

## 4 Use of Library Functions

You should use the `read()` and `write()` system calls to read and write 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. Do *not* use the I/O streams `stdin`, `stdout`, or `stderr`, as part of the purpose of this assignment is to learn about and use C system calls.

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 email the TAs.

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

| | | | |
|---|---|---|---|
| `assert()` | `isprint()` | `memmove()` | `str(r)chr()` |
| `exit()` | `ispunct()` | `memset()` | `strerror()` |
| `atoi()` | `isspace()` | `opendir()` | |
| `strtol()` | `isupper()` | `readdir()` | `strlen()` |
| `strtoll()` | `isxdigit()` | `closedir()` | `strpbrk()` |
| `isalnum()` | `malloc()` | `perror()` | |
| `isalpha()` | `free()` | `(v)s(n)printf()` | `strstr()` |
| `iscntrl()` | `realloc()` | `str(c)spn()` | `strtok()` |
| `isdigit()` | `memchr()` | `str(n)cat()` | |
| `isgraph()` | `memcmp()` | `str(n)cmp()` | `tolower()` |
| `islower()` | `memcpy()` | `str(n)cpy()` | `toupper()` |

### 4.1 Error handling

You are responsible for dealing with errors whenever you use the allowed system calls. For instance, you need to check the output of `write()` whenever you use it your code, to ensure that the system wrote the correct number of bytes. As this could get repetitive, you may want to make a helper function for this to do error checking whenever you need to `write()` output.

## 5 Support

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

### 5.1 Demo

We have provided a demo implementation of Sea Shell to show you the expected behavior of the program. It is located in `/course/cs033/bin/` with the name `cs033_shell_1_demo`. There is also a no prompt version of the demo which is in `/course/cs033/bin/` with the name `cs033_noprompt_shell_1_demo`. You do not need to give it any arguments to run. 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.

### 5.2 Tester

We have provided a test suite and testing program to test your shell. There are about 40 tests in `/course/cs033/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 Sea Shell should handle, either printing out an appropriate error or the output of a command, depending on the test. To use this script, run

```
cs033_shell_1_test -s <33noprompt> -u /course/cs033/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/cs033/pub/shell_1/<test>` 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 be 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 `cs033_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` 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.

## 6 GDB Tips for Sea Shell

### 6.1 Following Child Processes

When debugging your code to execute programs in Sea 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.

### 6.2 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).

## 7 Minimum Requirements for Conch Shell

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

For Conch Shell, it is imperative that your code for Sea Shell can successfully `fork` and `execv` new child processes based on command-line input. Conch Shell does *not* require correct implementations of the built-in shell commands or input/output redirection. Make sure you complete the other parts of the assignment before proceeding.

Here are some things to be aware of:

- A baseline Sea Shell implementation will *not* be released for work on Conch Shell. All of the code you write for these two assignments will be your own.
- Late days used on Sea Shell will *not* also apply to Conch Shell.
- If your Sea Shell project contains errors, you may receive up to 50% of the points lost if those mistakes are fixed by the time of your Conch Shell handin. To request these points back, please email the Head TAs when you turn in your Conch Shell.

# 8 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.
- *Correct use of system calls*: make sure you check the return value of each system call you use and handle errors accordingly. 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.
- *Style*: your code will be evaluated for its style.

# 9 Handing In

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

- Continously 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

To hand in the first part of your shell, run

```
cs033_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. Only your most recent handin will be graded.