

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-216БВ-24

Студент: Лукьянчук А.О.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 23.12.25

Москва, 2025

Постановка задачи

Вариант 16

Родительский процесс создает дочерний процесс. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись. Родительский процесс принимает от пользователя строки произвольной длины и пересыпает их в pipe1. Процесс child проверяет строки на валидность правилу: строка должна оканчиваться на «.» или «;». Если строка соответствует правилу, то она выводится в файл. Если строка не соответствует правилу, то в pipe2 выводится информация об ошибке. Родительский процесс полученные от child ошибки выводит в стандартный поток вывода.

Общий метод и алгоритм решения

Использованные системные вызовы:

- pid_t fork(void); – создает дочерний процесс.
- int pipe(int *fd); – создает односторонний канал для межпроцессного взаимодействия.
- int execl(const char *path, const char *arg, ...); – заменяет образ текущей программы на указанную.
- int dup2(int oldfd, int newfd); – создает копию файлового дескриптора.
- int open(const char *pathname, int flags, mode_t mode); – открывает файл по указанному пути.
- ssize_t read(int fd, void *buf, size_t count); – читает данные из файлового дескриптора.
- ssize_t write(int fd, const void *buf, size_t count); – записывает данные в файловый дескриптор.
- int close(int fd); – закрывает файловый дескриптор.
- pid_t wait(int *status); – ожидает изменения состояния дочернего процесса.

Алгоритм:

1. Родительский процесс создает два канала (pipe1 и pipe2).
2. Создается дочерний процесс с помощью fork().
3. Дочерний процесс перенаправляет свой стандартный ввод на чтение из pipe1, а стандартный вывод — на запись в pipe2.

4. Дочерний процесс запускает программу child.c, которая читает строки из стандартного ввода.
5. Каждая строка проверяется на соответствие правилу (оканчивается на '.' или ';').
6. Если строка соответствует правилу, она записывается в файл. Если нет — в pipe2 отправляется сообщение об ошибке.
7. Родительский процесс читает сообщения об ошибках из pipe2 и выводит их пользователю.

Код программы

```
[main.c]
#include "./include/parent.h"
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
    if (argc < 2) {
        char filename[256];
        const char prompt[] = "Enter filename: ";
        write(STDOUT_FILENO, prompt, sizeof(prompt) - 1);

        ssize_t bytes_read = read(STDIN_FILENO, filename, sizeof(filename) - 1);
        if (bytes_read <= 0)
            const char msg[] = "error: failed to read filename\n";
            write(STDERR_FILENO, msg, sizeof(msg) - 1);
            exit(EXIT_FAILURE);
    }

    if (bytes_read > 0 && filename[bytes_read - 1] == '\n') {
        filename[bytes_read - 1] = '\0';
    } else {
        filename[bytes_read] = '\0';
    }

    run_parent_process(filename);
} else {
    run_parent_process(argv[1]);
}
return 0;
}

[src/parent.c]
```

```
#include "../include/parent.h"
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <fcntl.h>

void run_parent_process(const char* filename) {
    int pipe1[2], pipe2[2];

    if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
        const char msg[] = "error: failed to create pipes\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }

    pid_t pid = fork();
    if (pid == -1) {
        const char msg[] = "error: failed to fork\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        close(pipe1[1]);
        close(pipe2[0]);

        dup2(pipe1[0], STDIN_FILENO);
        close(pipe1[0]);

        dup2(pipe2[1], STDOUT_FILENO);
        close(pipe2[1]);

        execl("./bin/child", "child", filename, NULL);

        const char msg[] = "error: failed to exec child\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    } else {
        close(pipe1[0]);
        close(pipe2[1]);

        char buffer[4096];
        ssize_t bytes_read;

        while ((bytes_read = read(STDIN_FILENO, buffer, sizeof(buffer))) > 0) {
            ssize_t written = write(pipe1[1], buffer, bytes_read);
            if (written == -1) {
                const char msg[] = "error: failed to write to pipe\n";
                write(STDERR_FILENO, msg, sizeof(msg) - 1);
                exit(EXIT_FAILURE);
            }
        }
    }
}
```

```

        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        break;
    }

    char response[4096];
    ssize_t nread = read(pipe2[0], response, sizeof(response) - 1);
    if (nread > 0) {
        response[nread] = '\0';

        if (strstr(response, "Error:") == response) {
            write(STDOUT_FILENO, response, nread);
        }
    }
}

close(pipe1[1]);
close(pipe2[0]);
wait(NULL);
}

}

[src/child.c]
#include "../include/child.h"
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

int check_rule(const char* str) {
    size_t len = strlen(str);
    if (len == 0) return 0;

    for (size_t i = len - 1; i > 0; --i) {
        if (str[i] != '\n' && str[i] != '\r' && str[i] != ' ' && str[i] != '\t') {
            return (str[i] == '.' || str[i] == ';');
        }
    }

    return (str[0] == '.' || str[0] == ';');
}

void run_child_process(const char* filename) {
    int file = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file == -1) {
        const char error_msg[] = "Error: failed to open file\n";
        write(STDOUT_FILENO, error_msg, sizeof(error_msg) - 1);
        exit(EXIT_FAILURE);
    }
}

