Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации Сибирский Государственный Университет Телекоммуникаций и Информатики СибГУТИ

Кафедра вычислительных систем

**Курсовая работа**

по дисциплине «Операционные системы»

Выполнил:

Студент группы ИВ-921

Ярошев Р.А.

Работу проверил:

ассистент кафедры ВС

Петрук Е. А.

Новосибирск 2021

**Оглавление**

[Введение 3](#__RefHeading___Toc16594_2779262880)

[Реализация командной оболочки файлового менеджера 4](#__RefHeading___Toc16596_2779262880)

[Запуск программы 8](#__RefHeading___Toc16614_2779262880)

[Сборка: 8](#__RefHeading___Toc16616_2779262880)

[Список литературы 9](#__RefHeading___Toc16618_2779262880)

[Листинг 10](#__RefHeading___Toc204_2990361199)

# Введение

В ходе данной работы будет реализована командная оболочка со встроенным функционалом в виде файлового менеджера для ОС GNU/Linux.

Данное приложение адаптированно под терминал (командную строку) Linux и представляет собой консольную программу с минимальным интерактивным интерфейсом. Другими словами, пользователю выводится приглашение для ввода команд, в случае ошибки, будет выведено соответствующее сообщение.

Задача, которую решает приложение это работа с файлами, поэтому в функционале реализованы данные операции:

* Создание файла
* Просмотр содержимого файла
* Перемещение, копирование
* Удаление
* Создание ссылки на файл
* Просмотр списка файлов в директории

Функционал файлового менеджера безусловно может быть расширен. Здесь будут описаны наиболее важные функции.

# Реализация командной оболочки файлового менеджера

Как было сказано во введении, приложение имеет минимальный интерактивный интерфейс, однако таковым его можно назвать с натяжкой. Оболочка предоставляет лишь данный функционал:

* Выводит приглашение для ввода команды,
* Запускает введенную команду, выводит результат,
* Поддерживает конвейер:

> command1 | command2 | command3.

Командная оболочка функционирует со всеми командами, которые есть в терминале. Мы хотим расширить ее возможности в области работы с файлами., добавлением функций файлового менеджера:

* create\_file,
* view\_file\_content,
* cut\_paste\_file,
* copy\_paste\_file,
* remove\_file,
* link\_create,
* print\_dir.

**Create\_file**

- открытие файла и получение файлового дескриптора осуществляются с помощью системного вызова open(). Системный вызов open() ассоциирует файл, на который указывает имя пути name с файловым дескриптором, возвращаемым в случае успеха. В качестве файловой позиции указывается его начало (нуль), и файл открывается для доступа в соответствии с заданными флагами (параметр flags ):

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open (const char \*name, int flags);

int open (const char \*name, int flags, mode\_t mode);

Аргумент flags — это поразрядное ИЛИ, состоящее из одного или нескольких флагов. Он должен указывать режим доступа, который может иметь одно из следующих значений: O\_RDONLY , O\_WRONLY или O\_RDWR . Эти аргументы соответственно означают, что файл может быть открыт только для чтения, только для записи или одновременно для того и другого.View\_file\_content

- просмотр содержимого файла. Осуществляется системным вызовом read(). Каждый вызов считывает не более len байт в памяти, на которые содержится указание в buf . Считывание происходит с текущим значением смещения, в файле, указанном в fd . При успешном вызове возвращается количество байтов, записанных в buf . При ошибке вызов возвращает -1 и устанавливает errno . Файловая позиция продвигается в зависимости от того, сколько байтов было считано с fd . Если объект, указанный в fd , не имеет возможности позиционирования (например, это файл символьного устройства), то считывание всегда начинается с «текущей» позиции:

#include <unistd.h>

ssize\_t read (int fd, void \*buf, size\_t len);

unsigned long word;

ssize\_t nr;

nr = read (fd, &word, sizeof (unsigned long));

if (nr == –1)

**Cut\_paste\_file**

- перемещение файла в новую директорию. Осуществляется системным вызовом rename - изменяет имя файла или каталога, указанного в oldfilename (srcPathToFile в нашем случае) на новое имя, указанное в newfilename (dstPathToFile в нашем случае).  
Если в oldfilename и newfilename указать разные пути, и, если это поддерживается системой, файл будет перемещен в новое место:

int cut\_paste\_file(const char \*srcPathToFile, const char \*dstPathToFile) {

if (rename(srcPathToFile, dstPathToFile) == -1) {

perror("Cut/paste error");

return -1;

}

return 0;

}

**Copy\_paste\_file**

- копирование файла. Осуществляется вызовом read (чтение), а затем write (запись). При вызове write() записывается некоторое количество байтов, меньшее или равное тому, что указано в count . Запись начинается с buf , установленного в текущую файловую позицию. Ссылка на нужный файл определяется по файловому дескриптору fd.

