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-210БВ-24

Студент: Вельдеватова Н.С.

Преподаватель: Бахарев В.Д.

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

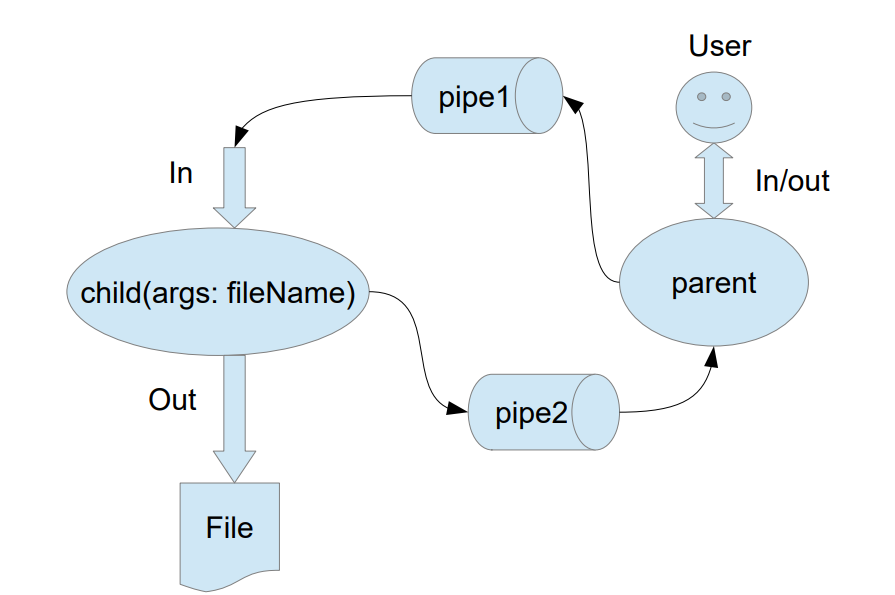
Дата: 02.10.25

Москва, 2025

**Постановка задачи**

**Вариант 2.**

**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 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». 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**

В данной лабораторной работе реализована программа для межпроцессного взаимодействия с использованием системных вызовов Unix/Linux. Основная задача — создать родительский процесс, который порождает дочерний процесс, передает ему данные через неименованные каналы (pipes), и получает результат обратно. Дочерний процесс обрабатывает входные данные (суммирует числа из строки), записывает результат в файл и выводит его на стандартный вывод.

**Использованные системные вызовы :**

* pid\_t fork(void); – создает дочерний процесс, копируя адресное пространство родителя. Возвращает 0 в дочернем процессе, PID дочернего в родительском или -1 при ошибке.
* int pipe(int \*fd); – создает неименованный канал для межпроцессного взаимодействия. Возвращает два файловых дескриптора: fd[0] для чтения, fd[1] для записи, или -1 при ошибке.
* int dup2(int oldfd, int newfd); – дублирует файловый дескриптор oldfd на newfd, закрывая newfd, если он был открыт. Используется для перенаправления stdin/stdout на pipes.
* int execv(const char \*path, char \*const argv[]); – заменяет текущий процесс образом программы по пути path с аргументами argv. Не возвращается при успехе, -1 при ошибке.
* ssize\_t read(int fd, void \*buf, size\_t count); – читает до count байт из файлового дескриптора fd в буфер buf. Возвращает количество прочитанных байт или -1 при ошибке.
* ssize\_t write(int fd, const void \*buf, size\_t count); – записывает count байт из буфера buf в файловый дескриптор fd. Возвращает количество записанных байт или -1 при ошибке.
* int close(int fd); – закрывает файловый дескриптор fd, освобождая ресурсы.
* pid\_t wait(int \*status); – ожидает завершения дочернего процесса и сохраняет статус в status. Возвращает PID завершившегося процесса или -1 при ошибке.
* int open(const char \*pathname, int flags, mode\_t mode); – открывает файл по пути pathname с флагами flags и правами mode. Возвращает файловый дескриптор или -1 при ошибке.
* ssize\_t readlink(const char \*pathname, char \*buf, size\_t bufsiz); – читает содержимое символьной ссылки pathname в буфер buf. Возвращает количество прочитанных байт или -1 при ошибке (используется для получения пути к программе).

