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-214Б-23

Студент: Шитов Н.В.

Преподаватель: Бахарев В.Д.

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

Дата: 22.10.24

Москва, 2025

**Постановка задачи**

**Вариант 4.**

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.

Родительский процесс создает дочерний процесс. Первой строчкой пользователь в консоль родительского процесса пишет имя файла, которое будет передано при создании дочернего процесса. Родительский и дочерний процесс должны быть представлены разными программами. Родительский процесс передает команды пользователя через pipe1, который связан с стандартным входным потоком дочернего процесса. Дочерний процесс принеобходимости передает данные в родительский процесс через pipe2. Результаты своей работы дочерний процесс пишет в созданный им файл. Допускается просто открыть файл и писать туда, не перенаправляя стандартный поток вывода.

4 вариант) Пользователь вводит команды вида: «число число число». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс производит деление первого числа, на последующие, а результат выводит в файл. Если происходит деление на 0, то тогда дочерний и родительский процесс завершают свою работу. Проверка деления на 0 должна осуществляться на стороне дочернего процесса. Числа имеют тип float. Количество чисел может быть произвольным.

**Общий метод и алгоритм решения**

Использованные системные вызовы:

* **pid\_t fork(void);** – создает дочерний процесс, который является точной копией родительского процесса. После вызова fork() оба процесса продолжают выполнение с точки, следующей за вызовом fork.
* int pipe(int fd); – создает однонаправленный канал для межпроцессного взаимодействия. Массив fd из двух элементов заполняется файловыми дескрипторами: fd[0] для чтения из канала, fd[1] для записи в канал.
* ssize\_t read(int fd, void buf, size\_t count); – читает данные из файлового дескриптора fd в буфер buf размером до count байт.
* ssize\_t write(int fd, const void buf, size\_t count); – записывает данные из буфера buf размером count байт в файловый дескриптор fd.
* int dup2(int oldfd, int newfd); – создает копию файлового дескриптора oldfd в указанном файловом дескрипторе newfd. Если newfd уже был открыт, он предварительно закрывается.
* pid\_t wait(int status); – приостанавливает выполнение родительского процесса до тех пор, пока один из его дочерних процессов не завершится.
* int execv(const char path, char const argv[]); – заменяет текущий образ процесса новым образом, загружаемым из исполняемого файла path с аргументами argv.
* int close(int fd); – закрывает файловый дескриптор, делая его доступным для повторного использования.

**Алгоритм работы программы:**

Программа состоит из двух основных компонентов: клиента (main.c) и сервера (server.c), которые взаимодействуют через pipe-каналы.

1. **Инициализация и проверка аргументов:**
   * Клиент проверяет наличие аргументов командной строки. При отсутствии аргументов выводит сообщение об использовании и завершает работу.
2. **Определение пути к программе:**
   * Клиент определяет полный путь к директории, где находится исполняемый файл, используя readlink("/proc/self/exe").
3. **Создание каналов связи:**
   * Создаются два pipe-канала: client\_to\_server для передачи данных от клиента к серверу и server\_to\_client для передачи результатов обратно.
4. **Создание дочернего процесса:**
   * С помощью fork() создается дочерний процесс, который будет выполнять роль сервера.
5. **Логика дочернего процесса (сервер):**
   * Дочерний процесс закрывает ненужные концы каналов и перенаправляет стандартные потоки ввода/вывода с помощью dup2():
     + STDIN\_FILENO перенаправляется на чтение из client\_to\_server
     + STDOUT\_FILENO перенаправляется на запись в server\_to\_client
   * Сервер запускает программу "lab1" (server.c) с помощью execv(), передавая ей имя файла для записи результатов.
6. **Логика родительского процесса (клиент):**
   * Родительский процесс закрывает ненужные концы каналов и входит в цикл обработки данных:
     + Читает данные со стандартного ввода (STDIN\_FILENO)
     + Передает прочитанные данные серверу через канал client\_to\_server
     + Читает ответ сервера из канала server\_to\_client и выводит его на стандартный вывод
   * Цикл прерывается при вводе пустой строки (символ новой строки)
7. **Обработка данных на сервере:**
   * Сервер открывает/создает файл для записи результатов
   * Читает данные из STDIN\_FILENO (который теперь связан с каналом client\_to\_server)
   * Обрабатывает данные с помощью функции bober(), которая:
     + Разбирает строку на числа с поддержкой отрицательных значений и десятичных дробей
     + Выполняет последовательное деление всех чисел (первое число делится на второе, результат делится на третье и т.д.)
     + Обрабатывает ошибку деления на ноль
   * Записывает результат в файл и отправляет сообщение "success" клиенту
8. **Завершение работы:**
   * После получения пустой строки клиент закрывает каналы и ожидает завершения дочернего процесса с помощью wait()
   * Сервер закрывает файл и завершает работу

