Home

Homework 2:
Shell

This assignment will teach you how to use the Unix system call interface and the shell by implementing a small shell, which we will refer to as the 238P shell.

You can do this assignment on any operating system that supports the Unix API (Linux Openlab machines, your laptop that runs Linux or Linux VM, and even MacOS, etc.). **You don't need to set up xv6 for this assignment** Submit your programs and the shell through Gradescope (see instructions at the bottom of this page).

NOTE: YOU CANNOT PUBLICLY RELEASE SOLUTIONS TO THIS HOMEWORK. It's ok to show your work to your future employer as a private Git repo, however any public release is prohibited.

For **Mac/OSX** users. The support of 32 bit applications is depricated in the latest version of your system. So if you already updated your system to macOS Catalina or have updated your XCode then we recommend you to do the homework at the Openlab machines.

NOTE: We are aware that there are several tutorials on writing shells online. This assignment itself borrows heavily from Stephen Brennan's blog post . We strongly encourage you to do this assignment without referring to the actual code in those implementations. You are welcome to look at broad concepts (which we also try to explain here), but the actual implementation should be your work.

NOTE: We recently were made aware of the GNU readline library . Bash (and other shells) rely heavily on it for auto-complete, moving the cursor around when entering input, and even reverse-search. For those interested, this is a really interesting read on the history of `readline` . For the purposes of this assignment, using `readline` **is not allowed**, as it would make several implementation details entirely trivial. We want you to learn by implementing a shell, including it's intricacies.

TIP: While building this assignment, several parts, like adding support for I/O redirection and pipes might not be immediately obvious, and are quite involved. We encourage you to take a

look at the xv6's shell `sh.c` , to get design clues. Note however, that you cannot take the xv6 implementation and submit it (or any other submissions from previous years). You might pass all the test cases, but you will receive a 0 on this assignment if you don't submit what is entirely your work.

We will build a shell in the following steps:

1. Reading and parsing a command.
2. Executing programs
3. Implementing support for I/O redirection
4. Implementing support for pipes.

Download the `sh238p.c` , and look it over. This is a skeleton of a simple UNIX shell:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <errno.h>

int main(int argc, char **argv){
    sh_loop();
    return EXIT_SUCCESS;
}
```

As we will see, the shell is a program that essentially waits for user input, executes commands, and repeats. We will keep our shell simple, by just calling a function `sh_loop`, that loops indefinitely, reading, interpreting and executing commands. Typically a shell does much more (steps related to initialization, configuration, termination, shutdown and so on).

If you put the above snippet into a file `sh238p.c`, you can compile it with a C compiler, such as gcc. On Openlab machines you can compile it with the following command:

```
$ gcc sh238p.c -o sh238p
```

Here gcc will compile your program as `sh238p`. (Note that the above file won't compile, as we have not definted `sh_loop` yet). In the rest of this part of the assignment you will convert `sh.c` into a shell.

Part 0: The basics

Top

The main job of a shell is to execute commands. One way to break this down is:

- **Read:** commands from the standard input.
- **Parse:** the command string by separating it into a program string and its argument string.
- **Execute:** the program, passing to it the appropriate arguments.

The `sh_loop()` function, hence can look something like the following.

```
void sh_loop(void){
    char *line;
    char **args;
    int status;
    do{
        printf("238p$ ");
        line = sh_read_line();
        args = sh_split_line(line);
        status = sh_execute(args);
        free(line);
        free(args);
    }while(status);
}
```

It runs in a loop, and it provides a prompt to the user every time the loop executes:

```
238p$
```

Once the user enters a command, it calls `sh_read_line` to read the command, `sh_split_line` to parse it, and finally `sh_execute` to execute the command. It then loops back, trying to do the same thing all over again. Note here that the termination of the loop is dependant on the `status` variable, which you will have to set appropriately when you write the `sh_execute` function.

Reading a Line

We do not want to test you on your skills with reading and parsing lines in C, which can be quite involved if one wants to handle several possible error situations. Hence, we provide you with a template for `sh_read_line()` below.

The shell has to read characters from stdin into a buffer to parse it. The thing to note is that you cannot know before hand, how much text a user is going to input as a command, and hence, you cannot know how much buffer to allocate. One strategy is to start with an allocation of small size using `malloc`, and then reallocate if we run out of memory in the buffer. We can use `getchar()` to read character by character from stdin in a `while` loop, until we see a newline character, or an EOF character. In case of the former, return the buffer which has been filled by command characters until this point, after null-terminating the buffer. In case of an EOF it is customary to exit the shell, which we do. Note that an EOF can be sent by typing `Ctrl` + `D`.

We encourage you to try out writing your `sh_read_line` function using `getchar()` as mentioned above, which is a good learning opportunity. More recently however, the `getline` function was added as a GNU extension to the C library, which makes our work a lot easier.

