Липецкий государственный технический университет

Факультет автоматизации и информатики

Кафедра автоматизированных систем управления

ЛАБОРАТОРНАЯ РАБОТА № 3

по дисциплине «Операционные системы»

Управление процессами в ОС GNU/Linux

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Студент | |  |  |  | Антонова И. С. |  |
|  |  | |  | подпись, дата |  | фамилия, инициалы |  |
|  | Группа | АС-20-1 |  |  |  |  |  |
|  |  | |  |  |  |  |  |
|  | Руководитель | |  |  |  |  |  |
|  |  | |  |  |  | Кургасов В. В. |  |
|  | ученая степень, звание | |  | подпись, дата |  | фамилия, инициалы |  |

Липецк 2023 г.

Задание кафедры

Ознакомиться с теоретическими сведениями. Исследовать примеры программ, приведенных в теоретической части. Написать, скомпилировать и отладить программу по заданию преподавателя. Составить отчет о выполненной работе.

Вариант 1.

Блок № 1:

«Списки процессов». В дочернем процессе получить список процессов, записать его в файл и завершить процесс. В родительском процессе дождаться окончания дочернего, получить список процессов и дописать его в созданный дочерним процессом файл. Сравнить списки процессов.

Блок № 2:

Написать программу, создающую два дочерних процесса с использованием двух вызовов fork(). Родительский и два дочерних процесса должны выводить на экран свой pid и pid родительского процесса и текущее время в формате: часы : минуты : секунды : миллисекунды. Используя вызов system(), выполнить команду ps -x в родительском процессе. Найти свои процессы в списке запущенных процессов.

Блок № 3:

Написать программу нахождения массива K последовательных значений функции y[i]=sin(2\*PI\*i/N) (где i=0, 1, 2...K-1) с использованием ряда Тейлора. Пользователь задаёт значения K, N и количество n членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный поток. Каждый поток выводит на экран свой pid и рассчитанное значение ряда. Головной процесс суммирует все члены ряда Тейлора, и полученное значение y[i] записывает в файл.

Цель работы

Изучить возможности, предоставляемые ОС GNU/Linux для создания и управления процессами.

Ход работы

Пример № 1

Создадим новый файл с помощью команды touch ex1.c. Результат представлен на рисунке 1.

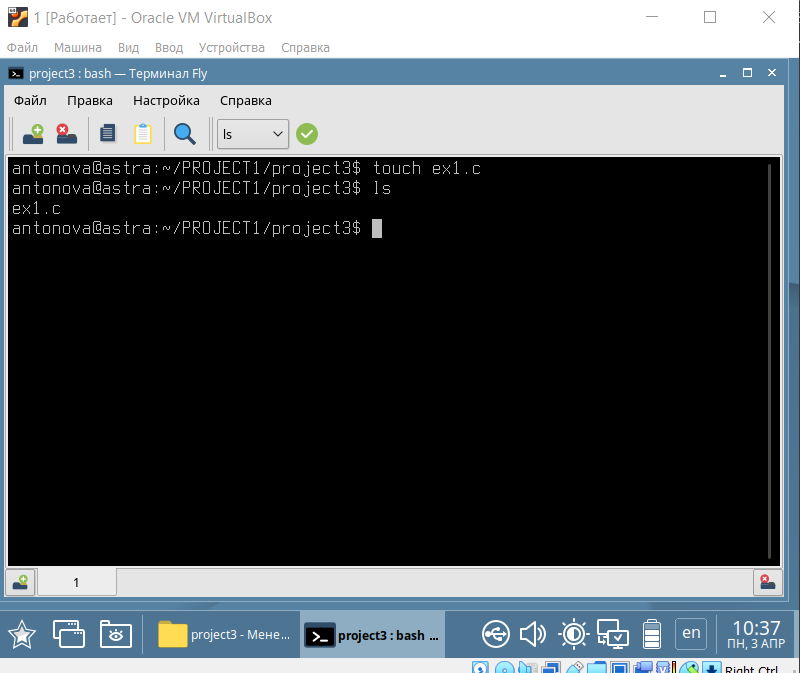


Рисунок 1 - Создание файла

Откроем данный файл с помощью редактора Vim и запишем код первого примера. Результат представлен на рисунке 2.

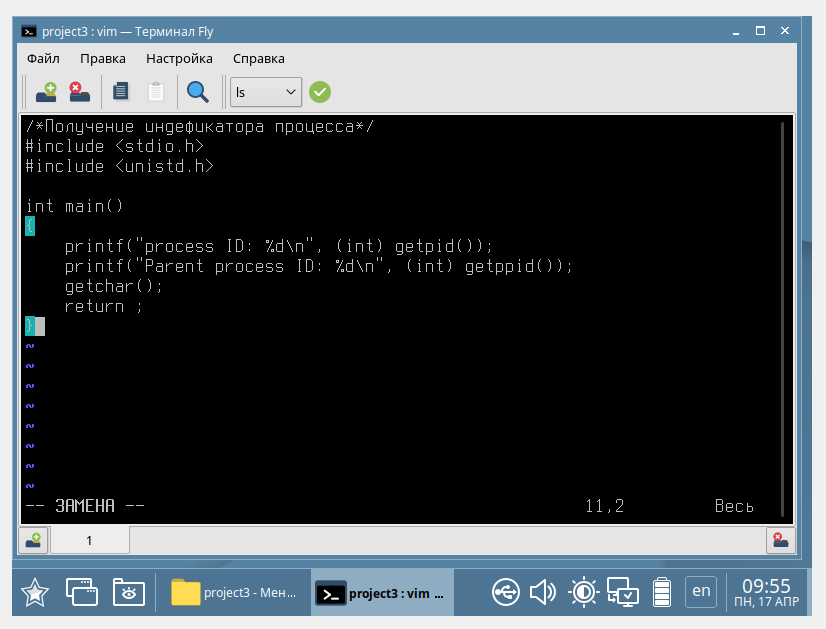


Рисунок 2 - Код программы

Скомпилируем код кнопкой F4 и запустим программу кнопкой F3. Результат представлен на рисунке 3.

