1. МИНОБРНАУКИ РОССИИ
2. САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
3. ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
4. «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
5. Кафедра Вычислительной техники

ОТЧЁТ

по лабораторной работе №4

по дисциплине «Организация процессов и программирования в среде Linux»

1. Тема: УПРАВЛЕНИЕ ПОТОКАМИ

|  |  |  |
| --- | --- | --- |
| Студент гр. 1335 |  | Максимов Ю Е |
| Преподаватель |  | Разумовский Г.В. |

Оглавление

[1. Введение 3](#__RefHeading___Toc336_311249038)

[1.1. Введение 3](#__RefHeading___Toc15338_3795591338)

[1.2. Порядок выполнения работы 3](#__RefHeading___Toc15340_3795591338)

[1.3. Содержание отчёта 4](#__RefHeading___Toc15342_3795591338)

[2. Тексты программ, распечатка входных и выходных файлов 5](#__RefHeading___Toc4255_890079619)

[2.1. main.cpp (часть 1) 5](#__RefHeading___Toc4257_890079619)

[2.2. main.cpp (часть 2) 10](#__RefHeading___Toc4259_890079619)

[2.3. Входной файл (часть 1) 14](#__RefHeading___Toc4261_890079619)

[2.4. Входной файл (часть 2) 16](#__RefHeading___Toc4263_890079619)

[2.5. Выходные файлы (часть 2) 18](#__RefHeading___Toc4265_890079619)

[3. Скриншот экрана вывода файла для первой программы 21](#__RefHeading___Toc4267_890079619)

[3.1. Компиляция 21](#__RefHeading___Toc4269_890079619)

[3.2. Часть 1 23](#__RefHeading___Toc4271_890079619)

[3.3. Часть 2 26](#__RefHeading___Toc4273_890079619)

[3. Вывод 27](#__RefHeading___Toc358_311249038)

[4. Список использованных источников 28](#__RefHeading___Toc360_311249038)

# 1. Введение

## 1.1. Введение

Тема работы: Управление потоками.

Цель работы: Знакомство с организацией потоков и способами синхронизации предков и потомков.

## 1.2. Порядок выполнения работы

1. Написать программу, которая открывает текстовый файл, порождает поток, а затем ожидает его завершения. Потоку в качестве параметра передаётся дескриптор файла. Поток выводит на экран класс планирования, текущий, минимальный и максимальный приоритеты, содержимое файла и закрывает файл. После завершения работы потока программа должна вывести текущий приоритет и проверить – закрыт ли файл, и если он не закрыт, то принудительно закрыть. Результат проверки должен быть выведен на экран.

2. Дважды откомпилировать программу при условии, когда поток закрывает и не закрывает файл. Затем последовательно запустить оба варианта.

3. Написать программу, которая открывает входной файл и 2 выходных файла. Затем она должна в цикле построчно читать входной файл и порождать 2 потока. Одному потоку передавать нечётную строку, а другому – чётную. Оба потока должны работать параллельно. Каждый поток записывает в свой выходной файл полученную строку и завершает работу. Программа должна ожидать завершения работы каждого потока и повторять цикл порождения потоков и чтения строк входного файла, пока не прочтёт последнюю строку, после чего закрыть все файлы.

4. Откомпилировать программу и запустить её.

## 1.3. Содержание отчёта

Отчёт по лабораторной работе должен содержать:

1. Цель и задания.

2. Тексты программ, распечатку входных и выходных файлов.

3. Скриншот экрана вывода файла для первой программы.

# 2. Тексты программ, распечатка входных и выходных файлов

## 2.1. main.cpp (часть 1)

// !!! compile w/ flag "-pthread" !!!

// i.e. "g++ -Wall -pthread -o "%e" "%f""

#include <iostream>

#include <fstream>

#include <string>

#include <pthread.h>

#include <unistd.h>

#include <fcntl.h>

using namespace std;

void \*ChildThread(void \*arg);

int main()

{

int handle\_check; // handle check

int handle; // handle

int schedualing\_policy; // schedualing policy field MAIN THREAD

struct sched\_param schedualing\_parameters; // struct for schedual priority MAIN THREAD

pthread\_t thread;

pthread\_attr\_t thread\_attributes;

cout << "---------- MAIN THREAD BEGIN ----------\n";

// ---------- OPENING TEXT FILE ----------

handle = open("/home/matmanbj/lorem\_ipsum.txt", O\_RDONLY);

if (handle == -1)

{

cout << "File's handle HAS NOT BEEN opened by PARENT thread!\n";

return -1; // end program with error

}

else

{

cout << "File's handle HAS BEEN opened by PARENT thread! Handle: " << handle << "\n";

}

// ---------- CREATING A THREAD (SENDING HANDLE AS A PARAMETER) ----------

pthread\_attr\_init(&thread\_attributes);

pthread\_create(&thread, &thread\_attributes, ChildThread, &handle);

pthread\_join(thread, nullptr);

// ---------- CURRENT PRIORITY OUTPUT ----------

cout << "---------- MAIN THREAD INFO ----------\n";

pthread\_getschedparam(pthread\_self(), &schedualing\_policy, &schedualing\_parameters); // getting parameters about MAIN thread

cout << "Current priority: " << schedualing\_parameters.sched\_priority << "\n"; // current priority

// ---------- CHECKING ON OPEN/CLOSED FILE (IF OPENED -- FORCIBLY CLOSE) ----------

handle\_check = fcntl(handle, F\_GETFD); // checking handle

if (handle\_check != -1)

{

cout << "File's handle HAS NOT BEEN closed by CHILD thread!\n";

close (handle); // forced closing the file (handle)

handle\_check = fcntl(handle, F\_GETFD); // checking handle again

if (handle\_check != -1)

{

cout << "File's handle HAS NOT BEEN closed by PARENT thread!\n";

return -2; // end program with error

}

else

{

cout << "File's handle HAS BEEN closed by PARENT thread!\n";

}

}

else

{

cout << "File's handle HAS BEEN closed by CHILD thread!\n";

}

cout << "---------- MAIN THREAD END ----------\n";

pthread\_attr\_destroy(&thread\_attributes); // cleaning memory w/ destroying thread attributes no loner required

return 0;

}

// ---------- ChildThread function ----------

void \*ChildThread(void \*arg)