```
char *sh_read_line(void){
```

```
    char *line = NULL;
    size_t bufsize = 0; // have getline allocate a buffer for us
    if(getline(&line, &bufsize, stdin) == -1){
        if (feof(stdin)){ // We recieved an EOF
            fprintf(stderr, "EOF\n");
            exit(EXIT_SUCCESS);
        }else{
            perror("sh238p: sh_read_line");
            exit(EXIT_FAILURE);
        }
    }
    return line;
}
```

We have given an implementation of the parser for you, but make sure you understand what `getline` is doing.

Parsing the Line

Now that we have the line inputted by the user, we need to parse it into a list of arguments. We won't be supporting backslash or quoting in our command line arguments. The list of arguments will be simply be separated by whitespace. What this means is a program call like `echo "hello world"` that would idealy parse to `echo` `hello world` will **not be tested in your shell**.

However, a program call like `echo hello world` is perfectly fine, and will be tested in your shell.

That being said, the parser, `sh_split_line`, should split the string into tokens, using whitespaces as delimiter. `strtok` comes to our rescue:

> `char *strtok(char *restrict str, const char *restrict delim);`
>
> The strtok() function breaks a string into a sequence of zero or more nonempty tokens. On the first call to strtok(), the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str must be NULL.
>
> Each call to strtok() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, strtok() returns NULL.
>
> Source: Extract from man7.org

```
#define SH_TOK_BUFSIZE 64
#define SH_TOK_DELIM " \t\r\n\a"

char **sh_split_line(char *line){
    int bufsize = SH_TOK_BUFSIZE;
    int position = 0;
    char **tokens = malloc(bufsize * sizeof(char *));
    char *token, **tokens_backup;

    if(!tokens){
        fprintf(stderr, "sh238p: allocation error\n");
        exit(EXIT_FAILURE);
    }

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

```
        if(position >= bufsize){
            bufsize += SH_TOK_BUFSIZE;
            tokens_backup = tokens;
            tokens = realloc(tokens, bufsize * sizeof(char *));
            if (!tokens){
                free(tokens_backup);
                fprintf(stderr, "sh238p: allocation error\n");
                exit(EXIT_FAILURE);
            }
        }

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

At the start of the function, we begin tokenizing by calling `strtok()` which returns a pointer to the first "token". What `strtok()` actually does is return pointers to within the string you give it (we call that pointer `token`), and places a null terminator `\0` at the end of each token. We store each pointer in an array (buffer) of character pointers called `tokens`.

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.

Part 1: Executing programs (30 Points)

Top

NOTE: For the rest of this assignment, *you* will be doing all the implementation. You are free to modify any functions that we provide, including their signatures. What we provide is a template which we encourage you to use, as we expect it will make things easier for you.

Now, finally we can come to the part where we make our tiny shell do what it was created for: starting to execute programs! By now, our shell should start and offer the user a prompt:

```
238p$
```

In this part of the assignment, you have to extend the shell to allow simple execution of external programs, for instance `ls`:

```
238p$ ls
bar.txt foo.txt sh238 sh.c
238p$
```

The execution of programs, of course, is handled by the `sh_execute` function.

```
int sh_execute(char **args){
    if (args[0] == NULL) {
        return 1;  // An empty command was entered.
```

```
    }
    return sh_launch(args);   // launch
}
```

You should do this by implementing the `sh_launch` function. Use the UNIX interface that we've discussed in class (the functions to clone processes, i.e., `fork()`, executing new processes, i.e., `exec()`, working with file descriptors i.e., `close(), dup(), open(), wait()`, etc. to implement the various shell features.

Remember to return an appropriate return value from `sh_launch` as the main loop `sh_loop` depends on it. Feel free to modify how you use the `status` variable in `sh_loop`. Print an error message when `exec` fails.

You might find it useful to look at the manual page for `exec`, for example, type

```
$ man 3 exec
```

and read about `execv`.

NOTE: When you type `ls` your shell may print an error message (unless there is a program named `ls` in your working directory or you are using a version of `exec` that searches `PATH`, i.e., `execlp`, `execvp`, or `execvpe`).

Now type the following:

```
238p$ /bin/ls
```

This should execute the program `/bin/ls`, which should print out the file names in your working directory. You can stop the 238P shell by inputting `Ctrl` + `D`, which should put you back in your computer's shell.

You may want to change the 238P shell to always try `/bin`, if the program doesn't exist in the current working directory, so that below you don't have to type "/bin" for each program, or (which is better) use one of the exec functions that search the `PATH` variable.

Your shell should handle arguments to the called program , i.e. this should work

```
238p$ ls /home
aburtsev
238p$
```

TIP: In GDB, if you want to debug child processes, `set follow-fork-mode child` is sometimes useful. This is a good reference .

Part 2: I/O redirection (30 Points)

Top

Now that you can execute commands, let us extend the features our shell provides. You have to implement I/O redirection commands so that you can run:

```
238p$ echo 238P is cool > x.txt
238p$ cat < x.txt
238P is cool
238p$
```

You should extend `sh_execute` to recognize the `>` and `<` characters. Remember to take a look at xv6's shell to get design clues.

You might find the man pages for `open` and `close` useful. Make sure you print an error message if one of the system calls you are using fails.

Part 3: Pipes (40 Points)

Top

Finally, you have to implement support for pipes so that you can run command pipelines such as:

```
238p$ ls | sort | uniq | wc -c
1199
238p$
```

You have to extend `sh_execute` to recognize the character `|`. You might find the man pages for `pipe`, `fork`, `close`, and `dup` useful.

Test that you can run the above pipeline. The `sort` program may be in the directory `/usr/bin/` and in that case you can type the absolute pathname `/usr/bin/sort` to run sort. (In your computer's shell you can type `which sort` to find out which directory in the shell's search path has an executable named `sort`.)

Your shell is on itself a normal program, therefore, its standard input/output can also be redirected by the process that starts it. Assuming the binary of your compiled shell is `./cs238p.bin`, you should be able to redirect the standard input of your shell by running the following command in the regular computer's shell (or even from within your own shell!):

```
$ ./sh238p.bin < my_commands.sh
```

Formal Description of Valid Inputs

Top

In any program on which the user has to input data, there are countless options and combinations that our program could receive, therefore, it is a good idea to agree in what is and what is not a valid input. In order to describe what "a correct" input looks like, let's use a powerful notation created precisely for this kind of jobs: the EBNF Notation.

> NOTE: **You DO NOT need to program anything about the EBNF notation**. This section is only for us to agree on what is considered a valid or invalid input for your shell. While grading your shell, **only valid inputs** will be given to it.

The EBNF Notation

> **Extended Backus–Naur Form (EBNF)** is a family of meta-syntax notations, any of which can be used to express a context-free grammar. EBNF is used to make a formal description of a formal language such as a computer programming language...
>
> Source: Wikipedia

In a nutshell, we start with **terminal symbols**, simple quoted characters or strings like `"-"` or `"hello"`. Through some **production rules**, we combine these terminal symbols to construct **non-terminal symbols**.

Then, we use those non-terminal symbols in new production rules to create even more sophisticated non-terminal symbols. We stop once we build the non-terminal symbol called **grammar**. This one will represent all the valid inputs for our shell.

Bellow, the syntax used to create the production rules and symbols:

Syntax	Name	Description	Example
`"..."`	Terminal string	Denotes a strings.	`"a"` or `"echo"`
`,`	Concatenation	Concatenate the symbols at the left and right of it.	`"a" , "b"` → `"ab"`
`;`	Termination	The end of a rule.	
`::=`	Definition	It assigns the rule on the right, to the symbol name on the left.	`my_sym ::= "a" , "b" ;`
`|`	Alternation	Pick one of the options.	`bit ::= "0" | "1" ;` → either `"0"` or `"1"`
`[...]`	Optional	Whatever is inside is optional. It can appear zero or one time.	`"ema" , [ "cs" ]` → either `"ema"` or `"emacs"`

Syntax	Name	Description	Example
`{...}`	Repetition	Whatever is inside could go many times. If followed by `*`, it appears "zero or more times". If followed by `+`, it appears "one or more times".	`{"a"}+` → any of `"a"`, `"aa"`... `{"b"}*` → any of `""`, `"b"`, `"bb"`...
`(...)`	Grouping	Just to group elements to evaluate them in a particular order.	`("a" | "b" ) , "c"` can be "ac" or "bc" `"a" | "b" , "c"` can be "a" or "bc"

The Grammar of your Shell

Let's start defining some terminal symbols that we will be using later, note that I just wrote some of the elements of letters and digits. The letters go all the way from A to Z and from a to z. The digits from 0 to 9:

```
letter    ::= "A"|"B"| ... |"Y"|"Z"|"a"|"b"| ... "y"|"z" ;
digit     ::= "0"|"1"| ... |"8"|"9" ;
symb      ::= "="|"."|","|"-"|"_" ;
```

As you can see, `symb` is a reduced set of non-alphanumeric items. Now, some basic definition of a characters, and spaces.

