Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Ульяновский государственный технический университет»
Кафедра «Информатика и вычислительная техника»

Операционные системы
(название дисциплины)

Лабораторная работа №1

«Разработка оболочки»
(название (тема) работы)

Выполнил:
студент группы ИВТАПбд-31
Молчанов А. В.
(Фамилия И. О.)

Проверил(а):
преподаватель, аспирант кафедры «ВТ»
(должность)
Беляев К. С.
(Фамилия И. О.)

Ульяновск
2023

Постановка задачи

Написать простую оболочку *myshell* со следующими свойствами:

1. Оболочка должна поддерживать следующие внутренние команды:

   a. *cd <directory>* – смена текущего каталога по умолчанию на *<directory>*. Если аргумент *<directory>* отсутствует, вывести текущий каталог. Если каталог отсутствует, вывести соответствующее сообщение об ошибке. Эта команда должна также соответствующим образом изменять переменную среды *PWD*;

   b. *clr* – очистка экрана;

   c. *dir <directory>* – вывод содержимого каталога *<directory>*;

   d. *environ* – вывод всех переменных среды;

   e. *echo <comment>* – вывод на экран *<comment>*, после которого выполняется переход на новую строку (множественные пробелы/табуляции могут быть сокращены до единственного пробела);

   f. *help* – вывод руководства пользователя с использованием фильтра *more*;

   g. *pause* – приостановка операций оболочки до нажатия клавиши *<Enter>*;

   h. *quit* – выход из оболочки;

   i. Среда оболочки должна содержать переменную *shell=<pathname>/myshell*, где *<pathname>/myshell* – полный путь к выполнимому файлу оболочки (не «прошитый» путь к вашему каталогу, а тот, откуда была выполнена оболочка).

2. Все прочие входные данные командной строки интерпретируются как вызовы программ, которые должны выполняться оболочкой с использованием механизмов *fork* и *exec* как собственные дочерние процессы. Программы должны выполняться в среде, содержащей переменную *parent=<pathname>/myshell*, где *<pathname>/myshell* такие же, как описано выше, в задании 1.i;

3. Оболочка должна быть в состоянии получать данные командной строки из файла. То есть оболочка вызывается с аргументом командной строки *myshell batchfile* и предполагается, что файл *batchfile* содержит набор командных строк для обработки оболочкой. По достижению конца файла должно быть выполнено завершение работы оболочки. Очевидно, что если оболочка вызывается без аргумента командной строки, то она запрашивает ввод от пользователя через приглашение на дисплее;

4. Оболочка должна поддерживать перенаправление ввода-вывода для *stdin* и/или *stdout*, то есть командная строка *programname argl arg2 < inputfile > outputfile* должна выполнить программу *programname* с аргументами *argl* и *arg2* с заменой файлового потока *stdin* файлом *inputfile*, а файлового потока *stdout* – файлом *outputfile*. Перенаправление *stdout* должно также быть возможным для внутренних команд *dir*, *environ*, *echo* и *help*. При перенаправлении вывода символ > должен приводить к созданию *outputfile*, если такового не существует, и его усечению, если он имеется. При перенаправлении >> файл *outputfile* создаётся, если он ещё не существует, а если существует, выполняется дозапись в конец файла;

5. Оболочка должна поддерживать фоновое выполнение программ. Амперсанд (&) в конце командной строки указывает, что оболочка должна вернуться к командной строке сразу после запуска данной программы;

6. Приглашение командной строки должно содержать путь к текущему каталогу.

Детали реализации

1.

a. Смена текущей рабочей директории на каталог из аргумента *<directory>* производится с помощью системной функции *chdir*. После перехода, вызывается функция *getcwd* для получения нового пути до текущей рабочей директории и его сохранения в переменную среды *PWD*:

```
int cd(char** args)
{
    if (args[1] == NULL)
    {
        printf("[ERROR] Expected argument for \"cd\" command!\n");
    } else if (chdir(args[1]) != 0)
    {
        printf("[ERROR] Couldn't change directory to \"%s\"!\n", args[1]);
    }

    if (getcwd(cwd, MAX_DIRECTORY_PATH) != NULL)
    {
        char* PWD = malloc(strlen("PWD=") + strlen(cwd) + 1);

        memcpy(PWD, "PWD=", strlen("PWD="));
        memcpy(PWD + strlen("PWD="), cwd, strlen(cwd) + 1);

        putenv(PWD);
    }

    return CONTINUE;
}
```

