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-216Б-24

Студент: Седов М. А

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 19.11.25

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<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. Возвращает PID дочернего процесса в родительском процессе, 0 в дочернем процессе, или -1 в случае ошибки.

● int pipe(int pipefd[2]) – создает неименованный канал (pipe) для передачи данных между процессами. Массив `pipefd` содержит два файловых дескриптора: `pipefd[0]` для чтения и `pipefd[1]` для записи. Возвращает 0 при успехе, -1 при ошибке.

● int dup2(int oldfd, int newfd) – переназначает файловый дескриптор. Копирует `oldfd` в `newfd`, закрывая `newfd` если он был открыт. Используется для перенаправления стандартных потоков ввода/вывода. Возвращает новый дескриптор при успехе, -1 при ошибке.

● int execv(const char *pathname, char *const argv[]) – заменяет образ памяти текущего процесса новым исполняемым файлом. `pathname` – путь к исполняемому файлу, `argv` – массив аргументов командной строки (должен заканчиваться NULL). При успехе не возвращает управление, при ошибке возвращает -1.

● ssize_t read(int fd, void *buf, size_t count) – читает данные из файлового дескриптора. Возвращает количество прочитанных байт, 0 при достижении конца файла, -1 при ошибке.

● ssize_t write(int fd, const void *buf, size_t count) – записывает данные в файловый дескриптор. Возвращает количество записанных байт, -1 при ошибке.

● int open(const char *pathname, int flags, mode_t mode) – открывает или создает файл. flags определяет режим доступа (O_WRONLY, O_CREAT, O_TRUNC и т.д.), mode – права доступа при создании файла. Возвращает файловый дескриптор при успехе, -1 при ошибке.

● int close(int fd) – закрывает файловый дескриптор. Освобождает ресурсы, связанные с дескриптором. Возвращает 0 при успехе, -1 при ошибке.

● pid_t waitpid(pid_t pid, int *status, int options) – ожидает завершения дочернего процесса. `pid` – PID дочернего процесса, `status` – указатель для сохранения статуса завершения, `options` – опции ожидания. Возвращает PID завершенного процесса при успехе, -1 при ошибке.

● ssize_t readlink(const char *pathname, char *buf, size_t bufsiz) – читает содержимое символической ссылки. Используется для получения пути к исполняемому файлу через `/proc/self/exe`. Возвращает количество прочитанных байт при успехе, -1 при ошибке.

● void _exit(int status) – завершает выполнение процесса немедленно, не вызывая функции очистки. status – код возврата процесса.

Алгоритм работы программы:

1. Родительский процесс (parent.c):

- Читает первую строку из стандартного ввода – имя файла для записи результатов

- Создает два pipe: `parent_to_child` для передачи команд и `child_to_parent` для получения ответов

- Вызывает `fork()` для создания дочернего процесса

- В дочернем процессе: закрывает ненужные концы pipe, перенаправляет stdin/stdout через `dup2()`, вызывает `execv()` для запуска программы child

- В родительском процессе: закрывает ненужные концы pipe, в цикле читает строки с числами из stdin, передает их дочернему процессу через pipe, получает ответы и выводит их в stdout

- После завершения ввода закрывает pipe и ожидает завершения дочернего процесса через `waitpid()`

2. Дочерний процесс (child.c):

- Получает имя файла как аргумент командной строки

- Открывает файл для записи через `open()` с флагами O_WRONLY | O_CREAT | O_TRUNC

- В цикле читает строки из stdin (который перенаправлен на pipe от родителя)

- Парсит строку, извлекая числа с плавающей точкой и вычисляя их сумму

- Форматирует результат и записывает его в файл и в stdout (который перенаправлен на pipe к родителю)

