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-214БВ-24

Студент: Александров М.С.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 02.10.25

Москва, 2025

Постановка задачи

Вариант 2.

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 при необходимости 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.

Пользователь вводит команды вида: «число число число<endline>». Далее эти числа передаются от родительского процесса в дочерний. Дочерний процесс считает их сумму и выводит её в файл. Числа имеют тип float. Количество чисел может быть произвольным.

Общий метод и алгоритм решения

Использованные системные вызовы:

- *pid_t fork(void)*; – создает дочерний процесс.
- *int pipe(int *fd)*; – создает однонаправленный канал для межпроцессорного взаимодействия;
- *int dup2(int oldfd, int newfd)*; – создает копию файлового дескриптора *oldfd* в указанном дескрипторе *newfd*.
- *int execv(const char *path, char *const argv[])*; — заменяет текущий образ процесса на новый исполняемый файл.
- *int open(const char* pathname, int flags, mode_t mode)*; – открывает файл по указанному пути с заданными флагами и правами доступа.
- *ssize_t read(int fd, void* buf, size_t count)*; – читает данные из файлового дескриптора в буфер.
- *ssize_t write(int fd, const void* buf, size_t count)*; – записывает данные из буфера в файловый дескриптор.
- *int close(int fd)*; – закрывает файловый дескриптор.
- *pid_t wait(int* status)*; – ожидает изменения состояния указанного дочернего процесса.
- *pid_t getpid(void)*; — возвращает PID текущего процесса. Используется для отладочного вывода.

В рамках лабораторной работы была реализована программа, состоящая из двух исполняемых файлов: родительского (*parent*) и дочернего (*child*). Родительский процесс создаёт дочерний с помощью системного вызова *fork*(), после чего заменяет его образ на отдельную программу с помощью *execv*(). Для обмена данными между процессами используется анонимный канал, созданный системным вызовом *pipe*().

Родительский процесс запрашивает у пользователя имя выходного файла, а затем читает из стандартного ввода строку, содержащую произвольное количество чисел типа *float*. Эта строка передаётся дочернему процессу через канал. С помощью системного вызова *dup2*() стандартный ввод дочернего процесса перенаправляется на чтение из канала, что позволяет ему получать данные, как будто они поступают из терминала.

Дочерний процесс (*child*) получает имя выходного файла как аргумент командной строки, читает строку с числами из своего стандартного ввода, парсит их, вычисляет сумму и записывает результат в указанный файл с помощью системных вызовов *open*() и *write*().

Программа демонстрирует кооперацию процессов через каналы (*pipe*) в соответствии с Unix-

философией: данные передаются байтами, процессы связаны иерархией, а стандартные потоки перенаправляются для организации межпроцессного взаимодействия.

Код программы

parent.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[] = "child";



int main(int argc, char *argv[]) {
    // проверка аргументов
    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);
    }

    // путь до директории
    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';
    }

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

// НЕОБЯЗАТЕЛЬНО
// // Child => Parent
// int server_to_client[2];
    // if (pipe(server_to_client) == -1) {
    //     const char msg[] = "error: failed to create pipe\n";
    //     write(STDERR_FILENO, msg, sizeof(msg));
    //     exit(EXIT_FAILURE);
    // }

// дочерний процесс
const pid_t child = fork();

switch (child) {
    case -1: {
```

```
            const char msg[] = "error: failed to spawn new process\n";
            write(STDERR_FILENO, msg, sizeof(msg));
            exit(EXIT_FAILURE);
        } break;

    case 0: { // дочерний
        {
            pid_t pid = getpid(); // получение PID дочернего

                    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]);

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

        {
                    char path[2048];
                    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 new exectuable image\n";
                        write(STDERR_FILENO, msg, sizeof(msg));
                        exit(EXIT_FAILURE);
```

```
                }
        }
    } break;

    default: { // PARENT
        {
            pid_t pid = getpid();