**Алгоритм решения:**

1. Родительский процесс создает два неименованных канала (pipes) для двусторонней связи с дочерним процессом.
2. Родительский процесс вызывает fork() для создания дочернего процесса.
3. В дочернем процессе перенаправляются стандартные потоки ввода/вывода на каналы с помощью dup2(), после чего выполняется execv() для запуска программы child с аргументом (именем файла).
4. Родительский процесс читает данные из стандартного ввода, передает их дочернему через pipe, закрывает канал записи и ждет завершения дочернего процесса.
5. Дочерний процесс читает данные из pipe, парсит числа, вычисляет сумму, записывает результат в указанный файл и выводит на stdout.
6. Родительский процесс читает результат из pipe дочернего процесса, проверяет на ошибки и выводит его на соответствующий поток (stdout или stderr).
7. Программа обрабатывает ошибки на каждом этапе, используя системные вызовы для чтения/записи и управления процессами. Входные данные — строка с числами, разделенными пробелами; выход — сумма с двумя знаками после запятой.

**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>

#include <string.h>

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

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[2048];

{

// 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[] = "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 parent\_to\_child[2];

if (pipe(parent\_to\_child) == -1) {

const char msg[] = "Failed to create pipe\n";

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

exit(EXIT\_FAILURE);

}

int child\_to\_parent[2];

if (pipe(child\_to\_parent) == -1) {

const char msg[] = "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[] = "Failed to spawn new process\n";

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

exit(EXIT\_FAILURE);

} break;

case 0: { // NOTE: We're a child

close(parent\_to\_child[1]);

dup2(parent\_to\_child[0], STDIN\_FILENO);

close(parent\_to\_child[0]);

dup2(child\_to\_parent[1], STDOUT\_FILENO);

close(child\_to\_parent[1]);

{

char path[4096];

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[] = "Failed to exec into new exectuable image\n";

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

exit(EXIT\_FAILURE);

}

}

} break;

default: { // NOTE: We're a parent

char buf[4096];

ssize\_t bytes;

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

if (bytes < 0) {

const char msg[] = "Failed to read from stdin\n";

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

exit(EXIT\_FAILURE);

}

write(parent\_to\_child[1], buf, bytes);

break; // NOTE: Exit after first input

}

close(parent\_to\_child[1]);

close(child\_to\_parent[1]);

char result\_buf[4096];

ssize\_t result\_bytes = read(child\_to\_parent[0], result\_buf, sizeof(result\_buf));

if (result\_bytes > 0) {

// NOTE: Check if the output from child starts with "ERROR: " prefix

// If yes, it's an error message, so write to STDERR; otherwise, it's the result, write to STDOUT

const char error\_prefix[] = "ERROR: ";

int output\_fd = STDOUT\_FILENO;

size\_t offset = 0;

if (result\_bytes >= sizeof(error\_prefix) - 1 && strncmp(result\_buf, error\_prefix, sizeof(error\_prefix) - 1) == 0) {

output\_fd = STDERR\_FILENO;

offset = sizeof(error\_prefix) - 1; // Skip the "ERROR: " prefix when writing

}

write(output\_fd, result\_buf + offset, result\_bytes - offset);

} else if (result\_bytes < 0) {

const char msg[] = "Failed to read from child pipe\n";

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

exit(EXIT\_FAILURE);

}

close(child\_to\_parent[0]);

int status;

if (wait(&status) == -1) {

const char msg[] = "Failed to wait for child\n";

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

exit(EXIT\_FAILURE);

}

if (WIFEXITED(status)) {

if (WEXITSTATUS(status) != 0) {

exit(EXIT\_FAILURE);

}

} else {

// NOTE: Child terminated abnormally

exit(EXIT\_FAILURE);

}

} break;

}

}

**child.c**

#include <stdint.h>

#include <stdbool.h>

#include <ctype.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdio.h>

#include <math.h>

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

char buf[4096];

ssize\_t bytes;

// 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(STDOUT\_FILENO, msg, sizeof(msg) - 1);

\_exit(EXIT\_FAILURE);

}

// NOTE: Read input data from standard input into the buffer

bytes = read(STDIN\_FILENO, buf, sizeof(buf) - 1);

if (bytes < 0) {

const char msg[] = "ERROR: Failed to read from stdin\n";

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

exit(EXIT\_FAILURE);

