`lash` Shell

Fall 2023

1 Intro

In this assignment, you will be writing a shell called `lash` (**L**oyol**a** **sh**ell). The `lash` shell will be an extension of `shittyshell`, the in-class assignment. 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 `ls -la /tmp`, your shell should run the program `/bin/ls` with the given arguments `-la` and `/tmp` (how does the shell know to run `/bin/ls`? It's something called the shell path; more on this below).

2 Specifications

2.1 Paths

Your shell shoould maintain a `PATH` variable which holds a default search path for commands. When the user tries to run a program without specifying a complete filesystem path (like `/bin/ls`), your shell should append the `PATH` variable to the beginning of the command and attempt to execute that binary file.

For example, suppose the user types `ps` on the command line, and your `PATH` variable contains "`/bin/`". Your program should concatenate these two strings together into a new temporary buffer:

```
char tmp[1000];
strcpy(tmp,PATH); // tmp now contains "/bin/"
strcat(tmp,cmd);  // strcat concatenates strings, like "+" in Java or python. tmp now has "/bin/ps"
```

Your program can check if the requested command actually exists in the filesystem by calling `access()`. See `man access` for details about how to call it.

2.2 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 `path` as built-in commands.

1. `exit`: When the user types exit, your shell should simply call the exit system call with 0 as a parameter.
2. `cd`: always take one argument (if you get 0 or ¿1 args, you should be signal an error to the user). To change directories, use the `chdir()` system call with the argument supplied by the user; if the `chdir()` fails, that is also an error that should be signaled to the user.
3. `path`: The path command takes 1 or more arguments, with each argument separated by whitespace from the others. Three options are supported: print, add, remove, and clear. Invalid arguments should print an error.
   (a) `print` prints out the current path variable.
   (b) `add` accepts 1 path. Your shell should append it to the beginning of the path list. For example, path add /usr/bin results in the path list containing /usr/bin and /bin (notice the order here). Your shell should not report an error if an invalid path is added. It should kindly accept it.

(c) `remove` accepts 1 path. It searches through the current path list and removes the corresponding one. If the path cannot be found, this is an error.

(d) `clear` takes no additional argument. It simply removes everything from the path list. If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands).

3 Grading

Grade Percentage	Description
10%	Shell launches new processes given a complete path and no arguments (like `/usr/bin/ls`, `/usr/bin/ps`, etc). It waits for the new process to finish before continuing.
10%	Default search path of `/bin/` works without specifying the full filesystem path of the binary.
25%	Shell launches new processes with multiple arguments, like `ls -l /tmp`
5%	`exit` command works
10%	`cd` command works
10%	`path print` works
10%	`path add` works
10%	`path remove` works
10%	`path clear` works
Extra Credit 5%	Redirection: entering a command like `ls > out.txt` writes the output of the command to file `out.txt`.
Extra Credit 5%	Multiple commands: Launching a command like `cmd1 arg1 arg2 ; cmd2 arg1 ; cmd3` will first run `cmd1` with its arguments, then run `cmd2` with its arguments, then run `cmd3`.

4 Parsing The Command String

Strings in C are implemented a simply arrays of characters. The last character of the string is the NULL terminator, with the value of 0. Consider the C string (character array below), which we can call `cmdstring`:

```
char cmdstring[] = "/bin/ls -a /tmp";
```

This string is stored in memory in the following way:

/	b	i	n	/	l	s	␣	-	a	␣	/	t	m	p	\0

The way it's stored now, it's one continuous C string with a NULL terminator (`\0`) at the end. Let's say we want to split this string up into three space-delimited pieces: (1) `/bin/ls` (2) `-a` (3) `/tmp` Each will be its own C string with a NULL terminator at the end. We can do this simply by replacing each space in the string with a NULL terminator.

```
unsigned int cmd_str_len = strlen(cmdstring);

for(unsigned int k = 0; k < cmd_str_len; k++) {
  if(cmdstring[k] == ' ') { // Check if character is a space
    cmdstring[k] = '\0';    // If so, replace with NULL terminator
  }
}
```

This code snippet will convert the array to the following form, with spaces replaced by NULL terminators:

/	b	i	n	/	l	s	\0	-	a	\0	/	t	m	p	\0

In its new form, there are actually three separate C strings that live in the same array. Remember, a C string is just a sequence of ASCII characters followed by a NULL terminator. Since there are now three NULL terminators in the array, there are three C strings in the same array.