            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]);

        char buf[4096];
        ssize_t 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);
        }

        write(client_to_server[1], buf, bytes);
        close(client_to_server[1]);

        wait(NULL);
    } break;
    }
}
```

child.c

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        char msg[1024];
        uint32_t len = snprintf(msg, sizeof(msg) - 1, "error: need filename and string of floats\n");
        write(STDERR_FILENO, msg, len);
        exit(EXIT_FAILURE);
    }

        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);
        }

    char buf[4096];
        ssize_t bytes = read(STDIN_FILENO, buf, sizeof(buf) - 1);
    if (bytes <= 0) {
        const char msg[] = "error: failed to read from stdin\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
```

```
}
buf[bytes] = '\0';

if (bytes > 0 && buf[bytes - 1] == '\n') {
    buf[bytes - 1] = '\0';
}

// парсинг
float sum = 0;
char *token = strtok(buf, " \t\n");
while (token) {
    char *endptr;
    float f = strtof(token, &endptr);
    if (endptr != token && *endptr == '\0') { sum += f; }
    token = strtok(NULL, " \t\n");
}

char output[100];
uint32_t len = snprintf(output, sizeof(output) - 1, "%.3f\n", sum);
if (len < 0 || len >= sizeof(output)) {
    const char msg[] = "error: snprintf failed\n";
    write(STDERR_FILENO, msg, sizeof(msg) - 1);
    exit(EXIT_FAILURE);
}

if (write(file, output, len) != len) {
    const char msg[] = "error: failed to write to file\n";
    write(STDERR_FILENO, msg, sizeof(msg) - 1);
    exit(EXIT_FAILURE);
}

    close(file);
```

```
    exit(EXIT_SUCCESS);
}
```

Протокол работы программы

Тестирование:

```
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ cc -o parent parent.c
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ cc -o child child.c
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ ./parent result.txt
23461: I'm a parent, my child has PID 23462
23462: I'm a child
1.5 2.3 -0.7
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ cat result.txt
3.100
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ ./parent result.txt
23464: I'm a parent, my child has PID 23465
23465: I'm a child
1.532 -1.532 6.432 0 1.111
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ ls
child  child.c  parent  parent.c  result.txt
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ cat result.txt
7.543
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ ./parent out2.txt
23472: I'm a parent, my child has PID 23473
23473: I'm a child
42.0
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ cat out2.txt
42.000
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ ./parent out3.txt
23476: I'm a parent, my child has PID 23477
23477: I'm a child

maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ cat out3.txt
0.000
maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$
```

Strace:

maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ strace -o trace.log ./parent test.txt

23620: I'm a parent, my child has PID 23621

23621: I'm a child

1.2 3.4 -1.3

maks-alex@DESKTOP-QFPFVP1:~/OS/lab1/src$ cat trace.log

execve("./parent", ["./parent", "test.txt"], 0x7ffe56b00a58 /* 28 vars */) = 0

brk(NULL) = 0x5630be67d000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9bf8989000

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=19375, ...}) = 0
mmap(NULL, 19375, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9bf8984000
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\0\220\243\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"..., 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 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"..., 784, 64) = 784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f9bf8772000
mmap(0x7f9bf879a000, 1605632, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f9bf879a000
mmap(0x7f9bf8922000, 323584, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7f9bf8922000
mmap(0x7f9bf8971000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7f9bf8971000
mmap(0x7f9bf8977000, 52624, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f9bf8977000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7f9bf876f000
arch_prctl(ARCH_SET_FS, 0x7f9bf876f740) = 0
set_tid_address(0x7f9bf876fa10)         = 23620
set_robust_list(0x7f9bf876fa20, 24)     = 0
rseq(0x7f9bf8770060, 0x20, 0, 0x53053053) = 0
mprotect(0x7f9bf8971000, 16384, PROT_READ) = 0
mprotect(0x56308c783000, 4096, PROT_READ) = 0
mprotect(0x7f9bf89c1000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY})
= 0
munmap(0x7f9bf8984000, 19375)           = 0
readlink("/proc/self/exe", "/home/maks-alex/OS/lab1/src/pare"..., 1023) = 34
pipe2([3, 4], 0)                        = 0
```

```
clone(child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7f9bf876fa10) = 23621

getpid()                = 23620

write(1, "23620: I'm a parent, my child ha"..., 44) = 44

close(3)                = 0

read(0, "1.2 3.4 -1.3\n", 4096)         = 13

write(4, "1.2 3.4 -1.3\n", 13)          = 13

close(4)                = 0

--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=23621, si_uid=1000,
si_status=0, si_utime=0, si_stime=0} ---

wait4(-1, NULL, 0, NULL)                = 23621

exit_group(0)                = ?

+++ exited with 0 +++
```

Вывод

В ходе выполнения лабораторной работы были успешно изучены и применены основные системные вызовы для работы с процессами и межпроцессным взаимодействием в ОС Linux. Была реализована программа, демонстрирующая создание процессов, организацию каналов связи между ними и перенаправление стандартных потоков ввода-вывода. Основными сложностями стали корректная обработка системных вызовов с проверкой ошибок, безопасное управление файловыми дескрипторами (особенно закрытие ненужных концов канала) и парсинг вещественных чисел.