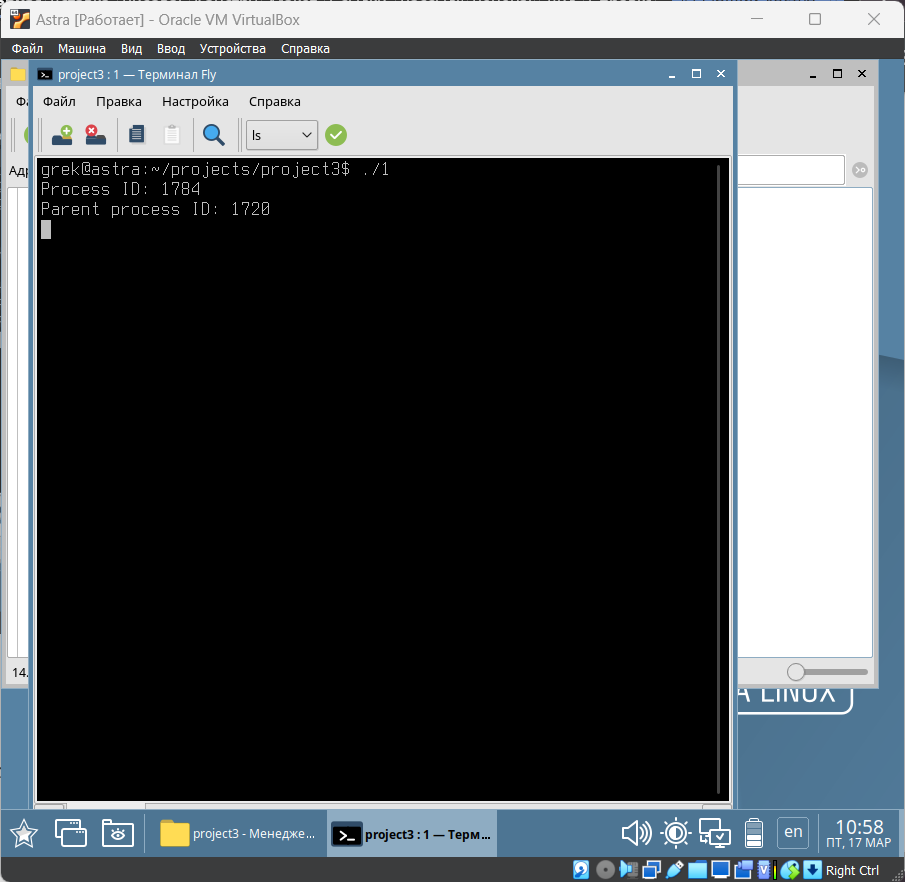


Рисунок 3 - Результат работы программы

После вывода идентификаторов процессов программа ожидает нажатия клавиши. В это время запустим отдельный терминал bash и введем команду ps -axf, выводящую список процессов. Результат представлен на рисунке 4.

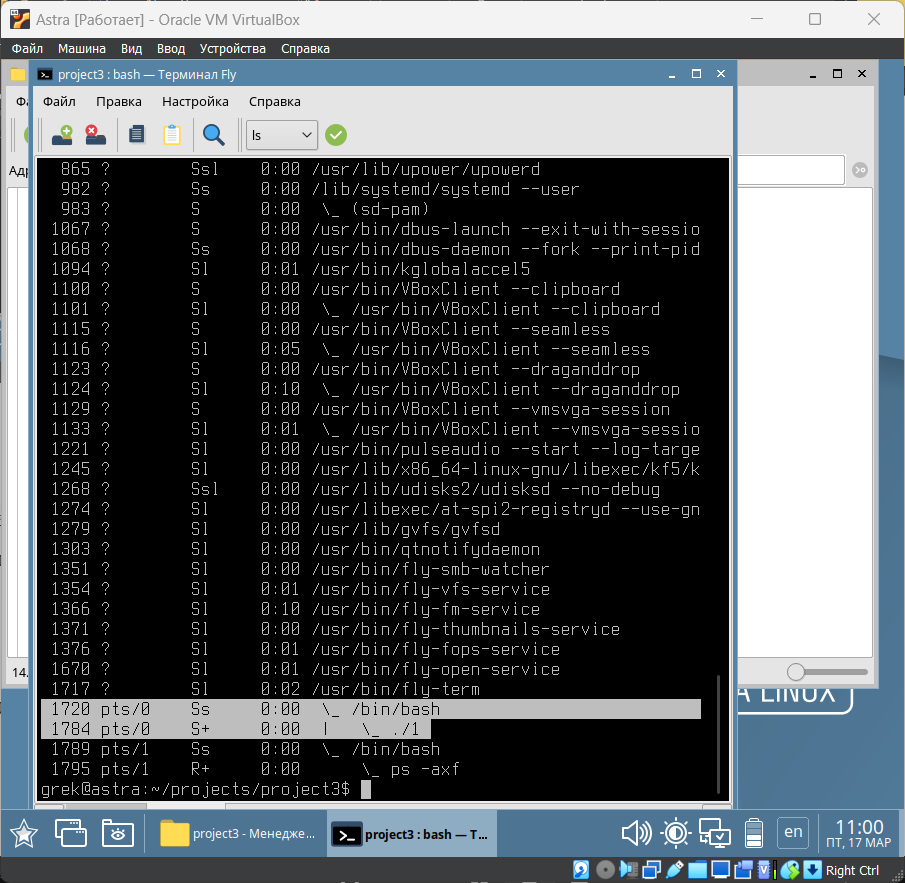


Рисунок 4 - Выполнение команды

В списке по полученным идентификаторам можно отыскать:

* родительский процесс – терминал bash (PID=1720);
* являющийся дочерним по отношению к bash процесс getpid (PID=1784);
* отдельно запущенный для вызова команды ps -axf терминал bash (PID=1789);
* процесс, отображающий список процессов ps (PID=1795).

Проделаем те же действия для других примеров.

Пример № 2.

Код программы представлен на рисунке 5.

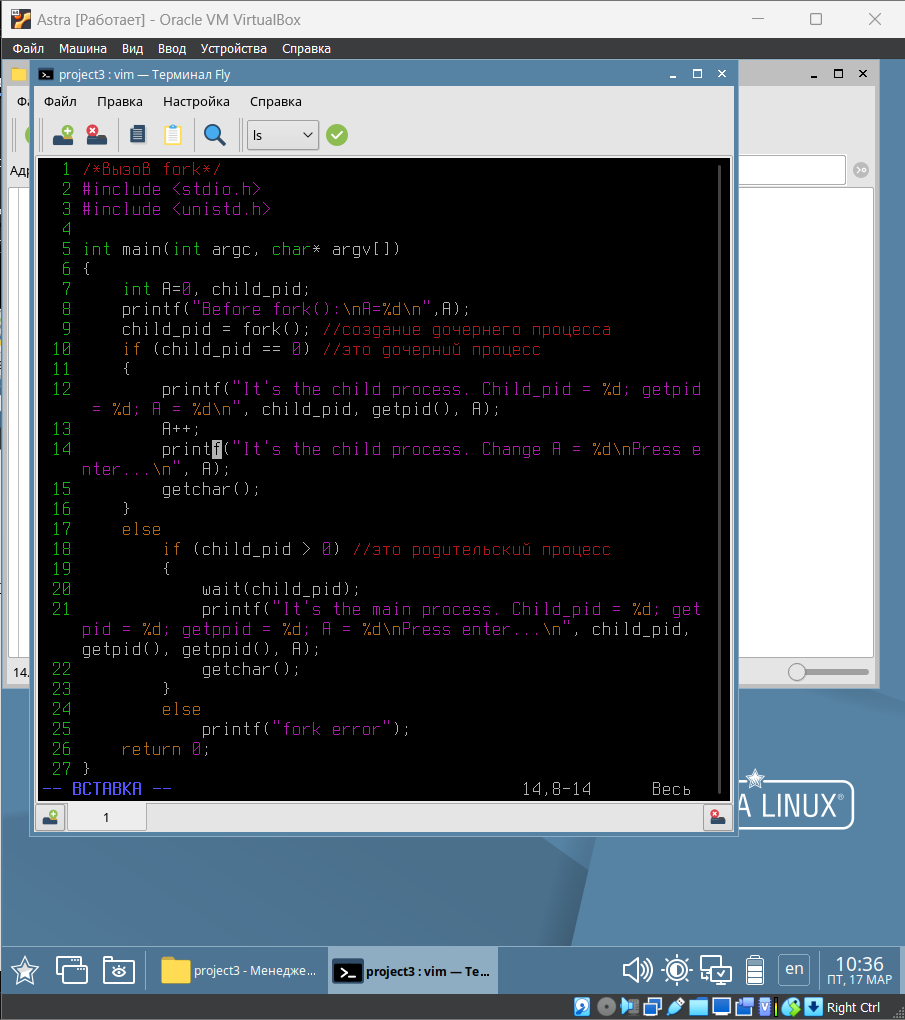


Рисунок 5 - Код программы

Результат работы программы представлен на рисунке 6.