{

int local\_handle\_check; // handle check local

int local\_schedualing\_policy; // schedualing policy field

int local\_close = 0; // 0 -- child thread DOES NOT CLOSES the handle, 1 -- chiled thread CLOSES the handle, other -- DOES NOT CLOSE by default

int local\_buffer\_counter = -1; // number of bytes read (-1 is for begin, 0 is for ending loop)

int local\_handle = \*((int\*)arg); // handle of opened file

struct sched\_param local\_schedualing\_parameters; // struct for schedual priority

cout << "---------- CHILD THREAD BEGIN ----------\n";

cout << "Close the handle by child process (0 -- don't close, 1 -- close, other -- don't close)?\n"; // close handle by child thread or not

cin >> local\_close; // user's choice

pthread\_getschedparam(pthread\_self(), &local\_schedualing\_policy, &local\_schedualing\_parameters); // getting parameters about CHILD thread

// ---------- SCHEDUALING POLICY, CURRENT, MIN & MAX PRIORITY, FILE OUTPUT ----------

cout << "---------- CHILD THREAD INFO ----------\n"

<< "Schedualing policy: " // schedualing policy

<< (local\_schedualing\_policy == SCHED\_FIFO ? to\_string(local\_schedualing\_policy) + " -- SCHED\_FIFO" : "")

<< (local\_schedualing\_policy == SCHED\_RR ? to\_string(local\_schedualing\_policy) + " -- SCHED\_RR" : "")

<< (local\_schedualing\_policy == SCHED\_OTHER ? to\_string(local\_schedualing\_policy) + " -- SCHED\_OTHER" : "") << "\n"

<< "Current priority: " << local\_schedualing\_parameters.sched\_priority << "\n" // current priority

<< "Minimal priority: " << sched\_get\_priority\_min(local\_schedualing\_policy) << "\n" // minimal priority

<< "Maximal priority: " << sched\_get\_priority\_max(local\_schedualing\_policy) << "\n" // maximal priority

<< "File:\n" // file output

<< "---------- BEGIN OF FILE ----------\n";

while(local\_buffer\_counter != 0) // 0 means no bytes to read

{

char local\_buffer[80];

size\_t n = sizeof(local\_buffer);

local\_buffer\_counter = read(local\_handle, &local\_buffer, n);

local\_buffer[local\_buffer\_counter] = '\0';

cout << local\_buffer;

}

cout << "\n";

cout << "---------- END OF FILE ----------\n";

if (local\_close == 0) // handle close chosen actions

{

cout << "File's handle SHOULD NOT BE closed by CHILD thread!\n";

}

else if (local\_close == 1)

{

cout << "File's handle SHOULD BE closed by CHILD thread!\n";

close(local\_handle);

}

else

{

cout << "File's handle SHOULD NOT BE closed by CHILD thread by default!\n";

}

local\_handle\_check = fcntl(local\_handle, F\_GETFD);

if (local\_handle\_check == -1) // handle close try check

{

cout << "File's handle HAS BEEN closed by CHILD thread!\n";

}

else

{

cout << "File's handle HAS NOT BEEN closed by CHILD thread!\n";

}

cout << "---------- CHILD THREAD END ----------\n";

pthread\_exit(NULL); // terminating calling thread, i.e. this CHILD process will be terminated

}

## 2.2. main.cpp (часть 2)

// !!! compile w/ flag "-pthread" !!!

// i.e. "g++ -Wall -pthread -o "%e" "%f""

#include <iostream>

#include <fstream>

#include <string>

#include <pthread.h>

#include <unistd.h>

#include <fcntl.h>

#include <errno.h>

using namespace std;

typedef struct // struct to give arguments (string & write file adress) to function

{

string local\_string;

ofstream \*write\_file;

} funcArg;

void isFileOpenMyFunc (ifstream \*local\_file, const string local\_file\_name);

void isFileOpenMyFunc (ofstream \*local\_file, const string local\_file\_name);

void\* threadFunction (void\* main\_argument);

// ---------- MAIN ----------

int main(int argc, char \*argv[])

{

pthread\_t thread\_1; // 1st thread to write odd strings (1, 3, 5...)

pthread\_t thread\_2; // 2nd thread to write even strings (2, 4, 6...)

bool is\_file\_end = false; // file read end indicator

funcArg main\_arguments\_1;

funcArg main\_arguments\_2;

cout << "---------- MAIN THREAD BEGIN ----------\n";

cout << "---------- FILE OPEN & CHECK ----------\n";

ifstream input\_file("/home/matmanbj/lorem\_ipsum\_2.txt"); // open file to read

ofstream output\_file\_1("output\_file\_1.txt"); // open file to write 1

ofstream output\_file\_2("output\_file\_2.txt"); // open file to write 2

main\_arguments\_1.write\_file = &output\_file\_1; // for struct

main\_arguments\_2.write\_file = &output\_file\_2; // for struct

isFileOpenMyFunc (&input\_file, "/home/matmanbj/lorem\_ipsum\_2.txt"); // check opening file to read

isFileOpenMyFunc (&output\_file\_1, "output\_file\_1.txt"); // check opening file to write 1

isFileOpenMyFunc (&output\_file\_2, "output\_file\_2.txt"); // check opening file to write 2

cout << "---------- READ & WRITE BEGINS ----------\n";

while (is\_file\_end == false) // while file's end didn't reached, do R/W

{

if(getline(input\_file, main\_arguments\_1.local\_string)) // if we can read 2n+1\_th string, then put characters into string

{

pthread\_create(&thread\_1, NULL, threadFunction, (void\*)&main\_arguments\_1); // create thread to write this string to file

}

else

{

is\_file\_end = true; // else indicate the loop to stop

}

if(getline(input\_file, main\_arguments\_2.local\_string)) // if we can read 2n\_th string, then put characters into string

{

pthread\_create(&thread\_2, NULL, threadFunction, (void\*)&main\_arguments\_2); // create thread to write this string to file

}

else

{

is\_file\_end = true; // else indicate the loop to stop

}

pthread\_join(thread\_1,NULL); // waiting for parallel threads termination

pthread\_join(thread\_2,NULL); // waiting for parallel threads termination

}

cout << "---------- READ & WRITE ENDS ----------\n";

