Basic-shell

> COS 331 - Operating Systems

Martin Nestorov

I, Martin Nestorov, have submitted software that is all my own work. I did not copy the program from someone else. No one but me developed the program.

The project has been completed successfully and with all bonus features required.

General

This is a bare bones shell, written in C, with the ability to both execute foreground and background processes. The shell initializes very little resources and is capable of handling only one command with its argument list. If specified with the `&` symbol, the command will respectively be executed in the background.

How to run

In order to compile and run this project:

```
# compile
gcc basic-shell.c -o basic-shell

# run
./basic-shell
```

If everything is okay, you will see the following:

```
=== Welcome to basic-shell 0.1.0 ===

[anarcroth@basic-shell] $
```

Implementation

The general program loop goes like this:

```
while TRUE:
    get()
    parse()
    execute()
```

Although this is a very basic approach, it yields great results!

Looking at the code closer, we can see that there are two main data-structures that are used.

```
char *line;
char **params;
```

The `line` variable is used in order to capture the users input. This would represent the whole line of the input, together with all of the additional arguments in the form of a single string, hence the `char *`. The `params` variable is representative of the actual command with any additional arguments passed to it. It is a 2D array, containing strings.

In order to get the input line and parse it, we use the `char *get_line()` function.

```
char *get_line(void)
{
    char *line = (char*)malloc(sizeof(char) * COMMAND_LENGHT);

    if (!line) {
        fprintf(stderr, "failed input allocation: %s", strerror(er
        exit(errno);
    }

    /* Exits the shell either through terminating or Ctrl + D */
    if (fgets(line, sizeof(char) * COMMAND_LENGHT, stdin) == NULL)
        exit(0);

    if (line[strlen(line) - 1] == '\n')
        line[strlen(line) - 1] = '\0';

    return line;
}
```

First, we allocate space for the whole line, where the constant `COMMAND_LENGHT` is defined like so: `#define COMMAND_LENGHT 1024`. Since `malloc` can fail, we do a check for that on the first `if` statement. Then we use the builtin `fgets` function from the `string.h` header file, in order to get any user input. This builtin function saves a lot of manual work for us, but we do have to make sure that we end the input of the user with a terminating character. That is what we do with the last `if`. We check if the last character of

the `line` array is a new line. If it is, we just replace it and return the whole line for further processing.

After we get the line, we need to separate it into tokens, in order to know what is the command and the accompanying options. We do that through the `char **parse(line)` function.

```
char **parse(char *comm)
{
    char **params = (char**)malloc(sizeof(char) * PARAMETER_LENGHT

    if (!params) {
        fprintf(stderr, "failed parameters allocation: %s", strerr
        exit(errno);
    }

    for (int i = 0; i < COMMAND_LENGHT; i++) {
        char *tmp = strsep(&comm, " ");

        if (tmp == NULL)
            break;

        if (*tmp == '&')
            bg = true;
        else
            params[i] = tmp;
    }
    return params;
}
```

This allocates space for a *2D* array for strings and we fill it in with each *space-delimited* argument. With the constant `#define PARAMETER_LENGHT 64` we are able to set the size of each parameter to be at most 8 bytes. The `strsep(&comm, " ")` function does exactly that. From a positive side, *a lot* of manual work is saved with this method, since we don't have to worry about iterating over all of the characters in the string, but at the same time, we loose the ability to clean the string from white-spaces. We also have to do a check if the end of the line is an `&`. If it is, we have to indicate it with the global variable: `bg = true`. Finally, we return the parameters.

We finally get to execute everything through `void execute(char`

`**params)`. This takes in the separated line.

```
void execute(char **params)
{
    if (*params[0] == '\0') /* Empty command */
        return;

    if (getppid() == 1)     /* Already a daemon */
        return;

    if (is_builtin(params[0]))
        exec_builtin(params);
    else
        exec_std(params);
}
```

Here we do two initial checks. First we make sure that an empty line does not cause problems. We just return *nothing* in that case. We also check that the current process is not a daemon with `if (getppid() == 1)`. If everything is fine, we then have to make two choices. By default, Linux does not have a few commands on demand. This means that each shell has to implement them manually. Things like `cd`, `exit`, `topd`, `bye`, etc. have to be built-in to the shell. That is why we define a global list that holds these functions.

```
char *builtin_commands[] = { "cd", "exit" };
```

Because the array `params[0]` holds the command itself, we pass that to the helper function `bool is_builtin(char *comm)` to check for a builtin command.

```
bool is_builtin(char *comm)
{
    for (int i = 0; i < BUILTIN_COMMS; i++)
        if (strcmp(builtin_commands[i], comm) == 0)
            return true;

    return false;
}
```

After we return from this, we are able to execute anything.

Executing a builtin function is as simple as just appending a functionality to an `if` statement.

```
void exec_builtin(char **params)
{
    if (strcmp(params[0], "cd") == 0)
        cd(params);
    else if (strcmp(params[0], "exit") == 0)
        exit(0);
}
```

Here we can see that some functions are easy to do and don't need a separate implementation. While others need a whole other function.

```
void cd(char **params)
{
    if (params[1] == NULL || chdir(params[1]) != 0)
        fprintf(stderr, "%s: %s\n", params[0], strerror(errno));
}
```

The `cd` function can fail so we make sure that it executed normally. There is nothing more to do here! Simple!

