Tutorial - Write a Shell in C

Stephen Brennan • 16 January 2015

It's easy to view yourself as "not a *real* programmer." There are programs out there that everyone uses, and it's easy to put their developers on a pedestal. Although developing large software projects isn't easy, many times the basic idea of that software is quite simple. Implementing it yourself is a fun way to show that you have what it takes to be a real programmer. So, this is a walkthrough on how I wrote my own simplistic Unix shell in C, in the hopes that it makes other people feel that way too.

The code for the shell described here, dubbed `lsh`, is available on GitHub.

University students beware! Many classes have assignments that ask you to write a shell, and some faculty are aware of this tutorial and code. If you're a student in such a class, you shouldn't copy (or copy then modify) this code without permission. And even then, I would advise against heavily relying on this tutorial.

Basic lifetime of a shell

Let's look at a shell from the top down. A shell does three main things in its lifetime.

- **Initialize**: In this step, a typical shell would read and execute its configuration files. These change aspects of the shell's behavior.
- **Interpret**: Next, the shell reads commands from stdin (which could be interactive, or a file) and executes them.
- **Terminate**: After its commands are executed, the shell executes any shutdown commands, frees up any memory, and terminates.

These steps are so general that they could apply to many programs, but we're going to use them for the basis for our shell. Our shell will be so simple that there won't be any configuration files, and there won't be any shutdown command. So, we'll just call the looping function and then terminate. But in terms of architecture, it's important to keep in mind that the lifetime of the program is more than just looping.

```
int main(int argc, char **argv)
{
```

```
  // Load config files, if any.

  // Run command loop.
  lsh_loop();

  // Perform any shutdown/cleanup.

  return EXIT_SUCCESS;
}
```

Here you can see that I just came up with a function, `lsh_loop()`, that will loop, interpreting commands. We'll see the implementation of that next.

Basic loop of a shell

So we've taken care of how the program should start up. Now, for the basic program logic: what does the shell do during its loop? Well, a simple way to handle commands is with three steps:

- **Read**: Read the command from standard input.
- **Parse**: Separate the command string into a program and arguments.
- **Execute**: Run the parsed command.

Here, I'll translate those ideas into code for `lsh_loop()`:

```
void lsh_loop(void)
{
  char *line;
  char **args;
  int status;

  do {
    printf("> ");
    line = lsh_read_line();
    args = lsh_split_line(line);
    status = lsh_execute(args);

    free(line);
    free(args);
  } while (status);
}
```

Let's walk through the code. The first few lines are just declarations. The do-while loop is more convenient for checking the status variable, because it executes once before checking its value. Within the loop, we print a prompt, call a function to read a line, call a

function to split the line into args, and execute the args. Finally, we free the line and arguments that we created earlier. Note that we're using a status variable returned by `lsh_execute()` to determine when to exit.

Reading a line

Reading a line from stdin sounds so simple, but in C it can be a hassle. The sad thing is that you don't know ahead of time how much text a user will enter into their shell. You can't simply allocate a block and hope they don't exceed it. Instead, you need to start with a block, and if they do exceed it, reallocate with more space. This is a common strategy in C, and we'll use it to implement `lsh_read_line()`.

```
#define LSH_RL_BUFSIZE 1024
char *lsh_read_line(void)
{
  int bufsize = LSH_RL_BUFSIZE;
  int position = 0;
  char *buffer = malloc(sizeof(char) * bufsize);
  int c;

  if (!buffer) {
    fprintf(stderr, "lsh: allocation error\n");
    exit(EXIT_FAILURE);
  }

  while (1) {
    // Read a character
    c = getchar();

    // If we hit EOF, replace it with a null character and return.
    if (c == EOF || c == '\n') {
      buffer[position] = '\0';
      return buffer;
    } else {
      buffer[position] = c;
    }
    position++;

    // If we have exceeded the buffer, reallocate.
    if (position >= bufsize) {
      bufsize += LSH_RL_BUFSIZE;
      buffer = realloc(buffer, bufsize);
      if (!buffer) {
        fprintf(stderr, "lsh: allocation error\n");
        exit(EXIT_FAILURE);
      }
    }
```

```
    }
}
```

