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

```
massakazu@WIN-IJEE655BLAV:~$ ./client logs
PID 713: I`m a parent, my child 1 has PID 714, child 2 has PID 715.
         To exit, press CTRL+C
PID 713: Write a message that needs to be changed: hello,     world!
PID 713: HELLO, WORLD!
PID 713: Write a message that needs to be changed: ^C
massakazu@WIN-IJEE655BLAV:~$ cat logs
(Child 1): My PID is 714. Processing: HELLO,     WORLD!
(Child 2): My PID is 715. Processing: HELLO, WORLD!
```

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.