// close all opened files

output\_file\_1.close();

output\_file\_2.close();

input\_file.close();

cout << "---------- MAIN THREAD END ----------\n";

return 0;

}

// ---------- isFileOpen function ----------

void isFileOpenMyFunc (ifstream \*local\_file, const string local\_file\_name) // check if the file open

{

if ((\*local\_file).is\_open()) //if (\*local\_file)

{

cout << "File to READ \"" << local\_file\_name << "\" HAS BEEN opened by PARENT thread!\n";

}

else // if the file isn't opened, throw an error

{

cout << "File to READ \"" << local\_file\_name << "\" HAS NOT BEEN opened by PARENT thread!\n"

<< "The program terminates w/ error!\n";

puts(("File to READ \"" + local\_file\_name + "\" HAS NOT BEEN opened by PARENT thread. Continuation is impossible!").c\_str());

}

}

void isFileOpenMyFunc (ofstream \*local\_file, const string local\_file\_name) // check if the file open

{

if ((\*local\_file).is\_open()) //if (\*local\_file)

{

cout << "File to WRITE \"" << local\_file\_name << "\" HAS BEEN opened by PARENT thread!\n";

}

else // if the file isn't opened, throw an error

{

cout << "File to WRITE \"" << local\_file\_name << "\" HAS NOT BEEN opened by PARENT thread!\n"

<< "The program terminates w/ error!\n";

puts(("File to WRITE \"" + local\_file\_name + "\" HAS NOT BEEN opened by PARENT thread. Continuation is impossible!").c\_str());

}

}

// ---------- threadFunction function ----------

void\* threadFunction(void\* main\_argument) // write to file

{

funcArg\* local\_argument = (funcArg\*) main\_argument;

\*(local\_argument->write\_file) << local\_argument->local\_string << "\n";

pthread\_exit(NULL);