- При ошибке парсинга отправляет сообщение об ошибке родителю

- После завершения ввода закрывает файл и завершает работу

Код программы

parent.c

```
1. #define _POSIX_C_SOURCE 200809L#include <errno.h>
```

```c
#include <fcntl.h>
#include <limits.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

    #define CHILD_PROGRAM_NAME "child.c"
    #define MAX_LINE_LENGTH 4096

    static size_t string_length(const char *text) {
          size_t length = 0;
          while (text[length] != '\0') ++length;
          return length;
    }

    static void write_all(int fd, const char *buffer, size_t length) {
          while (length > 0) {
                ssize_t written = write(fd, buffer, length);
                if (written < 0) _exit(EXIT_FAILURE);
                buffer += (size_t)written;
                length -= (size_t)written;
          }
    }

    static void fail(const char *message) {
          write_all(STDERR_FILENO, message, string_length(message));
          _exit(EXIT_FAILURE);
    }

    static ssize_t read_line(int fd, char *buffer, size_t capacity) {
          if (capacity == 0) return -1;

          size_t offset = 0;
          while (offset + 1 < capacity) {
                char ch;
                ssize_t bytes = read(fd, &ch, 1);
                if (bytes < 0) {
                      return -1;
                }
                if (bytes == 0) break;

                buffer[offset++] = ch;
                if (ch == '\n') break;
          }
          buffer[offset] = '\0';
          return (ssize_t)offset;
    }

    static void trim_trailing_newline(char *line) {
          size_t length = string_length(line);
          if (length == 0) {
                return;
          }
          if (line[length - 1] == '\n') line[length - 1] = '\0';
    }

    static void build_child_path(char *result, size_t capacity) {
          char executable_path[PATH_MAX];
          ssize_t len = readlink("/proc/self/exe", executable_path,
    sizeof(executable_path) - 1);
          if (len == -1) fail("error: failed to read /proc/self/exe\n");

          executable_path[len] = '\0';

          while (len > 0 && executable_path[len] != '/') --len;
          if (len == 0) fail("error: executable path is invalid\n");

          executable_path[len] = '\0';

          size_t dir_length = string_length(executable_path);
          size_t name_length = string_length(CHILD_PROGRAM_NAME);
          if (dir_length + 1 + name_length + 1 > capacity) fail("error: child path
    buffer is too small\n");
```

```
76.
77.         size_t index = 0;
78.         for (size_t i = 0; i < dir_length; ++i) result[index++] =
    executable_path[i];
79.         result[index++] = '/';
80.         for (size_t i = 0; i < name_length; ++i) result[index++] =
    CHILD_PROGRAM_NAME[i];
81.
82.         result[index] = '\0';
83.   }
84.
85.   static void forward_line(int output_fd, const char *line) {
86.         size_t length = string_length(line);
87.         if (length == 0 || line[length - 1] != '\n') {
88.               write_all(output_fd, line, length);
89.               write_all(output_fd, "\n", 1);
90.               return;
91.         }
92.         write_all(output_fd, line, length);
93.   }
94.
95.   int main(void) {
96.         char filename[MAX_LINE_LENGTH];
97.         ssize_t filename_len = read_line(STDIN_FILENO, filename,
    sizeof(filename));
98.         if (filename_len <= 0) fail("error: failed to read filename\n");
99.
100.        trim_trailing_newline(filename);
101.        if (string_length(filename) == 0) fail("error: filename must not be
    empty\n");
102.
103.        int parent_to_child[2];
104.        if (pipe(parent_to_child) == -1) fail("error: failed to create pipe\n");
105.
106.        int child_to_parent[2];
107.        if (pipe(child_to_parent) == -1) fail("error: failed to create pipe\n");
108.
109.        pid_t child = fork();
110.        if (child == -1) fail("error: failed to fork\n");
111.
112.        if (child == 0) {
113.              if (close(parent_to_child[1]) == -1 || close(child_to_parent[0]) ==
    -1) fail("error: failed to close descriptors in child\n");
114.
115.              if (dup2(parent_to_child[0], STDIN_FILENO) == -1) fail("error:
    failed to redirect stdin\n");
116.
117.              if (dup2(child_to_parent[1], STDOUT_FILENO) == -1) fail("error:
    failed to redirect stdout\n");
118.
119.              if (close(parent_to_child[0]) == -1 || close(child_to_parent[1]) ==
    -1) fail("error: failed to close redundant descriptors\n");
120.
121.              char child_path[PATH_MAX];
122.              build_child_path(child_path, sizeof(child_path));
123.
124.              char *const args[] = {CHILD_PROGRAM_NAME, filename, NULL};
125.              execv(child_path, args);
126.              fail("error: exec failed\n");
127.        }
128.
129.        if (close(parent_to_child[0]) == -1 || close(child_to_parent[1]) == -1)
    fail("error: failed to close descriptors in parent\n");
130.
131.        char line_buffer[MAX_LINE_LENGTH];
132.        while(true) {
133.              ssize_t line_length = read_line(STDIN_FILENO, line_buffer,
    sizeof(line_buffer));
134.              if (line_length < 0) fail("error: failed to read input line\n");
135.
136.              if (line_length == 0) break;
137.
138.              if (line_buffer[0] == '\n') break;
139.
140.              write_all(parent_to_child[1], line_buffer, (size_t)line_length);
141.
```

```
142.                char response[MAX_LINE_LENGTH];
143.                ssize_t response_length = read_line(child_to_parent[0], response,
   sizeof(response));
144.                if (response_length <= 0) fail("error: child response failed\n");
145.                forward_line(STDOUT_FILENO, response);
146.        }
147.
148.        if (close(parent_to_child[1]) == -1 || close(child_to_parent[0]) == -1)
   fail("error: failed to close pipes\n");
149.
150.        int status = 0;
151.        if (waitpid(child, &status, 0) == -1) {
152.                fail("error: waitpid failed\n");
153.        }
154.        return WIFEXITED(status) ? WEXITSTATUS(status) : EXIT_FAILURE;
   155.     }
```