**Код программы**

**main.c**

#include <stdint.h>

#include <stdbool.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

#include <ctype.h>

static char SERVER\_PROGRAM\_NAME[] = "lab1";

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

    if (argc == 1) {

        char msg[1024];

        uint32\_t len = snprintf(msg, sizeof(msg) - 1, "usage: %s filename\n", argv[0]);

        write(STDERR\_FILENO, msg, len);

        exit(EXIT\_SUCCESS);

    }

    // NOTE: Get full path to the directory, where program resides

    char progpath[1024];

    {

        // NOTE: Read full program path, including its name

        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);

        }

        // NOTE: Trim the path to first slash from the end

        while (progpath[len] != '/')

            --len;

        progpath[len] = '\0';

    }

    // NOTE: Open pipes

    int client\_to\_server[2]; // AB

    if (pipe(client\_to\_server) == -1) {

        const char msg[] = "error: failed to create pipe\n";

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

        exit(EXIT\_FAILURE);

    }

    int server\_to\_client[2]; // BA

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

        const char msg[] = "error: failed to create pipe\n";

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

        exit(EXIT\_FAILURE);

    }

    // NOTE: Spawn a new process

    const pid\_t child = fork();

    switch (child) {

    case -1: { // NOTE: Kernel fails to create another process

        const char msg[] = "error: failed to spawn new process\n";

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

        exit(EXIT\_FAILURE);

    } break;

    case 0: { // NOTE: We're a child, child doesn't know its pid after fork

        {

            pid\_t pid = getpid(); // NOTE: Get child PID

            char msg[64];

            const int32\_t length = snprintf(msg, sizeof(msg),

                "%d: I'm a child\n", pid);

            write(STDOUT\_FILENO, msg, length);

        }

        close(client\_to\_server[1]);

        close(server\_to\_client[0]);

        dup2(client\_to\_server[0], STDIN\_FILENO);

        close(client\_to\_server[0]);

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

        close(server\_to\_client[1]);

        {

            char path[1024];

            snprintf(path, sizeof(path) - 1, "%s/%s", progpath, SERVER\_PROGRAM\_NAME);

            // NOTE: args[0] must be a program name, next the actual arguments

            // NOTE: `NULL` at the end is mandatory, because `exec\*`

            //       expects a NULL-terminated list of C-strings

            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 new exectuable image\n";

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

                exit(EXIT\_FAILURE);

            }

        }

    } break;

    default: { // NOTE: We're a parent, parent knows PID of child after fork

        {

            pid\_t pid = getpid(); // NOTE: Get parent PID

            char msg[64];

            const int32\_t length = snprintf(msg, sizeof(msg),

                "%d: I'm a parent, my child has PID %d\n", pid, child);

            write(STDOUT\_FILENO, msg, length);

        }

        close(client\_to\_server[0]);

        close(server\_to\_client[1]);

        char buf[4096];

        ssize\_t bytes;

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

            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') {

                // NOTE: When Enter is pressed with no input, then exit client

                break;