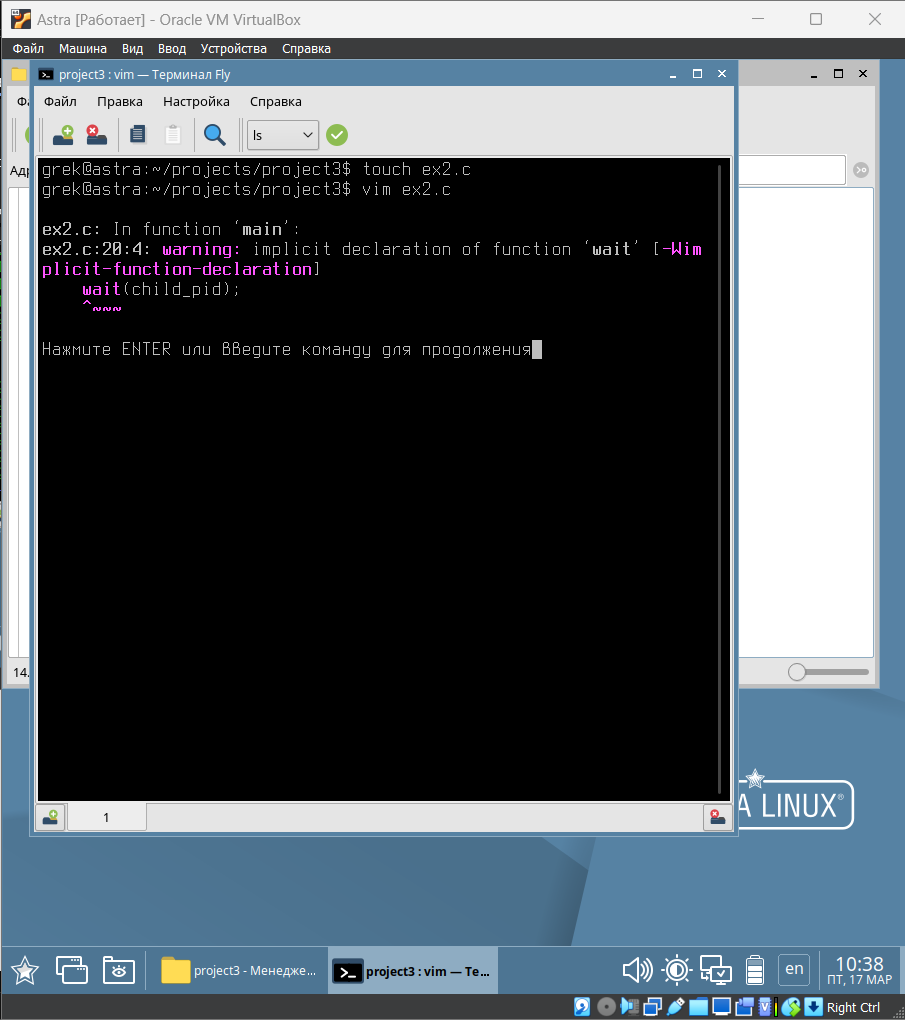


Рисунок 6 - Результат работы программы

Для исправления данной ошибки добавим два заголовочных файла sys/types.h и sys/wait.h в код программы. Результат работы программы представлен на рисунке 7.

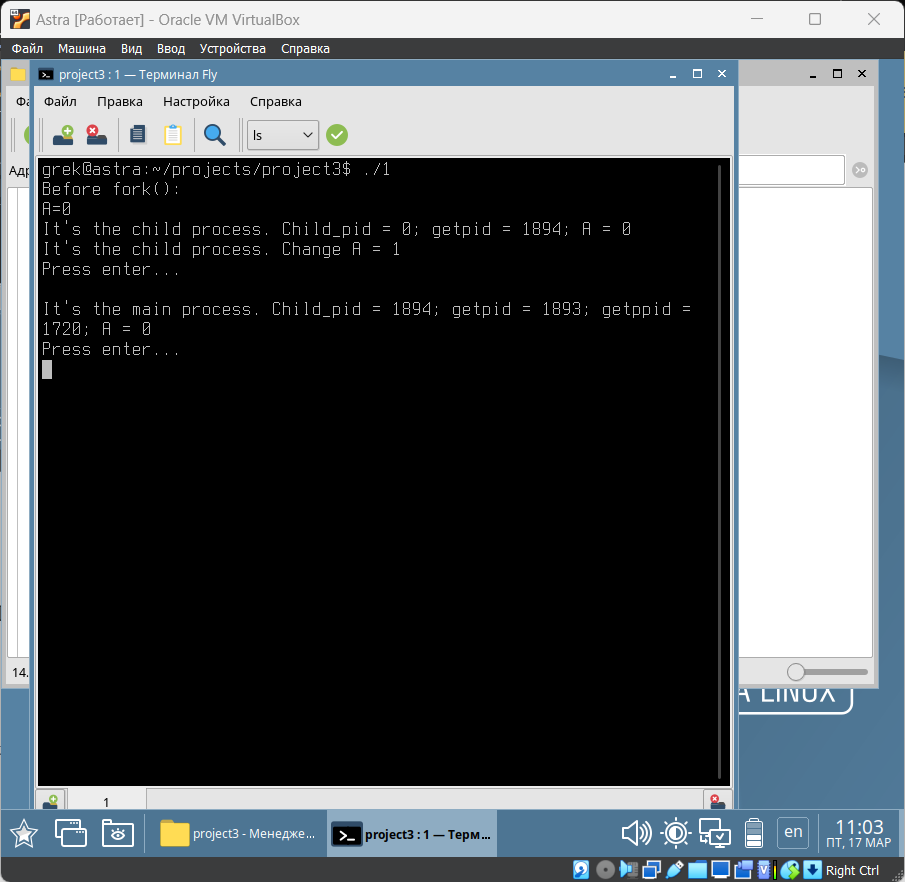


Рисунок 7 - Результат работы программы

В отдельном терминале до завершения дочернего процесса выполним команду ps -axf. Результат представлен на рисунке 8.

