Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Ульяновский государственный технический университет»

Кафедра «Вычислительная техника»

Дисциплина «Операционные системы»

**Лабораторная работа 1**

**Разработка оболочки**

Выполнил

Студент группы ИВТАСбд-42

Зяббаров А.А.

Проверил:

преподаватель кафедры «ВТ»

Беляев К.С.

Ульяновск

2024

# Постановка задачи

Разработать простую оболочку командной строки, которая удовлетворяет указанным в задании критериям, и реализовать её на конкретной платформе UNIX.

# Реализация

Хорошо, давайте рассмотрим более подробно, как работают некоторые ключевые части кода, связанные с обработкой команд в данной оболочке.

1. Обработка внешних команд (executeExternalCommand):

Функция executeExternalCommand предназначена для выполнения внешних команд, то есть тех, которые не являются встроенными в саму оболочку. Давайте разберем основные шаги этой функции:

- Определение флага фонового выполнения:

Команда может выполняться в фоновом режиме, если последний токен в команде равен "&". Если фоновое выполнение обнаружено, флаг background устанавливается в true, и последний токен обнуляется.

- Создание дочернего процесса (fork):

С помощью fork создается новый процесс. В этом новом процессе будет выполняться внешняя команда.

- Перенаправление ввода (dup2):

Если указан файл для ввода (inputFile), происходит открытие файла и его файловый дескриптор используется для замены стандартного ввода текущего процесса. Таким образом, внешняя команда читает данные из указанного файла вместо стандартного ввода.

- Обработка фонового выполнения:

Если команда выполняется в фоновом режиме (background), то ввод, вывод и ошибки перенаправляются в /dev/null, чтобы не влиять на текущий терминал.

- Замена текущего процесса на новую программу (execvp):

В дочернем процессе происходит замена текущего процесса на новую программу с помощью execvp. Если программа не может быть выполнена, выводится сообщение об ошибке.

- Ожидание завершения дочернего процесса:

В родительском процессе, если фоновое выполнение не установлено, родитель ждет завершения дочернего процесса с помощью waitpid.

1. Обработка внутренних команд (executeInternalCommand):

Функция executeInternalCommand предназначена для выполнения внутренних команд, таких как cd, clr, dir, environ, echo, pause, quit и help. Давайте рассмотрим, как обрабатываются некоторые из них:

- Команда "cd" - изменение текущей директории:

Если первый токен в команде равен "cd", то функция проверяет количество токенов. Если нет аргумента, то директория изменяется на текущую. В противном случае, директория изменяется на указанную, и обновляются переменные окружения OLDPWD и PWD.

- Команда "clr" - очистка экрана:

С помощью ANSI escape-последовательности printf("\x1B[2J\x1B[H") происходит очистка экрана.

- Команда "dir" - вывод списка файлов:

С использованием библиотеки <dirent.h> функция открывает текущую или указанную директорию и выводит список файлов в этой директории.

- Команда "environ" - вывод переменных окружения:

С использованием внешней переменной \_\_environ функция выводит все переменные окружения.

- Команда "echo" - вывод аргументов:

Функция просто выводит аргументы, указанные после "echo".

- Команда "pause" - ожидание нажатия клавиши Enter:

Функция выводит сообщение и ждет нажатия клавиши Enter с помощью getchar().

- Команда "quit" - выход из оболочки:

При встрече команды "quit" происходит завершение программы с помощью exit(EXIT\_SUCCESS).

- Команда "help" - вывод справки:

Функция executeHelpCommand использует команду less для вывода содержимого файла "readme", который предположительно содержит справку по использованию оболочки.

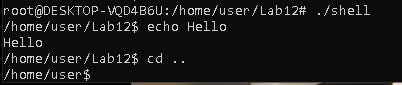
1. Обработка ввода-вывода (updateInputOutput, proccessInput):

- Функция updateInputOutput обновляет переменные ввода-вывода (inputFile, outputFile, appendOutput) в зависимости от наличия символов <, >, >> в команде.

- Функция proccessInput осуществляет токенизацию введенной строки и вызывает updateInputOutput. Затем она обрабатывает внутренние и внешние команды в соответствии с полученными токенами и переменными ввода-вывода.