But wait! Why would we store three separate strings in the same variable? The C variable `cmdstring` has only one name, but we are cramming three distinct objects into it. How can we keep track of these three different objects?

The answer is that we need to somehow record the starting memory address of each of the three strings. That way, even though they live next to each other in a contiguous blob of memory, we can individually find each one of the strings. Let's declare an array of pointers (we'll call it `argz`) that we will use to save the memory addresses of the start of our strings.

```
char *argz[4] = {NULL, NULL, NULL, NULL}; // Addresses of each string in cmdstring
unsigned int cmd_str_len = strlen(cmdstring);

unsigned int stringnum = 0;
argz[stringnum++] = &cmdstring[0];
for(unsigned int k = 0; k < cmd_str_len; k++) {
  if(cmdstring[k] == ' ') { // Check if character is a space
    cmdstring[k] = '\0';    // If so, replace with NULL terminator
    argz[stringnum++] = &cmdstring[k+1]; // Save the start address of the next string
  }
}
```

The array `argz` holds four pointers: three valid string pointers followed by a NULL pointer (to indicate the end of the array). This array `argz` will become the `argv` when we call `execv()` to launch a new process. In this example, we have statically allocated it (with a fixed length of 4). In real life, 4 pointers will not be enough. It should be dynamically allocated (with `malloc()`) at runtime to hold as many arguments as the user types.

Aside: Quotes In C, there are two different kinds of quotes: single quotes (', beteen semicolon and enter on the keyboard) and double quotes (", same key as single quote). Single quotes are used to enclose **one** ASCII character. For example, `'a'` gives us the ASCII value for lowercase *a*, 97. Only one character may be enclosed inside a pair of single quotes. Double quotes define a NULL-terminated C string. For example, `"neil"` creates a 5-byte object in memory and populates it with `n e i l \0`

5 Storing the `PATH` Variable

The `PATH` variable contains one or more strings. There are a few different approaches to storing the `PATH` variable:

1. One big `char` array that stores all of the `PATH`s. `PATH`s are separated by some delimiter (like `:`).
2. Multiple `char` arrays (one for each `PATH`).

One big array Most UNIX/Linux shells use this approach. For example, on my Mac, the `PATH` variable looks like the following:

```
neil@mac ~ $ echo $PATH
/opt/homebrew/bin:/opt/homebrew/sbin:/usr/local/bin:/System/Cryptexes/App/usr/bin:/usr/bin:/bin:/usr/sbin:/sbi
↪  Fusion.app/Contents/Public:/Library/TeX/texbin:/var/run/com.apple.security.cryptexd/codex.system/bootstrap
```

The shell maintains one big long string to store all of the search paths, and it parses that string each time the user types a command. You could do this by (1) making a copy of the `PATH` string[1], then (2) iterating through it, replacing the `:` character with a NULL terminator like we did in §4.

[1] To make a copy of a string, first create a new `char` array, then use the `strcpy()` function to copy the original string into the new array.

Multiple Arrays If you choose to use multiple arrays, you won't have to parse the `PATH` string each time the user enters a command, but keeping track of the location of the strings becomes a little more complicated. For each directory in the `PATH`, you need to create a `char` array. The easiest way to create a new array is with `malloc()`:

```
char *PATHVAR[10]; // Create 10 pointers to PATH strings
memset(PATHVAR, 0, sizeof(PATHVAR)); // initialize PATH pointers to NULL
char cmd[] = "/usr/local/bin"; // Path we get from user command
char *new_path_dir = malloc(strlen(cmd) + 1); // allocate mem to store new PATH
PATHVAR[0] = new_path_dir; // Point to the new path
strcpy(PATHVAR[0], cmd); // Copy the user-specified path into the PATHVAR
```