Рисунок 1

b. Очистка экрана производится с использованием системной функции *system(“clear”)*;

c. Для вывода содержимого каталога *<directory>* использовались такие специальные структуры данных, как *DIR* (сама директория) и *dirent* (её содержимое). Функция *opendir* использовалась для открытия папки, *readdir* – для чтения её содержимого, а *closedir* – для закрытия каталога:

```c
void dir_print(char** args)
{
    DIR *directory;
    struct dirent *directory_entity;

    if (args[1] != NULL)
    {
        directory = opendir(args[1]);
    }
    else
    {
        directory = opendir(".");
    }

    while ((directory_entity = readdir(directory)))
        printf("%s ", directory_entity->d_name);

    printf("\n");

    if (directory_entity != NULL)
    {
        if (strcmp(".", directory_entity->d_name) && strcmp("..", directory_entity->d_name))
            closedir(directory);
    }
}
```

Рисунок 2

d. Вывод всех переменных среды из внешнего (*extern*) массива строк под названием *environ* осуществляется перебором его элементов циклом *while* и функцией для вывода *printf*;

e. Вывод на экран аргумента <*comment*> так же, как и вывод переменных среды реализован с использованием цикла *while* и функции *printf*. Вывод производится по отдельным словам (токенам) через пробел, так как функция парсинга пользовательского ввода *split* вызывается перед исполнением той или иной команды из него;

f. Вывод руководства пользователя с использованием фильтра *more* выполнен с применением так называемых каналов данных *pipe*, которые позволяют подать вывод одной выполненной команды (*help*) на ввод другой (*more*):

```
if (pid < 0)
{
    printf("[ERROR] Couldn't create child process!\n");
}
else if (pid == 0)
{
    if (pipe_descriptor != STDIN_FILENO)
    {
        // Перенаправляем предыдущий канал на стандартный ввод
        dup2(pipe_descriptor, STDIN_FILENO);
        close(pipe_descriptor);
    }

    // Перенаправляем стандартный вывод stdout на текущий канал
    dup2(p[1], STDOUT_FILENO);
    close(p[1]);

    printf("%s", help_text); // Выполняем команду "help" с перенаправленным выводом

    exit(1);
}
else
{
    // Закрываем для чтения предыдущий канал, так как его больше не существует (дочерний процесс завершил свою работу)
    close(pipe_descriptor);
    // Закрываем для записи текущий канал
    close(p[1]);

    // Сохраняем то, что мы прочитали (результат выполнения команды "help")
    pipe_descriptor = p[0];

    if (pipe_descriptor != STDIN_FILENO)
    {
        // Перенаправляем это на стандартный ввод
        dup2(pipe_descriptor, STDIN_FILENO);
        close(pipe_descriptor);
    }

    // Выполняем команду "more" с перенаправленным вводом
    if (execvp(more_execvp[0][0], more_execvp[0]) == -1)
    {
        printf("[ERROR] Couldn't execute unknown command!\n");
    }
    else
    {
        printf("\nThe end of the help command has been reached!\n");
    }
}
```

Рисунок 3

g. Приостановка операций оболочки до нажатия клавиши <*Enter*> представляет из себя бесконечный цикл *while* с функцией получения кода нажатой клавиши *getch* внутри и условием, пока этот код не будет равен числу 10 – коду клавиши <*Enter*>;

h. Выход из оболочки не ограничивается одним лишь вызовом системной функции *exit(0)*, а завершает перед этим, в первую очередь, главный (*foreground*) процесс и все фоновые (*background*) процессы из списка:

```
int quit()
{
    bg_task* background_task;

    // Игнорируем (SIG_IGN) сигналы, посылаемые дочерними (SIGCHLD) процессами при изменении их статуса (завершён, приостановлен или возобновлён)
    signal(SIGCHLD, SIG_IGN);

    if (!shell_tasks.foreground.finished)
        kill_foreground();

    for (size_t i = 0; i < shell_tasks.iterator; ++i)
    {
        background_task = &shell_tasks.background[i];

        if (!background_task->finished)
            kill(background_task->pid, SIGTERM); // SIGTERM - сигнал для запроса завершения процесса по его идентификатору (PID)
    }

    return EXIT;
}
```

