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: Лабутин М. А.

Преподаватель: Бахарев В.Д.

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

Дата: 02.10.25

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

* int write(int fd, const void \*buf, size\_t count); – записывает данные из буфера по файловому дескриптору.
* int dup2(int oldfd, int newfd); – перенаправляет файловый дескриптор, позволяя процессу использовать канал (pipe) вместо стандартного ввода/вывода.
* int read(int fd, void\* buf, size\_t count); - читает данные из файла по файловому дескриптору и записывает в buf.
* int execl(const char \*path, const char \*arg, ...); - заменяет текущий процесс новым процессом, загружая и выполняя указанную программу;
* pid\_t waitpid(pid\_t pid, int \*wstatus, int options); - ожидает завершения конкретного дочернего процесса.
* int close(int fd); – закрывает файловый дескриптор.

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**

**parent.c**

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <string.h>

#define BUFFER\_SIZE 1024

#define EXIT\_SUCCESS 0

#define EXIT\_FAILURE 1

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

    if (argc != 1) {

        printf("Usage: %s\n", argv[0]);

        printf("No arguments needed. Program reads from stdin and writes to stdout.\n");

        return EXIT\_FAILURE;

    }

    int pipe\_parent\_to\_child1[2];

    int pipe\_child1\_to\_child2[2];

    int pipe\_child2\_to\_parent[2];

    // Создаем каналы

    if (pipe(pipe\_parent\_to\_child1) == -1 ||

        pipe(pipe\_child1\_to\_child2) == -1 ||

        pipe(pipe\_child2\_to\_parent) == -1) {

        perror("pipe");

        exit(EXIT\_FAILURE);

    }

    // Создаем Child1

    pid\_t child1 = fork();

    if (child1 == -1) {

        perror("fork");

        exit(EXIT\_FAILURE);

    }

    if (child1 == 0) {

        close(pipe\_parent\_to\_child1[1]);

        close(pipe\_child1\_to\_child2[0]);

        close(pipe\_child2\_to\_parent[0]);

        close(pipe\_child2\_to\_parent[1]);

        dup2(pipe\_parent\_to\_child1[0], STDIN\_FILENO);

        dup2(pipe\_child1\_to\_child2[1], STDOUT\_FILENO);

        close(pipe\_parent\_to\_child1[0]);

        close(pipe\_child1\_to\_child2[1]);

        execl("./child1", "child1", NULL);

        perror("execl child1");

        exit(EXIT\_FAILURE);

    }

    // Создаем Child2

    pid\_t child2 = fork();

    if (child2 == -1) {

        perror("fork");

        exit(EXIT\_FAILURE);

    }

    if (child2 == 0) {

        close(pipe\_parent\_to\_child1[0]);

        close(pipe\_parent\_to\_child1[1]);

        close(pipe\_child1\_to\_child2[1]);

        close(pipe\_child2\_to\_parent[0]);

        dup2(pipe\_child1\_to\_child2[0], STDIN\_FILENO);

        dup2(pipe\_child2\_to\_parent[1], STDOUT\_FILENO);

        close(pipe\_child1\_to\_child2[0]);

        close(pipe\_child2\_to\_parent[1]);

        execl("./child2", "child2", NULL);

        perror("execl child2");

        exit(EXIT\_FAILURE);

    }

    close(pipe\_parent\_to\_child1[0]);

    close(pipe\_child1\_to\_child2[0]);

    close(pipe\_child1\_to\_child2[1]);

    close(pipe\_child2\_to\_parent[1]);

    char buffer[BUFFER\_SIZE];

    ssize\_t bytes\_read;

    printf("The parent process is running. Enter the lines (Ctrl+D to end):\n");

    while (fgets(buffer, BUFFER\_SIZE, stdin) != NULL) {

        write(pipe\_parent\_to\_child1[1], buffer, strlen(buffer));

        bytes\_read = read(pipe\_child2\_to\_parent[0], buffer, BUFFER\_SIZE - 1);

        if (bytes\_read > 0) {

            buffer[bytes\_read] = '\0';

            printf("Result: %s\n", buffer);

        }

    }

    close(pipe\_parent\_to\_child1[1]);

    close(pipe\_child2\_to\_parent[0]);

    waitpid(child1, NULL, 0);

    waitpid(child2, NULL, 0);

    printf("The parent process is completed.\n");

    return EXIT\_SUCCESS;

}

**child1.c**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <ctype.h>

#include <string.h>

#define BUFFER\_SIZE 1024

#define EXIT\_SUCCESS 0

#define EXIT\_FAILURE 1

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

    if (argc != 1) {

        fprintf(stderr, "Usage: %s\n", argv[0]);

        fprintf(stderr, "This program reads from stdin and converts text to uppercase.\n");

        return EXIT\_FAILURE;

    }

    char buffer[BUFFER\_SIZE];

    ssize\_t bytes\_read;

    printf("Child1 started. PID: %d\n", getpid());

    while ((bytes\_read = read(STDIN\_FILENO, buffer, BUFFER\_SIZE - 1)) > 0) {

        buffer[bytes\_read] = '\0';

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

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

        }

        write(STDOUT\_FILENO, buffer, bytes\_read);

    }

    return EXIT\_SUCCESS;

}

**child2.c**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <ctype.h>

#define BUFFER\_SIZE 1024

#define EXIT\_SUCCESS 0

#define EXIT\_FAILURE 1

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

    if (argc != 1) {

        fprintf(stderr, "Usage: %s\n", argv[0]);

        fprintf(stderr, "This program reads from stdin and processes spaces.\n");

        return EXIT\_FAILURE;

    }

    char buffer[BUFFER\_SIZE];

    char result[BUFFER\_SIZE];

    ssize\_t bytes\_read;

    printf("Child2 started. PID: %d\n", getpid());

    while ((bytes\_read = read(STDIN\_FILENO, buffer, BUFFER\_SIZE - 1)) > 0) {

        buffer[bytes\_read] = '\0';

        char \*src = buffer;

        char \*dst = result;

        int space\_count = 0;

        int in\_space\_sequence = 0;

        while (\*src) {

            if (isspace((unsigned char)\*src)) {

                if (!in\_space\_sequence) {

                    in\_space\_sequence = 1;

                    space\_count = 1;

                } else {

                    space\_count++;

                }

            } else {

                if (in\_space\_sequence) {

                    if (space\_count % 2 != 0) {

                        \*dst++ = ' ';

                    }

                    in\_space\_sequence = 0;

                    space\_count = 0;

                }

                \*dst++ = \*src;

            }

            src++;

        }

        if (in\_space\_sequence && space\_count % 2 != 0) {

            \*dst++ = ' ';

        }

        \*dst = '\0';

        // Отправляем результат родителю

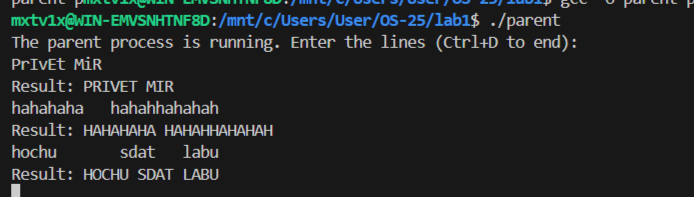
        write(STDOUT\_FILENO, result, strlen(result));

    }

    return EXIT\_SUCCESS;

}

**Протокол работы программы**



**Вывод**

В ходе выполнения данной лабораторной работы было изучено взаимодействие процессов через каналы (pipe) и механизмы их работы при создании дочерних процессов с помощью fork(). Я научился управлять процессами в ОС. Также обеспечил обмен данных между процессами посредством каналов.