But if we are executing a normal command, we invoke the `void exec_std(char **params)` method.

```
void exec_std(char **params)
{
    pid_t pid = fork();
    if (pid < 0) {
        fprintf(stderr, "fork error: %s\n", strerror(errno));
        return;
    }
    else if (pid == 0) {
        execvp(params[0], params);
        fprintf(stderr, "%s: %s\n", params[0], strerror(errno));
        return;
    }
```

```
    if (bg) {
        push_to_bg();
        return;
    }
    else {
        int chid_status;
        waitpid(pid, &chid_status, 0);
        return;
    }
}
```

Here we use the `fork()` method to create a child copy of the process. We do a check to make sure it didn't fail. Then we execute it, together with the rest of the parameters through the `execvp(params[0], params)` method. If the process **does not** fail, then we get to either wait for it in the foreground, or push it to the background. By using the global flag `bg`, we can see what we have to do.

When we have to push to the background, then we have several steps.

```
void push_to_bg(void)
{
    bg = false;

    setsid(); /* Obtain a new process group */

    int i = open("/dev/null", O_RDWR); /* Handle standart I/O */
    dup(i);
    dup(i);

    signal(SIGCHLD, SIG_IGN); /* Ignore child */
    signal(SIGTSTP, SIG_IGN); /* Ignore tty signals */
}
```

This function first resets the value of the flag. Then we obtain a new process group for the child process. We do basic *IO* handling with `dip(i)` and finally, we ignore the child signals with `signal(SIGCHLD, SIG_IGN)`.

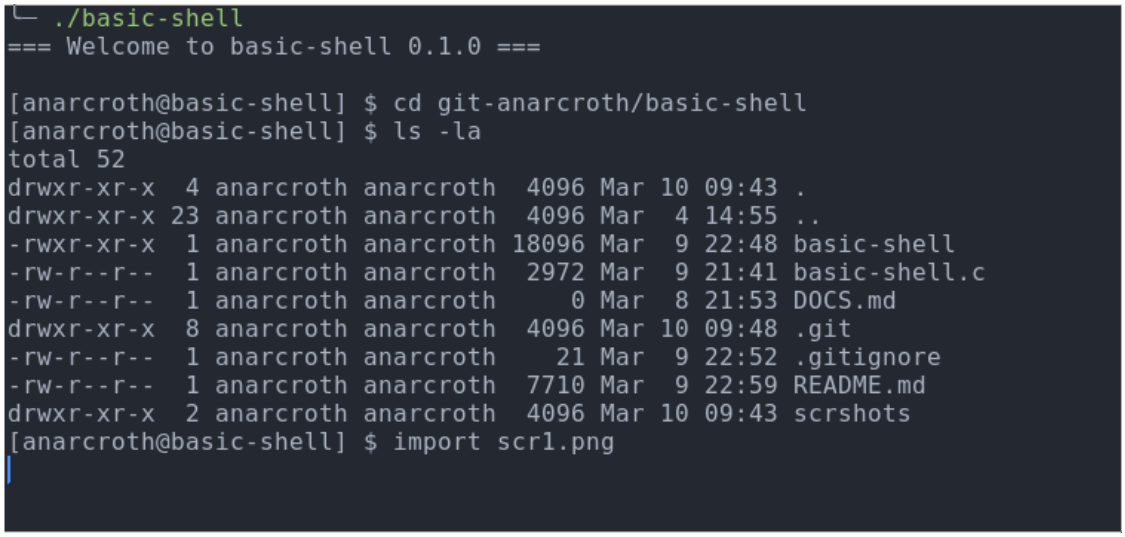
When we have to wait for a command to wait, we simply just do the

following.

```
else {
        int chid_status;
        waitpid(pid, &chid_status, 0);
        return;
}
```

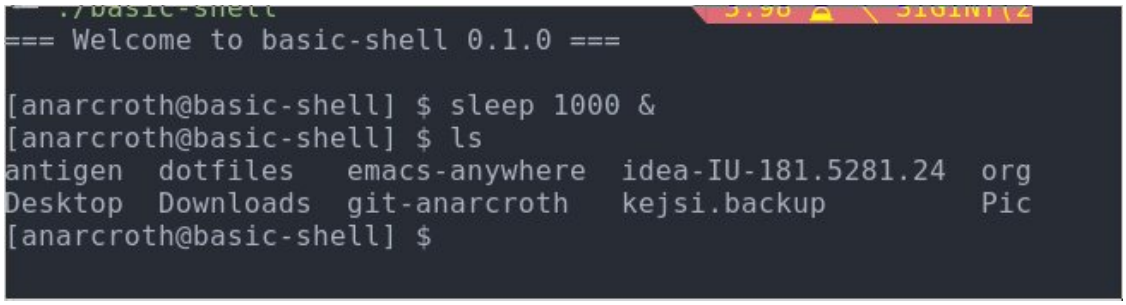
This `waitpid(pid, &chid_status, 0)` function makes sure that we wait for the child process to end.

Screenshots



Here we can see the initial screen when we run the *basic-shell* with its welcoming banner. By default, the shell starts from the `$HOME` directory. We then navigate to the directory of the project with the `cd` command. Then we list all of the contents with the `ls -la` command and options. After that, we invoke the `import` command which actually takes a picture of a selected area. We use that in order to *snap* the basic-shell!

We then move the taken picture with the `mv` command in the `scrshots` directory and list its contents.



And here is a test of a background process. We can see that the `sleep 100 &` command has been disassociated from the parent process and is running in the background, while the prompt is active for more work. We can prove this by checking the processes with `htop`. We can see that from `basic-shell` there is a `sleep` command running!

Statistics

functions	global vars	constants	headers
9	2	3	10

characters	words	lines
2972	433	163

Size of executable: 20K

Coding style: Linux style