Рисунок 4

i. Функция *init_environ* позволяет установить такие переменные среды как *SHELL* и *PARENT* при запуске оболочки, используя в свою очередь системные функции для копирования и выделения областей памяти *memcpy* и *malloc*, а также *getcwd*, *strlen* и *putenv* для получения текущей рабочей директории, вычисления длины строки и сохранения строки переменной среды соответственно:

```
void init_environ()
{
    if (getcwd(cwd, MAX_DIRECTORY_PATH) != NULL)
    {
        memcpy(cwd + strlen(cwd), "/myshell.exe", strlen("/myshell.exe"));

        char* SHELL = malloc(strlen("SHELL=") + strlen(cwd));
        memcpy(SHELL, "SHELL=", strlen("SHELL="));
        memcpy(SHELL + strlen("SHELL="), cwd, strlen(cwd) + 1);

        char* PARENT = malloc(strlen("PARENT=") + strlen(cwd));
        memcpy(PARENT, "PARENT=", strlen("PARENT="));
        memcpy(PARENT + strlen("PARENT="), cwd, strlen(cwd) + 1);

        putenv(SHELL);
        putenv(PARENT);
    }
}
```

Рисунок 5

2.

Выполнение прочих команд возлагается на функцию *launch*, которая использует функцию *fork* для создания дочернего процесса и функцию *execvp* для выполнения поданных команд. Также, данная функция отдельно обрабатывает фоновые процессы, если таковые имеются (*is_background_task == true*), и перенаправляет ввод-вывод. Родительский процесс, который породил дочерний, ждёт его завершения с применением функции *waitpid*:

```
int launch(char** args)
{
    pid_t pid;
    pid = fork();

    if (pid < 0)
    {
        printf("[ERROR] Couldn't create child process!\n");
    }
    else if (pid == 0)
    {
        if (is_background_task)
            // Помещаем фоновый (дочерний) процесс в новую группу процессов. Главный (родительский) процесс при этом просто продолжает свою работу
            setpgid(0, 0);

        if (input_redirection)
            open_file_descriptor(0);

        if (output_redirection)
            open_file_descriptor(1);

        if (execvp(args[0], args) == -1)
            printf("[ERROR] Couldn't execute unknown command!\n");

        exit(1);
    }
    else
    {
        // Если текущий процесс - фоновый, то добавляем его в список фоновых процессов shell_tasks.background и не ждём (waitpid) его завершения в главном процессе
        if (is_background_task)
        {
            if (add_background(pid, args[0]) == 0)
                quit();
        }
        else
        {
            set_foreground(pid);

            if (waitpid(pid, NULL, 0) == -1)
            {
                if (errno != EINTR)
                    printf("[ERROR] Couldn't track the completion of the process!\n");
            }
        }
    }

    return CONTINUE;
}
```

Рисунок 6

3.

Исполняемый файл оболочки *myshell.exe* можно запустить как с аргументом *batchfile* для получения из него командных строк для их обработки оболочкой, так и без него.

При передачи второго аргумента, файл открывается для чтения при помощи функции *fopen*. Далее, в бесконечном цикле *do while*, который характеризует непрерывную и безошибочную работу командной строки, происходит построчное считывание *batchfile* с использованием функции *fgets*, разделение считанных строк на слова функцией *split* и их сохранение в массив *args*, который затем передаётся в качестве аргумента функции *execute*. По достижению конца пакетного файла производится завершение работы оболочки.

При отсутствии второго аргумента, перед выполнением описанных выше действий, вызываются функции *display* и *readline*. Первая отображает так называемое приглашение командной строки, которое содержит в себе путь к текущему каталогу (задание 6), а вторая функция считывает пользовательский ввод в строковую переменную *line*, которая является эквивалентом строковой переменной *fileline* при работе с пакетным файлом:

```
if (argc == 2)
{
    if ((batchfile = fopen(argv[1], "r")) == NULL)
    {
        printf("[ERROR] Failed to open the file!\n");
        exit(2); // 2 - серьёзная ошибка
    }
}

init_environ();

do
{
    if (argc != 2)
    {
        display();

        line = readline();

        if (line == NULL)
            exit(1); // 1 - незначительная проблема
        args = split(line);

        if (args == NULL)
        {
            free(line);
            exit(2);
        }
    }
    else
    {
        if (fgets(fileline, MAX_FILE_CAPACITY, batchfile) == NULL)
        {
            exit(0); // 0 - ошибок нет
        }
        else
        {
            fileline[strcspn(fileline, "\n")] = 0;
            args = split(fileline);
        }

        if (args == NULL)
            exit(2);
    }

    status = execute(args); // Пытаемся выполнить введённую пользователем команду и получаем статус о её выполнении

    free(line);
    free(args);

} while (status); // Командная оболочка запущена, пока status == CONTINUE и прерывается, когда status == EXIT

return 0;
```