Общий принцип работы оболочки заключается в том, что она создает новые процессы для выполнения команд, перенаправляет ввод и вывод, обрабатывает внутренние команды самостоятельно, а внешние команды выполняет, вызывая соответствующие системные вызовы.

# Тестирование







# Вывод

Проделав данную лабораторную работу, было рассмотрено и реализовано функциональное ядро простой командной оболочки на языке программирования C. Основные механизмы работы оболочки, включая выполнение и команд, обработку ввода пользователя, а также перенаправление ввода/вывода были тщательно изучены и освоены. Код оболочки структурирован и включает в себя реализацию различных системных вызовов для создания дочерних процессов, перенаправления потоков и управления файлами. Полученный опыт в программировании системных утилит позволил лучше понять внутреннее устройство командных интерпретаторов и их базовые принципы функционирования.

# Приложение. Код myshell.c

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <dirent.h>

#include "myshell.h"

#include "util.h"

extern char \_\_environ;

// Функция для выполнения внешней команды

void executeExternalCommand(char \*tokens[], int tokenCount, char \*inputFile, char \*outputFile, bool appendOutput)

{

bool background = false;

// Проверяем, выполняется ли команда в фоновом режиме

if (tokenCount > 0 && strcmp(tokens[tokenCount - 1], "&") == 0)

{

background = true;

tokens[tokenCount - 1] = NULL;

}

// Создаем новый процесс

pid\_t childPid = fork();

if (childPid == 0) // Дочерний процесс

{

// Перенаправляем ввод, если указан файл для ввода

if (inputFile)

{

int inputFd = open(inputFile, O\_RDONLY);

int oldinputFd = dup(0);

if (inputFd < 0)

{

fprintf(stderr, "Не удалось открыть файл для ввода!\n");

exit(EXIT\_FAILURE);

}

if (dup2(inputFd, STDIN\_FILENO) == -1)

{

fprintf(stderr, "Не удалось перенаправить ввод!\n");

}

close(inputFd);

}

// Если команда выполняется в фоновом режиме

if (background)

{

// Перенаправляем ввод/вывод/ошибки в /dev/null

int devNullFd = open("/dev/null", O\_RDWR);

if (devNullFd < 0)

{

fprintf(stderr, "Не удалось получить доступ к /dev/null!\n");

exit(EXIT\_FAILURE);

}

if (!inputFile)

{

dup2(devNullFd, STDIN\_FILENO);

}

if (!outputFile)

{

dup2(devNullFd, STDOUT\_FILENO);

}

dup2(devNullFd, STDERR\_FILENO);

close(devNullFd);

}

else // Если команда не выполняется в фоновом режиме

{

// Заменяем текущий процесс на новую программу

execvp(tokens[0], tokens);

fprintf(stderr, "Не удалось выполнить программу!\n");

exit(EXIT\_FAILURE);

}

}

else if (childPid > 0) // Родительский процесс

{

if (!background)

{

// Ждем завершения дочернего процесса

waitpid(childPid, NULL, 0);

}

}

else // Ошибка при создании процесса

{

fprintf(stderr, "Не удалось создать процесс!\n");

}

}

// Функция для выполнения команды "help"

void executeHelpCommand()

{

FILE \*lessPipe;

// Создаем pipe для less

lessPipe = popen("less", "w");

if (lessPipe == NULL)

{

perror("Не удалось создать pipe для less.");

return;

}

// Получаем путь к исполняемому файлу оболочки

char \*readmePath = strdup(getenv("SHELL"));

// Удаляем имя исполняемого файла из пути

char \*match = strstr(readmePath, SHELL\_NAME);

if (match != NULL)

{

size\_t len = strlen(SHELL\_NAME);

memmove(match, match + len, strlen(match + len) + 1);

}

// Открываем файл readme для чтения

int readmeFileFd = open(strcat(readmePath, "readme"), O\_RDONLY);

int oldinputFd = dup(0);

// Проверяем, удалось ли открыть файл

if (readmeFileFd < 0)

{

fprintf(stderr, "Не удалось открыть файл для чтения!\n");

if (pclose(lessPipe) == -1)

{

perror("Не удалось закрыть pipe для потока.");

return;

}

return;

}

// Перенаправляем ввод на содержимое файла

if (dup2(readmeFileFd, STDIN\_FILENO) == -1)