}

buf[bytes] = '\0';

float sum = 0.0f;

int count = 0;

char \*ptr = buf;

char \*endptr;

while (\*ptr) {

// NOTE: Skip whitespace characters in the input, but stop at newline

if (isspace(\*ptr) && \*ptr != '\n') {

ptr++;

continue;

}

if (\*ptr == '\n') {

break;

}

float num = strtof(ptr, &endptr);

if (num == HUGE\_VALF || num == -HUGE\_VALF) {

const char msg[] = "ERROR: Number out of range\n";

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

exit(EXIT\_FAILURE);

}

if (ptr == endptr) {

const char msg[] = "ERROR: Invalid character in input\n";

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

exit(EXIT\_FAILURE);

}

sum += num;

count++;

if (isinf(sum)) {

const char msg[] = "ERROR: Sum overflow\n";

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

exit(EXIT\_FAILURE);

}

ptr = endptr;

}

if (count == 0) {

const char msg[] = "ERROR: No numbers provided\n";

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

exit(EXIT\_FAILURE);

}

// NOTE: Format the computed sum as a string and write it to file

char sum\_str[64];

int len = snprintf(sum\_str, sizeof(sum\_str), "%.2f\n", sum);

int32\_t written = write(file, sum\_str, len);

if (written != len) {

const char msg[] = "ERROR: Failed to write to file\n";

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

exit(EXIT\_FAILURE);

}

// NOTE: Also write the sum to standard output

int32\_t stdout\_written = write(STDOUT\_FILENO, sum\_str, len);

if (stdout\_written != len) {

const char msg[] = "ERROR: Failed to write to stdout\n";

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

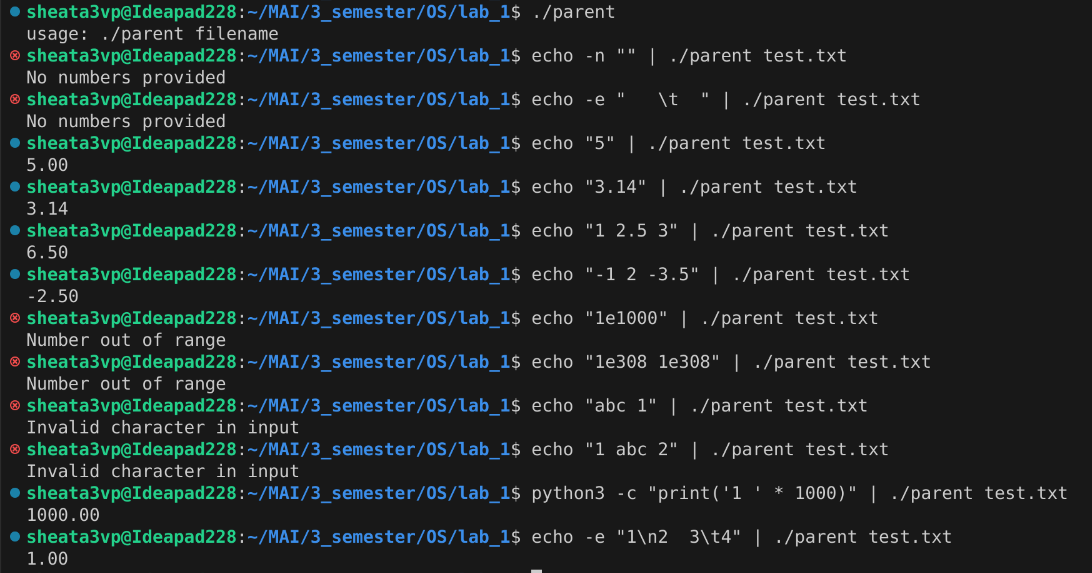
exit(EXIT\_FAILURE);

}

close(file);

}

**Протокол работы программы**

****

**Вывод**

**В результате выполнения лабораторной работы была успешно реализована система межпроцессного взаимодействия с использованием системных вызовов Unix/Linux, где родительский процесс управляет дочерним через неименованные каналы, обеспечивая передачу данных и обработку результатов. Программа корректно суммирует числа из входного потока, записывает результат в файл и выводит его на экран, демонстрируя базовые принципы многозадачности и синхронизации процессов. Код получился компактным и надежным, с обработкой ошибок на каждом этапе, что позволяет использовать его как основу для более сложных задач в операционных системах.**