

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и
прикладной математики
Кафедра вычислительной математики и
программирования

**Лабораторная работа №2 по курсу
"Операционные системы"**

УПРАВЛЕНИЕ ПРОЦЕССАМИ

Студент: Сеимов Максим Сергеевич

Группа: М8О-208Б-21

Вариант: 9

Преподаватель: Соколов Андрей Алексеевич

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2022

1 Постановка задачи

Цель работы

Целью лабораторной работы №2 является приобретение практических навыков в:

- Управлении процессами в ОС
- Применении системных вызовов `fork`, `open`, `read` и т.д.
- Обеспечение обмена данных между процессами посредством каналов

Задание

Необходимо составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем (в данном случае Linux). В результате работы программа (основной процесс) должен создать для решения 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.

Вариант 9

В варианте №9 нужно написать программу, представленную двумя процессами: родительский принимает у пользователя имя файла, создаёт дочерний, который из введённого файла считывает команды вида [число число число\n], и отправляет в родительский процесс результаты деления первого числа на последующие в строке; числа представлены типом `float`.

Общие сведения о программе

Программа представлена двумя исполняемыми файлами **lab2** и **child**, компилируемые из файлов **main.c** и **child.c** соответственно. Используются заголовочные файлы **sys/types.h**, **sys/stat.h**, **fcntl.h**, **wait.h**, **unistd.h**, **stdlib.h**, **stdio.h**, **string.h**. В программе используются следующие системные вызовы:

- **fork** - Создание дочернего процесса (копия родительского с другим идентификатором)
- **execve** - Замена образа памяти процесса другим исполняемым файлом
- **waitpid** - Ожидание завершения дочернего процесса
- **exit** - Завершение выполнения процесса и возвращение статуса
- **pipe** - Создание неименованного канала для передачи данных между процессами
- **dup2** - Переназначение файлового дескриптора
- **open** - Открытие или создание файла
- **close** - Закрытие файла

Общий метод и алгоритм решения

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы вышеописанных системных вызовов
2. Написать код дочернего процесса, выполняющего простейшие операции деления для команд указанного типа
3. В коде родительского процесса создать **pipe**, после вызова **fork** в дочерней ветке переназначить файловые дескрипторы файла, открытого на чтение, и записывающего конца неименованного канала как стандартные потоки ввода-вывода, после чего загрузить в память файл **child**; в родительской ветке ждать завершения работы дочернего процесса
4. После завершения дочернего процесса вывести в родительском результаты из канала для передачи данных в стандартный поток вывода.

Основные файлы программы

main.c:

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4  #include <wait.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <string.h>
9
10 #define STR_LEN 128
11
12 char *const child_args[] = { "child", NULL };
13 char *child_env[] = { NULL };
14
15 size_t strlen_new(char *s) {
16     for (size_t i = 0; i < strlen(s); ++i) {
17         if (s[i] == '\n')
18             return i;
19     }
20     return strlen(s);
21 }
22
23 int main() {
24
25     char *fname_buf = calloc(sizeof(char), STR_LEN);
26     if (read(0, fname_buf, STR_LEN) < 0) {
27         perror("Read error");
28         exit(EXIT_FAILURE);
29     }
30
31     char *fname = malloc(sizeof(char) * strlen_new(fname_buf));
32     strncpy(fname, fname_buf, strlen_new(fname_buf));
33     free(fname_buf);
34
35     pid_t fd[2];
36     if (pipe(fd) != 0) {
```

```

37     perror("Pipe error");
38     exit(EXIT_FAILURE);
39 }
40
41 int filedес;
42 if ((filedes = open(fname, O_RDONLY)) < 0) {
43     perror(fname);
44     exit(EXIT_FAILURE);
45 }
46
47 pid_t pid = fork();
48
49 if (pid < 0) {
50     perror("Fork error");
51     exit(EXIT_FAILURE);
52 }
53
54 if (pid == 0) { // Child
55
56     close(fd[0]);
57     if (dup2(filedes, 0) < 0) {
58         perror("Descriptor redirection error (input)");
59         exit(EXIT_FAILURE);
60     }
61     if (dup2(fd[1], 1) < 0) {
62         perror("Descriptor redirection error (output)");
63         exit(EXIT_FAILURE);
64     }
65     execve(child_args[0], child_args, child_env);
66     perror("Execve error");
67 }
68
69 else { // Parent
70
71     int child_exit_status;
72     if (waitpid(pid, &child_exit_status, 0) < 0) {
73         perror("Waitpid error");
74     }
75     if (WEXITSTATUS(child_exit_status) == 1) {
76         char *err1 = "Child process error exit, divison by zero\n";
77         write(2, err1, strlen(err1));
78         exit(EXIT_FAILURE);
79     }
80
81     else if (WEXITSTATUS(child_exit_status) == 4) {
82         char *err2 = "Child process error exit, expected float number\n";
83         write(2, err2, strlen(err2));
84         exit(EXIT_FAILURE);
85     }
86
87     else if (WEXITSTATUS(child_exit_status) != 0) {
88         perror("Child process error");
89         exit(EXIT_FAILURE);
90     }
91
92     int t;
93     float result;
94     while ((t = read(fd[0], &result, sizeof(float))) == sizeof(float)) {
95
96         char *result_string = calloc(sizeof(char), STR_LEN);

```

