

Project 3

Due Feb 20 by 11:59pm **Points** 16 **Available** until Feb 20 at 11:59pm

This assignment was locked Feb 20 at 11:59pm.

CS537 Spring 2023, Project 3

Updates

- Removed references to batch mode
- Updated spec document to clarify that you do not need to verify program names, and that on multi command lines, run until first failure then stop.
- Added another example command to clarify what pipes need to support

Administrivia

- **Due Date** by February 20th, 2023 at 11:59 PM
- Questions: We will be using Piazza for all questions.
- Collaboration: The assignment has to be done by yourself. Copying code (from others) is considered cheating. [Read this \(http://pages.cs.wisc.edu/~remzi/Courses/537/Spring2018/dontcheat.html\)](http://pages.cs.wisc.edu/~remzi/Courses/537/Spring2018/dontcheat.html) for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
- Some tests will be provided at `~cs537-1/tests/P3`. Read more about the tests, including how to run them, by executing the command `cat ~cs537-1/tests/P3/README` on any lab machine. Note these test cases are not complete, and you are encouraged to create more on your own.
- Handing it in: Copy your file named `smash.c` to `~cs537-1/handin/login/P3` where login is your CS login.
- **Slip Days**
 - In case you need extra time on projects, you each will have 2 slip days for individual projects and 2 slip days for group projects (4 total slip days for the semester). After the due date we will make a copy of the handin directory for on time grading. To use a slip days you will submit your files with and `additional` file `slipdays.txt` in your regular project handing directory. This file should include one thing only and that is a single number, which is the number of slip days you want to use (ie. 1 or 2). Each consecutive day we will make a copy of any directories which contain one of these slipdays.txt files.
 - After using up your slip days you can get up to 80% if turned in a day late or 60% if 2 days late.
 - Any exception will need to be requested from the instructors.
 - Example slipdays.txt

Unix Shell

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

There are three specific objectives to this assignment:

- To further familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

Overview

In this assignment, you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Unix. If you don't know what shell you are running, it's probably `bash`. One thing you should do on your own time is to learn more about your shell, by reading the man pages or other online materials.

Program Specifications

Basic Shell: `smash`

Your basic shell, called `smash` (short for Super Madison Shell, naturally), is basically an interactive loop: it repeatedly prints a prompt `smash>` (note the space after the greater-than sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types `exit`. The name of your final executable should be `smash`.

The shell can be invoked with no arguments; anything else is an error. Here is how it will be called:

```
prompt> ./smash
smash>
```

At this point, `smash` is running, and ready to accept commands. Type away!

The mode above is called *interactive* mode, and allows the user to type commands directly.

You should structure your shell such that it creates a process for each new command (the exception are *built-in commands*, discussed below). Your basic shell should be able to parse a command and run the program corresponding to the command. For example, if the user types `/bin/ls -la /tmp`, your shell should run the program `/bin/ls` with the given arguments `-la` and `/tmp`.


Structure

Basic Shell

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command `exit`, at which point it exits. That's it!

For reading lines of input, you should use `getline()`. This allows you to obtain arbitrarily long input lines with ease. The shell will be run in *interactive mode*, where the user types a command (one at a time) and the shell acts on it.

To parse the input line into constituent pieces, you might want to use `strsep()`. Read the man page (carefully) for more details.

To execute commands, look into `fork()`, `exec()`, and `wait()/waitpid()`. Your shell will have to wait for any program you launch to finish before processing the next command. See the man pages for these functions, and also read the relevant [book chapter](http://www.ostep.org/cpu-api.pdf)  (<http://www.ostep.org/cpu-api.pdf>) for a brief overview.

You will note that there are a variety of commands in the `exec` family; for this project, you must use `execv`. You should **not** use the `system()` library function call to run a command. Remember that if `execv()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified.

Built-in Commands

Whenever your shell accepts a command, it should check whether the command is a **built-in command** or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the `exit` built-in command, you simply call `exit(0);` in your smash source code, which then will exit the shell.

In this project, you should implement `exit`, `cd`, and `pwd` as built-in commands.

- `exit`: When the user types `exit`, your shell should simply call the `exit` system call with 0 as a parameter. It is an error to pass any arguments to `exit`.
- `cd`: `cd` always take one argument (0 or >1 args should be signaled as an error). To change directories, use the `chdir()` system call with the argument supplied by the user; if `chdir` fails, that is also an error.
- `pwd`: When the user types `pwd`, your shell should print the current working directory. To read the current working directory, use the `*getcwd(char *buf, size_t size)` system call. If `getcwd`

fails, print the error specified in the error section to stderr in the shell.

- loop: When the user types loop, it will run the next command a specified number of times. This is detailed more below in the loop section.

Redirection

Many times, a shell user prefers to send the output of a program to a file rather than to the screen. Usually, a shell provides this nice feature with the `>` character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature, but with a slight twist (explained below).

For example, if a user types `/bin/ls -la /tmp > output`, nothing should be printed on the screen. Instead, the standard output of the `ls` program should be rerouted to the file `output`. In addition, the standard error output of the program should be rerouted to the file `output` (the twist is that this is a little different than standard redirection). However, if the program cannot be found (i.e., mistyped `/bin/ls` as `/bin/ll`), an error should be reported, but not to be redirected to `output`.

If the `output` file exists before you run your program, you should simply overwrite it (after truncating it).

The exact format of redirection is a command (and possibly some arguments) followed by the redirection symbol followed by a filename. Multiple redirection operators or multiple files to the right of the redirection sign are errors. Redirection without a command is also not allowed - an error should be printed out, instead of being redirected.