The first part is a lot of declarations. If you hadn't noticed, I prefer to keep the old C style of declaring variables before the rest of the code. The meat of the function is within the (apparently infinite) `while (1)` loop. In the loop, we read a character (and store it as an `int`, not a `char`, that's important! EOF is an integer, not a character, and if you want to check for it, you need to use an `int`. This is a common beginner C mistake.). If it's the newline, or EOF, we null terminate our current string and return it. Otherwise, we add the character to our existing string.

Next, we see whether the next character will go outside of our current buffer size. If so, we reallocate our buffer (checking for allocation errors) before continuing. And that's really it.

Those who are intimately familiar with newer versions of the C library may note that there is a `getline()` function in `stdio.h` that does most of the work we just implemented. To be completely honest, I didn't know it existed until after I wrote this code. This function was a GNU extension to the C library until 2008, when it was added to the specification, so most modern Unixes should have it now. I'm leaving my existing code the way it is, and I encourage people to learn it this way first before using `getline`. You'd be robbing yourself of a learning opportunity if you didn't! Anyhow, with `getline`, the function becomes easier:

```
char *lsh_read_line(void)
{
  char *line = NULL;
  ssize_t bufsize = 0; // have getline allocate a buffer for us

  if (getline(&line, &bufsize, stdin) == -1){
    if (feof(stdin)) {
      exit(EXIT_SUCCESS);  // We recieved an EOF
    } else  {
      perror("readline");
      exit(EXIT_FAILURE);
    }
  }

  return line;
}
```

This is not 100% trivial because we still need to check for EOF or errors while reading. EOF (end of file) means that either we were reading commands from a text file which we've reached the end of, or the user typed Ctrl-D, which signals end-of-file. Either way, it means we should exit successfully, and if any other error occurs, we should fail after printing the error.

Parsing the line

OK, so if we look back at the loop, we see that we now have implemented `lsh_read_line()`, and we have the line of input. Now, we need to parse that line into a list of arguments. I'm going to make a glaring simplification here, and say that we won't allow quoting or backslash escaping in our command line arguments. Instead, we will simply use whitespace to separate arguments from each other. So the command `echo "this message"` would not call echo with a single argument `this message`, but rather it would call echo with two arguments: `"this` and `message"`.

With those simplifications, all we need to do is "tokenize" the string using whitespace as delimiters. That means we can break out the classic library function `strtok` to do some of the dirty work for us.

```
#define LSH_TOK_BUFSIZE 64
#define LSH_TOK_DELIM " \t\r\n\a"
char **lsh_split_line(char *line)
{
  int bufsize = LSH_TOK_BUFSIZE, position = 0;
  char **tokens = malloc(bufsize * sizeof(char*));
  char *token;

  if (!tokens) {
    fprintf(stderr, "lsh: allocation error\n");
    exit(EXIT_FAILURE);
  }

  token = strtok(line, LSH_TOK_DELIM);
  while (token != NULL) {
    tokens[position] = token;
    position++;

    if (position >= bufsize) {
      bufsize += LSH_TOK_BUFSIZE;
      tokens = realloc(tokens, bufsize * sizeof(char*));
      if (!tokens) {
        fprintf(stderr, "lsh: allocation error\n");
        exit(EXIT_FAILURE);
      }
    }

    token = strtok(NULL, LSH_TOK_DELIM);
  }
  tokens[position] = NULL;
  return tokens;
}
```

If this code looks suspiciously similar to `lsh_read_line()`, it's because it is! We are using the same strategy of having a buffer and dynamically expanding it. But this time, we're doing it with a null-terminated array of pointers instead of a null-terminated array of characters.

