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: Мицкевич А.А.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 15.10.25

Москва, 2025

**Постановка задачи**

**Вариант 12.**

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); – создаёт новый процесс.

 int pipe(int \*fd); – создаёт канал между двумя процессами. Возвращает массив fd.

fd[0] – файловый дескриптор для чтения

fd[1] – файловый дескриптор для записи

 ssize\_t readlink(char \*path, char \*buf, size\_t bufsz); – считывает содержимое символьной ссылки и записывает в buf.

 int write(int fd, const void \*buf, size\_t count); – записывает данные из буфера по указанному файловому дескриптору.

 int read(int fd, void \*buf, size\_t count); – читает данные из файла по файловому дескриптору и сохраняет в буфер.

 int execv(const char \*path, char \*const argv[]); – заменяет текущий процесс новым процессом, запуская указанную программу.

 pid\_t wait(int \*wstatus); – ожидает завершения одного из дочерних процессов, возвращает информацию о его завершении.

 pid\_t waitpid(pid\_t pid, int \*wstatus, int options); – ожидает завершения конкретного дочернего процесса.

Межпроцессное взаимодействие реализовано с использованием системных вызовов. Родительский процесс создаёт два дочерних: первый выполняет преобразование всех символов в верхний регистр, второй – удаляет служебные (лишние) пробелы. Обмен данными между процессами осуществляется через канал, созданный с помощью системного вызова pipe. Взаимодействие с пользователем происходит исключительно через родительский процесс.

**Код программы**

**client.c**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <unistd.h>

#include <sys/wait.h>

static char PROCESSOR\_1[] = "processor1";

static char PROCESSOR\_2[] = "processor2";

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

    if (argc == 1) {

        char message[512];

        int message\_length = snprintf(message, sizeof(message), "Usage: %s <filename>\n", argv[0]);

        write(STDERR\_FILENO, message, message\_length);

        exit(EXIT\_FAILURE);

    }

    char program\_path[1024];

    ssize\_t path\_length = readlink("/proc/self/exe", program\_path, sizeof(program\_path) - 1);

    if (path\_length == 0) {

        const char error[] = "Error: Failed to read program path\n";

        write(STDERR\_FILENO, error, sizeof(error));

        exit(EXIT\_FAILURE);

    }

    while (path\_length > 0 && program\_path[path\_length] != '\\' && program\_path[path\_length] != '/') {

        --path\_length;

    }

    program\_path[path\_length] = '\0';

    int pipe\_client\_to\_1[2];

    int pipe\_1\_to\_2[2];

    int pipe\_2\_to\_client[2];

    if (pipe(pipe\_client\_to\_1) == -1) {

        const char error[] = "Error: Failed to create pipe [client to processor 1]\n";

        write(STDERR\_FILENO, error, sizeof(error));

        exit(EXIT\_FAILURE);

    }

    if (pipe(pipe\_1\_to\_2) == -1) {

        const char error[] = "Error: Failed to create pipe [processor 1 to processor 2]\n";

        write(STDERR\_FILENO, error, sizeof(error));

        exit(EXIT\_FAILURE);

    }

    if (pipe(pipe\_2\_to\_client) == -1) {

        const char error[] = "Error: Failed to create pipe [processor 2 to client]\n";

        write(STDERR\_FILENO, error, sizeof(error));

        exit(EXIT\_FAILURE);

    }

    const pid\_t child1 = fork();

    switch (child1) {

        case -1: {

            const char error[] = "Error: Failed to spawn new process\n";

            write(STDERR\_FILENO, error, sizeof(error));

            exit(EXIT\_FAILURE);