}

## 2.3. Входной файл (часть 1)

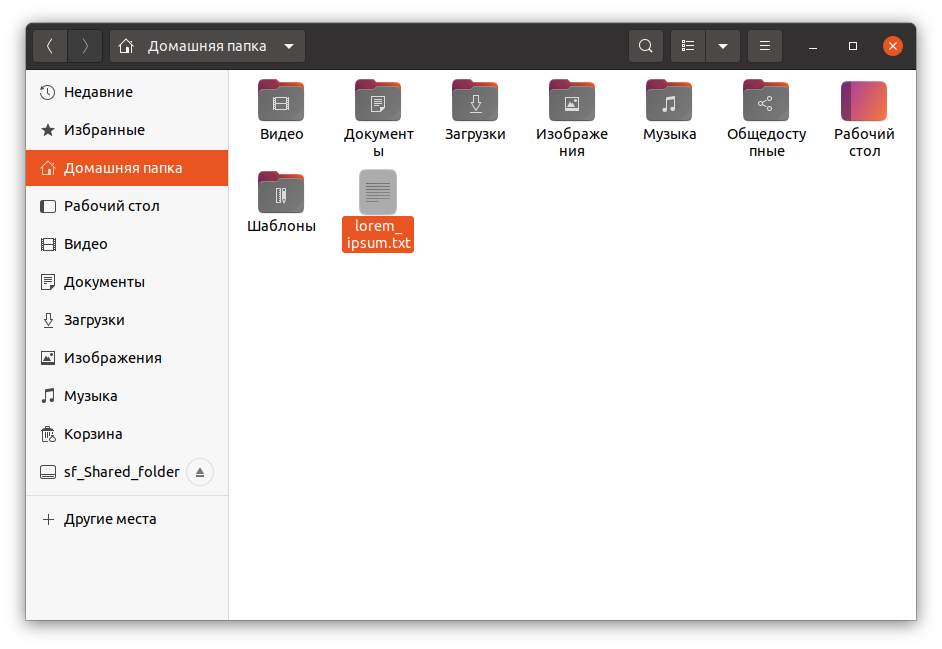


Рисунок 1. Входной файл для первой программы



Рисунок 2. Входной файл для первой программы

## 2.4. Входной файл (часть 2)

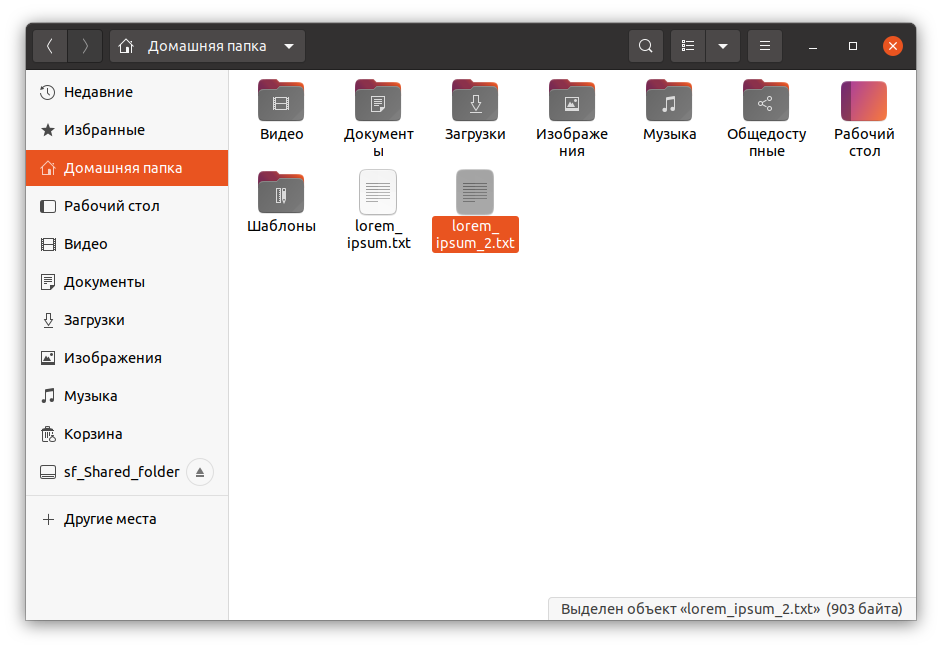


Рисунок 3. Входной файл для второй программы

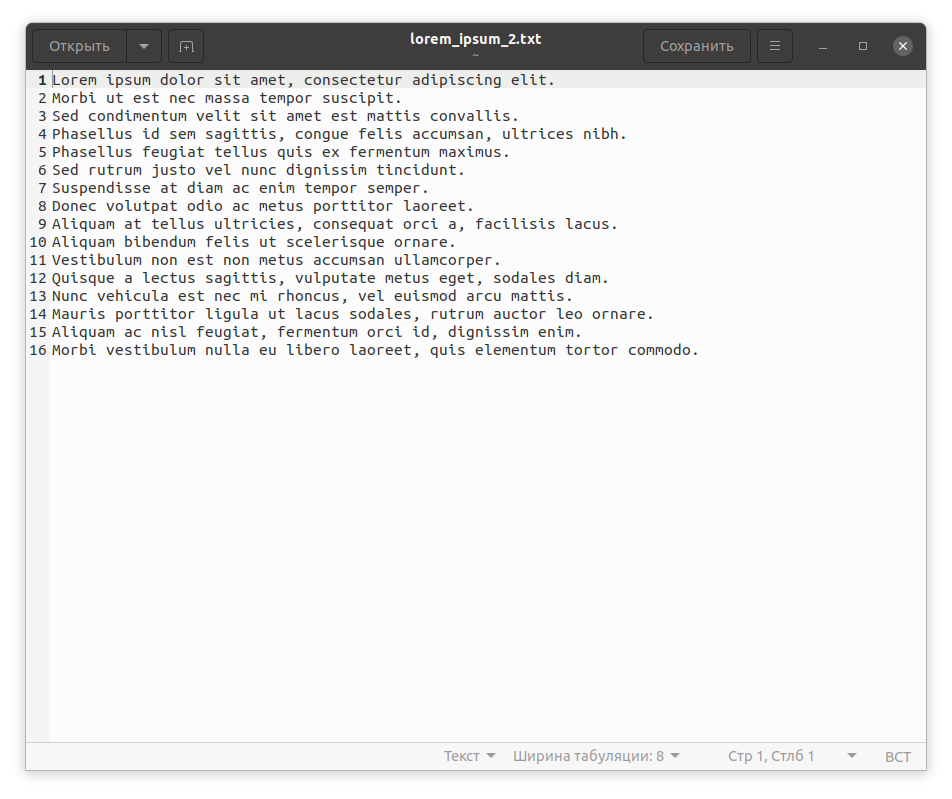


Рисунок 4. Входной файл для второй программы

## 2.5. Выходные файлы (часть 2)

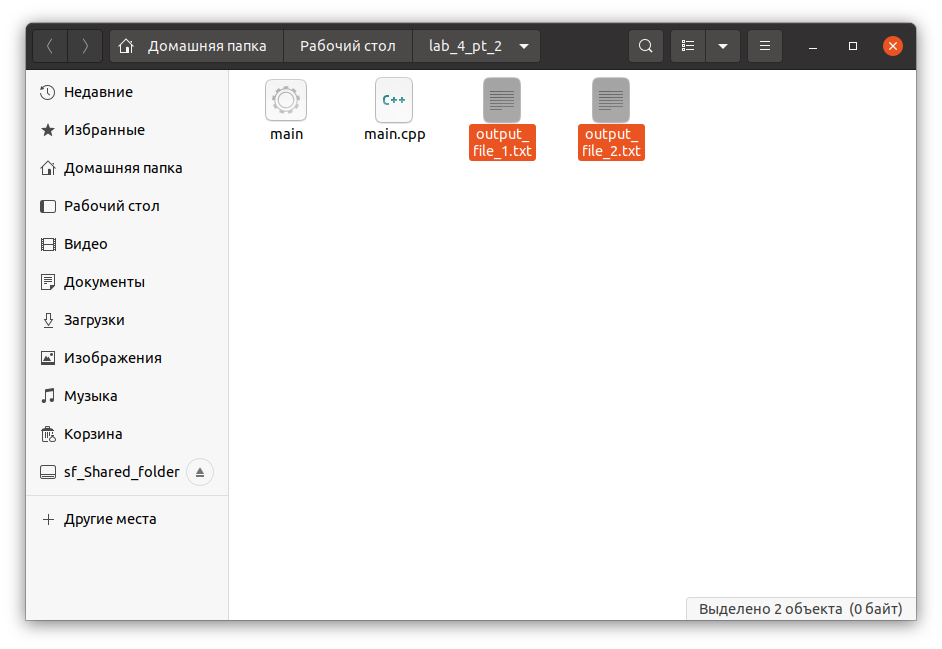


Рисунок 5. Выходные файлы для второй программы

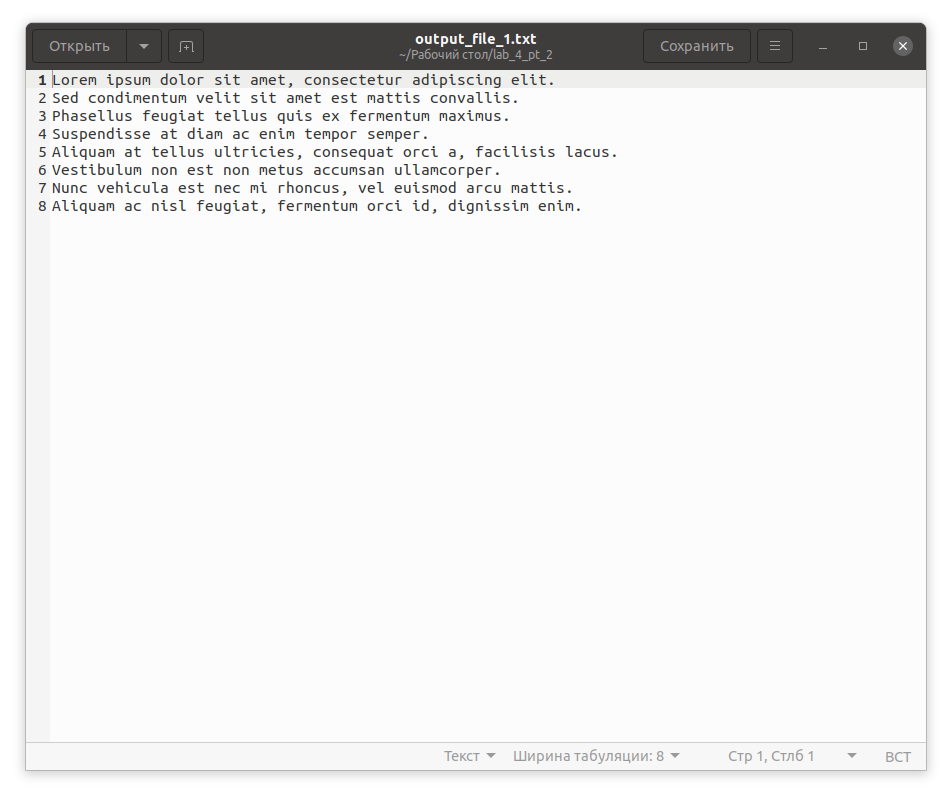


Рисунок 6. Первый выходной файл для второй программы