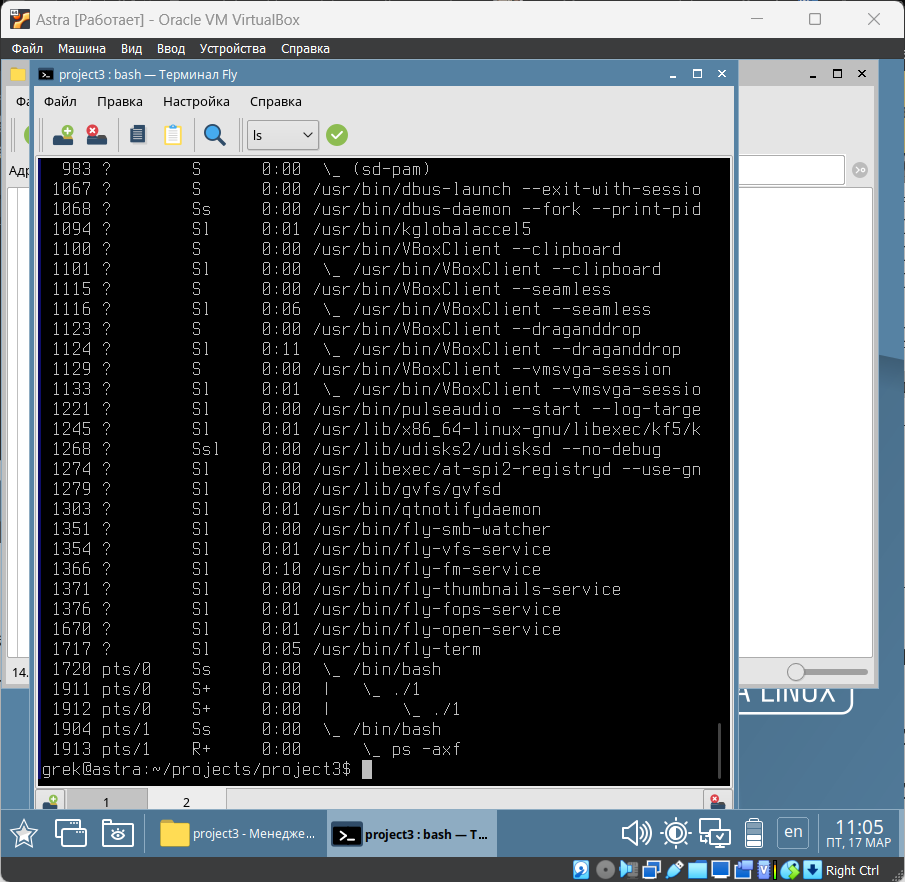


Рисунок 8 - Выполнение команды

Программа создает дочерний процесс. Для определения «кто есть кто» используется значение, возвращаемое вызовом fork. Перед вызовом fork переменной A присваивается значение 0. В дочернем процессе переменная увеличивается на 1. Родительский процесс ожидает окончания дочернего, после чего выводит значение переменной А.

Пример № 3

Код программы представлен на рисунке 9.

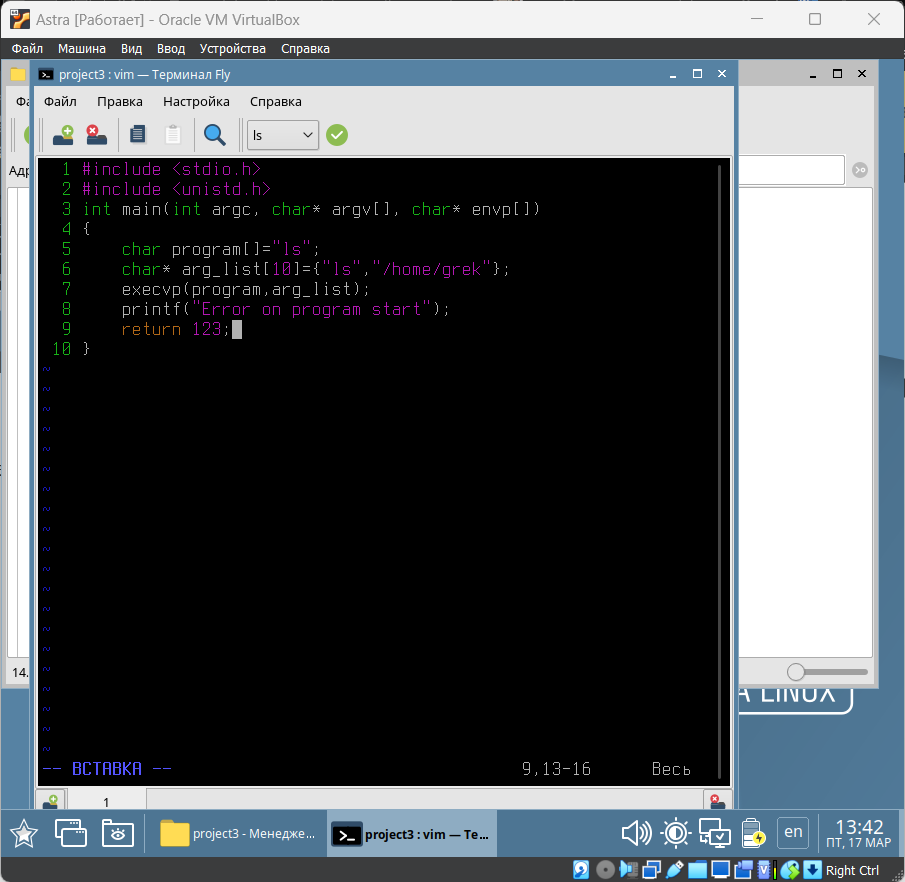


Рисунок 9 - Код программы

Результат работы программы представлен на рисунке 10.

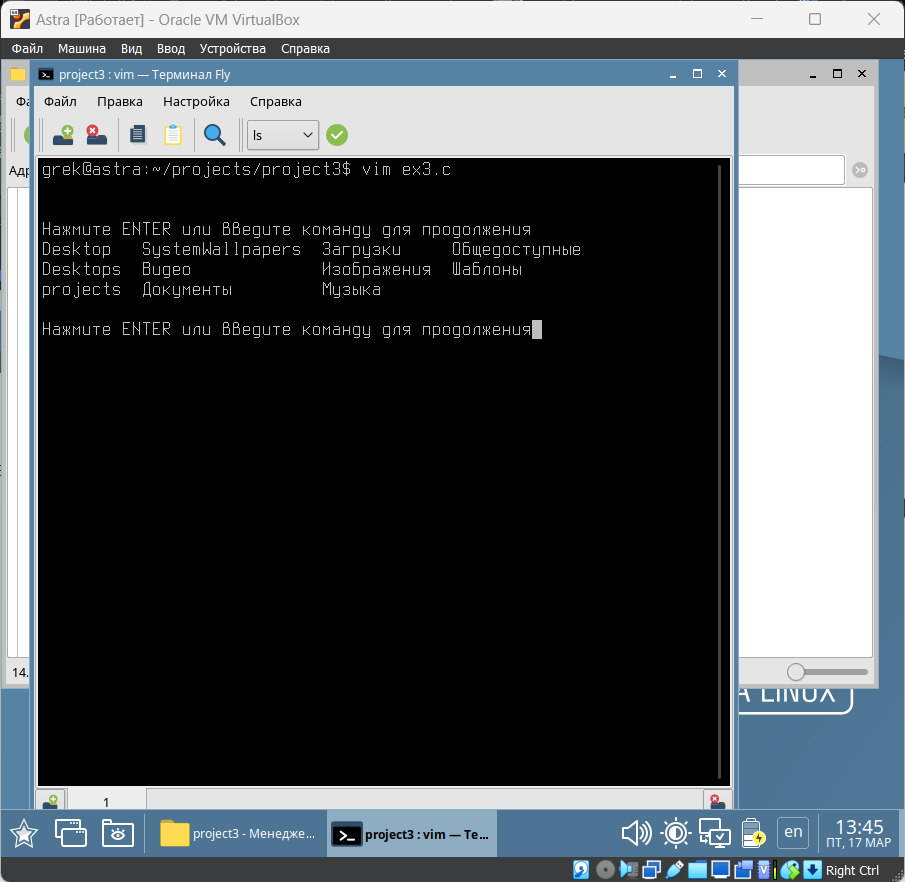


Рисунок 10 - Результат работы программы

Программа запускает команду ls с аргументом /home/grek. После успешного завершения ls команды, указанные после execvp не выполняются.