```

```

char buffer[4096];
ssize_t bytes_read;

while ((bytes_read = read(STDIN_FILENO, buffer, sizeof(buffer) - 1)) > 0) {
    buffer[bytes_read] = '\0';

    char* line = buffer;
    char* next_line;

    while ((next_line = strchr(line, '\n')) != NULL) {
        *next_line = '\0';

        if (check_rule(line)) {

            size_t line_len = strlen(line);
            write(file, line, line_len);
            write(file, "\n", 1);
        } else {
            const char error_msg[] = "Error: String does not end with '.' or
';'\n";
            write(STDOUT_FILENO, error_msg, sizeof(error_msg) - 1);
        }

        line = next_line + 1;
    }

    if (strlen(line) > 0) {
        if (check_rule(line)) {
            size_t line_len = strlen(line);
            write(file, line, line_len);
            write(file, "\n", 1);
            write(STDOUT_FILENO, line, line_len);
            write(STDOUT_FILENO, "\n", 1);
        } else {
            const char error_msg[] = "Error: String does not end with '.' or
';'\n";
            write(STDOUT_FILENO, error_msg, sizeof(error_msg) - 1);
        }
    }
}

close(file);
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        const char error_msg[] = "Error: missing filename argument\n";
        write(STDOUT_FILENO, error_msg, sizeof(error_msg) - 1);
        return EXIT_FAILURE;
    }
}

```

```

    run_child_process(argv[1]);
    return 0;
}

[include/child.h]
#ifndef CHILD_H
#define CHILD_H

void run_child_process(const char* filename);

#endif
[include/parent.h]
#ifndef PARENT_H
#define PARENT_H

void run_parent_process(const char* filename);

#endif

[Makefile]
CC = gcc
CFLAGS = -Wall -Wextra -I./include
SRC_DIR = src
BIN_DIR = bin

.PHONY: all clean directories

all: directories $(BIN_DIR)/main

directories:
    mkdir -p $(BIN_DIR)

$(BIN_DIR)/main: main.c $(SRC_DIR)/parent.c $(SRC_DIR)/child.c
    $(CC) $(CFLAGS) -o $(BIN_DIR)/child $(SRC_DIR)/child.c
    $(CC) $(CFLAGS) -o $@ main.c $(SRC_DIR)/parent.c

clean:
    rm -rf $(BIN_DIR)

```

Протокол работы программы

Тестирование:

\$ make

```
gcc -Wall -Wextra -I./include -o bin/child src/child.c
gcc -Wall -Wextra -I./include -o bin/main main.c src/parent.c
$ ./bin/main
Enter filename: output.txt
test input 1 .
test input 2 ;
another string without punctuation
Error: String does not end with '.' or ';'
valid string.;

^D
$ cat output.txt
test input 1 .
test input 2 ;
valid string.;
```

Вывод

В ходе выполнения лабораторной работы были изучены основные механизмы межпроцессного взаимодействия в Linux с использованием неименованных каналов (pipe). Была реализована программа, в которой родительский и дочерний процессы обмениваются данными через два канала: один для передачи строк, другой — для сообщений об ошибках.

Основные сложности возникли с корректным закрытием файловых дескрипторов после перенаправления стандартных потоков, что могло приводить к зависанию программы. Также требовалось аккуратно обрабатывать строки, учитывая символы перевода строки при проверке условия. В целом, работа позволила получить практический опыт работы с системными вызовами для создания процессов и организации взаимодействия между ними.