{

fprintf(stderr, "Не удалось перенаправить ввод!\n");

if (pclose(lessPipe) == -1)

{

perror("Не удалось закрыть pipe для потока.");

return;

}

return;

}

// Читаем содержимое файла и пишем в pipe для less

char buffer[MAX\_INPUT\_SIZE];

size\_t bytesReadCount;

while ((bytesReadCount = read(STDIN\_FILENO, buffer, sizeof(buffer))) > 0)

{

if (fwrite(buffer, 1, bytesReadCount, lessPipe) != bytesReadCount)

{

perror("Не удалось записать данные в поток.");

break;

}

}

// Очищаем буфер ввода

fflush(stdin);

dup2(oldinputFd, STDIN\_FILENO);

close(readmeFileFd);

// Закрываем pipe для less

if (pclose(lessPipe) == -1)

{

perror("Не удалось закрыть pipe для потока.");

return;

}

}

// Функция для выполнения внутренних команд

bool executeInternalCommand(char \*tokens[], int tokenCount, char \*outputFile, bool appendOutput)

{

if (tokenCount == 0)

{

return true;

}

// Команда "cd" - изменение текущей директории

if (strcmp(tokens[0], "cd") == 0)

{

if (tokenCount == 1)

{

if (chdir(".") != 0)

{

fprintf(stderr, "Недопустимый путь.\n");

}

}

else

{

if (chdir(tokens[1]) != 0)

{

fprintf(stderr, "Недопустимый путь.\n");

}

else

{

// Устанавливаем переменные окружения OLDPWD и PWD

setenv("OLDPWD", getenv("PWD"), 1);

setenv("PWD", getcwd(NULL, 0), 1);

}

}

return true;

}

// Команда "clr" - очистка экрана

else if (strcmp(tokens[0], "clr") == 0)

{

printf("\x1B[2J");

printf("\x1B[H");

return true;

}

// Команда "dir" - вывод списка файлов в текущей или указанной директории

else if (strcmp(tokens[0], "dir") == 0)

{

struct dirent \*entry;

DIR \*dir = ((tokenCount == 1 || (tokenCount > 1 && !!outputFile && tokens[1] == NULL)) ? opendir(".") : opendir(tokens[1]));

if (dir == NULL)

{

fprintf(stderr, "Не удалось открыть директорию!\n");

return true;

}

char \*filenames[MAX\_DIRECTORY\_CONTENTS];

int filesCnt = 0;

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

{

filenames[filesCnt++] = strdup(entry->d\_name);

}

closedir(dir);

return true;

}

// Команда "environ" - вывод переменных окружения

else if (strcmp(tokens[0], "environ") == 0)

{

char env = \_\_environ;

while (\*env)

{

printf("%s\n", \*env);

env++;

}

return true;

}

// Команда "echo" - вывод аргументов

else if (strcmp(tokens[0], "echo") == 0)

{

for (int i = 1; i < tokenCount; i++)

{

if (tokens[i] == NULL)

{

break;

}

printf("%s%s", tokens[i], ((i == tokenCount - 1) ? "\n" : " "));

}

return true;

}

// Команда "pause" - ожидание нажатия клавиши Enter

else if (strcmp(tokens[0], "pause") == 0)

{

printf("Нажмите Enter для продолжения...");

getchar();

return true;

}

// Команда "quit" - выход из оболочки

else if (strcmp(tokens[0], "quit") == 0)

{

exit(EXIT\_SUCCESS);

}

// Команда "help" - вывод справки

if (strcmp(tokens[0], "help") == 0)

{

executeHelpCommand();

return true;

}

return false;

}

// Переменные для хранения имени файла ввода, вывода и флага дозаписи

char \*inputFile = NULL;

char \*outputFile = NULL;

bool appendOutput = false;

// Обновление переменных ввода-вывода на основе аргументов командной строки

void updateInputOutput(char \*tokens[], int tokenCount)

{

inputFile = NULL;

outputFile = NULL;

for (int i = 0; i < tokenCount - 1; i++)

{

if (!inputFile && strcmp(tokens[i], "<") == 0)

{

if (i < tokenCount - 1)

{

inputFile = tokens[i + 1];

tokens[i] = NULL;

}

else

{

fprintf(stderr, "Отсутствует файл ввода!\n");

return;

