**Myshell**

**Design:**

There are two main aspects to the design of this shell: parsing the command and running the command. Built-ins are also discussed as part of the design.

**Parsing Command**

-char \*\*tokenize\_command(char \*command, int \*next)

Break command into tokens separated on whitespace, return the resulting array of String tokens. A NULL string is added to the end of tokens, since argv passed to exec() is NULL-string terminated.

-int parse\_command(struct program data \*\*\*pdata, int \*next, char \*\*tokens, int size)

Loop over each token in **tokens**. Extract information from the **tokens** and store it in an array of structs; **pdata**. Each element in **pdata** carries information for a single program (by program I mean built-in or executable).

There are a few things to keep track of. We know that each program needs an argv. This argv is a subarray of ‘tokens’, which will include at least one string: the program name. One of the jobs of our parser is to find the start and end of each argv for each program.

As for the start of argv: **arg\_start**, it is initially just the index of the first token, at least for the first program. After the first program, **arg\_start** is also found easily. The only way to launch multiple programs is through using ‘|’ or ‘&’, therefore the **arg\_start** of a new program is just the index where we found the ‘|’ or ‘&’ plus one (i.e index of the next token).

Now for finding the end of argv. The end of argv is marked by a NULL string. Our **tokens** array already has a NULL string in the end. In the simplest case, we have one program with no operators, so the entire **tokens** array is the argv of the program. In other, more complicated cases, we have multiple programs. Each program needs to have the end of its argv marked by a NULL string. This is not so hard. As mentioned above, the only time we can have multiple programs is by using pipes or background processing (| or & respectively). It turns out, whenever we find a ‘|’ or ‘&’, we replace that token with NULL, to mark the end of argv for the current program.

The parser detects invalid operators in the following way:

>, >>, <, |, & cannot be the first argument (that is reserved for program name)

>, >>, <, | cannot be the last argument, because all those operators require operands that come after them.

Cases in which multiple programs are run in the same command are as follows: either there is a pipe ‘|’ or there is an ‘&’ and the ‘&’ IS NOT the last token. In both cases, a new struct is allocated for the new program, and pdata is doubled in size if it’s not big enough. We get the argc for this new program by subtracting the current index from the number of tokens. NOTE: number of tokens DOES NOT include the final NULL-string at the end of the tokens array.

\*NOTE\* argc is used for built-in commands, for simple checking of number of args. It could’ve been excluded, as it is possible to loop over our null-terminated string to get the arg count, but it’s probably better to not do that.

**Built-ins**

Shell built-ins are stored in an array of structs, each struct holds the name of the built-in and a pointer to its function. This is useful because all we have to do is loop through the array and check if an entered program name matches a built-in function. If it’s found, we make note of the index, so that when it’s time to run the command, we know exactly which function to call.

**Running Command**

-char \*find\_executable(char \*name)

Search the PATH environment variable for an executable whose name matches **name**

-bool contains\_slash(char \*string)

This is used to quickly tell if the user entered a path to an executable

-void find\_program(char \*pname, char \*\*exec\_path, struct built\_in \*b, int \*ibuilt\_in)

Looks for the program **pname**.

If **pname** has a slash, it is must be a path, it is checked for validity then saved to **exec\_path**

Otherwise, we look for a builtin called **pname**, if found its index is saved in **ibuilt\_in**

Otherwise, look for an executable in the PATH called **pname**, if found, save to **exec\_path**

-int run\_command(struct program\_data \*\*pdata, size\_t size, struct built\_in \*b)

Loop over each element in pdata. Find program with the same name as the one in the current element. If the program can’t be found, return error. If saving the stdio or opening files for redirection/piping fails, restore the shell’s io, free resources (fds, pipes, allocated strings), then return error.

We have 3 slightly different ways to execute programs.

1. If the program is a built-in, and there’s no background processing or piping involved with this particular program, then the shell simply executes a function call
2. If the program is a built-in, and there IS background processing and/or piping, the shell forks a child, then that child calls the function
3. Otherwise, the program is an executable. The shell forks and execs into the executable’s code.

\*NOTE\* piping flag is stored in the program\_data struct of the program which writes to the pipe. Since our pdata array is sequential, this is perfectly fine. To find if a program must read from a pipe, check the is\_piped flag of the program\_data previous to this program.

Example:

pdata[0] {

is\_piped = true;

}

pdata[1] {

is\_piped = false;

}

Say we want to know if pdata[1] must read from a pipe. Simply do the following check:

If(pdata[1 - 1].is\_piped) {

//handling code here

}

This type of check is used in multiple places throughout the code. Two separate flags, such as is\_piped\_in and is\_piped\_out could be used, however, why use additional memory when you already have the information?

**Testing**

The majority of testing was done through executing multiple variations of commands, with and without operators. Operators were used together in order to ensure that the shell can handle them. Multiple chains of piping were tested. Also, background execution was testing using the sleep(1) comand. Command errors such as starting with an operator were tested. Also, ending with an operator such as >>, >, <, | is also an error case, which was tested.

Each built-in command was tested with its multiple valid and invalid arg combonations. Built-ins were also tested with operators.

*The parser was tested using the following code snippet:*

struct program\_data \*\*pdata = NULL;

int last\_index = 0;

parse\_command(&pdata, &last\_index, tokens, size);

for(int i = 0; i <= last\_index; i++) {

if(pdata[i]->argv != NULL) {

while(\*(pdata[i]->argv) != NULL) {

printf("%s ", \*(pdata[i]->argv));

++(pdata[i]->argv);

}

puts("");

}

printf("argc: %d\n", pdata[i]->argc);

if(pdata[i]->input\_file) printf("input\_file: %s\n", pdata[i]->input\_file);

if(pdata[i]->output\_file) printf("output\_file: %s\n", pdata[i]->output\_file);

printf("append\_output: %d\n", pdata[i]->append\_output);

printf("is\_piped: %d\n", pdata[i]->is\_piped);

printf("is\_daemon: %d\n", pdata[i]->is\_daemon);

}

*Built-ins were tested as such (parsing was skipped for simplicity):*

int size = 0;

char \*\*tokens = tokenize\_command(line, &size);

int ibuilt\_in = find\_builtin(tokens[0], b);

if(ibuilt\_in != -1) b[ibuilt\_in].func(size - 1, tokens);

*Testing find\_executable:*

setenv("PATH", "/bin:/usr/bin:/home", 1);

char \*exec = find\_executable("ls");

printf("Executable path: %s\n", exec);

*Testing batch mode:*

Shell was executed with a batch file (Batch file included in project folder)