```

97         gcvt(result, 7, result_string);
98         if (write(1, result_string, strlen(result_string)) < 0) {
99             perror("Can't write result to stdout");
100             exit(EXIT_FAILURE);
101         }
102
103         char newline_c = '\n';
104         if (write(1, &newline_c, 1) < 0) {
105             perror("Can't write \n to stdout");
106             exit(EXIT_FAILURE);
107         }
108     }
109
110     close(fd[0]);
111     close(fd[1]);
112     close(filedes);
113 }
114
115 return 0;
116 }

```

child.c:

```

1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  #define STR_LEN 32
8
9  int first = 1;
10
11 int read_float(int fd, float *f) {
12     char *buf = calloc(sizeof(char), STR_LEN);
13     char c = 0;
14     short i = 0;
15
16     if (read(fd, &c, 1) < 0) {
17         exit(3);
18     }
19
20     if ((c == ' ') || (c == '\n') || (c == '\0'))
21         return -1;
22
23     while (c != ' ' && c != '\n') {
24         buf[i++] = c;
25         read(fd, &c, 1);
26     }
27
28     *f = strtod(buf, NULL);
29
30     if (c == '\n') {
31         return 0;
32     }
33
34     return 1;
35 }
36
37 int exec_command() {
38

```

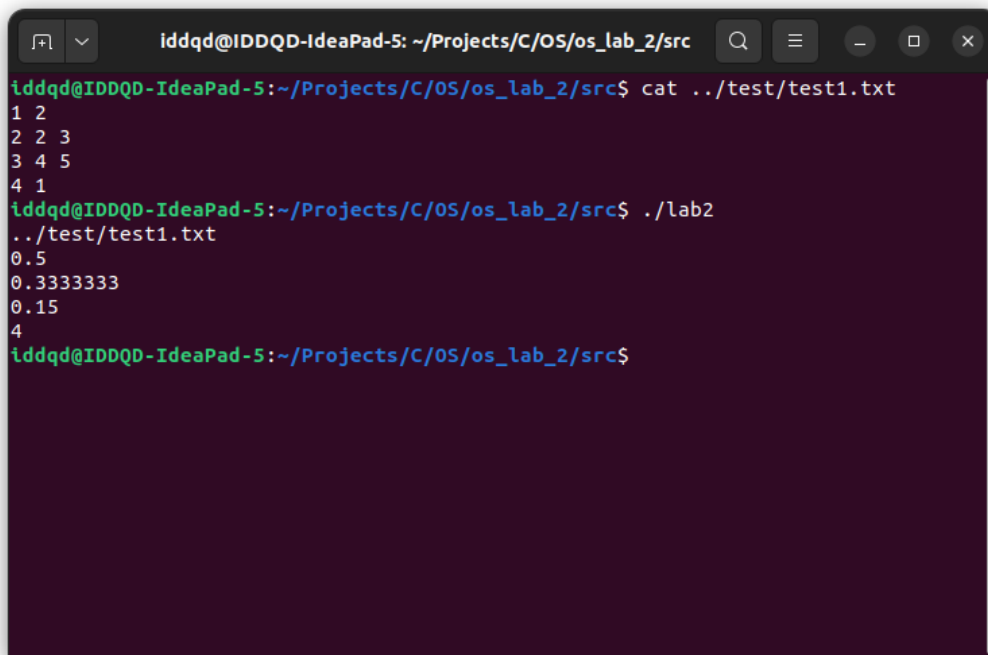
```