At the start of the function, we begin tokenizing by calling `strtok`. It returns a pointer to the first token. What `strtok()` actually does is return pointers to within the string you give it, and place `\0` bytes at the end of each token. We store each pointer in an array (buffer) of character pointers.

Finally, we reallocate the array of pointers if necessary. The process repeats until no token is returned by `strtok`, at which point we null-terminate the list of tokens.

So, once all is said and done, we have an array of tokens, ready to execute. Which begs the question, how do we do that?

How shells start processes

Now, we're really at the heart of what a shell does. Starting processes is the main function of shells. So writing a shell means that you need to know exactly what's going on with processes and how they start. That's why I'm going to take us on a short diversion to discuss processes in Unix-like operating systems.

There are only two ways of starting processes on Unix. The first one (which almost doesn't count) is by being Init. You see, when a Unix computer boots, its kernel is loaded. Once it is loaded and initialized, the kernel starts only one process, which is called Init. This process runs for the entire length of time that the computer is on, and it manages loading up the rest of the processes that you need for your computer to be useful.

Since most programs aren't Init, that leaves only one practical way for processes to get started: the `fork()` system call. When this function is called, the operating system makes a duplicate of the process and starts them both running. The original process is called the "parent", and the new one is called the "child". `fork()` returns 0 to the child process, and it returns to the parent the process ID number (PID) of its child. In essence, this means that the only way for new processes is to start is by an existing one duplicating itself.

This might sound like a problem. Typically, when you want to run a new process, you don't just want another copy of the same program – you want to run a different program. That's what the `exec()` system call is all about. It replaces the current running program with an entirely new one. This means that when you call `exec`, the operating system stops your process, loads up the new program, and starts that one in its place. A process never returns from an `exec()` call (unless there's an error).

With these two system calls, we have the building blocks for how most programs are run on Unix. First, an existing process forks itself into two separate ones. Then, the child uses `exec()` to replace itself with a new program. The parent process can continue doing other things, and it can even keep tabs on its children, using the system call `wait()`.

Phew! That's a lot of information, but with all that background, the following code for launching a program will actually make sense:

```
int lsh_launch(char **args)
{
  pid_t pid, wpid;
  int status;

  pid = fork();
  if (pid == 0) {
    // Child process
    if (execvp(args[0], args) == -1) {
      perror("lsh");
    }
    exit(EXIT_FAILURE);
  } else if (pid < 0) {
    // Error forking
    perror("lsh");
  } else {
    // Parent process
    do {
      wpid = waitpid(pid, &status, WUNTRACED);
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
  }

  return 1;
}
```

Alright. This function takes the list of arguments that we created earlier. Then, it forks the process, and saves the return value. Once `fork()` returns, we actually have *two* processes running concurrently. The child process will take the first if condition (where `pid == 0`).

In the child process, we want to run the command given by the user. So, we use one of the many variants of the `exec` system call, `execvp`. The different variants of `exec` do slightly different things. Some take a variable number of string arguments. Others take a list of strings. Still others let you specify the environment that the process runs with. This particular variant expects a program name and an array (also called a vector, hence the 'v') of string arguments (the first one has to be the program name). The 'p' means that instead of providing the full file path of the program to run, we're going to give its name, and let the operating system search for the program in the path.

If the exec command returns -1 (or actually, if it returns at all), we know there was an error. So, we use `perror` to print the system's error message, along with our program name, so users know where the error came from. Then, we exit so that the shell can keep running.

The second condition (`pid < 0`) checks whether `fork()` had an error. If so, we print it and keep going – there's no handling that error beyond telling the user and letting them decide if they need to quit.

The third condition means that `fork()` executed successfully. The parent process will land here. We know that the child is going to execute the process, so the parent needs to wait for the command to finish running. We use `waitpid()` to wait for the process's state to change. Unfortunately, `waitpid()` has a lot of options (like `exec()`). Processes can change state in lots of ways, and not all of them mean that the process has ended. A process can either exit (normally, or with an error code), or it can be killed by a signal. So, we use the macros provided with `waitpid()` to wait until either the processes are exited or killed. Then, the function finally returns a 1, as a signal to the calling function that we should prompt for input again.