Рисунок 7

```c
void display()
{
    uid_t uid = geteuid();
    struct passwd *pw = getpwuid(uid);

    if (pw != NULL)
        printf("%s%s%s:", PRIMARY_COLOR, pw->pw_name, RESET_COLOR);

    if (getcwd(cwd, MAX_DIRECTORY_PATH) != NULL)
        printf("%s%s%s", SECONDARY_COLOR, cwd, RESET_COLOR);

    printf(": ");
}

char* readline()
{
    char* line = NULL;
    size_t size = 0;
    ssize_t line_length;

    if ((line_length = getline(&line, &size, stdin)) == -1)
    {
        if (errno != 0)
            printf("[ERROR] Couldn't read from stdin!\n");

        free(line);
        printf("\n");

        return NULL;
    }

    if (line[line_length - 1] == '\n')
        line[line_length - 1] = '\0';

    return line;
}
```

Рисунок 8

```
char** split(char* line)
{
    size_t position  = 0;
    size_t buffer_size = DEFAULT_BUFF_SIZE;

    char* token;
    char** tokens = (char**)malloc(sizeof(char*) * buffer_size);

    input_redirection = false;
    output_redirection = false;
    append_output_file = false;
    is_background_task = false;

    if (tokens == NULL)
    {
        printf("[ERROR] Couldn't allocate buffer for splitting!\n");
        return NULL;
    }

    token = strtok(line, TOKENS_DELIMITERS);

    while (token != NULL)
    {
        if (token[0] == '<')
        {
            input_redirection = true;
            input_file_name = strtok(NULL, TOKENS_DELIMITERS);
        }
        else if (token[0] == '>')
        {
            if (&token[1] != NULL && token[1] == '>')
                append_output_file = true;
            output_redirection = true;

            output_file_name = strtok(NULL, TOKENS_DELIMITERS);
        }
        else if (token[0] == '&')
        {
            is_background_task = true;
        }
        else
        {
            tokens[position++] = token;
        }

        if (position >= buffer_size)
        {
            buffer_size *= 2;
            tokens = (char**)realloc(tokens, buffer_size * sizeof(char*));

            if (tokens == NULL)
            {
                printf("[ERROR] Couldn't reallocate buffer for tokens!\n");
                return NULL;
            }
        }

        token = strtok(NULL, TOKENS_DELIMITERS);
    }

    tokens[position] = NULL;
    return tokens;
}
```

Рисунок 9

```c
int execute(char** args)
{
    if (args[0] == NULL) {
        return CONTINUE;
    } else if (strcmp(args[0], "cd") == 0) {
        return cd(args);
    } else if (strcmp(args[0], "clr") == 0) {
        return clear();
    } else if (strcmp(args[0], "dir") == 0) {
        return dir(args);
    } else if (strcmp(args[0], "environ") == 0)
        return env();
    } else if (strcmp(args[0], "echo") == 0) {
        return echo(args);
    } else if (strcmp(args[0], "help") == 0) {
        return help();
    } else if (strcmp(args[0], "pause") == 0) {
        return pause();
    } else if (strcmp(args[0], "quit") == 0) {
        return quit();
    } else if (strcmp(args[0], "jobs") == 0) {
        return jobs();
    } else {
       return launch(args);
    }
}
```

Рисунок 10

4.

