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-211БВ-24

Студент: Рыбин В.В.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 21.09.25

Москва, 2025

Постановка задачи

Вариант 12.

Группа вариантов 3. Родительский процесс создает два дочерних процесса. Перенаправление стандартных потоков ввода-вывода показано на картинке выше. Child1 и Child2 можно «соединить» между собой дополнительным каналом. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1. Процесс child1 и child2 производят работу над строками. Child2 пересылает результат своей работы родительскому процессу. Родительский процесс полученный результат выводит в стандартный поток вывода.

Child1 переводит строки в верхний регистр. Child2 убирает все задвоенные пробелы.

Общий метод и алгоритм решения

Использованные системные вызовы:

- pid_t fork(void); – создает дочерний процесс.
- int pipe(int *fd); – создает канал между двумя файловыми дескрипторами. Возвращает -1, если возникла ошибка при создании. Заполняет массив fd.
  fd[0] – файловый дескриптор для чтения
  fd[1] – файловый дескриптор для записи
- ssize_t readlink(char *path, char* buf, ssize_t bufsiz); - считывает содержание символической ссылки и записывает в buf.
- int write(int fd, const void *buf, size_t count); – записывает данные из буфера по файловому дескриптору.
- int read(int fd, void* buf, size_t count); - читает данные из файла по файловому дескриптору и записывает в buf.
- int execv(const char *path, char *const argv[]); - заменяет текущий процесс новым процессом, загружая и выполняя указанную программу
- pid_t wait(int *wstatus); - ожидает завершения любого из дочерних процессов, возвращает информацию о его завершении
- pid_t waitpid(pid_t pid, int *wstatus, int options); - ожидает завершения конкретного дочернего процесса.

Я реализовал межпроцессорное взаимодействие с помощью системных вызовов. Есть родительский процесс, который порождает два дочерних процесса. Первый преобразует все символы в верхний регистр, а второй удаляет все сдвоенные пробелы. Общаются между собой процессы с помощью канала, созданным функцией pipe. Пользователь общается только с родительским процессом.

Код программы

client.c

```c
#include <unistd.h> // системные вызовы

#include <stdint.h> // для uint32_t, int32_t
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
```

```c
static char SERVER_PROGRAM_SERVER_1[] = "server_1";
static char SERVER_PROGRAM_SERVER_2[] = "server_2";


int main(int argc, char * argv[]) {
    if (argc == 1) {
        char msg[1024];
        uint32_t len = snprintf(msg, sizeof(msg) - 1, "usage: %s filename\n",
argv[0]);
        write(STDERR_FILENO, msg, len);
        exit(EXIT_FAILURE);
    }


    char progpath[1024];
    {
        ssize_t len = readlink("/proc/self/exe", progpath, sizeof(progpath) - 1);

        if (len == -1) {
            const char msg[] = "error: failed to read full program path\n";
            write(STDERR_FILENO, msg, sizeof(msg));
            exit(EXIT_FAILURE);
        }

        while (progpath[len] != '/') {
            --len;
        }

        progpath[len] = '\0';
    }

    int pipe_client_to_child1[2];
    int pipe_child1_to_child2[2];
    int pipe_child2_to_client[2];


    if (pipe(pipe_client_to_child1) == -1) {
        const char msg[] = "error: failed to create pipe client to server 1\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    if (pipe(pipe_child1_to_child2) == -1) {
        const char msg[] = "error: failed to create pipe server 1 to server 2\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    if (pipe(pipe_child2_to_client) == -1) {
        const char msg[] = "error: failed to create pipe server 2 to client\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
```

```
    }


    const pid_t child1 = fork(); // создаю новый процесс

    switch (child1) { // если ребенок 1
        case -1: {
            const char msg[] = "error: failed to spawn new process\n";
            write(STDERR_FILENO, msg, sizeof(msg));
            exit(EXIT_FAILURE);
        } break;

        case 0: {
            close(pipe_child2_to_client[0]); // закрыл child2 - client
            close(pipe_child2_to_client[1]);

            close(pipe_client_to_child1[1]); // закрыл запись клиент - child1
            close(pipe_child1_to_child2[0]); // закрыл чтение child1 - child2

            dup2(pipe_client_to_child1[0], STDIN_FILENO); // перенаправил чтение из
клиента в стандартный поток ввода
            close(pipe_client_to_child1[0]);

            dup2(pipe_child1_to_child2[1], STDOUT_FILENO); // перенаправил запись в
сервер 2 в стандартный поток вывода
            close(pipe_child1_to_child2[1]);

            {
                char path[4096];
                snprintf(path, sizeof(path) - 1, "%s/%s", progpath,
SERVER_PROGRAM_SERVER_1);

                char *const args[] = {SERVER_PROGRAM_SERVER_1, argv[1], NULL};

                int32_t status = execv(path, args);

                if (status == -1) {
                    const char msg[] = "error: failed to exec into new executable
image\n";
                    write(STDERR_FILENO, msg, sizeof(msg));
                    exit(EXIT_FAILURE);
                }
            }
        } break;

        default: { // если родительский процесс
            const pid_t child2 = fork(); // создаю второго ребенка

            switch (child2) {
                case -1: {
                    const char msg[] = "error: failed to spawn new process\n";
                    write(STDERR_FILENO, msg, sizeof(msg));
                    exit(EXIT_FAILURE);
```

```
            } break;

            case 0: { // если второй ребенок
                close(pipe_client_to_child1[0]); // закрываем pipe client-
server 1
                close(pipe_client_to_child1[1]);

                close(pipe_child1_to_child2[1]); // закрываем запись server 1 -
server 2
                close(pipe_child2_to_client[0]); // закрываем чтение сервер 2 -
клиент

                dup2(pipe_child1_to_child2[0], STDIN_FILENO);
                close(pipe_child1_to_child2[0]);

                dup2(pipe_child2_to_client[1], STDOUT_FILENO);
                close(pipe_child2_to_client[1]);

                {
                    char path[4096];
                    snprintf(path, sizeof(path) - 1, "%s/%s", progpath,
SERVER_PROGRAM_SERVER_2);

                    char *const args[] = {SERVER_PROGRAM_SERVER_2, argv[1],
NULL};

                    int32_t status = execv(path, args);

                    if (status == -1) {
                        const char msg[] = "error: failed to exec into new
executable image\n";
                        write(STDERR_FILENO, msg, sizeof(msg));
                        exit(EXIT_FAILURE);
                    }
                }
            } break;

            default: { // логика родителя :
            //  получаем данные от клиента - отправляем данные на сервер 1 -
получаем данные из сервера 2
                pid_t pid = getpid();
                {

                    char msg[256];

                    const int32_t length = snprintf(
                        msg, sizeof(msg),
                        "PID %d: I`m a parent, my child 1 has PID: %d, child 2
has PID: %d.\n           To exit, press CTRL+C\n",
                        pid, child1, child2);
                    write(STDOUT_FILENO, msg, length);
                }

                close(pipe_client_to_child1[0]);
```

```
                    close(pipe_child1_to_child2[0]);
                    close(pipe_child1_to_child2[1]);
                    close(pipe_child2_to_client[1]);

                    char buf[8096];
                    ssize_t bytes;

                    char msg[1024];
                    const int32_t length = snprintf(msg, sizeof(msg),
                        "PID %d: Write a message that needs to be changed: ", pid);
                    write(STDOUT_FILENO, msg, length);

                    while (bytes = read(STDIN_FILENO, buf, sizeof(buf))) {
                        if (bytes < 0) {
                            const char msg[] = "error: failed to read from
stdin\n";

                            write(STDERR_FILENO, msg, sizeof(msg));
                            exit(EXIT_FAILURE);
                        }
                        else if (buf[0] == '\n') {
                            break;
                        }

                        write(pipe_client_to_child1[1], buf, bytes);

                        bytes = read(pipe_child2_to_client[0], buf, bytes);

                        char info_pid[32];
                        int32_t length_info_pid = snprintf(info_pid,
sizeof(info_pid), "PID %d: ", pid);

                        write(STDOUT_FILENO, info_pid, length_info_pid);
                        write(STDOUT_FILENO, buf, bytes);
                        write(STDOUT_FILENO, msg, length);

                    }

                    close(pipe_client_to_child1[1]);
                    close(pipe_child2_to_client[0]);

                    waitpid(child1, NULL, 0);
                    waitpid(child2, NULL, 0);
                } break;
            }
        } break;
    }
}
```

server_1.c

```
#include <stdint.h>
#include <ctype.h>

#include <stdlib.h>
```

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


int main(int argc, char * argv[]) {
    char buf[4096];
    ssize_t bytes;

    pid_t pid = getpid();

    int32_t file = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC | O_APPEND, 0600);
    if (file == -1) {
        const char msg[] = "error: failed to open requested file\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    while (bytes = read(STDIN_FILENO, buf, sizeof(buf))) {
        if (bytes < 0) {
            const char msg[] = "error: failed to read stdin\n";
            write(STDERR_FILENO, msg, sizeof(msg));
            exit(EXIT_FAILURE);
        }

        for (uint32_t i = 0; i < bytes; ++i) {
            buf[i] = toupper(buf[i]);
        }

        {
            char log_msg[4200];
            uint32_t log_length = snprintf(log_msg, sizeof(log_msg), "(Child 1)PID
%d: %s", pid, buf);

            // записываем в логи
            uint32_t written = write(file, log_msg, log_length);
            if (written != log_length) {
                const char msg[] = "error: failed to write in file\n";
                write(STDERR_FILENO, msg, sizeof(msg));
                exit(EXIT_FAILURE);
            }

            written = write(STDOUT_FILENO, buf, bytes);
            if (written != bytes) {
                const char msg[] = "error: failed to echo\n";
                write(STDERR_FILENO, msg, sizeof(msg));
                exit(EXIT_FAILURE);
            }
        }
    }

    if (bytes == 0) {
        const char term = '\0';
        write(file, &term, sizeof(term));
```

```
    }
    close(file);
}
```

server_2.c

```
#include <stdint.h>

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


int main(int argc, char * argv[]) {
    char buf[4096];
    ssize_t bytes;

    pid_t pid = getpid();

    int32_t file = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC | O_APPEND, 0600);
    if (file == -1) {
        const char msg[] = "error: failed to open requested file\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    while (bytes = read(STDIN_FILENO, buf, sizeof(buf))) {
        if (bytes < 0) {
            const char msg[] = "error: failed to read stdin\n";
            write(STDERR_FILENO, msg, sizeof(msg));
            exit(EXIT_FAILURE);
        }

        char answer[4096];
        int index = 0;
        for (uint32_t i = 0; i < bytes; ++i) {
            if (i != (bytes - 1) && buf[i] == buf[i + 1] && buf[i] == ' ') {
                ++i;
                continue;
            }
            answer[index] = buf[i];
            ++index;
        }

        {
            char log_msg[8096];
            uint32_t log_length = snprintf(log_msg, sizeof(log_msg), "(Child 2)PID
%d: %s", pid, answer);


            uint32_t written = write(file, log_msg, log_length);
            if (written != log_length) {
                const char msg[] = "error: failed to write in file\n";
```

```
                write(STDERR_FILENO, msg, sizeof(msg));
                exit(EXIT_FAILURE);
            }

            written = write(STDOUT_FILENO, answer, index);
            if (written != index) {
                const char msg[] = "error: failed to echo\n";
                write(STDERR_FILENO, msg, sizeof(msg));
                exit(EXIT_FAILURE);
            }
        }
    }

    if (bytes == 0) {
        const char term = '\0';
        write(file, &term, sizeof(term));
    }
    close(file);
}
```

Протокол работы программы

```
PID 34134: I`m a parent, my child 1 has PID: 34135, child 2 has PID: 34136.
           To exit, press CTRL+C
PID 34134: Write a message that needs to be changed: vova   qwerty asdfg   lwlw
PID 34134: VOVA QWERTY ASDFG LWLW
PID 34134: Write a message that needs to be changed: Лабораторная   работа   номер   1
PID 34134: Лабораторная работа номер 1
PID 34134: Write a message that needs to be changed: Тестирование лаборатной     работы
PID 34134: Тестирование лаборатной работы
PID 34134: Write a message that needs to be changed: ^C
```

Вывод

В ходе данной лабораторной работы я научился управлять процессами в ОС. Также обеспечил обмен данных между процессами посредством каналов.