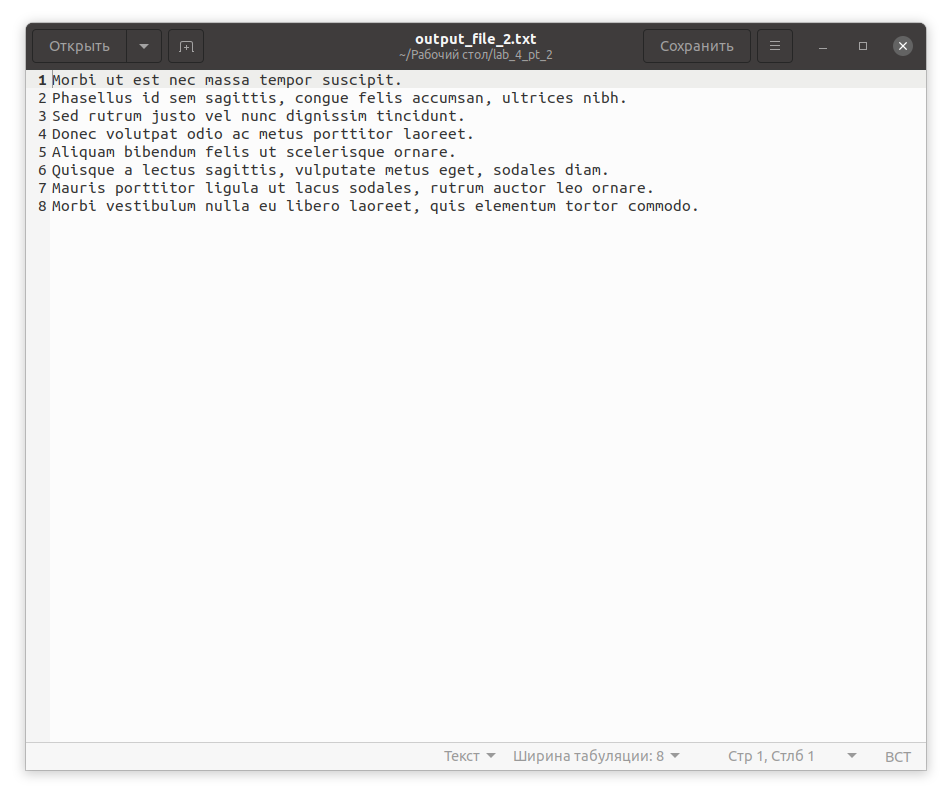


Рисунок 7. Второй выходной файл для второй программы

# 3. Скриншот экрана вывода файла для первой программы

## 3.1. Компиляция

Для запуска программы необходимо выставить специальный флаг «-pthread», который отвечает за сборку программы с функциями процессов. Иначе – будет ошибка сборки.

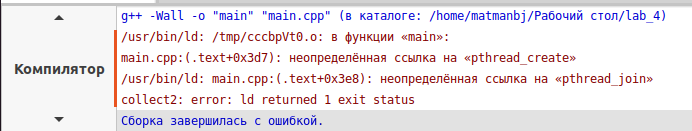


Рисунок 8. Сборка с ошибкой при отсутствии флага «-pthread»

В IDE Geany были выставлены следующие флаги, для компиляции: «g++ -Wall -o "%e" "%f"»; для сборки: «g++ -Wall -pthread -o "%e" "%f"».