Пример № 4

Код программы представлен на рисунке 11.

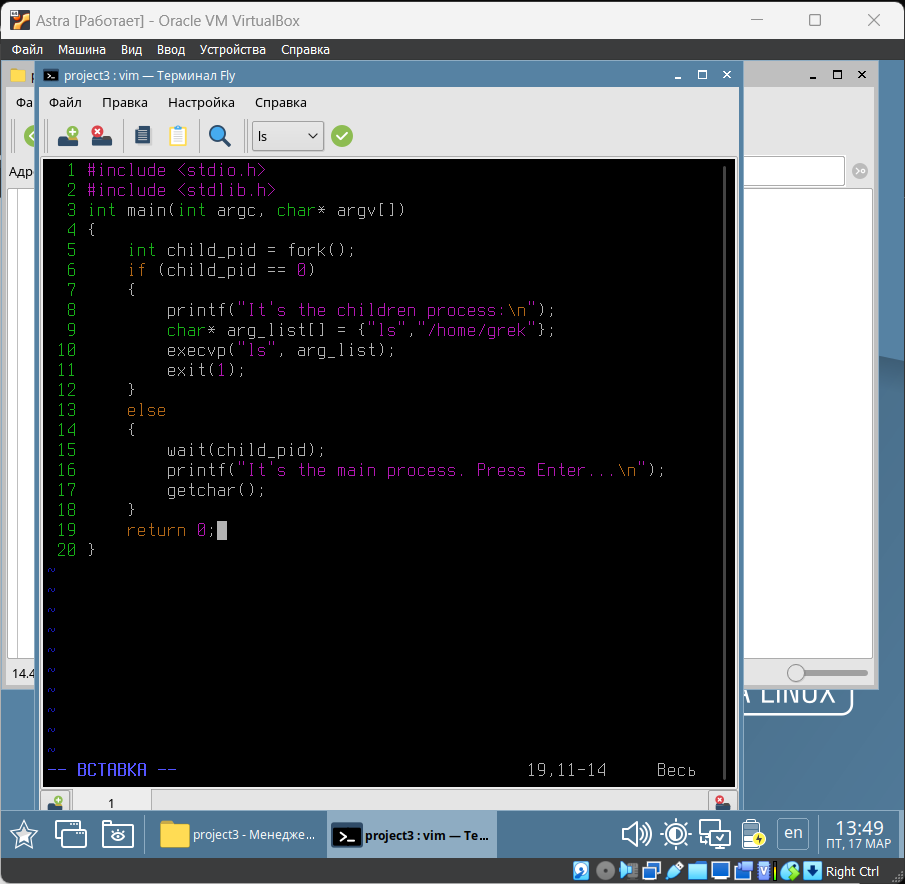


Рисунок 11 - Код программы

Результат работы программы представлен на рисунке 12.

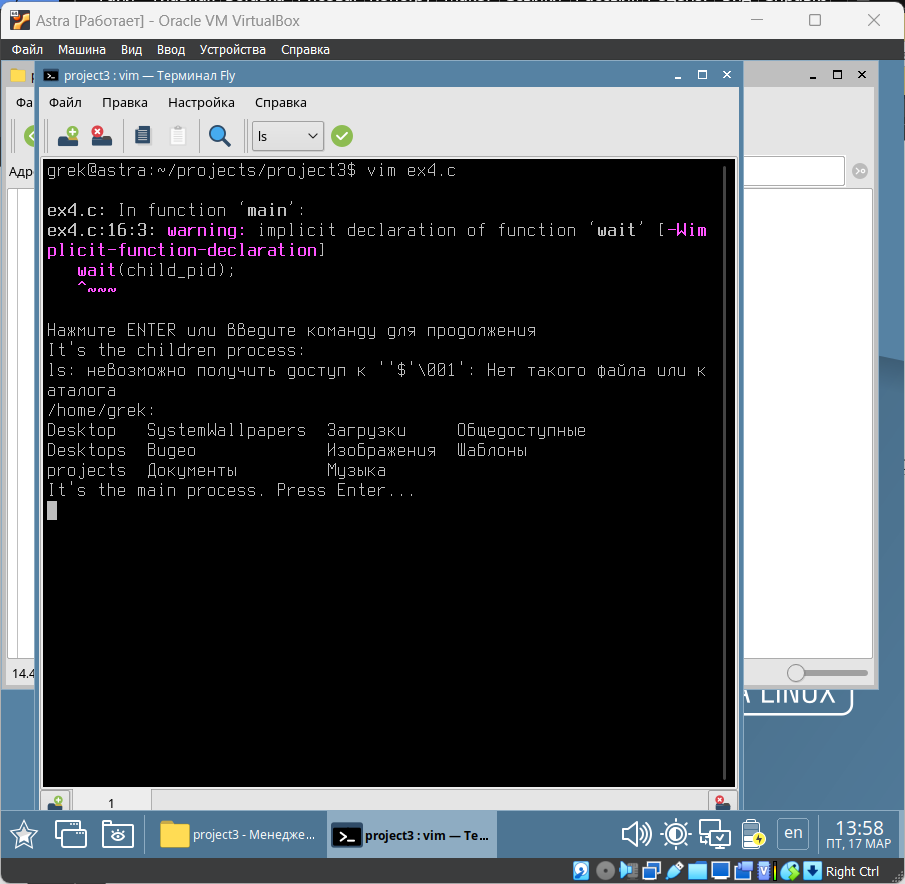


Рисунок 12 - Результат работы программы

При последовательном использовании вызовов fork и exec выполняемый процесс вначале разветвляется на два, а затем вместо второго запускается другая программа. Программа создает дочерний процесс, в котором выполняется вызов exec.

Пример № 5

Код программы представлен на рисунке 13.

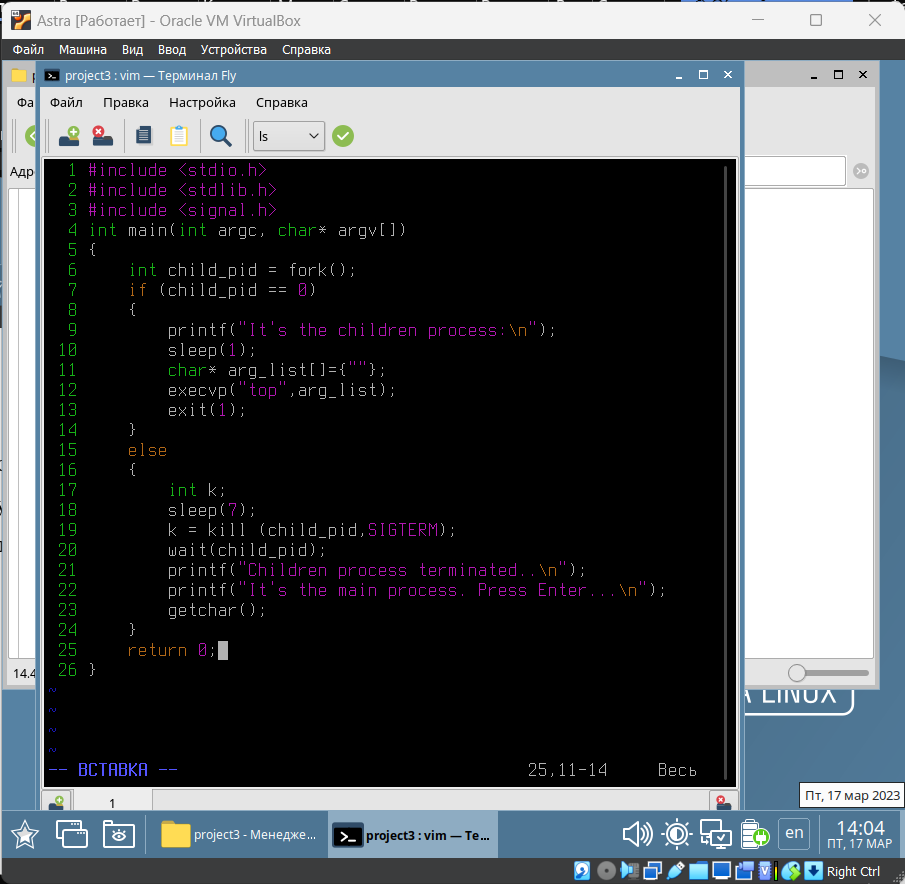


Рисунок 13 - Код программы

Результат работы программы представлен на рисунке 14.