child.c

```
1. #include <errno.h>
2. #include <fcntl.h>
3. #include <limits.h>
4. #include <stdbool.h>
5. #include <stdint.h>
6. #include <stdlib.h>
7. #include <string.h>
8. #include <sys/wait.h>
9. #include <unistd.h>
10.
11.   #define CHILD_PROGRAM_NAME "child.c"
12.   #define MAX_LINE_LENGTH 4096
13.
14.   static size_t string_length(const char *text) {
15.         size_t length = 0;
16.         while (text[length] != '\0') ++length;
17.         return length;
18.   }
19.
20.   static void write_all(int fd, const char *buffer, size_t length) {
21.         while (length > 0) {
22.               ssize_t written = write(fd, buffer, length);
23.               if (written < 0) _exit(EXIT_FAILURE);
24.               buffer += (size_t)written;
25.               length -= (size_t)written;
26.         }
27.   }
28.
29.   static void fail(const char *message) {
30.         write_all(STDERR_FILENO, message, string_length(message));
31.         _exit(EXIT_FAILURE);
32.   }
33.
34.   static ssize_t read_line(int fd, char *buffer, size_t capacity) {
35.         if (capacity == 0) return -1;
36.
37.         size_t offset = 0;
38.         while (offset + 1 < capacity) {
39.               char ch;
40.               ssize_t bytes = read(fd, &ch, 1);
41.               if (bytes < 0) {
42.                     return -1;
43.               }
44.               if (bytes == 0) break;
45.
46.               buffer[offset++] = ch;
47.               if (ch == '\n') break;
48.         }
49.         buffer[offset] = '\0';
50.         return (ssize_t)offset;
51.   }
52.
53.   static void trim_trailing_newline(char *line) {
54.         size_t length = string_length(line);
55.         if (length == 0) {
56.               return;
```

```
57.         }
58.         if (line[length - 1] == '\n') line[length - 1] = '\0';
59.   }
60.
61.   static void build_child_path(char *result, size_t capacity) {
62.         char executable_path[PATH_MAX];
63.         ssize_t len = readlink("/proc/self/exe", executable_path,
   sizeof(executable_path) - 1);
64.         if (len == -1) fail("error: failed to read /proc/self/exe\n");
65.
66.         executable_path[len] = '\0';
67.
68.         while (len > 0 && executable_path[len] != '/') --len;
69.         if (len == 0) fail("error: executable path is invalid\n");
70.
71.         executable_path[len] = '\0';
72.
73.         size_t dir_length = string_length(executable_path);
74.         size_t name_length = string_length(CHILD_PROGRAM_NAME);
75.         if (dir_length + 1 + name_length + 1 > capacity) fail("error: child path
   buffer is too small\n");
76.
77.         size_t index = 0;
78.         for (size_t i = 0; i < dir_length; ++i) result[index++] =
   executable_path[i];
79.         result[index++] = '/';
80.         for (size_t i = 0; i < name_length; ++i) result[index++] =
   CHILD_PROGRAM_NAME[i];
81.
82.         result[index] = '\0';
83.   }
84.
85.   static void forward_line(int output_fd, const char *line) {
86.         size_t length = string_length(line);
87.         if (length == 0 || line[length - 1] != '\n') {
88.               write_all(output_fd, line, length);