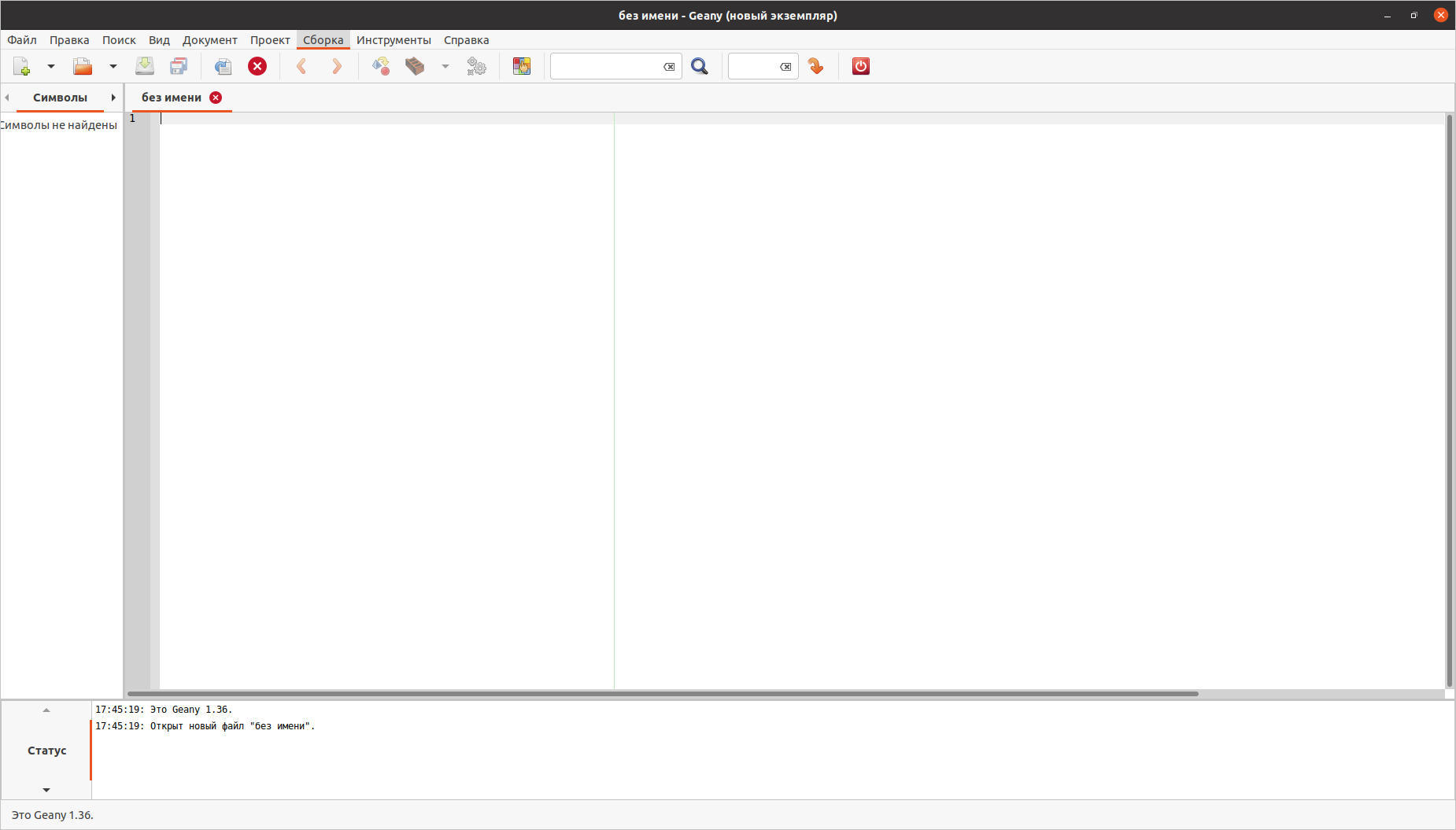


Рисунок 9. Установка флага «-pthread» в IDE «Geany»

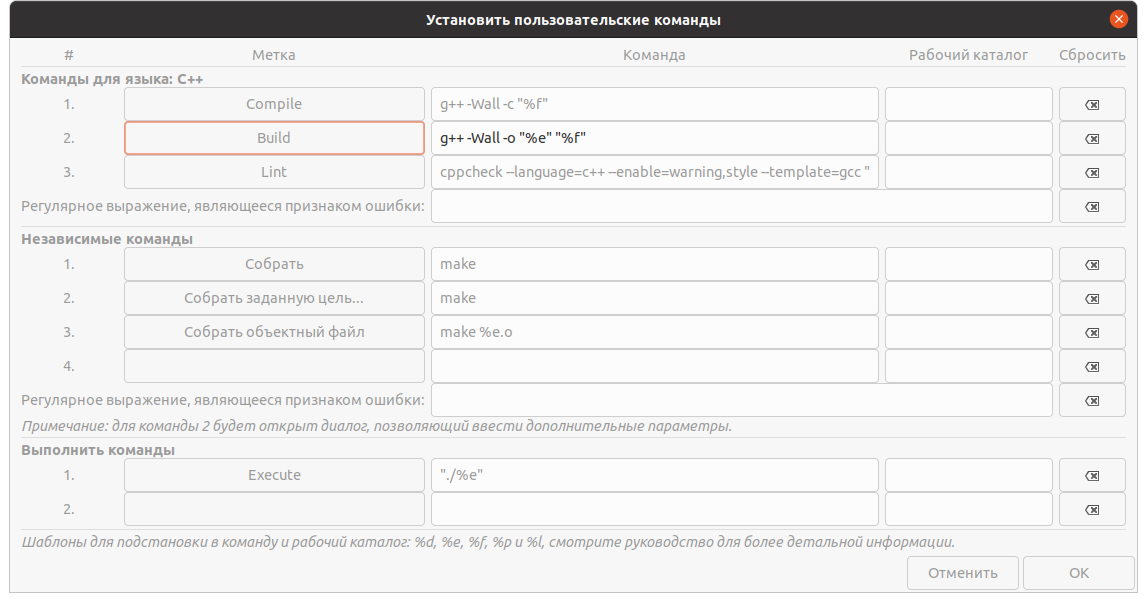


Рисунок 10. Установка флага «-pthread» в IDE «Geany»

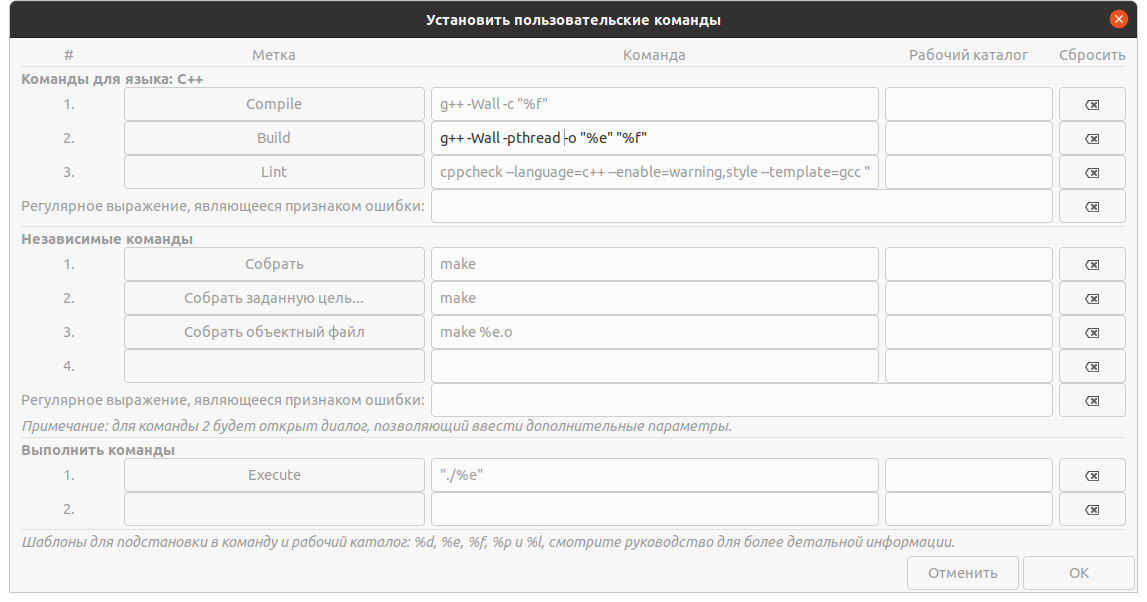


Рисунок 11. Установка флага «-pthread» в IDE «Geany»