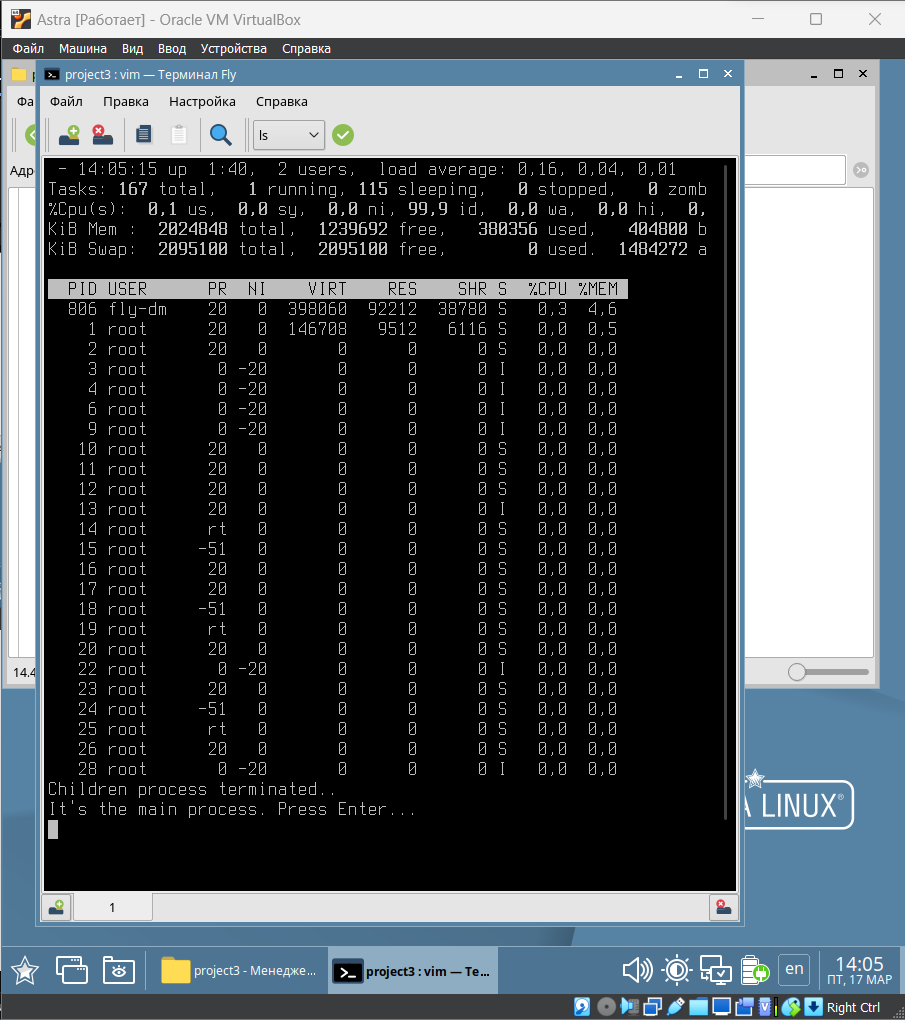


Рисунок 14 - Результат работы программы

Программа создает дочерний процесс и после паузы отправляет ему сигнал завершения.

Выполним задания по варианту.

1. «Списки процессов». В дочернем процессе получить список процессов, записать его в файл и завершить процесс. В родительском процессе дождаться окончания дочернего, получить список процессов и дописать его в созданный дочерним процессом файл. Сравнить списки процессов.

Код программы

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

**int** **main**() {

**pid\_t** pid;

**int** status;

**FILE** \*fp;

**char** buffer[**1024**];

// создание дочернего процесса

pid = fork();

**if** (pid == -**1**) {

printf("Ошибка создания дочернего процесса.**\n**");

exit(EXIT\_FAILURE);

} **else** **if** (pid == **0**) { // дочерний процесс

// открытие файла для записи

fp = fopen("processes.txt", "w");

**if** (fp == NULL) {

printf("Ошибка открытия файла.**\n**");

exit(EXIT\_FAILURE);

}

// получение списка процессов и запись в файл

system("ps -eo pid,user,cmd > processesD.txt");

// закрытие файла и завершение процесса

fclose(fp);

exit(EXIT\_SUCCESS);

} **else** { // родительский процесс

// ожидание завершения дочернего процесса

waitpid(pid, &status, **0**);

// открытие файла для дописывания

fp = fopen("processes.txt", "a");

**if** (fp == NULL) {

printf("Ошибка открытия файла.**\n**");

exit(EXIT\_FAILURE);

}

// получение списка процессов и дописывание в файл

system("ps -eo pid,user,cmd >> processes1.txt");

// закрытие файла

fclose(fp);

// сравнение списков процессов

fp = popen("diff -u processes1.txt processesD.txt", "r");

**while** (fgets(buffer, **sizeof**(buffer), fp) != NULL) {

printf("%s", buffer);

}

pclose(fp);

}

**return** **0**;

}

Результат работы программы представлен на рисунке 15.