89.               write_all(output_fd, "\n", 1);
90.               return;
91.         }
92.         write_all(output_fd, line, length);
93.   }
94.
95.   int main(void) {
96.         char filename[MAX_LINE_LENGTH];
97.         ssize_t filename_len = read_line(STDIN_FILENO, filename,
   sizeof(filename));
98.         if (filename_len <= 0) fail("error: failed to read filename\n");
99.
100.        trim_trailing_newline(filename);
101.        if (string_length(filename) == 0) fail("error: filename must not be
   empty\n");
102.
103.        int parent_to_child[2];
104.        if (pipe(parent_to_child) == -1) fail("error: failed to create pipe\n");
105.
106.        int child_to_parent[2];
107.        if (pipe(child_to_parent) == -1) fail("error: failed to create pipe\n");
108.
109.        pid_t child = fork();
110.        if (child == -1) fail("error: failed to fork\n");
111.
112.        if (child == 0) {
113.              if (close(parent_to_child[1]) == -1 || close(child_to_parent[0]) ==
   -1) fail("error: failed to close descriptors in child\n");
114.
115.              if (dup2(parent_to_child[0], STDIN_FILENO) == -1) fail("error:
   failed to redirect stdin\n");
116.
117.              if (dup2(child_to_parent[1], STDOUT_FILENO) == -1) fail("error:
   failed to redirect stdout\n");
118.
119.              if (close(parent_to_child[0]) == -1 || close(child_to_parent[1]) ==
   -1) fail("error: failed to close redundant descriptors\n");
120.
121.              char child_path[PATH_MAX];
122.              build_child_path(child_path, sizeof(child_path));
```

```
123.
124.                 char *const args[] = {CHILD_PROGRAM_NAME, filename, NULL};
125.                 execv(child_path, args);
126.                 fail("error: exec failed\n");
127.         }
128.
129.         if (close(parent_to_child[0]) == -1 || close(child_to_parent[1]) == -1)
     fail("error: failed to close descriptors in parent\n");
130.
131.         char line_buffer[MAX_LINE_LENGTH];
132.         while(true) {
133.                 ssize_t line_length = read_line(STDIN_FILENO, line_buffer,
     sizeof(line_buffer));
134.                 if (line_length < 0) fail("error: failed to read input line\n");
135.
136.                 if (line_length == 0) break;
137.
138.                 if (line_buffer[0] == '\n') break;
139.
140.                 write_all(parent_to_child[1], line_buffer, (size_t)line_length);
141.
142.                 char response[MAX_LINE_LENGTH];
143.                 ssize_t response_length = read_line(child_to_parent[0], response,
     sizeof(response));
144.                 if (response_length <= 0) fail("error: child response failed\n");
145.                 forward_line(STDOUT_FILENO, response);
146.         }
147.
148.         if (close(parent_to_child[1]) == -1 || close(child_to_parent[0]) == -1)
     fail("error: failed to close pipes\n");
149.
150.         int status = 0;
151.         if (waitpid(child, &status, 0) == -1) {
152.                 fail("error: waitpid failed\n");
153.         }
154.         return WIFEXITED(status) ? WEXITSTATUS(status) : EXIT_FAILURE;
155.  }
```

