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**

#include <unistd.h> *// системные вызовы*

#include <stdint.h> *// для uint32\_t, int32\_t*

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

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>

#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';

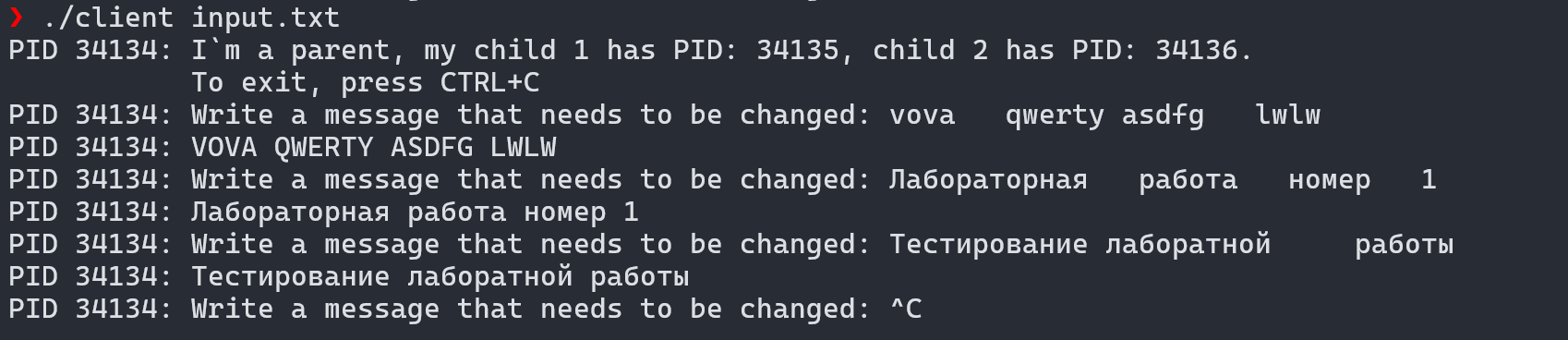
        write(file, &term, sizeof(term));

    }

    close(file);

}

**Протокол работы программы**

****

**Вывод**

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