Shell Builtins

You may have noticed that the `lsh_loop()` function calls `lsh_execute()`, but above, we titled our function `lsh_launch()`. This was intentional! You see, most commands a shell executes are programs, but not all of them. Some of them are built right into the shell.

The reason is actually pretty simple. If you want to change directory, you need to use the function `chdir()`. The thing is, the current directory is a property of a process. So, if you wrote a program called `cd` that changed directory, it would just change its own current directory, and then terminate. Its parent process's current directory would be unchanged. Instead, the shell process itself needs to execute `chdir()`, so that its own current directory is updated. Then, when it launches child processes, they will inherit that directory too.

Similarly, if there was a program named `exit`, it wouldn't be able to exit the shell that called it. That command also needs to be built into the shell. Also, most shells are configured by running configuration scripts, like `~/.bashrc`. Those scripts use commands that change the operation of the shell. These commands could only change the shell's operation if they were implemented within the shell process itself.

So, it makes sense that we need to add some commands to the shell itself. The ones I added to my shell are `cd`, `exit`, and `help`. Here are their function implementations below:

```
/*
  Function Declarations for builtin shell commands:
 */
```

```
int lsh_cd(char **args);
int lsh_help(char **args);
int lsh_exit(char **args);

/*
  List of builtin commands, followed by their corresponding functions.
 */
char *builtin_str[] = {
  "cd",
  "help",
  "exit"
};

int (*builtin_func[]) (char **) = {
  &lsh_cd,
  &lsh_help,
  &lsh_exit
};

int lsh_num_builtins() {
  return sizeof(builtin_str) / sizeof(char *);
}

/*
  Builtin function implementations.
*/
int lsh_cd(char **args)
{
  if (args[1] == NULL) {
    fprintf(stderr, "lsh: expected argument to \"cd\"\n");
  } else {
    if (chdir(args[1]) != 0) {
      perror("lsh");
    }
  }
  return 1;
}

int lsh_help(char **args)
{
  int i;
  printf("Stephen Brennan's LSH\n");
  printf("Type program names and arguments, and hit enter.\n");
  printf("The following are built in:\n");

  for (i = 0; i < lsh_num_builtins(); i++) {
    printf("  %s\n", builtin_str[i]);
  }

  printf("Use the man command for information on other programs.\n");
  return 1;
```

```
}

int lsh_exit(char **args)
{
  return 0;
}
```

There are three parts to this code. The first part contains *forward declarations* of my functions. A forward declaration is when you declare (but don't define) something, so that you can use its name before you define it. The reason I do this is because `lsh_help()` uses the array of builtins, and the arrays contain `lsh_help()`. The cleanest way to break this dependency cycle is by forward declaration.

The next part is an array of builtin command names, followed by an array of their corresponding functions. This is so that, in the future, builtin commands can be added simply by modifying these arrays, rather than editing a large "switch" statement somewhere in the code. If you're confused by the declaration of `builtin_func`, that's OK! I am too. It's an array of function pointers (that take array of strings and return an int). Any declaration involving function pointers in C can get really complicated. I still look up how function pointers are declared myself![1]

Finally, I implement each function. The `lsh_cd()` function first checks that its second argument exists, and prints an error message if it doesn't. Then, it calls `chdir()`, checks for errors, and returns. The help function prints a nice message and the names of all the builtins. And the exit function returns 0, as a signal for the command loop to terminate.