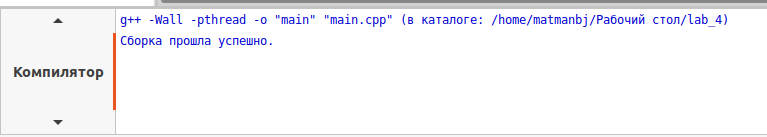


Рисунок 12. Сборка без ошибки при наличии флага «-pthread»

## 3.2. Часть 1

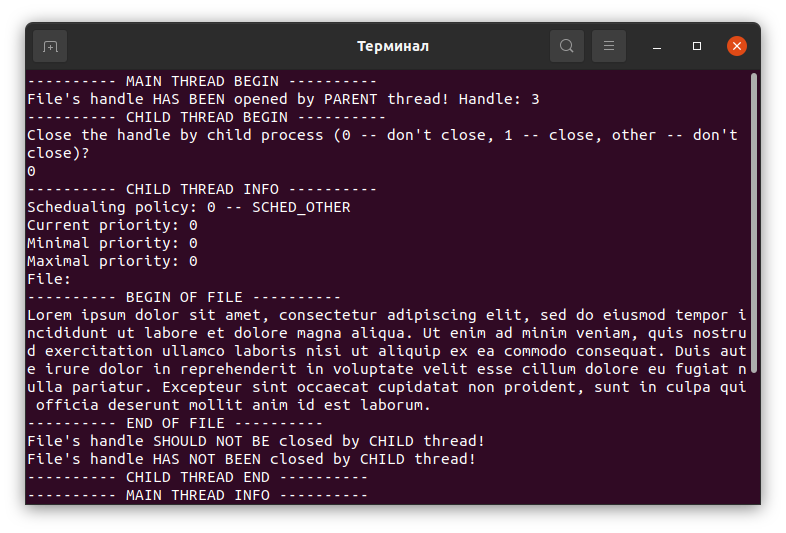


Рисунок 13. Запуск первой программы без закрытия дескриптора файла

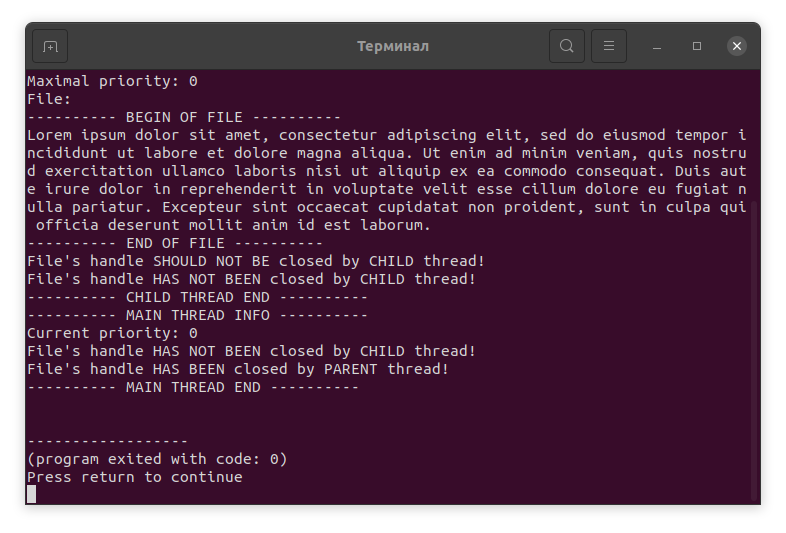


Рисунок 14. Запуск первой программы без закрытия дескриптора файла

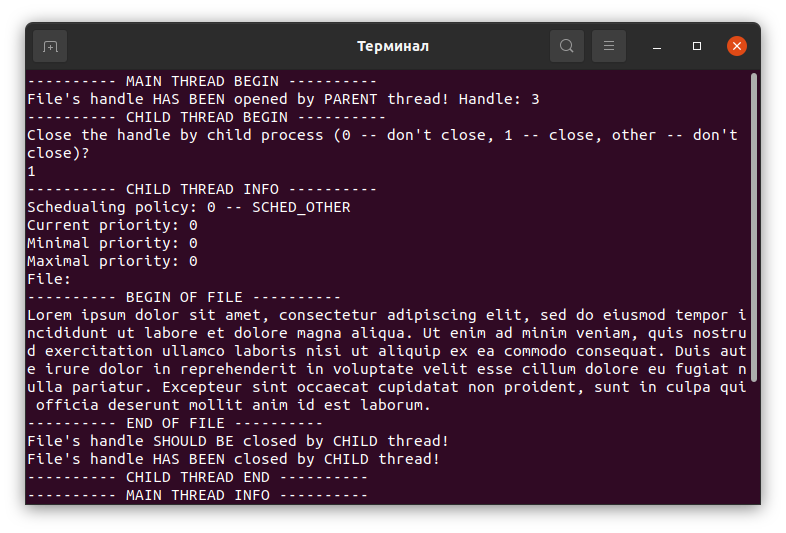


Рисунок 15. Запуск первой программы с закрытием дескриптора файла

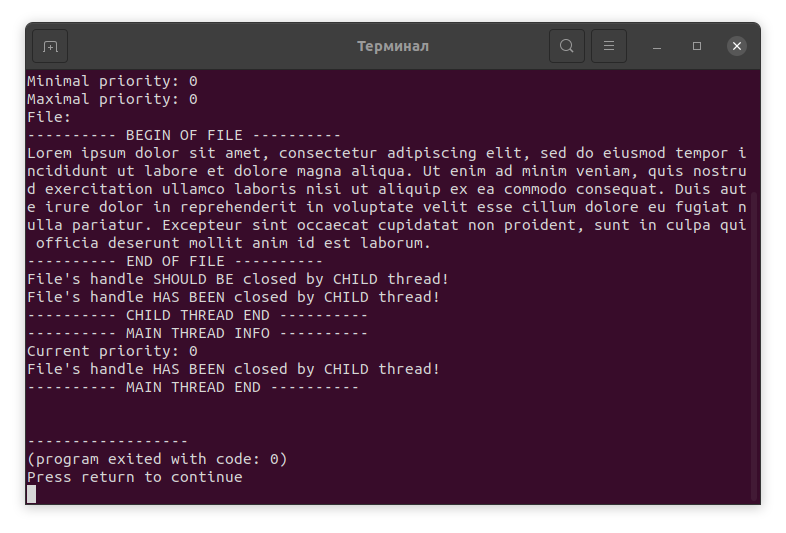


Рисунок 16. Запуск первой программы с закрытием дескриптора файла

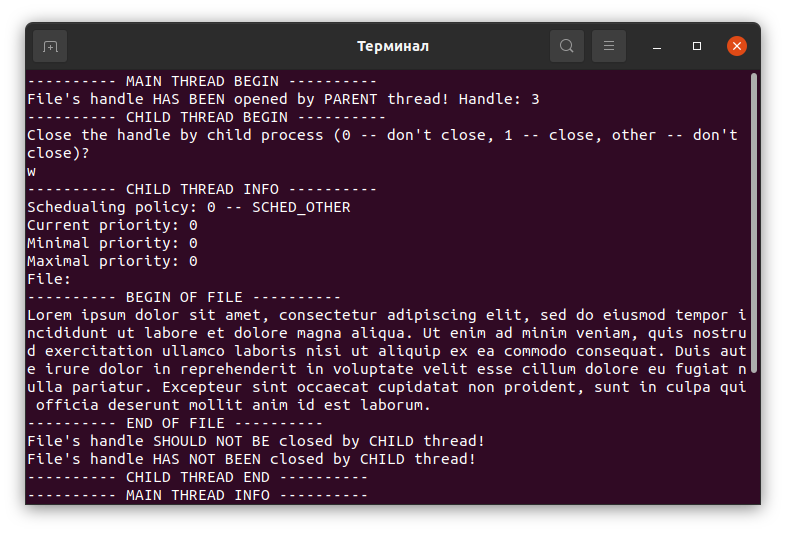


Рисунок 17. Запуск первой программы по умолчанию (без закрытия дескриптора файла)

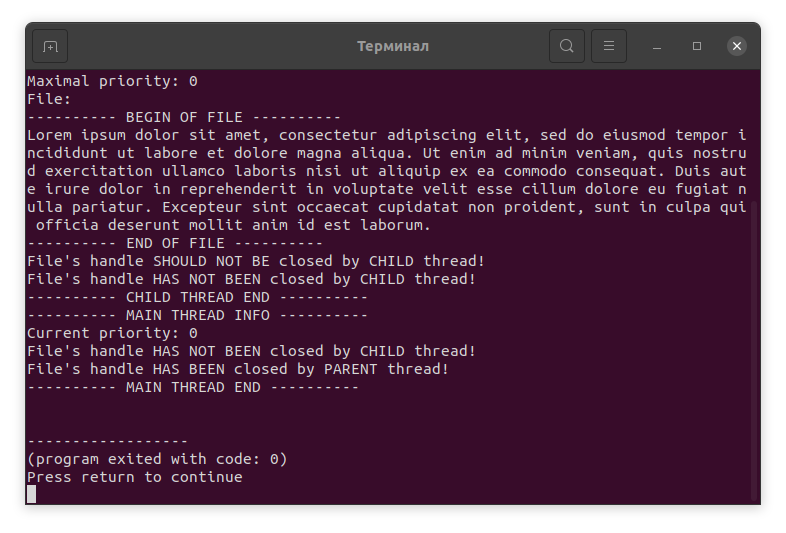


Рисунок 18. Запуск первой программы по умолчанию (без закрытия дескриптора файла)

## 3.3. Часть 2

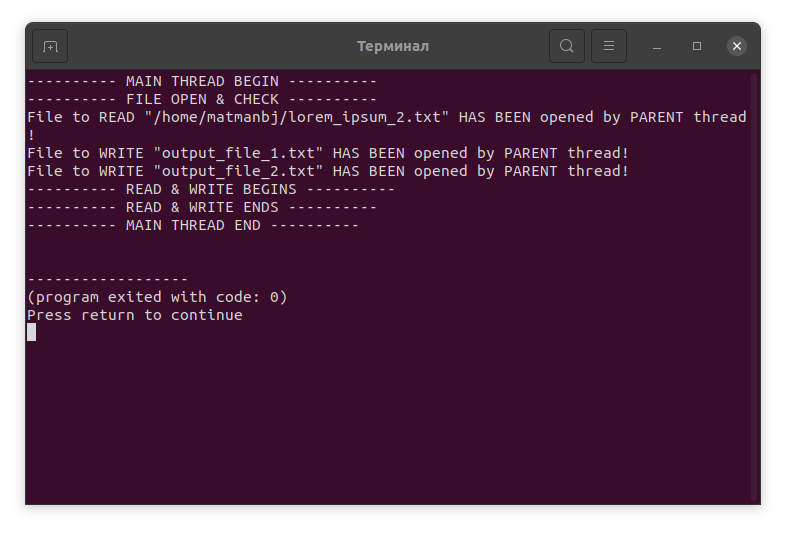


Рисунок 19. Запуск второй программы

# 3. Вывод

В ходе выполнения лабораторной работы №4 «Управление потоками» были изучены системные функции, позволяющие создавать потоки и управлять ими. Были написаны 2 программы: в первой программе дочерний поток осуществлял чтение файла по дескриптору, а затем по выбору пользователя закрывал его или оставлял открытым; во второй программе два дочерних потока осуществляли чтение файла, а затем записывали в новый, где один поток читал и записывал чётные строки, а другой – нечётные. Были использованы функции «pthread\_create()» и «pthread\_join()», которые создавали и ожидали поток соответственно. Таким образом и было произведено знакомство с организацией потоков и со способами синхронизации предков и потомков.

# 4. Список использованных источников

1. Онлайн-курс «Организация процессов и программирование в среде Linux» в LMS Moodle [сайт]. URL: <https://vec.etu.ru/moodle/course/view.php?id=9703>.

2. Разумовский Г.В. Организация процессов и программирование в среде Linux: учебно-методическое пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2018. 40с.