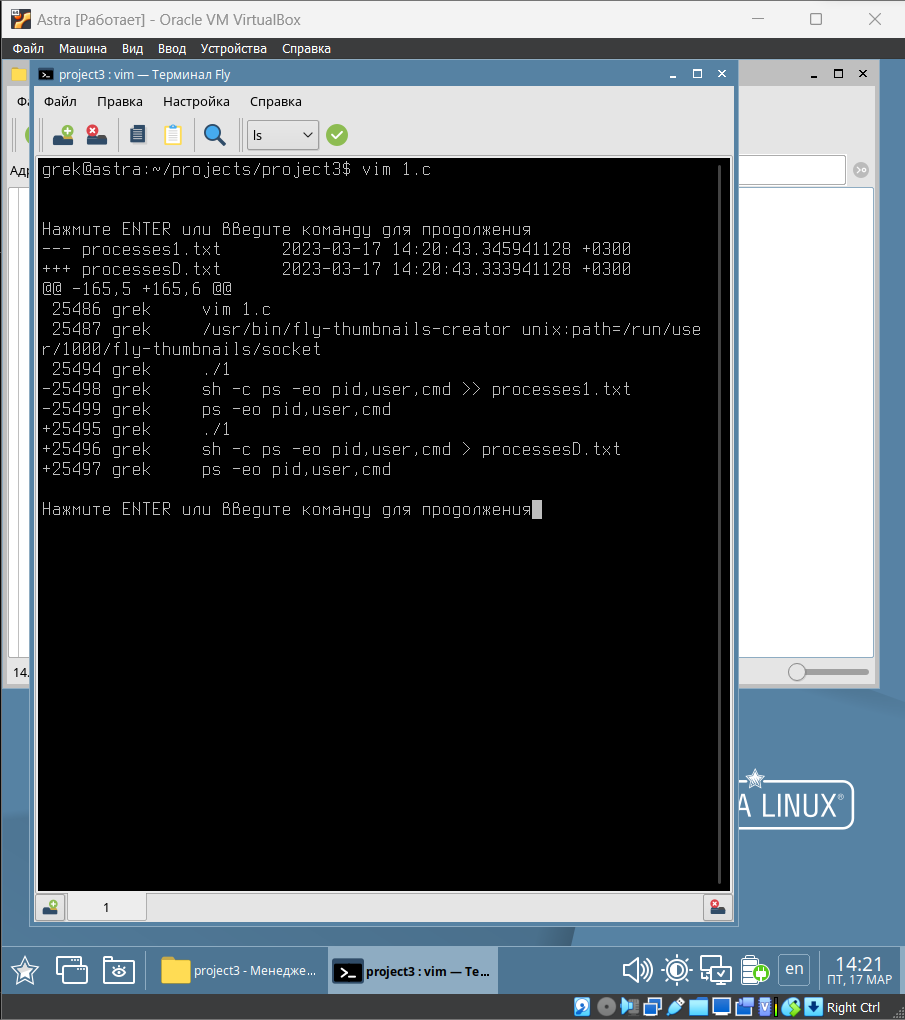


Рисунок 15 - Результат работы программы

1. Написать программу, создающую два дочерних процесса с использованием двух вызовов fork(). Родительский и два дочерних процесса должны выводить на экран свой pid и pid родительского процесса и текущее время в формате: часы : минуты : секунды : миллисекунды. Используя вызов system(), выполнить команду ps -x в родительском процессе. Найти свои процессы в списке запущенных процессов.

Код программы

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/timeb.h>

#include <time.h>

**int** **main**()

{

**int** status;

**pid\_t** child1, child2;

**if** (child1 = fork() == **0**)

{

printf("Child: %d**\n**", getpid());

printf("Parent: %d**\n**", getppid());

}

**else**

{

**if** (child2 = fork() == **0**)

{

printf("Child: %d**\n**", getpid());

printf("Parent: %d**\n**", getppid());

}

**else**

{

system("ps -x");

waitpid(child1, &status, **0**);

printf("Process (1) exit successfully: %s**\n**", (WIFEXITED(status) ? "true" : "false"));

printf("child exitcode = %i**\n**", WEXITSTATUS(status));

waitpid(child2, &status, **0**);

printf("Process (2) exit successfully: %s**\n**", (WIFEXITED(status) ? "true" : "false"));

printf("child exitcode = %i**\n**", WEXITSTATUS(status));

//system("ps -x");

printf("Main process: %d**\n**", getpid());

}

}

**struct** timeb sys\_time;

**struct** tm \*loctime;

ftime(&sys\_time);

loctime = localtime(&sys\_time.time);

printf("%d:%d:%d:%d**\n**", loctime->tm\_hour, loctime->tm\_min, loctime->tm\_sec, sys\_time.millitm);

**return** **0**;

}

Результат работы программы представлен на рисунках 16 и 17 .