Протокол работы программы

Тестирование:

Программа была протестирована на различных входных данных. Примеры тестов:

Тест 1: Простые числа

```
1. $ ./build/lab_01_parent
2. results.txt
3. 1.5 2.5 3.0
4. sum: 7.0
5. 10.25 20.75
6. sum: 31.0
7. -5.5 10.5
8. sum: 5.0
9.
10. $ cat results.txt
11. sum: 7.0
12. sum: 31.0
13. sum: 5.0
14.
```

Тест 2: Множество чисел в одной строке

1. $ ./build/lab_01_parent
2. output.txt
3. 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10.0
4. sum: 59.5
5. 0.1 0.2 0.3 0.4 0.5

6. sum: 1.5
7.
8. $ cat output.txt
9. sum: 59.5
10. sum: 1.5

Тест 3: Обработка ошибок

1. $ ./build/lab_01_parent
2. test.txt
3. 1.5 2.5 abc
4. error: invalid input
5. 3.0 4.0
6. sum: 7.0
7. invalid
8. error: invalid input
9.
10. $ cat test.txt
11. sum: 7.0

Тест 4: Отрицательные числа и нули

1. $ ./build/lab_01_parent
2. negative.txt
3. -10.5 5.5
4. sum: -5.0
5. 0.0 0.0 0.0
6. sum: 0.0
7. -1.1 -2.2 -3.3
8. sum: -6.6
9.
10. $ cat negative.txt
11. sum: -5.0
12. sum: 0.0
13. sum: -6.6

Strace:

Для анализа системных вызовов использовалась утилита `strace` с флагом `-f` для отслеживания дочерних процессов. Программа была запущена через WSL:

$ strace -f -o strace_output.txt ./build/lab_01_parent

Ниже представлен фрагмент вывода `strace` с выделенными системными вызовами, используемыми в нашей программе. Написанные ниже строки соответствуют системным вызовам, вызванным непосредственно из нашего кода:

```
320   execve("./build/lab_01_parent", ["./build/lab_01_parent"], 0x7ffd7a7c80d8 /* 22 vars
*/) = 0

...

320   read(0, "t", 1)                   = 1

320   read(0, "e", 1)                   = 1

...

320   read(0, "\n", 1)                  = 1

320   **pipe2([3, 4], 0)                  = 0**  // parent_to_child (pipe)

320   **pipe2([5, 6], 0)                  = 0**  // child_to_parent (pipe)

320   **clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7e20786a6a10) = 321**  // fork

strace: Process 321 attached

...

320   close(3 <unfinished ...>

321   **close(4 <unfinished ...>**  // закрытие write конца parent_to_child в child

320   <... close resumed>)              = 0

321   <... close resumed>)              = 0

320   close(6 <unfinished ...>

321   **close(5 <unfinished ...>**  // закрытие read конца child_to_parent в child

320   <... close resumed>)              = 0

321   <... close resumed>)              = 0

...

321   **dup2(3, 0 <unfinished ...>**  // перенаправление stdin на parent_to_child[0]

321   <... dup2 resumed>)               = 0

...

321   **dup2(6, 1 <unfinished ...>**  // перенаправление stdout на child_to_parent[1]

321   <... dup2 resumed>)               = 1

...

321   **close(3 <unfinished ...>**  // закрытие после dup2

321   <... close resumed>)              = 0
```

```
321   **close(6 <unfinished ...>**  // закрытие после dup2

321   <... close resumed>)              = 0

321   **readlink("/proc/self/exe",  <unfinished ...>**  // получение пути к исполняемому
файлу

321   <... readlink resumed>"/mnt/c/MAI/3_sem/OS/OS_LAB_1/bui"..., 4095) = 48

321   **execve("/mnt/c/MAI/3_sem/OS/OS_LAB_1/build/lab_01_child", ["lab_01_child",
"test_output.txt"], 0x7ffc61753238 /* 22 vars */ <unfinished ...>**  // execv

...

321   <... execve resumed>)             = 0

...

321   **openat(AT_FDCWD, "test_output.txt", O_WRONLY|O_CREAT|O_TRUNC, 0600) = 3**  // open

321   **read(0, "1", 1)                   = 1**  // чтение из stdin (pipe)

321   read(0, ".", 1)                   = 1

321   read(0, "5", 1)                   = 1

...

321   read(0, "\n", 1)                  = 1

321   **write(3, "sum: ", 5)              = 5**  // запись в файл

321   **write(3, "7.0", 3)                = 3**  // запись в файл

321   **write(3, "\n", 1)                 = 1**  // запись в файл

321   **write(1, "sum: ", 5 <unfinished ...>**  // запись в stdout (pipe к родителю)

321   <... write resumed>)              = 5

321   **write(1, "7.0", 3 <unfinished ...>**  // запись в stdout (pipe к родителю)

321   <... write resumed>)              = 3

321   **write(1, "\n", 1 <unfinished ...>**  // запись в stdout (pipe к родителю)

321   <... write resumed>)              = 1

...

320   **read(5,  <unfinished ...>**  // родитель читает ответ от child

320   <... read resumed>"s", 1)         = 1

320   read(5, "u", 1)                   = 1

...

320   read(5, "\n", 1)                  = 1

320   **write(1, "sum: 7.0\n", 9)         = 9**  // родитель выводит в stdout

...

321   **read(0,  <unfinished ...>**  // конец ввода

321   <... read resumed>"", 1)          = 0

320   **close(4)                          = 0**  // закрытие write конца parent_to_child в
parent
```

```
320   close(5 <unfinished ...>
321   **close(3 <unfinished ...>**  // закрытие файла
320   <... close resumed>)              = 0
321   <... close resumed>)              = 0
320   **wait4(321,  <unfinished ...>**  // waitpid
...
321   **exit_group(0)                     = ?**  // _exit
321   +++ exited with 0 +++
320   <... wait4 resumed>[{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 321
...
320   exit_group(0)                     = ?
320   +++ exited with 0 +++
```