При успешном выполнении возвращается количество записанных байтов, а файловая позиция обновляется соответственно. При ошибке возвращается -1 и устанавливается соответствующее значение errno . Вызов write() может вернуть 0:

#include <unistd.h>

ssize\_t write (int fd, const void \*buf, size\_t count);

const char \*buf = "My ship is solid!";

ssize\_t nr;

nr = write (fd, buf, strlen (buf));

if (nr == –1)

**Remove\_file**

- удаление файла. Осуществляется вызовом unlink - удаляет указанный файл из каталога. В случае успеха функция возвращает 0, а при неудаче -1:

int remove\_file(const char \*pathToFile) {

if (unlink(pathToFile) == -1) {

perror("File deletion error");

return -1;

}

return 0;

}

**Link\_create**

- создание ссылки на файл. Осуществляется вызовом link() - жесткая ссылка - создает новую ссылку по пути newpath для существующего файла oldpath , а затем возвращает 0 . По выполнении и oldpath , и newpath ссылаются на один и тот же файл. В случае сбоя вызов возвращает -1:

#include <unistd.h>

int link (const char \*oldpath, const char \*newpath);

int ret;

ret = link ("/home/kidd/privateer", /home/kidd/pirate");

if (ret)

perror ("link");

**Print\_dir**

- вывод файлов директории. Осуществляется вызовом opendir(), readdir(). Opendir() - создает поток каталога, представляющий каталог, определенный через name, readdir() - читает записи из потока, созданного opendir(), и возвращает записи одну за другой из указанного объекта DIR:

int find\_file\_in\_dir (const char \*path, const char \*file)

{

struct dirent \*entry;

int ret = 1;

DIR \*dir;

dir = opendir (path);

errno = 0;

while ((entry = readdir (dir)) != NULL) {

if (strcmp(entry->d\_name, file) == 0) {

ret = 0;

**break**;

}

}

if (errno && !entry)

perror ("readdir");

closedir (dir);

return ret;

}

# Запуск программы

## Сборка:

$ gcc main.c

Запуск:

$ ./a.out

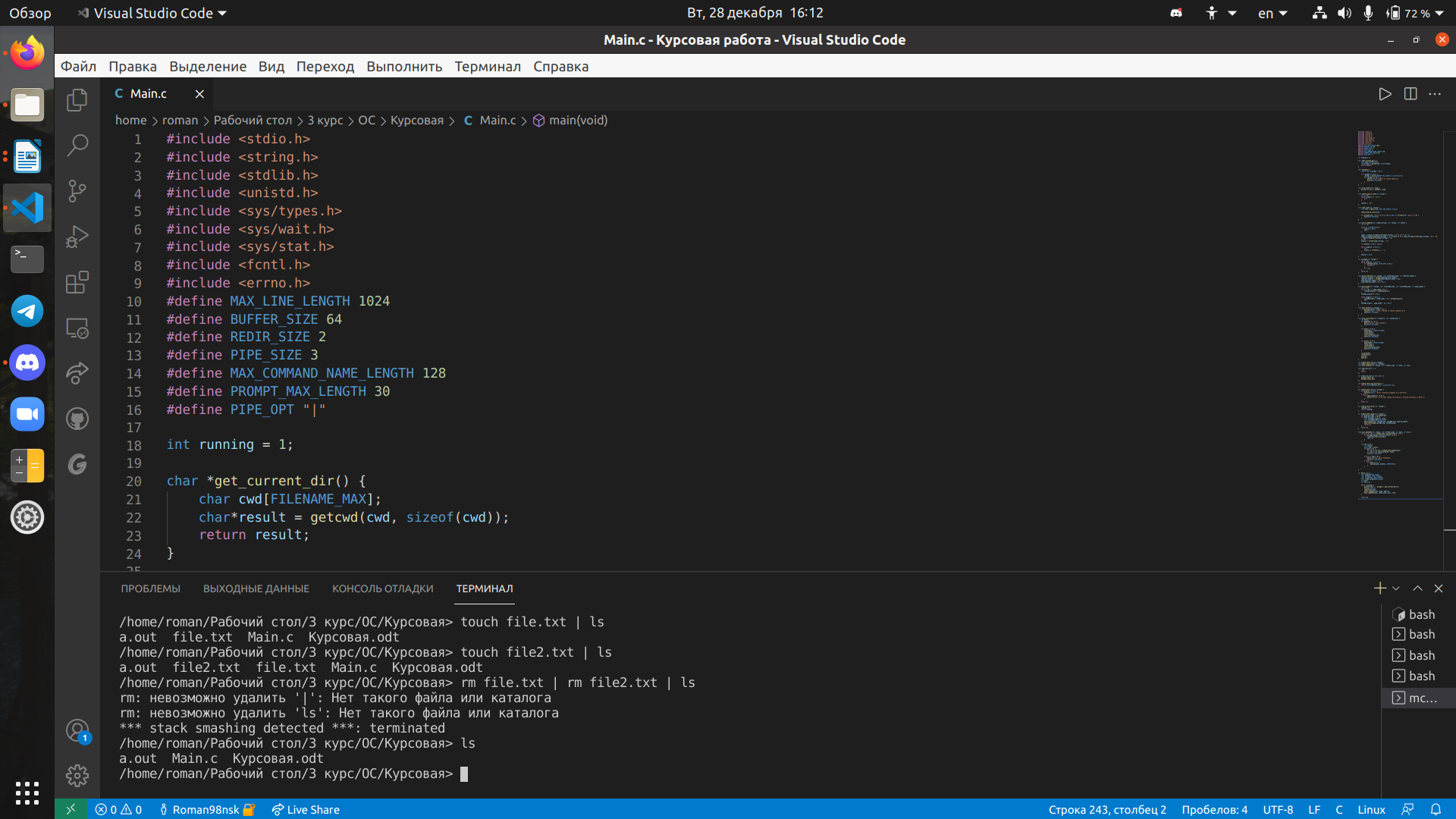
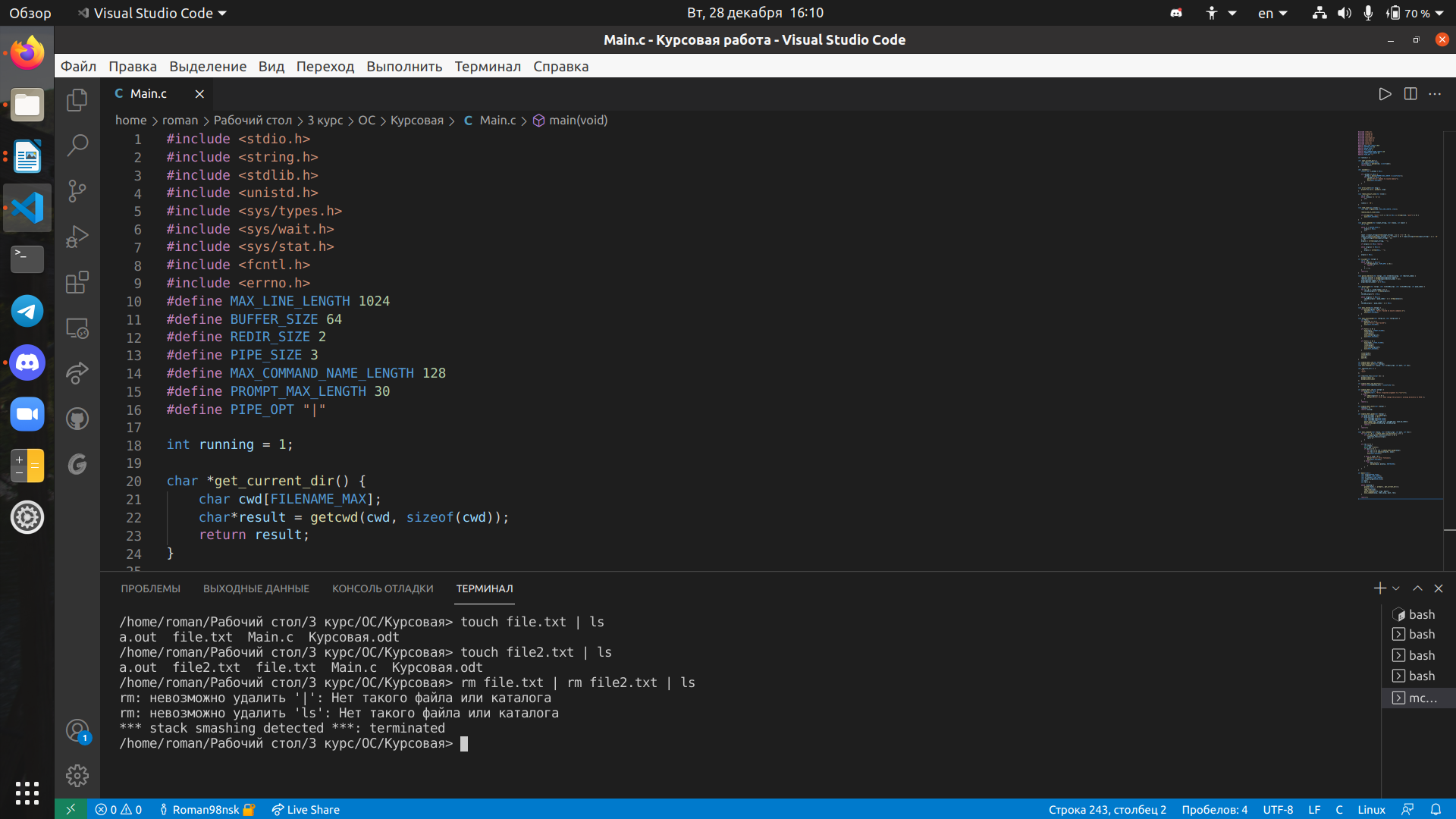