39     int read_result;
40
41     float init_num = 0, div_num = 0;
42
43     if ((read_result = read_float(0, &init_num)) == -1) {
44
45         if (first == 1)
46             exit(4);
47         else
48             return 0;
49     }
50
51     else if (read_result == 0) {
52         if (write(1, &init_num, sizeof(float)) < 0) {
53             exit(2);
54         }
55         first = 0;
56     }
57
58     else {
59         first = 0;
60         while ((read_result = read_float(0, &div_num)) != -1) {
61
62             if (div_num == 0) {
63                 exit(1);
64             }
65
66             init_num /= div_num;
67
68             if (read_result == 0) {
69                 break;
70             }
71         }
72
73         if (write(1, &init_num, sizeof(float)) < 0) {
74             exit(2);
75         }
76     }
77
78     return 1;
79 }
80
81
82 int main() {
83     while (exec_command() == 1) {
84         ;
85     }
86
87     char c = '\0';
88     if (write(1, &c, sizeof(char)) < 0) {
89         exit(2);
90     }
91
92     return 0;
93 }

```

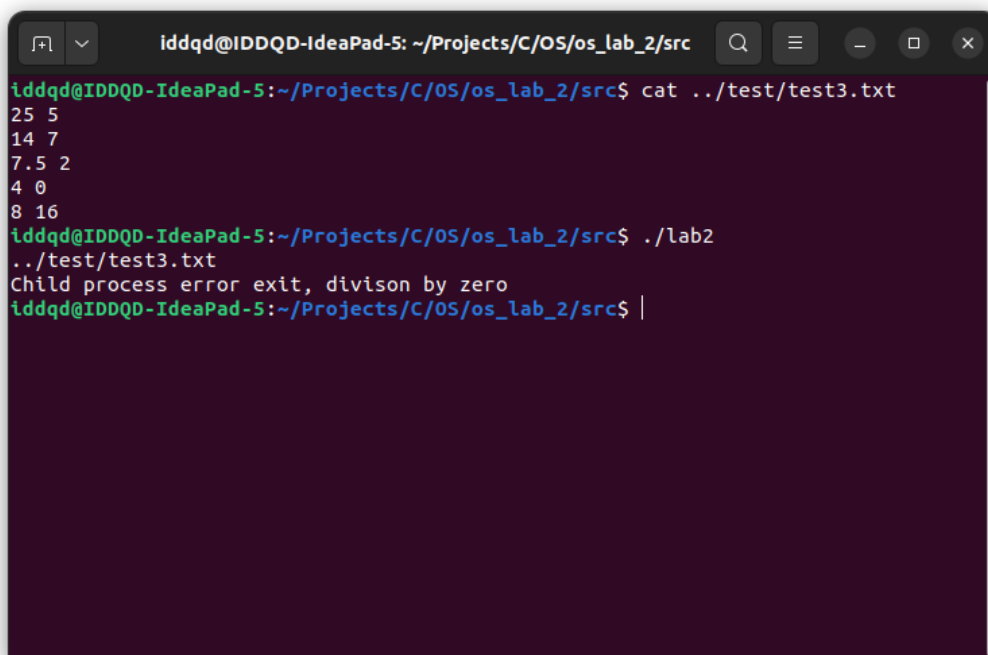
Пример работы

Запуск программы на нескольких тестах:

A terminal window titled 'iddqd@IDDQD-IdeaPad-5: ~/Projects/C/OS/os_lab_2/src'. The user enters 'cat ../test/test1.txt' and the output is:
1 2
2 2 3
3 4 5
4 1
Then the user enters './lab2' and the output is:
../test/test1.txt
0.5
0.3333333
0.15
4
The prompt returns to the user.

```
iddqd@IDDQD-IdeaPad-5: ~/Projects/C/OS/os_lab_2/src$ cat ../test/test1.txt
1 2
2 2 3
3 4 5
4 1
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ ./lab2
../test/test1.txt
0.5
0.3333333
0.15
4
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$
```

Рис. 1: Простейший пример запуска программы с несколькими командами

A terminal window titled 'iddqd@IDDQD-IdeaPad-5: ~/Projects/C/OS/os_lab_2/src'. The user enters 'cat ../test/test3.txt' and the output is:
25 5
14 7
7.5 2
4 0
8 16
Then the user enters './lab2' and the output is:
../test/test3.txt
Child process error exit, divison by zero
The prompt returns to the user.

```
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ cat ../test/test3.txt
25 5
14 7
7.5 2
4 0
8 16
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ ./lab2
../test/test3.txt
Child process error exit, divison by zero
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ |
```

Рис. 2: При делении на ноль хотя бы в одной команде оба процесса завершаются

```
iddqd@IDDQD-IdeaPad-5: ~/Projects/C/OS/os_lab_2/src
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ cat ../test/test4.txt
25
8
16.92
0
1
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ ./lab2
../test/test4.txt
25
8
16.92
0
1
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ |
```

Рис. 3: Если в команде содержится только одно число, оно и будет результатом выполнения

```
iddqd@IDDQD-IdeaPad-5: ~/Projects/C/OS/os_lab_2/src
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ cat ../test/test6.txt
10.0 1.22 +3.6623 -0.023
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ ./lab2
../test/test6.txt
-97.3102
iddqd@IDDQD-IdeaPad-5:~/Projects/C/OS/os_lab_2/src$ |
```

Рис. 4: Числа могут записываться как целые или как десятичные дроби, также возможно использование знаков

Вывод

В ходе выполнения задания были изучены принципы работы используемых системных вызовов, а также было получено понимание устройства отдельных частей системы в целом (речь о файловых дескрипторах, процессах, их идентификаторах и каналах передачи данных). Исходный код программы писался без привычных для задач прошлых семестров функций ввода-вывода `printf` и `scanf`, что, на мой взгляд, демонстрирует функциональность системных вызовов с выбором дескриптора на запись и чтение, которые помимо стандартных потоков могут взаимодействовать с файлами и другими процессами.