```
char      ::= letter | digit | symb ;
space     ::= { " " }+ ;
```

Now some more interesting ones.

```
word      ::= { char }+ ;
glob_word ::= { char | "*" | "?" }+ ;
file_path ::= "/" | ( ["/"] , { glob_word , ["/"] }+ ) ;
```

If you are not supporting globbing yet, then, for now, consider `glob_word ::= word`. Note that the file_path can be an absolute path, a relative path, a directory, etc. Some examples are:

- `/`
- `/folder`
- `./*/file-name.txt`
- `../sibling_dir/.hidden_file`
- `just-a,weird..name`
- `/***/?/./file`

Let's define now the arguments of a program call, and the program call itself:

```
arguments ::= space , { word | space | file_path }+ ;
prog_call ::= file_path , [arguments] ;
```

Your program call could (or could not) have arguments. And its arguments could contain paths, spaces and words. Additionally, note that the name of a program is just a file_path. for example, it could be:

- `/bin/cat` an absolute path to the binary.
- `cat` in which case your shell will search for binaries using the `$PATH` variable (using one of the family of `exec` functions).
- `./cat` in which case, the file should be in the local directory.

Let's define the stdin/stdout redirections, pipe, and background task symbols. If you are not yet supporting some of these functionalities, consider its symbol `::= "" ;` for now:

```
redir_in  ::= space , "<" , space , file_path ;
redir_out ::= space , ">" , space , file_path ;
pipe      ::= space , "|" , space , prog_call ;
bg        ::= space , "&" ;
```

As you can see, the `<`, `>`, `|`, and `&` symbols are always surrounded by spaces, which is nice for our parser!

Now, let's define what a full command looks like, what a separator of commands is, and how multiple commands will be passed to our shell. The last one becomes the **grammar** of our shell:

```
command ::= prog_call , [redir_in] , { pipe }* , [redir_out] , [bg] ;
separator ::= space , ";" , space ;
grammar ::= command , { separator , command }* ;
```

If you are not supporting multiple commands separated by `;` in a single input yet, then, for now consider `grammar ::= command ;`.

That's it. With this notation you can check whether any possible user input is valid or not.

Your Assignment

Top

You shall implement the described shell with functionality for executing programs, I/O redirection, and inter-process pipes.

Following, a list of extra functionality you may implement for extra credit.

Extra credit 1 (10%): Support for `cd`

It is a useful exercise to figure out how why `cd` doesn't work when provided as a command line argument to our shell, and make it work.

```
238p$ echo $PWD
/home/username/cs238P/hw2/
238p$ cd ../hw1
238p$ echo $PWD
/home/username/cs238P/hw1/
```

Extra credit 2 (10%): Support for `history`

This is another built-in shell command which displays a history of the commands entered in the current session of shell invocation. Note that using the GNU `readline` library is **not allowed**.

```
238p$ perl
238p$ dos2unix
238p$ history
1 perl
2 dos2unix
3 history
```

Extra credit 3 (10%): Support for globbing

Shells typically support globbing, which looks for the `*` and `?`, etc. pattern matchers in the command and perform a pathname expansion and replace the glob with matching filenames when it invokes the program. For example:

```
238p$ cp *.jpg /some/other/location
```

will copy all files with .jpg in the current directory to `/some/other/location`

Extra credit 4 (10%): Support for `;` separated commands in one line

You can usually run a list of commands in one line in most of the popular shells around, by separating the commands by a `;`:

```
238p$ cmd1 ; cmd2 ; cmd3
```

Extra credit 5 (10%): Support for `&` for background processes

One can typically ask the shell to run a command in the *background* by appending a `&` at the end. The command is then run as a job, asynchronously.

```
238p$ cmd arg1 arg2 &
```

Submit your work

Top

Submit your solution (the zip archive) through Gradescope in the assignment called HW2 - Shell . Pack your shell, `sh238p.c` (lower cases) into a zip archive and submit it. **Please note that `sh238p.c` must be in the root of the zip archive, not inside yet another folder**.

You can resubmit as many times as you wish. If you have any problems with the structure the autograder will tell you. The structure of the zip file should be the following:

```
homework2.zip
└── sh238p.c
```

238P Operative Systems, Fall 2021, University of California, Irvine

Instructor: Anton Burtsev

TAs: Claudio Parra, Elahe Khatibi

Updated: October, 2021