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

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

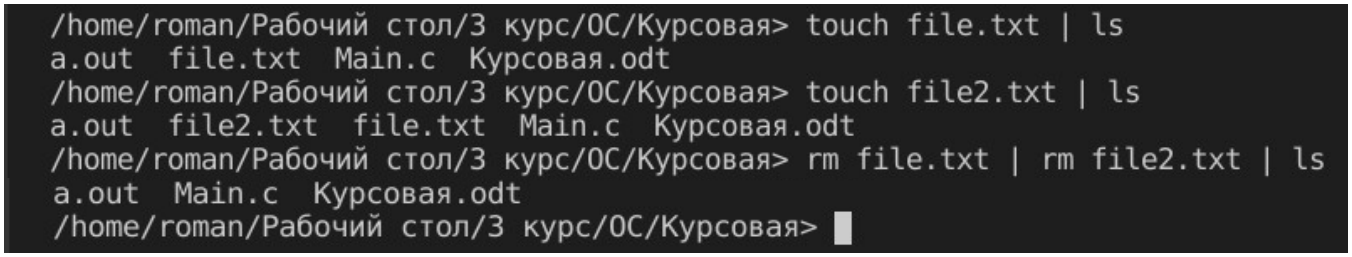

Рисунок 1. Запуск

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

```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
};
```

```c
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;
}
```