            }

            write(client\_to\_server[1], buf, bytes);

            bytes = read(server\_to\_client[0], buf, sizeof(buf) - 1);

            if (bytes > 0) {

                buf[bytes] = '\0';  // Добавляем нулевой терминатор

                write(STDOUT\_FILENO, buf, bytes);

                }

        }

        close(client\_to\_server[1]);

        close(server\_to\_client[0]);

        wait(NULL);

    } break;

    }

}

**server.c**

#include <stdint.h>

#include <stdbool.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdio.h>

float bober(char buf[]) {

    float numbers[100];

    int count = 0;

    int i = 0;

    while (buf[i] != '\0' && buf[i] != '\n') {

        while (buf[i] == ' ') i++;

        if (buf[i] == '\0' || buf[i] == '\n') break;

        float num = 0;

        int is\_negative = 0;

        int has\_decimal = 0;

        float decimal\_place = 0.1f;

        if (buf[i] == '-') {

            is\_negative = 1;

            i++;

        }

        while ((buf[i] >= '0' && buf[i] <= '9') || buf[i] == '.') {

            if (buf[i] == '.') {

                has\_decimal = 1;

            } else {

                if (!has\_decimal) {

                    num = num \* 10 + (buf[i] - '0');

                } else {

                    num += (buf[i] - '0') \* decimal\_place;

                    decimal\_place \*= 0.1f;

                }

            }

            i++;

        }

        if (is\_negative) {

            num = -num;

        }

        numbers[count++] = num;

        while (buf[i] == ' ') i++;

    }

    if (count == 0) {

        return 0;

    }

    if (count == 1) {

        return numbers[0];

    }

    float result = numbers[0];

    for (int j = 1; j < count; j++) {

        if (numbers[j] == 0) {

            const char msg[] = "error: division by zero\n";

            write(STDOUT\_FILENO, msg, sizeof(msg));

            exit(EXIT\_FAILURE);

        }

        result /= numbers[j];

    }

    return result;

}

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

    char buf[4096];

    ssize\_t bytes;

    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(STDOUT\_FILENO, msg, sizeof(msg));

        exit(EXIT\_FAILURE);

    }

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

        if (bytes < 0) {

            const char msg[] = "error: failed to read from stdin\n";

            write(STDOUT\_FILENO, msg, sizeof(msg));

            exit(EXIT\_FAILURE);

        }

        float result = bober(buf);

        bytes = snprintf(buf, sizeof(buf), "%.4f\n", result);

        {

            int32\_t written = write(file, buf, bytes);

            if (written != bytes) {

                const char msg[] = "error: failed to write to file\n";

                write(STDOUT\_FILENO, msg, sizeof(msg));

                exit(EXIT\_FAILURE);

            }

        }

        const char msg[] = "success\n";

        write(STDOUT\_FILENO, msg, sizeof(msg));

    }

    if (bytes == 0) {

        const char term = '\0';

        write(file, &term, sizeof(term));

    }

    close(file);

}

**Протокол работы программы**

**Тестирование:**

┌──(flow㉿kali)-[~/Documents/vsc/oc/lab1]

└─$ strace -o trace.log ./client output.txt

25 5

success

1.1 1.2 1.3

success

1234 2

success

10000 2 2 2 2 2 2

success

0 1 2 3

success

11 0

error: division by zero

success

123 1

success

**Strace:**

execve("./client", ["./client", "output.txt"], 0x7ffd711d2c68 /\* 55 vars \*/) = 0

brk(NULL) = 0x55c661b1c000

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f00e4069000

access("/etc/ld.so.preload", R\_OK) = -1 ENOENT (Нет такого файла или каталога)

openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3

fstat(3, {st\_mode=S\_IFREG|0644, st\_size=94067, ...}) = 0

mmap(NULL, 94067, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7f00e4052000

close(3) = 0

openat(AT\_FDCWD, "/lib/x86\_64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\236\2\0\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 840, 64) = 840

