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: Захарченко М.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 02.10.25

Москва, 2025

Постановка задачи

Вариант 8.

Родительский процесс создает дочерний процесс. Первой строчкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия файла с таким именем на чтение. Стандартный поток ввода дочернего процесса переопределяется открытым файлом. Дочерний процесс читает команды из стандартного потока ввода. Стандартный поток вывода дочернего процесса перенаправляется в pipe1. Родительский процесс читает из pipe1 и прочитанное выводит в свой стандартный поток вывода. Родительский и дочерний процесс должны быть представлены разными программами.

В файле записаны команды вида:«число число число<endline>». Дочерний процесс производит деление первого числа команда, на последующие числа в команде, а результат выводит в стандартный поток вывода. 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 int. 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 канал для межпроцессного взаимодействия, возвращает -1 если возникла ошибка при создании. Массив fd состоит из двух элементов: fd[0] — файловый дескриптор для чтения и fd[1] — файловый дескриптор для записи
- ssize_t readlink(char* path, char*, buf, ssize_t bufsize); - читает символическую ссылку, содержащую путь к исполняемому файлу текущего процесса и записывает в buf
- int close(int __fd); - закрывает файловый дескриптор
- *int* dup2(*int __fd*, *int __fd2*); - дублирует файловый дескриптор, заменяя старый
- int open(const char* pathname, int flags, mode_t mode); - открывает файл
- *int* execv(const *char* *__*path*, *char* *const *__*argv*; - заменяет текущий образ процесса исполняемым файлом
- *ssize_t* read(*int __fd*, *void *__buf*, *size_t __nbytes*); - читает данные из файлового дескриптора
- *ssize_t* write(*int __fd*, const *void *__buf*, *size_t __n*); - записывает данные в файловый дескриптора
- *pid_t* wait(*int *__stat_loc*); - ожидает завершения дочернего процесса
- void exit(int status); - завершает процесс указанным статусом

В рамках лабораторной работы было реализовано межпроцессное взаимодействие. В родительский процесс подается файл, который будет использован как стандартный поток ввода дочернего процесса. Дочерний процесс заменяется другим исполняемым файлом, который производит обработку данных согласно заданию и отправляет обработанные данные в стандартный поток вывода дочернего процесса, который в свою очередь перенаправляется в канал записи для родительского процесса. Сам родительский процесс читает данные из канала и выводит их в свой стандартный поток вывода.