Рисунок 1. Запуск

# Список литературы

* Керниган Б., Ритчи Д. Язык программирования Си (The C programming language). - 2-е изд. — М.: Вильямс, 2007.
* Стивенс У. UNIX: взаимодействие процессов. — СПб.: Питер, 2001.
* 42 команды Linux которые вы должны знать: [Электронный ресурс]. М., 2020. URL: <https://losst.ru/42-komandy-linux-kotorye-vy-dolzhny-znat>. (Дата обращения: 27.12.2021).

## Листинг

main.c

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h>

#define MAX\_LINE\_LENGTH 1024

#define BUFFER\_SIZE 64

#define REDIR\_SIZE 2

#define PIPE\_SIZE 3

#define MAX\_COMMAND\_NAME\_LENGTH 128

#define PROMPT\_MAX\_LENGTH 30

#define PIPE\_OPT "|"

int running = 1;

char \*get\_current\_dir() {

char cwd[FILENAME\_MAX];

char\*result = getcwd(cwd, sizeof(cwd));

return result;

}

char \*prompt() {

static char \*\_prompt = NULL;

if (\_prompt == NULL) {

\_prompt = malloc(PROMPT\_MAX\_LENGTH \* sizeof(char));

if (\_prompt == NULL) {

perror("Error: Unable to locate memory");

exit(EXIT\_FAILURE);

}

}

}

void error\_alert(char \*msg) {

printf("%s %s**\n**", prompt(), msg);

}

void remove\_end\_of\_line(char \*line) {

int i = 0;

while (line[i] != '**\n**') {

i++;

}

line[i] = '**\0**';

}

void read\_line(char \*line) {

char \*ret = fgets(line, MAX\_LINE\_LENGTH, stdin);

remove\_end\_of\_line(line);

if (strcmp(line, "exit") == 0 || ret == NULL || strcmp(line, "quit") == 0) {

exit(EXIT\_SUCCESS);

}

}

void parse\_command(char \*input\_string, char \*\*argv, int \*wait) {

int i = 0;

while (i < BUFFER\_SIZE) {

argv[i] = NULL;

i++;

}

\*wait = (input\_string[strlen(input\_string) - 1] == '&') ? 0 : 1;

input\_string[strlen(input\_string) - 1] = (\*wait == 0) ? input\_string[strlen(input\_string) - 1] = '**\0**'

: input\_string[strlen(input\_string) - 1];

i = 0;

argv[i] = strtok(input\_string, " ");

if (argv[i] == NULL) return;

while (argv[i] != NULL) {

i++;

argv[i] = strtok(NULL, " ");

}

argv[i] = NULL;

}

int is\_pipe(char \*\*argv) {

int i = 0;

while (argv[i] != NULL) {

if (strcmp(argv[i], PIPE\_OPT) == 0) {

return i;