fstat(3, {st\_mode=S\_IFREG|0755, st\_size=2003408, ...}) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 840, 64) = 840

mmap(NULL, 2055800, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f00e3e5c000

mmap(0x7f00e3e84000, 1462272, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x28000) = 0x7f00e3e84000

mmap(0x7f00e3fe9000, 352256, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x18d000) = 0x7f00e3fe9000

mmap(0x7f00e403f000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1e2000) = 0x7f00e403f000

mmap(0x7f00e4045000, 52856, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f00e4045000

close(3) = 0

mmap(NULL, 12288, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f00e3e59000

arch\_prctl(ARCH\_SET\_FS, 0x7f00e3e59740) = 0

set\_tid\_address(0x7f00e3e59a10) = 119646

set\_robust\_list(0x7f00e3e59a20, 24) = 0

rseq(0x7f00e3e59680, 0x20, 0, 0x53053053) = 0

mprotect(0x7f00e403f000, 16384, PROT\_READ) = 0

mprotect(0x55c630a4d000, 4096, PROT\_READ) = 0

mprotect(0x7f00e40a5000, 8192, PROT\_READ) = 0

prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0

munmap(0x7f00e4052000, 94067) = 0

readlink("/proc/self/exe", "/home/flow/Documents/vsc/oc/lab1"..., 1023) = 39

**pipe2([3, 4], 0) = 0**

**pipe2([5, 6], 0) = 0**

rt\_sigprocmask(SIG\_BLOCK, ~[], [], 8) = 0

**clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7f00e3e59a10) = 119647**

rt\_sigprocmask(SIG\_SETMASK, [], NULL, 8) = 0

**close(3) = 0**

**close(6) = 0**

**read(0, "25 5\n", 4096) = 5**

**write(4, "25 5\n", 5) = 5**

**read(5, "success\n", 4095) = 8**

**write(1, "success\n", 8) = 8**

**read(0, "1.1 1.2 1.3\n", 4096) = 12**

**write(4, "1.1 1.2 1.3\n", 12) = 12**

**read(5, "success\n", 4095) = 8**

**write(1, "success\n", 8) = 8**

**read(0, "1234 2\n", 4096) = 7**

**write(4, "1234 2\n", 7) = 7**

**read(5, "success\n", 4095) = 8**

**write(1, "success\n", 8) = 8**

**read(0, "10000 2 2 2 2 2 2 \n", 4096) = 19**

**write(4, "10000 2 2 2 2 2 2 \n", 19) = 19**

**read(5, "success\n", 4095) = 8**

**write(1, "success\n", 8) = 8**

**read(0, "0 1 2 3\n", 4096) = 8**

**write(4, "0 1 2 3\n", 8) = 8**

**read(5, "success\n", 4095) = 8**

**write(1, "success\n", 8) = 8**

**read(0, "11 0\n", 4096) = 5**

**write(4, "11 0\n", 5) = 5**

**read(5, "error: division by zero\nsuccess\n", 4095) = 32**

**write(1, "error: division by zero\nsuccess\n", 32) = 32**

**read(0, "123 1\n", 4096) = 6**

**write(4, "123 1\n", 6) = 6**

**read(5, "success\n", 4095) = 8**

**write(1, "success\n", 8) = 8**

**read(0, "\n", 4096) = 1**

**close(4) = 0**

**close(5) = 0**

**wait4(-1, NULL, 0, NULL) = 119647**

--- SIGCHLD {si\_signo=SIGCHLD, si\_code=CLD\_EXITED, si\_pid=119647, si\_uid=1000, si\_status=0, si\_utime=0, si\_stime=0} ---

exit\_group(0) = ?

+++ exited with 0 +++

**Вывод**

Данная программа реализует межпроцессное взаимодействие через pipe-каналы между клиентом и сервером. Сервер выполняет арифметическую операцию последовательного деления чисел, полученных от клиента. Программа демонстрирует практическое применение системных вызовов для организации параллельных вычислений.