Код программы

parent.c

```
#include <stdint.h>
#include <stdbool.h>
#include <fcntl.h>

#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

static char CHILD_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);
	}
	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';
	}

	int child_to_parent[2];

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

	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: {
```

```
	close(child_to_parent[0]);

	int32_t file = open(argv[1], O_RDONLY);

	dup2(file, STDIN_FILENO);
	close(file);

	dup2(child_to_parent[1], STDOUT_FILENO);
	close(child_to_parent[1]);
		{
			char path[1024];
			snprintf(path, sizeof(path), "%s/%s", progpath,
CHILD_PROGRAM_NAME);

			char *const args[] = {CHILD_PROGRAM_NAME, 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: {

		close(child_to_parent[1]);

		char buf[4096];
		ssize_t bytes;

		while (bytes = read(child_to_parent[0], 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(STDOUT_FILENO, buf, bytes);
		}

		close(child_to_parent[0]);
		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>

void intToStr(char* number, int num){
    char temp[12];
    bool negative = false;
    int len = 0;

    if(num < 0){
        num = -num;
        negative = true;
    }

    do{
        temp[len++] = (num % 10) + '0';
        num /= 10;
    } while(num > 0);

    if(negative){
        temp[len++] = '-';
    }

    for(int i = 0; i < len; ++i){
        number[i] = temp[len - i - 1];
    }
    number[len] = '\0';
}

int strToInt(char* number){
    int result = 0;
    int start = 0;
    if(number[0] == '-'){
        start = 1;
    }

    for(int i = start; i < strlen(number); ++i){
        result = result * 10 + (number[i] - '0');
    }

    if(start){
        return -result;
    }
    return result;
}

void compare(char* res, int num1, int num2){
    char stringNumberOne[12];
    char stringNumberTwo[12];
    intToStr(stringNumberOne, num1);
    intToStr(stringNumberTwo, num2);

    strcpy(res, stringNumberOne);
```

```
    strcat(res, " ");

    strcat(res, stringNumberTwo);
    strcat(res, "\n");
}

int main(int argc, char **argv) {
        char buf[4096];
        ssize_t bytes;

    int numbers[3];
    int count = 0, strIndex = 0;
    char strNumber[12];
    char outputString[25];

        while (bytes = read(STDIN_FILENO, buf, sizeof(buf))) { // STDIN_FILENO = input_file
                if (bytes < 0) {
                        const char msg[] = "error: failed to read from stdin\n";
                        write(STDERR_FILENO, msg, sizeof(msg));
                        exit(EXIT_FAILURE);
                }

                // Transform data
                for (uint32_t i = 0; i < bytes; ++i) {
        if(buf[i] == '-' || (buf[i] >= '0' && buf[i] <= '9')){
           strNumber[strIndex++] = buf[i];
        } else if(buf[i] == ' ' || buf[i] == '\n'){
           strNumber[strIndex] = '\0';
           numbers[count++] = strToInt(strNumber);

           strNumber[0] = '\0';
           strIndex = 0;

           if(count == 3 && buf[i] == '\n'){
              if(numbers[1] == 0 || numbers[2] == 0){
                 const char msg[] = "error: division by zero\n";
                 write(STDERR_FILENO, msg, sizeof(msg));
                 exit(EXIT_FAILURE);
              }
              int result1 = numbers[0] / numbers[1];
              int result2 = numbers[0] / numbers[2];
              count = 0;
              outputString[0] = '\0';
              compare(outputString, result1, result2);
              write(STDOUT_FILENO, outputString, strlen(outputString));
           } else if(count < 3 && buf[i] == '\n'){
              const char msg[] = "error: in line must be 3 numbers\n";
              write(STDERR_FILENO, msg, sizeof(msg));
              exit(EXIT_FAILURE);
           }
        }
                }
        }
    return 0;
}
```

Протокол работы программы

```
❯ cat < input.txt
12 2 4
15 3 5
90 10 9
100 100
27 0 9
12 3 4
56 78 32
❯ ./parent input.txt
error: in line must be 3 numbers
6 3
5 3
9 10
 ~/os_labs                                                    00:30:48
```

```
❯ cat < input.txt
12 2 4
15 3 5
90 10 9
100 100 25
27 0 9
12 3 4
56 78 32
❯ ./parent input.txt
error: division by zero
6 3
5 3
9 10
1 4
 ~/os_labs                                                    00:32:53
```

```
❯ cat < input.txt
12 2 4
15 3 5
90 10 9
100 100 25
27 3 9
12 3 4
56 78 32
❯ ./parent input.txt
6 3
5 3
9 10
1 4
9 3
4 3
0 1
 ~/os_labs                                                    00:34:03
```

```
❯ cat < input.txt
12 2 4
15 3 5
9.1 10 9
100 100 25
27 3 9
12 3 4
56 78 32
❯ ./parent input.txt
error: numbers must be natural
6 3
5 3
 ~/os_labs                                                    00:35:34
```

Вывод

В ходе работы было изучено управление процессами в операционной системе, а также реализован обмен данных между процессами посредством каналов. Была составлена программа на языке Си, осуществляющая управление процессами по схеме «родитель-потомок» с использование системных вызовов fork() и execv(). Реализовано межпроцессное взаимодействие по обмену данных, перенаправление потоков ввода-вывода (STDIN дочернего процесса на входной файл; STDOUT дочернего класса в канал связи), обработка ошибок и синхронизация процессов при помощи системного вызова wait(). Таким образом, цель работы по приобретению практических навыков в управлении процессами ОС и обмена данных между процессами при помощи каналов - достигнутаю