

How a basic shell program works

Jonas Ciplickas

CS 220 – Intro to Software Development

Team Autocomplete – Sprint 1

April 29th, 2018

High level breakdown

During a shell's lifetime, three tasks are performed:

1. Initialize

1. Read and execute configuration files

2. Interpret

1. Reads commands from stdin and executes them
 1. Interactive
 2. File

3. Terminate

1. After commands executed, it shuts down

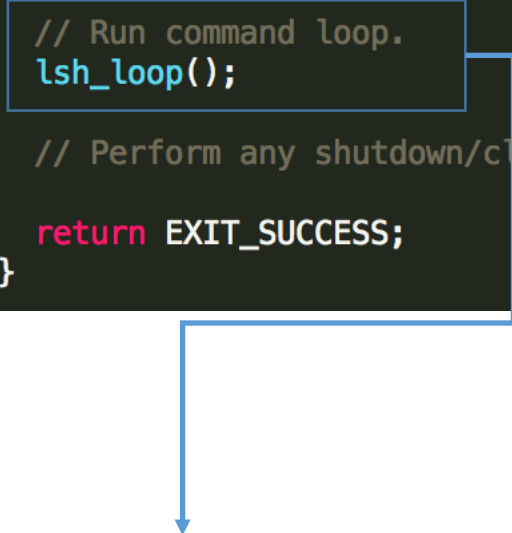
Implementation – main and lsh_loop

```
int main(int argc, char **argv)
{
    // Load config files, if any.

    // Run command loop.
    lsh_loop();

    // Perform any shutdown/cleanup.

    return EXIT_SUCCESS;
}
```



main function only has
one command: lsh_loop,
w

```
/**
 * @brief Loop getting input and executing it.
 */
void lsh_loop(void)
{
    char *line;
    char **args;
    int status;

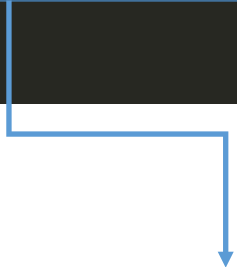
    do {
        printf("> ");
        line = lsh_read_line();
        args = lsh_split_line(line);
        status = lsh_execute(args);

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

Five things: Creates the interface, reads the line,
splits the line, executes the line, and frees up all
the memory

Implementation – reading a line

```
char *lsh_read_line(void)
{
    char *line = NULL;
    size_t bufsz = 0; // have getline allocate a buffer for us
    getline(&line, &bufsz, stdin);
    return line;
}
```



Uses the stdio.h function getline to read the input from stdin and returns it, so that split_line can prepare the command for execution.

Implementation – Parsing the line

```
#define LSH_TOK_BUFSIZE 64
#define LSH_TOK_DELIM " \t\r\n\a"
/**
 * @brief Split a line into tokens (very naively).
 * @param line The line.
 * @return Null-terminated array of tokens.
 */
```

```
char **lsh_split_line(char *line)
```

```
{
    int bufsize = LSH_TOK_BUFSIZE, position = 0;
    char **tokens = malloc(bufsize * sizeof(char*));
    char *token, **tokens_backup;
```

```
    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_backup = tokens;
            tokens = realloc(tokens, bufsize * sizeof(char*));
            if (!tokens) {
                free(tokens_backup);
                fprintf(stderr, "lsh: allocation error\n");
                exit(EXIT_FAILURE);
            }
        }
    }
```

```
    token = strtok(NULL, LSH_TOK_DELIM);
```

```
    tokens[position] = NULL;
    return tokens;
}
```

Allocates buffers for reading the line

Tokenizes the words in the command and adds them to a list of strings. Note that whitespace is what delineates commands

If necessary, allocates more space

Resets the token so the loop can move on to the next input

Returns a pointer to the list of inputs

Implementation – Executing arguments, 1

```
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);
}
```

lsh_split_line will return one of two types of functions:

1. a “built in” function
2. A different function

Checks if the command line argument corresponds to a builtin function. If so, it executes the builtin function

Otherwise, it uses a new function, lsh_launch, to execute the function

Implementation – Built-In Functions

```
/*
 * Function Declarations for builtin shell commands:
 */
int lsh_cd(char **args);
int lsh_help(char **args);
int lsh_exit(char **args);
int lsh_read(char **args);
int lsh_hack(char **args);

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

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

The shell comes with several functions “built-in” to it. That is, the author can define custom processes that the user can input and the shell will execute.

In this particular shell, you can add a built-in function by adding it in three places, as pictured:

1. declarations at the top (instead of a .h file)
2. In the builtin_str array
3. In the builtin_fun array of function pointers

Implementation – Built-in example

```
int lsh_read(char **args)
{
    if (args[1] == NULL)
    {
        fprintf(stderr, "lsh: expected argument to \"cd\\n\"");
    }
    else
    {
        print_file(args[1]);
    }
    return 1;
}
```

```
int print_file(char *filename)
{
    char buffer[500];
    FILE *fp;

    // Error condition
    if ((fp = fopen(filename, "r")) == NULL)
    {
        printf("Could not open %s.\n", filename);
        exit(1);
    }

    char *tokenPtr;

    while ( !feof(fp))
    {
        // read in the line and make sure it was successful
        if (fgets(buffer, 500, fp) != NULL)
        {
            tokenPtr = strtok(buffer, "");

            while(tokenPtr != NULL)
            {
                printf("%s", tokenPtr);
                tokenPtr = strtok(NULL, "");
            }
        }
    }

    printf("\n");

    return 1;
}
```

Here's an example of a built-in function that I created. What's important to notice from this is at *it's written and functions exactly like a normal function would*. The shell script does all the heavy lifting in terms of shell-specific things; built-in functions work exactly as any other would, except instead of using them in a main() function, you get inputs from the shell command line

Implementation – Starting a process

The shell also supports launching processes that aren't built-in functions!

```
int lsh_launch(char **args)
{
    pid_t pid;
    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 {
            waitpid(pid, &status, WUNTRACED);
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
    }

    return 1;
}
```

pid_t is a built-in type that keeps track of the ID of the current process

The fork() function is a system process that duplicates the current process into a "child process", so two are running concurrently. This allows the system to do multiple things at once

In the child process, run the command from stdin

The parent process, which waits for the child process to run – it "stalls" the shell while the child runs the command, and, when the child is finished with its process, stops waiting and lets the function return so the shell can keep running

The end!

At a pretty basic level, that's how the shell works.

Credit to Stephen Brenner for most of the code and the tutorial, which can be found [here](#).