}

i = -~i;

}

return 0;

}

void parse\_redirect(char \*\*argv, char \*\*redirect\_argv, int redirect\_index) {

redirect\_argv[0] = strdup(argv[redirect\_index]);

redirect\_argv[1] = strdup(argv[redirect\_index + 1]);

argv[redirect\_index] = NULL;

argv[redirect\_index + 1] = NULL;

}

void parse\_pipe(char \*\*argv, char \*\*child01\_argv, char \*\*child02\_argv, int pipe\_index) {

int i = 0;

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

child01\_argv[i] = strdup(argv[i]);

}

child01\_argv[i++] = NULL;

while (argv[i] != NULL) {

child02\_argv[i - pipe\_index - 1] = strdup(argv[i]);

i++;

}

child02\_argv[i - pipe\_index - 1] = NULL;

}

void exec\_child(char \*\*argv) {

if (execvp(argv[0], argv) < 0) {

fprintf(stderr, "Error: Failed to execte command.**\n**");

exit(EXIT\_FAILURE);

}

}

void exec\_child\_pipe(char \*\*argv\_in, char \*\*argv\_out) {

int fd[2];

if (pipe(fd) == -1) {

perror("Error: Pipe failed");

exit(EXIT\_FAILURE);

}

if (fork() == 0) {

dup2(fd[1], STDOUT\_FILENO);

close(fd[0]);

close(fd[1]);

exec\_child(argv\_in);

exit(EXIT\_SUCCESS);

}

if (fork() == 0) {

dup2(fd[0], STDIN\_FILENO);

close(fd[1]);

close(fd[0]);

exec\_child(argv\_out);

exit(EXIT\_SUCCESS);

}

close(fd[0]);

close(fd[1]);

wait(0);

wait(0);

}

int simple\_shell\_cd(char \*\*args);

int simple\_shell\_exit(char \*\*args);

void exec\_command(char \*\*args, char \*\*redir\_argv, int wait, int res);

char \*builtin\_str[] = {

"cd",

"exit"

};

int (\*builtin\_func[])(char \*\*) = {

&simple\_shell\_cd,

&simple\_shell\_exit

};

int simple\_shell\_num\_builtins() {

return sizeof(builtin\_str) / sizeof(char \*);

}

int simple\_shell\_cd(char \*\*argv) {

if (argv[1] == NULL) {

fprintf(stderr, "Error: Expected argument to **\"**cd**\"\n**");

} else {

if (chdir(argv[1]) != 0) {

perror("Error: Error when change the process's working directory to PATH.");

}

}

return 1;

}

int simple\_shell\_exit(char \*\*args) {

running = 0;

return running;

}

int simple\_shell\_pipe(char \*\*args) {

int pipe\_op\_index = is\_pipe(args);

if (pipe\_op\_index != 0) {

char \*child01\_arg[PIPE\_SIZE];

char \*child02\_arg[PIPE\_SIZE];

parse\_pipe(args, child01\_arg, child02\_arg, pipe\_op\_index);

exec\_child\_pipe(child01\_arg, child02\_arg);

return 1;

}

return 0;

}

void exec\_command(char \*\*args, char \*\*redir\_argv, int wait, int res) {

for (int i = 0; i < simple\_shell\_num\_builtins(); i++) {

if (strcmp(args[0], builtin\_str[i]) == 0) {

(\*builtin\_func[i])(args);

res = 1;

}

}

if (res == 0) {

int status;

pid\_t pid = fork();

if (pid == 0) {

if (res == 0) res = simple\_shell\_pipe(args);

if (res == 0) execvp(args[0], args);

exit(EXIT\_SUCCESS);

} else if (pid < 0) {

perror("Error: Error forking");

exit(EXIT\_FAILURE);

} else {

if (wait == 1) {

waitpid(pid, &status, WUNTRACED);

}

}

}

}

int main(void) {

char \*args[BUFFER\_SIZE];

char line[MAX\_LINE\_LENGTH];

char t\_line[MAX\_LINE\_LENGTH];

char \*redir\_argv[REDIR\_SIZE];

int wait;

int res = 0;

while (running) {

printf("%s%s> ", prompt(), get\_current\_dir());

fflush(stdout);

read\_line(line);

parse\_command(line, args, &wait);

exec\_command(args, redir\_argv, wait, res);

}

return 0;

}