        } break;

        case 0: {

            close(pipe\_2\_to\_client[0]);

            close(pipe\_2\_to\_client[1]);

            close(pipe\_client\_to\_1[1]);

            close(pipe\_1\_to\_2[0]);

            dup2(pipe\_client\_to\_1[0], STDIN\_FILENO);

            close(pipe\_client\_to\_1[0]);

            dup2(pipe\_1\_to\_2[1], STDOUT\_FILENO);

            close(pipe\_1\_to\_2[1]);

            char path[2048];

            snprintf(path, sizeof(path), "%s/%s", program\_path, PROCESSOR\_1);

            char \* const args[] = {PROCESSOR\_1, argv[1], NULL};

            int status = execv(path, args);

            if (status == -1) {

                const char error[] = "Error: Failed to execute new executable image\n";

                write(STDERR\_FILENO, error, sizeof(error));

                exit(EXIT\_FAILURE);

            }

        } break;

        default: {

            const pid\_t child2 = fork();

            switch (child2) {

                case -1: {

                    const char error[] = "Error: Failed to spawn new process\n";

                    write(STDERR\_FILENO, error, sizeof(error));

                    exit(EXIT\_FAILURE);

                } break;

                case 0: {

                    close(pipe\_client\_to\_1[0]);

                    close(pipe\_client\_to\_1[1]);

                    close(pipe\_1\_to\_2[1]);

                    close(pipe\_2\_to\_client[0]);

                    dup2(pipe\_1\_to\_2[0], STDIN\_FILENO);

                    close(pipe\_1\_to\_2[0]);

                    dup2(pipe\_2\_to\_client[1], STDOUT\_FILENO);

                    close(pipe\_2\_to\_client[1]);

                    char path[2048];

                    snprintf(path, sizeof(path), "%s/%s", program\_path, PROCESSOR\_2);

                    char \* const args[] = {PROCESSOR\_2, argv[1], NULL};

                    int status = execv(path, args);

                    if (status == -1) {

                        const char error[] = "Error: Failed to execute new executable image\n";

                        write(STDERR\_FILENO, error, sizeof(error));

                        exit(EXIT\_FAILURE);

                    }

                } break;

                default: {

                    pid\_t pid = getpid();

                    {

                        char message[256];

                        const int length = snprintf(

                            message, sizeof(message),

                            "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, message, length);

                    }

                    close(pipe\_client\_to\_1[0]);

                    close(pipe\_1\_to\_2[0]);

                    close(pipe\_1\_to\_2[1]);

                    close(pipe\_2\_to\_client[1]);

                    char buffer[4048];

                    ssize\_t bytes;

                    char message[1024];

                    const int length = snprintf(message, sizeof(message),

                        "PID %d: Write a message that needs to be changed: ", pid);

                    write(STDOUT\_FILENO, message, length);

                    while (bytes = read(STDIN\_FILENO, buffer, sizeof(buffer))) {

                        if (bytes < 0) {

                            const char message[] = "Error: Failed to read stdin\n";

                            write(STDERR\_FILENO, message, sizeof(message));

                            exit(EXIT\_FAILURE);

                        } else if (buffer[0] == '\n') break;

                        write(pipe\_client\_to\_1[1], buffer, bytes);

                        bytes = read(pipe\_2\_to\_client[0], buffer, bytes);

                        char info\_pid[32];

                        int length\_info\_pid = snprintf(info\_pid, sizeof(info\_pid), "PID %d: ", pid);

                        write(STDOUT\_FILENO, info\_pid, length\_info\_pid);

                        write(STDOUT\_FILENO, buffer, bytes);

                        write(STDOUT\_FILENO, message, length);

                    }

                    close(pipe\_client\_to\_1[1]);

                    close(pipe\_2\_to\_client[0]);

                    waitpid(child1, NULL, 0);

                    waitpid(child2, NULL, 0);

                } break;

            }

        } break;

    }

    return 0;

}

**processor1.c**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <unistd.h>

#include <fcntl.h>

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

    pid\_t pid = getpid();

    char buffer[4096];

    ssize\_t bytes;

    int logs = open(argv[1], O\_WRONLY | O\_CREAT | O\_APPEND, 0600);

    if (logs < 0) {

        const char error[] = "Error: Failed to open log file\n";

        write(STDERR\_FILENO, error, sizeof(error) - 1);

        exit(EXIT\_FAILURE);