Putting together builtins and processes

The last missing piece of the puzzle is to implement `lsh_execute()`, the function that will either launch a builtin, or a process. If you're reading this far, you'll know that we've set ourselves up for a really simple function:

```
int lsh_execute(char **args)
{
  int i;

  if (args[0] == NULL) {
    // An empty command was entered.
    return 1;
  }

  for (i = 0; i < lsh_num_builtins(); i++) {
    if (strcmp(args[0], builtin_str[i]) == 0) {
      return (*builtin_func[i])(args);
    }
```

```
    }

    return lsh_launch(args);
}
```

All this does is check if the command equals each builtin, and if so, run it. If it doesn't match a builtin, it calls `lsh_launch()` to launch the process. The one caveat is that `args` might just contain NULL, if the user entered an empty string, or just whitespace. So, we need to check for that case at the beginning.

Putting it all together

That's all the code that goes into the shell. If you've read along, you should understand completely how the shell works. To try it out (on a Linux machine), you would need to copy these code segments into a file (`main.c`), and compile it. Make sure to only include one implementation of `lsh_read_line()`. You'll need to include the following headers at the top. I've added notes so that you know where each function comes from.

- `#include <sys/wait.h>`
  - `waitpid()` and associated macros
- `#include <unistd.h>`
  - `chdir()`
  - `fork()`
  - `exec()`
  - `pid_t`
- `#include <stdlib.h>`
  - `malloc()`
  - `realloc()`
  - `free()`
  - `exit()`
  - `execvp()`
  - `EXIT_SUCCESS`, `EXIT_FAILURE`
- `#include <stdio.h>`
  - `fprintf()`
  - `printf()`
  - `stderr`
  - `getchar()`
  - `perror()`
- `#include <string.h>`
  - `strcmp()`
  - `strtok()`

Once you have the code and headers, it should be as simple as running `gcc -o main main.c` to compile it, and then `./main` to run it.

Alternatively, you can get the code from GitHub. That link goes straight to the current revision of the code at the time of this writing– I may choose to update it and add new features someday in the future. If I do, I'll try my best to update this article with the details and implementation ideas.

Wrap up

If you read this and wondered how in the world I knew how to use those system calls, the answer is simple: man pages. In `man 3p` there is thorough documentation on every system call. If you know what you're looking for, and you just want to know how to use it, the man pages are your best friend. If you don't know what sort of interface the C library and Unix offer you, I would point you toward the POSIX Specification, specifically Section 13, "Headers". You can find each header and everything it is required to define in there.

Obviously, this shell isn't feature-rich. Some of its more glaring omissions are:

- Only whitespace separating arguments, no quoting or backslash escaping.
- No piping or redirection.
- Few standard builtins.
- No globbing.

The implementation of all of this stuff is really interesting, but way more than I could ever fit into an article like this. If I ever get around to implementing any of them, I'll be sure to write a follow-up about it. But I'd encourage any reader to try implementing this stuff yourself. If you're met with success, drop me a line in the comments below, I'd love to see the code.

And finally, thanks for reading this tutorial (if anyone did). I enjoyed writing it, and I hope you enjoyed reading it. Let me know what you think in the comments!

Edit: In an earlier version of this article, I had a couple nasty bugs in `lsh_split_line()`, that just happened to cancel each other out. Thanks to /u/munmap on Reddit (and other commenters) for catching them! Check this diff to see exactly what I did wrong.

Edit 2: Thanks to user ghswa on GitHub for contributing some null checks for `malloc()` that I forgot. He/she also pointed out that the manpage for `getline()` specifies that the first argument should be freeable, so `line` should be initialized to `NULL` in my `lsh_read_line()` implementation that uses `getline()`.

Edit 3: It's 2020 and we're still finding bugs, this is why software is hard. Credit to harishankarv on Github, for finding an issue with my "simple" implementation of

`lsh_read_line()` that depends on `getline()`. See this issue for details – the text of the blog is updated.

Footnotes

1. **Edit 4/Footnote:** It's 2021, over 6.5 years since writing this tutorial. I now work on operating systems in C for a living. I just wanted to say that I still do not remember how to declare a function pointer. I still need to Google it every time. ↩

Share on: