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: Губеев Д.И.

Преподаватель: Бахарев В.Д.

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

Дата: 02.10.25

Москва, 2025

**Постановка задачи**

**Вариант 4.**

Пользователь вводит команды вида: «число число число<endline>». 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.
* int pipe(int \*fd); – создает анонимный канал, возвращает 0 при успехе, -1 при ошибке
* ssize\_t read(int fd, void \*buf, size\_t count) - читает данные из файла по дескриптору fd в буфер buf. Возвращает количество прочитанных байт
* ssize\_t write(int fd, const void \*buf, size\_t count) - записывает данные в файл по дескриптору fd. Возвращает число записанных байт
* int open(const char \*pathname, int flag, mode\_t mode) - Открывает файл, возвращает файловый дескриптор
* int close(int fd) - закрываает файловый дескриптор
* int execv(const char \*path, char \*const argv[]) - заменяет текущий процесс новым. Возвращает -1 при ошибке
* int dup2(int fd, int fd2) - замена файлового дескриптора
* pid\_t wait(int \*wstatus) - ожидает завершения дочернего процесса. Возвращает pid завершившегося процесса
* pid\_t getpid(void) - возвращает pid текущего процесса
* ssize\_t readlink(const char \*pathname, char \*buf, size\_t bufsiz) - считывает путь, на который указывает символьная ссылка

Клиент и сервер общаются через пайпы. Клиент запускает сервер в дочернем процессе, перенаправляя ему стандартный ввод и вывод на пайпы. Из консоли клиент читает числа, отправляет их серверу. Сервер принимает строку чисел из stdin, разбивает на отдельные значения и делит первое число на каждое из следующих. Результаты сервер записывает в выходной файл. Если встречается деление на ноль, сервер и клиент завершают свою работу.

**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>

static char SERVER\_PROGRAM\_NAME[] = "server.out";

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));

if (bytes == 0) { // division by zero

break;

}

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 <ctype.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#include <fcntl.h>

#include <math.h>

void divideNumbers(char \* input, int32\_t file) {

char buf[4096];

strncpy(buf, input, sizeof(buf) - 1);

buf[sizeof(buf) - 1] = '\0';

char \* saveptr;

char \* token = strtok\_r(buf, " ", &saveptr);

float numbers[100];

int count = 0;

while (token != NULL && count < 100) {

numbers[count++] = strtof(token, NULL);

token = strtok\_r(NULL, " ", &saveptr);

}

if (count < 2) {

const char msg[] = "error: need at least two numbers\n";

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

return;

}

float dividend = numbers[0];

for (int i = 1; i < count; i++) {

if (fabs(numbers[i]) < 1e-6) {

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

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

exit(EXIT\_FAILURE);

}

float result = dividend / numbers[i];

char resultStr[50];

int len = snprintf(resultStr, sizeof(resultStr), "%f ", result);

if (write(file, resultStr, len) == -1) {

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

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

exit(EXIT\_FAILURE);

}

}

if (write(file, "\n", 1) == -1) {

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

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

exit(EXIT\_FAILURE);

}

}

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

char buf[4096];

ssize\_t bytes;

pid\_t pid = getpid();

// NOTE: `O\_WRONLY` only enables file for writing

// NOTE: `O\_CREAT` creates the requested file if absent

// NOTE: `O\_TRUNC` empties the file prior to opening

// NOTE: `O\_APPEND` subsequent writes are being appended instead of overwritten

int32\_t file = open(argv[1], O\_WRONLY | O\_CREAT | O\_TRUNC | O\_APPEND, 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))) {

if (bytes < 0) {

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

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

exit(EXIT\_FAILURE);

}

buf[bytes] = '\0';

divideNumbers(buf, file);

{

{

const char msg[] = "Server received: ";

write(STDERR\_FILENO, msg, sizeof(msg) - 1);

}

// NOTE: Echo back to client

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

if (written != bytes) {

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

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

exit(EXIT\_FAILURE);

}

}

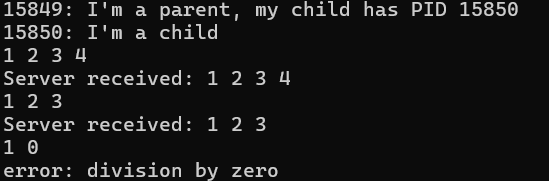
}

close(file);

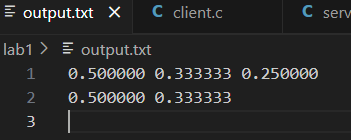
}

**Протокол работы программы**

**Входные данные:**

****

**Выходные данные:**

****

**Вывод**

При выполнении данной лабораторной работы я научился работать с процессами в ОС. На практике увидел работу fork, pipe и других системных вызовов. Научился обрабатывать ввод, вывод как поток байтов, преобразовывать его в строки.