    }

    while (bytes = read(STDIN\_FILENO, buffer, sizeof(buffer))) {

        if (bytes < 0) {

            const char error[] = "Error: Failed to read stdin\n";

            write(STDERR\_FILENO, error, sizeof(error) - 1);

            exit(EXIT\_FAILURE);

        }

        for (int i = 0; i < bytes; i++) {

            buffer[i] = toupper(buffer[i]);

        }

        char log\_buffer[4250];

        int log\_length = snprintf(log\_buffer, sizeof(log\_buffer), "(Child 1): My PID is %d. Processing: %s", pid, buffer);

        int bytes\_written = write(logs, log\_buffer, log\_length);

        if (bytes\_written != log\_length) {

            const char error[] = "Error: Failed to write to log file\n";

            write(STDERR\_FILENO, error, sizeof(error) - 1);

            exit(EXIT\_FAILURE);

        }

        bytes\_written = write(STDOUT\_FILENO, buffer, bytes);

        if (bytes\_written != bytes) {

            const char error[] = "Error: Failed to forward data to next process\n";

            write(STDERR\_FILENO, error, sizeof(error) - 1);

            exit(EXIT\_FAILURE);

        }

    }

    close(logs);

    return 0;

}

**processor2.c**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

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

    pid\_t pid = getpid();

    char buffer[4096];

    ssize\_t bytes;

    int logs = open(argv[1], O\_WRONLY | O\_CREAT | O\_APPEND, 0600);

    if (logs < 0) {

        const char error[] = "Error: Failed to open log file\n";

        write(STDERR\_FILENO, error, sizeof(error));

        exit(EXIT\_FAILURE);

    }

    while (bytes = read(STDIN\_FILENO, buffer, sizeof(buffer))) {

        if (bytes < 0) {

            const char error[] = "Error: Failed to read stdin\n";

            write(STDERR\_FILENO, error, sizeof(error));

            exit(EXIT\_FAILURE);

        }

        char result[4096];

        int end\_index = 0;

        for (int i = 0; i < bytes; i++) {

            if (!(end\_index != 0 && result[end\_index - 1] == ' ' && buffer[i] == ' ')) {

                result[end\_index] = buffer[i];

                end\_index++;

            }

        }

        char log\_buffer[4250];

        int log\_length = snprintf(log\_buffer, sizeof(log\_buffer), "(Child 2): My PID is %d. Processing: %s", pid, result);

        int bytes\_written = write(logs, log\_buffer, log\_length);

        if (bytes\_written != log\_length) {

            const char error[] = "Error: Failed to write to log file\n";

            write(STDERR\_FILENO, error, sizeof(error) - 1);

            exit(EXIT\_FAILURE);

        }

        bytes\_written = write(STDOUT\_FILENO, result, end\_index);

        if (bytes\_written != end\_index) {

            const char error[] = "Error: Failed to forward data to next process\n";

            write(STDERR\_FILENO, error, sizeof(error) - 1);

            exit(EXIT\_FAILURE);

        }

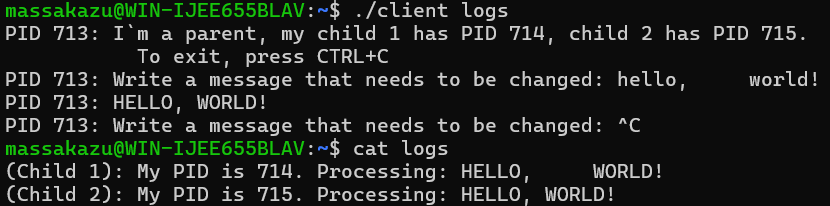
    }

    close(logs);

    return 0;

}

**Протокол работы программы**



**Вывод**

В ходе выполнения лабораторной работы было реализовано межпроцессное взаимодействие в Unix-подобной операционной системе с использованием системных вызовов fork, pipe, read и write. Родительский процесс успешно создаёт два дочерних, каждый из которых выполняет отдельную обработку данных: приведение символов к верхнему регистру и удаление лишних пробелов. Обмен информацией между процессами организован через каналы, что обеспечивает корректную передачу данных без прямого доступа к памяти друг друга.