Ключевые системные вызовы из нашего кода:

1. **readlink("/proc/self/exe", ...)** (строка 82) – получение пути к исполняемому файлу родительского процесса для построения пути к дочернему процессу

2. **pipe2([3, 4], 0)** и **pipe2([5, 6], 0)** (строки 50-51) – создание двух каналов для двусторонней связи между процессами (pipe)

3. **clone(...)** (строка 52) – создание дочернего процесса (fork)

4. **close()** (строки 53, 57, 58, 62, 74, 78, 103, 117, 215, 217, 218) – закрытие ненужных файловых дескрипторов

5. **dup2()** (строки 66, 70) – перенаправление стандартных потоков ввода/вывода на pipe

6. **execve()** (строка 86) – замена образа процесса на дочернюю программу (execv)

7. **openat()** (строка 128) – открытие файла для записи результатов (open)

8. **read()** (строки 34-49, 61, 65, 69, 73, 77, 81, 85, 88-92, 94, 129-140, 147, 151, 155, 156, 158-162, 164-175, 178, 179-189, 196, 200, 204, 205, 207-212, 214) – чтение данных из stdin/pipe

9. **write()** (строки 93, 141-143, 144, 148, 152, 163, 176, 190-192, 193, 197, 201, 213) – запись данных в файл и в stdout/pipe

10. **wait4()** (строка 220) – ожидание завершения дочернего процесса (waitpid)

11. **exit_group()** (строки 222, 226) – завершение процесса (_exit)

Все системные вызовы проверяются на ошибки: возвращаемые значения сравниваются с -1 (или 0 для некоторых вызовов), и при ошибке программа выводит сообщение и завершается.

Вывод

В ходе выполнения лабораторной работы была реализована программа для межпроцессного взаимодействия между родительским и дочерним процессами с использованием неименованных каналов (pipe). Программа успешно выполняет поставленную задачу: родительский процесс

передает строки с числами дочернему процессу, который вычисляет их сумму и записывает результаты в файл.

Основные достижения: реализовано корректное использование системных вызовов fork(), pipe(), dup2(), execv(), read(), write(), open(), close(), waitpid() и других для организации межпроцессного взаимодействия. Все системные вызовы проверяются на ошибки, что обеспечивает надежность работы программы. Программа корректно обрабатывает различные входные данные, включая отрицательные числа, нули и некорректный ввод.

При выполнении работы были изучены механизмы создания процессов, перенаправления потоков ввода/вывода и организации двусторонней связи между процессами через неименованные каналы. Программа успешно прошла тестирование и готова к демонстрации.