Note:

- Don't worry about redirection for built-in commands (e.g., we will **not** test what happens when you type `pwd > file`).
- Don't worry about the order of `stdout` and `stderr`. In other words, if a process writes to both, the output could be jumbled up. (This is okay!)
- There will always be a space before and after a redirection command.

Multiple Commands

What if a shell user would like to type in multiple commands in a single line? Sometimes, they might turn in a long list of commands, prepare some popcorns and, wait until all of them to finish. This is supported by semicolons:

```
smash> cmd1 ; cmd2 args1 args2 ; cmd3 args1
```

Here, your shell first runs first `cmd1`, when it completes run `cmd2`, and when `cmd2` completes run `cmd3` and wait for it to finish.

Note:

- Your shell should support multiple built-in commands, such as `pwd ; cd foo ; pwd`
- Redirection should be supported (e.g., `cmd1 > output ; cmd 2`).
- Empty commands are allowed (i.e., `cmd ;`, `; cmd`).
- There will always be at least one space between a semicolon and a command.

Pipes

What if a shell user would like to take the output from one command and use it on the input of another? We can use pipes to help users do this.

Pipes connect the STDOUT of the first command to the STDIN of the second

```
smash> /bin/cat file.txt | /bin/sort
```

In the case above your command first run `cat` on your file and take the output of command and send it to the input of `sort`. This will then print a sorted version of your file to STDOUT.

Note:

- Don't worry about piping built in commands, such as `pwd | /bin/grep test`
- Redirection should be supported (e.g., `ls test.txt | /bin/grep test > output.txt`).
- Chaining pipes with semicolons should be supported (i.e., `ls test.txt | /bin/grep test ; ls test.csv | /bin/grep column`).
- Chaining pipes should be supported (i.e., `ls test.txt | /bin/grep test | /bin/grep test`).
- Looping on pipes should be supported (i.e., `loop 5 /bin/ls | /bin/grep test`).
 - This will run `pwd` 5 times and then pipe the output separate 5 times to `grep`
- There will always be a space between a pipe symbol `|` and a command.

Loops

Finally, shell users may want to run a command multiple times without having to type it over and over. This is supported with the `loop` built in.

```
smash> loop 3 cmd args
```

Here, here your shell will run `cmd` 3 times sequentially, waiting for completion between runs

Note:

- Your shell should support looping on built-in commands, such as `loop 4 cd ..`
- Redirection should be supported (e.g., `loop 5 cmd1 > output`). This will rewrite file output 5 times, not append to file output.
- Your shell does **not** need to support nested loops (e.g. `loop 5 loop 5 cmd`)
- Your shell should support multiple commands in loops (e.g. `loop 5 cmd1 ; cmd2`)
 - In this case cmd1 will run 5 times, and then after all complete cmd2 will run once

Program Errors

The one and only error message. You should print this one and only error message whenever you encounter an error of any type:

```
char error_message[30] = "An error has occurred\n";
write(STDERR_FILENO, error_message, strlen(error_message));
```

The error message should be printed to stderr (standard error), as shown above.

After most errors, your shell simply *continue processing* after printing the one and only error message.

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there are any program-related errors (e.g., invalid arguments to `ls` when you run it, for example), the shell does not have to worry about that (rather, the program will print its own error messages and exit).

For multiple commands, syntax errors (e.g., `/bin/ls; > output`) should prevent the entire line from executing, but invalid programs names (e.g., a mistyped `/bin/ls`, like `/bin/lss`) should print an error for that command and then stop.

Miscellaneous Hints

The code below can be used to split a line from the user into args:

```
/// description: Takes a line and splits it into args similar to how argc
///             and argv work in main
/// line:       The line being split up. Will be mangled after completion
///             of the function<
/// args:       a pointer to an array of strings that will be filled and
///             allocated with the args from the line
/// num_args:   a point to an integer for the number of arguments in args
/// return:     returns 0 on success, and -1 on failure
int lexer(char *line, char ***args, int *num_args){
    *num_args = 0;
    // count number of args
    char *l = strdup(line);
    if(l == NULL){
        return -1;
    }
}
```

```

char *token = strtok(l, " \t\n");
while(token != NULL){
    (*num_args)++;
    token = strtok(NULL, " \t\n");
}
free(l);
// split line into args
*args = malloc(sizeof(char **) * *num_args);
*num_args = 0;
token = strtok(line, " \t\n");
while(token != NULL){
    char *token_copy = strdup(token);
    if(token_copy == NULL){
        return -1;
    }
    (*args)[(*num_args)++] = token_copy;
    token = strtok(NULL, " \t\n");
}
return 0;
}

```

Remember

to get the **basic functionality** of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as `/bin/ls`).

Next, add built-in commands. Then, try working on redirection. Finally, think about multiple commands. These requires a little more effort on parsing, but each should not be too hard to implement. It is recommended that you separate the process of parsing and execution - parse first, look for syntax errors (if any), and then finally execute the commands.

At some point, you should make sure your code is robust to white space of various kinds, including spaces () and tabs (\t). There will be always at least one piece of white space before and after commands, arguments, and operators, however there may be more. Note that you will be able to tokenize on whitespace and pull out all the commands, arguments, and operators. For example, your shell should accept commands like:

```
smash> ls ; /bin/ls > output ; cd /usr
```

Check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. It's also just good programming sense.

Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don't be gentle – other users certainly won't be.

Finally, keep versions of your code. More advanced programmers will use a source control system such as git. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around,

you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.