Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №1 по курсу

«Операционные системы»

Группа: М8О-210БВ-24

Студент: Самойлов Д. А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 02.10.25

Москва, 2025

Постановка задачи

Вариант 20.

Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами.Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Правило фильтрации: строки длины больше 10 символов отправляются в pipe2, иначе в pipe1. Дочерние процессы инвертируют строки.

Общий метод и алгоритм решения

Использованные системные вызовы:

- pid_t fork(void); – создает дочерний процесс.
- int pipe(int *fd); – создание неименованного канала для передачи данных между процессами
- void exit(int status) - завершения выполнения процесса и возвращение статуса
- int open(const char *pathname, int flags, mode_t mode) - открытие\создание файла
- int close(int fd) - закрыть файл
- int dup2(int oldfd, int newfd) - переназначение файлового дескриптора
- pid_t wait(int *status) - Ожидание завершения дочернего процесса
- int execl(const char* path, const char* args, …) - замена образа памяти процесса

Код программы

client.c

```c
#include <ctype.h>
#include <unistd.h>
#include <wait.h>
#include <stdio.h>
#include <stdlib.h>

#include "invert.h"

#define FILE_LEN 50
```

```c
int main() {
    int p1[2], p2[2];
    if (pipe(p1) == -1 || pipe(p2) == -1) {
        const char msg[] = "pipe() failed\n";
        write(STDERR_FILENO, msg, strlen(msg));
        exit(EXIT_FAILURE);
    }
    char filename1[FILE_LEN], filename2[FILE_LEN];
    ssize_t n = read(STDIN_FILENO, filename1, FILE_LEN - 1); // -терминирующий ноль
    if (n <= 0) {
        const char msg[] = "read() failed\n";
        write(STDERR_FILENO, msg, strlen(msg));
        exit(EXIT_FAILURE);
    }
    filename1[n - 1] = 0; // без \n
    ssize_t m = read(STDIN_FILENO, filename2, FILE_LEN - 1);
    if (m <= 0) {
        const char msg[] = "read() failed\n";
        write(STDERR_FILENO, msg, strlen(msg));
        exit(EXIT_FAILURE);
    }
    filename2[m - 1] = 0; // без \n



    pid_t pid1 = fork();
    if (pid1 == 0) {
        dup2(p1[0], STDIN_FILENO);
        close(p1[0]);
        close(p1[1]);
        close(p2[0]);
        close(p2[1]);
        execl("./server", "server", filename1, NULL);
        const char msg[] = "exec server failed\n";
        // если сюда попадаем, то это значит что наш процесс не смог развернуться,
        // ведь при exec вся старая программа стирается и разворачивается новая
        write(STDOUT_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }
    pid_t pid2 = fork();
    if (pid2 == 0) {
        dup2(p2[0], STDIN_FILENO);
        close(p1[0]);
        close(p1[1]);
        close(p2[0]);
```

```
        close(p2[1]);
        execl("./server", "server", filename2, NULL);
        const char msg[] = "exec server failed\n";
        write(STDOUT_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }
    // Parent
    close(p1[0]);
    close(p2[0]);

    ssize_t bytes_read;
    char line[BUFFER_SIZE], buff[BUFFER_SIZE];
    size_t pos = 0;
    while ((bytes_read = read(STDIN_FILENO, line, BUFFER_SIZE - 1))) {
        if (bytes_read < 0) {
            const char msg[] = "read() failed\n";
            write(STDERR_FILENO, msg, strlen(msg));
            exit(EXIT_FAILURE);
        }
        line[bytes_read] = 0;
        for (int i = 0; i <= bytes_read; ++i) {
            if (line[i] == '\n' || line[i] == '\0') {
                buff[i] = 0;
                if (pos > 0) {
                    // чтобы не закидывать пустые строки в ответ
                    size_t len = strlen(buff);
                    if (len > 10) {
                        write(p2[1], buff, len);
                        write(p2[1], "\n", 1);
                    }
                    else {
                        write(p1[1], buff, len);
                        write(p1[1], "\n", 1);
                    }
                }
                pos = 0;
            }
            else {
                buff[pos++] = line[i];
            }
        }
    }
    close(p1[1]);
    close(p2[1]);
```

```
    wait(NULL);
    wait(NULL);

    exit(EXIT_SUCCESS);
}
```

server.c

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

#include "invert.h"

int main(int argc, char* argv[]) {
    if (argc < 2) {
        const char msg[] = "Not enough arguments\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(1);
    }
    char* filename = argv[1];
    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC | O_APPEND, 0644);
    if (fd == -1) {
        const char msg[] = "Child error opening file\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(1);
    }
    ssize_t bytes_read;
    char line[BUFFER_SIZE], inv_buff[BUFFER_SIZE], buff[BUFFER_SIZE];
    size_t pos = 0;
    while ((bytes_read = read(STDIN_FILENO, line, BUFFER_SIZE - 1))) {
        if (bytes_read < 0) {
            const char msg[] = "read() failed\n";
            write(STDERR_FILENO, msg, strlen(msg));
            exit(EXIT_FAILURE);
        }
        line[bytes_read] = 0;

        // Обрабатываем буфер посимвольно для разбиения на строки
```

```
        for (int i = 0; i <= bytes_read; i++) {
            if (line[i] == '\n' || line[i] == 0) {
                buff[pos] = 0;

                if (pos > 0) {
                    // Инвертируем строку
                    char* res = reverse(buff, inv_buff);

                    // Добавляем перенос строки и выводим
                    char output[BUFFER_SIZE];
                    snprintf(output, BUFFER_SIZE, "%s\n",
res);
                    size_t len = strlen(output);
                    write(STDOUT_FILENO, output, len);
                    write(fd, output, len);
                }
                pos = 0;
            }
            else {
                buff[pos++] = line[i];
            }
        }
    }
    close(fd);
    exit(0);
}
```

Протокол работы программы

test1

```
$ ./client
file1.txt
file2.txt
1234
123
12
1
12345678910
aaaaaaaaaaaaab
$ cat < file1.txt
4321
321
21
```

```
1
$ cat < file2.txt
01987654321
baaaaaaaaaaaaa
```

test2

```
$ ./client
input.txt
output.txt
hello world
test string
12345

$ cat < input.txt
dlrow olleh
gnirts tset
54321

$ cat < output.txt
hello world
test string
12345
```

test3

```
$ ./client.c
data.in
data.out
apple banana cherry
python java c++

$ cat < data.in
yrrehc ananab elppa
++c avaj nohtyp

$ cat < data.out
apple banana cherry
python java c++
```

test4

```
$ ./client
in.txt
out.txt
This is a test file
```

```
Multiple lines
of text

$ cat in.txt
elif tset a si sihT
senil elpitluM
txet fo

$ cat < out.txt
This is a test file
Multiple lines
of text
```

Вывод

При выполнении лабораторной работы я отработал на практике вышеперечисленные системные вызовы. Разобрался в низкоуровневой работе процессов и файлов. В процессе выполнения работы наиболее сложными этапами были переназначения файловых дескрипторов и синтаксические особенности работы строк с языка си.