}

}

else if (!outputFile && strcmp(tokens[i], ">") == 0)

{

if (i < tokenCount - 1)

{

outputFile = tokens[i + 1];

tokens[i] = NULL;

}

else

{

fprintf(stderr, "Отсутствует файл вывода!\n");

return;

}

}

else if (!outputFile && strcmp(tokens[i], ">>") == 0)

{

if (i < tokenCount - 1)

{

outputFile = tokens[i + 1];

tokens[i] = NULL;

appendOutput = true;

}

else

{

fprintf(stderr, "Отсутствует файл вывода!\n");

return;

}

}

}

}

// Обработка входной строки

void proccessInput(char input[], char \*tokens[], int tokenCount)

{

// Разбиваем строку на токены

tokenize(input, tokens, &tokenCount);

// Обновляем переменные ввода-вывода

updateInputOutput(tokens, tokenCount);

if (tokenCount > 0)

{

int outputFd = -1;

int oldoutputFd = dup(STDOUT\_FILENO);

if (outputFile)

{

// Открываем файл вывода

if (appendOutput)

{

outputFd = open(outputFile, O\_CREAT | O\_RDWR | O\_APPEND, 0644);

}

else

{

outputFd = open(outputFile, O\_CREAT | O\_WRONLY | O\_TRUNC, 0644);

}

// Проверяем, удалось ли открыть файл

if (outputFd < 0)

{

fprintf(stderr, "Не удалось открыть файл вывода!\n");

exit(EXIT\_FAILURE);

}

// Перенаправляем вывод в файл

if (dup2(outputFd, STDOUT\_FILENO) == -1)

{

fprintf(stderr, "Не удалось перенаправить вывод!\n");

}

}

// Выполняем внутреннюю или внешнюю команду

if (!executeInternalCommand(tokens, tokenCount, outputFile, appendOutput))

{

executeExternalCommand(tokens, tokenCount, inputFile, outputFile, appendOutput);

}

if (!!outputFile)

{

// Сбрасываем буфер вывода и восстанавливаем стандартный вывод

fflush(stdout);

dup2(oldoutputFd, STDOUT\_FILENO);

close(outputFd);

}

}

}

// Главная функция

int main(int argc, char \*argv[])

{

// Если указан файл с командами в качестве аргумента

if (argc == 2)

{

FILE \*cmd\_source = fopen(argv[1], "r");

if (cmd\_source == NULL)

{

fprintf(stderr, "Не удалось открыть файл с командами!\n");

exit(EXIT\_FAILURE);

}

char buffer[MAX\_INPUT\_SIZE];

char \*buf\_tokens[MAX\_TOKENS];

int buf\_tokenCount = 0;

// Читаем команды из файла и обрабатываем их

while (fgets(buffer, sizeof(buffer), cmd\_source) != NULL)

{

proccessInput(buffer, buf\_tokens, buf\_tokenCount);

}

fclose(cmd\_source);

return EXIT\_SUCCESS;

}

// Получаем текущую директорию

char \*cwd = getcwd(NULL, 0);

if (cwd == NULL)

{

fprintf(stderr, "Cannot get current directory!\n");

exit(EXIT\_FAILURE);

}

// Выделяем память для строки с путем до файла оболочки

char \*shellPath = (char \*)malloc(strlen(cwd) + strlen(SHELL\_NAME) + 2);

if (shellPath == NULL)

{

fprintf(stderr, "malloc failed!\n");

free(cwd);

exit(EXIT\_FAILURE);

}

// Формируем путь к файлу оболочки и выставляем переменную окружения

sprintf(shellPath, "%s/%s", cwd, SHELL\_NAME);

setenv("SHELL", shellPath, 1);

free(cwd);

free(shellPath);

char input[MAX\_INPUT\_SIZE];

char \*tokens[MAX\_TOKENS];

int tokenCount = 0;

while (true)

{

char \*current\_dir = getcwd(NULL, 0);

printf("%s$ ", current\_dir);

free(current\_dir);

// Если не можем получить входную строку или достигнут конец файла

if (fgets(input, MAX\_INPUT\_SIZE, stdin) == NULL)

{

break;

}

proccessInput(input, tokens, tokenCount);

}

return EXIT\_SUCCESS;

}