Перенаправление ввода-вывода реализовано в полной мере, как того требует задание, в функции *open_file_descriptor*, которая принимает тип дескриптора (0 – ввод, 1 – вывод) в качестве аргумента. Используя такие системные функции как *open*, *dup2* и *close*, дескрипторы стандартного ввода-вывода *stdin* и *stdout* открываются с использованием необходимого файла (входного или выходного), дублируются (переопределяются) и после этого закрываются для экономии памяти. Перечисленные внутренние команды, такие как *dir*, *environ*, *echo* и *help* вызывают функцию *open_file_descriptor* с аргументом 1 (перенаправление вывода):

```
void open_file_descriptor(int descriptor_type)
{
    int descriptor = -1;

    switch (descriptor_type)
    {
        case 0:
            if ((descriptor = open(input_file_name, O_RDONLY, 0)) < 0)
                printf("[ERROR] Failed to open the input file!\n");

            dup2(descriptor, STDIN_FILENO);
            close(descriptor);
            break;

        case 1:
            if (append_output_file)
            {
                if ((descriptor = open(output_file_name, O_CREAT | O_RDWR | O_APPEND, 0644)) < 0)
                    printf("[ERROR] Failed to open the output file!\n");
            }
            else if ((descriptor = creat(output_file_name, 0644)) < 0)
                printf("[ERROR] Failed to open the output file!\n");

            dup2(descriptor, STDOUT_FILENO);
            close(descriptor);
            break;

        default:
            printf("[ERROR] An unsupported file descriptor type was specified!\n");
            break;
    }
}
```

Рисунок 11

5.

Символ ‘&’ в конце командной строки позволяет начать фоновое выполнение команды, запустив её, и передав управление обратно главному (*foreground*) процессу оболочки. Функция *add_background* позволяет добавить фоновый процесс и всю связанную с ним информацию в соответствующий список структуры *shell_tasks* и её вспомогательные поля, а функция под названием *catch_finished_task* вызывается исключительно по системному сигналу *SIGCHLD*, который характеризует любое изменение статуса дочерних процессов (завершён, приостановлен или возобновлён), так как все фоновые процессы запускаются как дочерние от основного процесса оболочки:

```
typedef struct fg_task_t fg_task;
struct fg_task_t // Структура, характеризующая главный процесс командной оболочки
{
    pid_t pid; // Идентификатор (ID) процесса
    bool finished; // Состояние процесса
};

typedef struct bg_task_t bg_task;
struct bg_task_t // Структура, характеризующая фоновый процесс командной оболочки
{
    pid_t pid; // Идентификатор (ID) процесса
    bool finished; // Состояние процесса
    char command[DEFAULT_BUFF_SIZE]; // Команда для выполнения
};

typedef struct tasks_t tasks;
struct tasks_t // Вспомогательная структура, хранящая в себе все активные процессы программы
{
    fg_task foreground; // Информация о главном процессе
    bg_task* background; // Список фоновых процессов
    size_t iterator; // Итератор списка фоновых процессов
    size_t size; // Размер списка фоновых процессов
};
```

Рисунок 12

```c
int add_background(pid_t pid, char* command)
{
    bg_task* background_task;

    // Если закончилось место в списке фоновых процессов shell_tasks.background
    if (shell_tasks.iterator >= shell_tasks.size)
    {
        // Выделяем дополнительную память под новый процесс
        shell_tasks.size = shell_tasks.size * 2 + 1
        shell_tasks.background = (bg_task*)realloc(shell_tasks.background, sizeof(bg_task) * shell_tasks.size);

        if (shell_tasks.background == NULL)
        {
            printf("[ERROR] Couldn't reallocate buffer for background tasks!\n");
            return EXIT;
        }
    }

    // Сохраняем информацию о процессе в списке
    background_task = &shell_tasks.background[shell_tasks.iterator];
    background_task->pid = pid;
    background_task->finished = false;
    strcpy(background_task->command, command);

    printf("[%zu] %d\n", shell_tasks.iterator, background_task->pid);

    shell_tasks.iterator += 1; // Берём следующий фоновый процесс

    return CONTINUE;
}
```

tasks shell_tasks

Рисунок 13

```c
void catch_finished_task()
{
    bg_task* background_task;
    pid_t pid = waitpid(-1, NULL, 0); // Ловим завершение любого (главного или фонового) из запущенных процессов

    // В зависимости от типа процесса, помечаем его как завершённый
    if (pid == shell_tasks.foreground.pid)
    {
        shell_tasks.foreground.finished = true;
    }
    else
    {
        for (size_t i = 0; i < shell_tasks.iterator; ++i)
        {
            background_task = &shell_tasks.background[i];

            if (background_task->pid == pid)
            {
                printf("\n[%zu]+ Done\n", i);
                background_task->finished = true;
                break;
            }
        }
    }
}
```

Рисунок 14