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: Бахшалиев М.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 02.10.25

Москва, 2025

Постановка задачи

Вариант 19.

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (pipe). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы. Правило фильтрации: с вероятностью 80% строки отправляются в pipe1, иначе в pipe2. Дочерние процессы удаляют все гласные из строк.

Общий метод и алгоритм решения

В данной лабораторной работе реализована клиент-серверная архитектура с использованием межпроцессного взаимодействия через каналы (pipes). Клиентская программа (client.c) создает два дочерних процесса, каждый из которых запускает серверную программу (server.c). Клиент читает строки из стандартного ввода и распределяет их между серверами с вероятностным распределением (80% и 20%). Каждый сервер обрабатывает полученные данные (удаляет гласные буквы) и записывает результат в указанный файл.

Использованные системные вызовы:

1. pid_t fork(void) – создает дочерний процесс. Используется для создания двух процессов-серверов.
2. int pipe(int fd[2]) – создает однонаправленный канал для обмена данными между процессами. Клиент создает два канала для взаимодействия с серверами.
3. ssize_t read(int fd, void *buf, size_t count) – читает данные из файлового дескриптора. Используется для чтения из стандартного ввода и каналов.
4. ssize_t write(int fd, const void *buf, size_t count) – записывает данные в файловый дескриптор. Применяется для отправки данных в каналы и записи в файлы.
5. int dup2(int oldfd, int newfd) – переназначает файловый дескриптор. Используется для перенаправления стандартного ввода дочерних процессов на чтение из каналов.
6. int execv(const char *path, char *const argv[]) – заменяет текущий образ процесса новым. Запускает серверную программу в дочерних процессах.
7. pid_t waitpid(pid_t pid, int *status, int options) – ожидает завершения дочернего процесса. Гарантирует корректное завершение работы серверов.
8. int open(const char *pathname, int flags, mode_t mode) – открывает файл. Сервер использует его для создания выходного файла.
9. int close(int fd) – закрывает файловый дескриптор. Освобождает ресурсы каналов и файлов.
10. ssize_t readlink(const char *path, char *buf, size_t bufsiz) – читает значение символической ссылки. Используется для определения пути к директории с программой.

Алгоритм работы:

1. Клиент:
    - Проверяет аргументы командной строки.
    - Определяет путь к директории с программой через /proc/self/exe.
    - Создает два канала (pipe1, pipe2).
    - Запускает два дочерних процесса через fork().
    - Каждый дочерний процесс:
        - Перенаправляет стандартный ввод на чтение из соответствующего канала.
        - Запускает сервер через execv() с указанием выходного файла.
    - Родительский процесс:
        - Читает строки из стандартного ввода.
        - Распределяет строки между каналами с вероятностью 80%/20%.
        - Завершает ввод при получении пустой строки.
        - Закрывает каналы и ожидает завершения серверов через waitpid().
2. Сервер:
    - Открывает указанный файл для записи.
    - Читает данные из стандартного ввода (перенаправленного из канала).
    - Удаляет все гласные буквы (английские и русские) из полученных данных.
    - Записывает модифицированные данные в выходной файл.
    - Завершает работу после закрытия канала.

Особенности реализации:

- Вероятностное распределение нагрузки между серверами (80%/20%).
- Поддержка многобайтовых символов (русские буквы).
- Корректное управление ресурсами (закрытие неиспользуемых дескрипторов).
- Обработка ошибок на всех критических этапах.

Код программы

client.c:

```
#include <stdint.h>
#include <stdbool.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <string.h>

static char SERVER_PROGRAM_NAME[] = "server";

int main(int argc, char **argv) {
    if (argc != 3) {
        char msg[1024];
        uint32_t len = snprintf(msg, sizeof(msg) - 1, "usage: %s filename1
filename2\n", argv[0]);
        write(STDERR_FILENO, msg, len);
        exit(EXIT_SUCCESS);
    }

    srand(time(NULL));

    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 pipe1[2];
    if (pipe(pipe1) == -1) {
        const char msg[] = "error: failed to create pipe1\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }
```

```
int pipe2[2];
if (pipe(pipe2) == -1) {
    const char msg[] = "error: failed to create pipe2\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    exit(EXIT_FAILURE);
}

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

case 0: {
    close(pipe1[1]);
    dup2(pipe1[0], STDIN_FILENO);
    close(pipe1[0]);

    {
        char path[1024];
        snprintf(path, sizeof(path) - 1, "%s/%s", progpath, SERVER_PROGRAM_NAME);
        char *const args[] = {SERVER_PROGRAM_NAME, argv[1], NULL};
        int32_t status = execv(path, args);

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

default: {
    const pid_t child2 = fork();
    switch (child2) {
    case -1: {
        const char msg[] = "error: failed to spawn second child process\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    } break;
```

```
        case 0: {
            close(pipe2[1]);
            dup2(pipe2[0], STDIN_FILENO);
            close(pipe2[0]);

            {
                char path[1024];
                snprintf(path, sizeof(path) - 1, "%s/%s", progpath,
SERVER_PROGRAM_NAME);
                char *const args[] = {SERVER_PROGRAM_NAME, argv[2], NULL};
                int32_t status = execv(path, args);

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

        default: {
            close(pipe1[0]);
            close(pipe2[0]);

            char buf[4096];
            ssize_t bytes;

            printf("Input string:\n");

            while ((bytes = read(STDIN_FILENO, buf, sizeof(buf))) > 0) {
                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;
                }

                int r = rand() % 100;
                if (r < 80) {
                    write(pipe1[1], buf, bytes);
                } else {
                    write(pipe2[1], buf, bytes);
                }
```

```
            }

            close(pipe1[1]);
            close(pipe2[1]);

            waitpid(child1, NULL, 0);
            waitpid(child2, NULL, 0);

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

    return 0;
}
```

server.c:

```
#include <stdint.h>
#include <stdbool.h>
#include <ctype.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

int is_vowel(char c) {
    c = tolower(c);
    return (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' || c == 'y');
}

void remove_vowels(char *str, ssize_t length) {
    char *src = str;
    char *dst = str;

    for (ssize_t i = 0; i < length; i++) {
        if (!is_vowel(*src)) {
            *dst++ = *src;
        }
        src++;
    }
}
```

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

        remove_vowels(buf, bytes);

        int32_t written = write(file, buf, bytes);
        if (written != bytes) {
            const char msg[] = "error: failed to write to file\n";
            write(STDERR_FILENO, msg, sizeof(msg));
            exit(EXIT_FAILURE);
        }
    }

    close(file);
    return 0;
}
```

Протокол работы программы

Тестирование:

```
$ ./client file1.txt file2.txt
Input string:
Hello World
This is a test
```

```
Another line

And one more

Last line
```

Вывод

В ходе лабораторной работы были успешно освоены механизмы межпроцессного взаимодействия в операционных системах семейства Linux. Реализована клиент-серверная архитектура с использованием каналов (pipes) для передачи данных между процессами. Практически применены ключевые системные вызовы для управления процессами и межпроцессной коммуникации.

Проблемы и сложности: корректное управление файловыми дескрипторами при работе с несколькими каналами; обеспечение правильного завершения процессов и избежание "зомби-процессов"; обработка ошибок при системных вызовах для повышения надежности программы.

Пожелания: расширить функциональность для обработки многобайтовых кодировок (UTF-8); добавить возможность динамического изменения вероятностного распределения; реализовать логирование для отладки распределения данных между процессами.