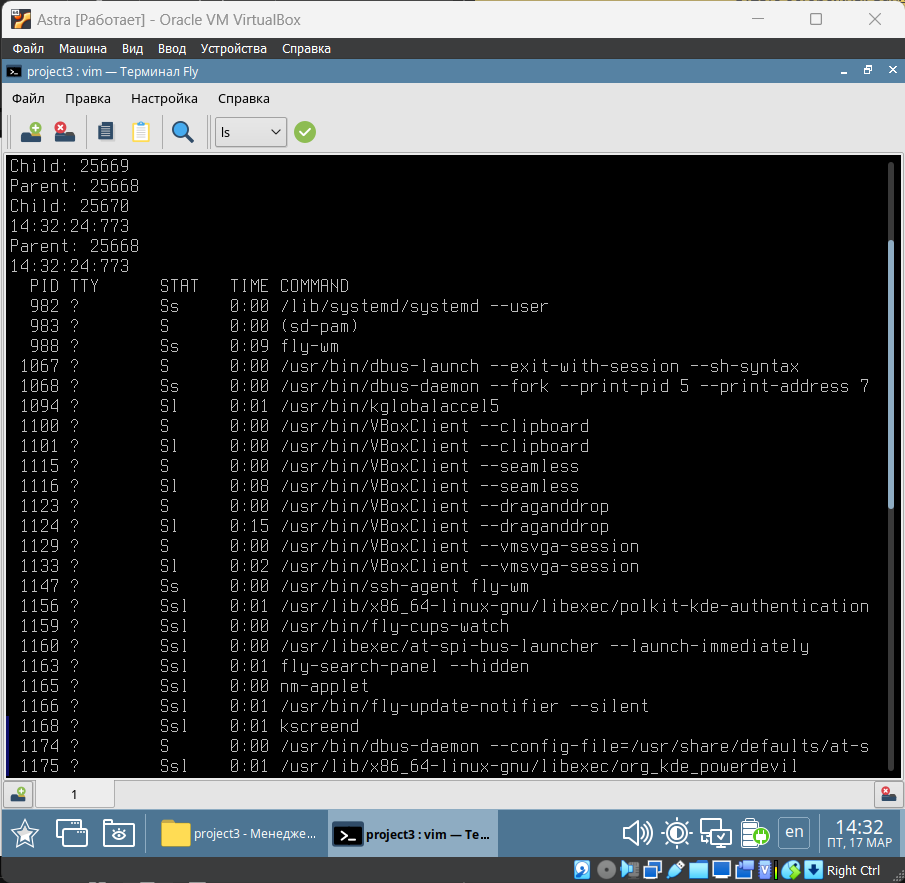


Рисунок 16 - Результат работы программы

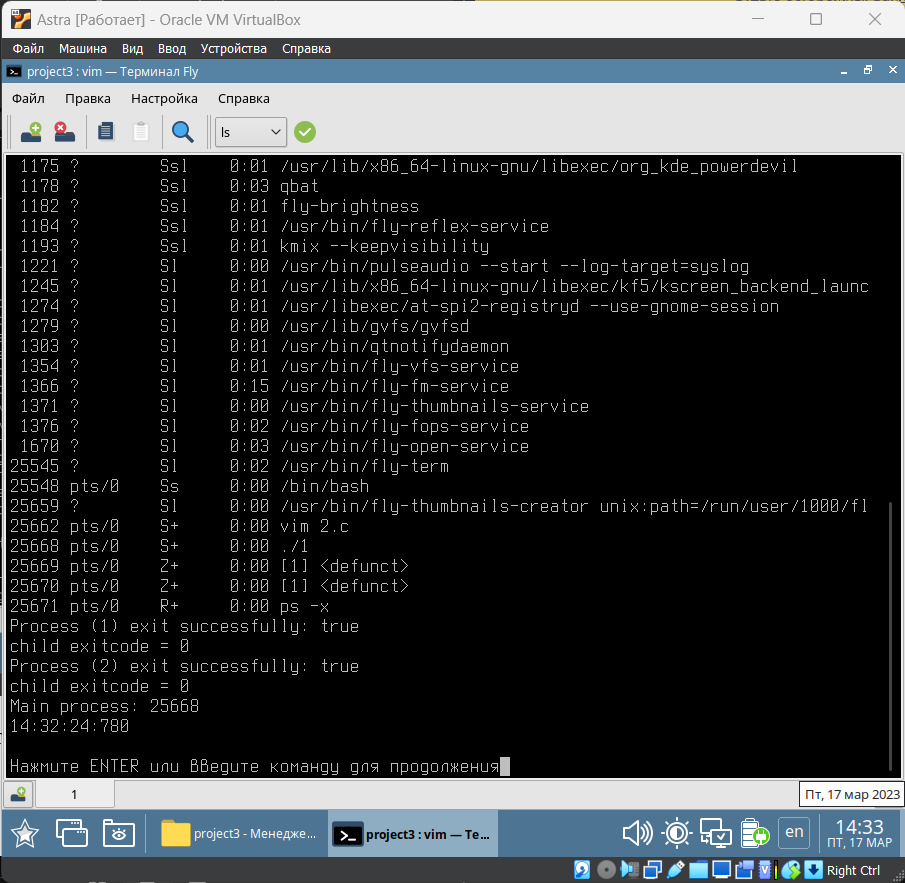


Рисунок 17 - Результат работы программы

1. Написать программу нахождения массива K последовательных значений функции y[i]=sin(2\*PI\*i/N) (где i=0, 1, 2...K-1) с использованием ряда Тейлора. Пользователь задаёт значения K, N и количество n членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный поток. Каждый поток выводит на экран свой pid и рассчитанное значение ряда. Головной процесс суммирует все члены ряда Тейлора, и полученное значение y[i] записывает в файл.

Код программы

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <math.h>

// Общие переменные

**double** \*partial\_sums;

**int** K, N, n;

// Функция, которую будут выполнять потоки

**void** \***calculate\_partial\_sum**(**void** \*arg) {

**int** thread\_id = \*(**int**\*)arg;

**int** i;

**double** sum = **0.0**;

**double** sign = (thread\_id % **2** == **0**) ? **1.0** : -**1.0**;

**int** start\_index = thread\_id \* K / n;

**int** end\_index = (thread\_id + **1**) \* K / n;

**for** (i = start\_index; i < end\_index; i++) {

sum += sign \* sin(**2.0** \* **3.14** \* i / N);

sign \*= -**1.0**;

}

partial\_sums[thread\_id] = sum;

printf("Thread %d (PID %d) finished calculating partial sum: %lf**\n**", thread\_id, getpid(), sum);

pthread\_exit(NULL);

}

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

// Получаем аргументы из командной строки

K = atoi(argv[**1**]);

N = atoi(argv[**2**]);

n = atoi(argv[**3**]);

// Выделяем память для массива сумм и идентификаторов потоков

partial\_sums = malloc(**sizeof**(**double**) \* n);

**int** \*thread\_ids = malloc(**sizeof**(**int**) \* n);

**int** i;

// Создаем потоки

**pthread\_t** \*threads = malloc(**sizeof**(**pthread\_t**) \* n);

**for** (i = **0**; i < n; i++) {

thread\_ids[i] = i;

pthread\_create(&threads[i], NULL, calculate\_partial\_sum, &thread\_ids[i]);

}

// Ждем завершения всех потоков

**for** (i = **0**; i < n; i++) {

pthread\_join(threads[i], NULL);

}

// Считаем общую сумму

**double** total\_sum = **0.0**;

**for** (i = **0**; i < n; i++) {

total\_sum += partial\_sums[i];

}

// Записываем результат в файл

**FILE** \*output\_file = fopen("output.txt", "w");

fprintf(output\_file, "%lf**\n**", total\_sum);

fclose(output\_file);

// Освобождаем память

free(partial\_sums);

free(thread\_ids);

free(threads);

**return** **0**;

}

Результат работы программы представлен на рисунке 18.

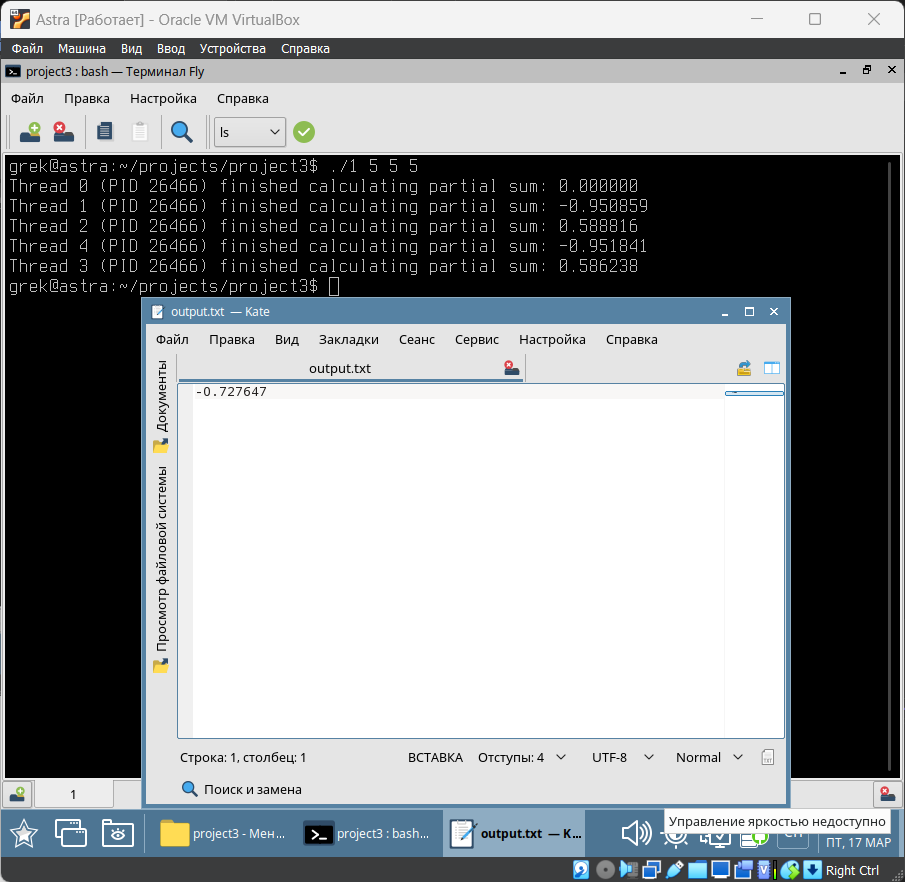


Рисунок 18 - Результат работы программы

Контрольные вопросы

1. Почему в листинге 7 переменная А сохранила значение 0 несмотря на то, что в дочернем процессе она была увеличена на 1?

У дочернего процесса свое адресное пространство, поэтому если в нем изменить значение переменной, то это никак не повлияет на значение переменной в родительском процессе.

1. В списке процессов, листинг 8, второй терминал bash имеет PID=1538, меньший, чем запущенная программа fork. Ошибка ли это?

Для того, чтобы процессы знали, кто из них родительский, а кто – дочерний существует два способа. Один из способов основан на получении идентификатора при помощи вызова getpid. У родительского процесса он будет равен идентификатору до выполнения вызова fork. У дочернего он будет иметь большее значение, т.к. процесс запущен позже родительского.

1. Почему в листинге 9 не была выведена на дисплей строка «Error on program start»?

После успешного завершения ls команды, строки кода после execvp не выполняются.

1. Что делает вызов wait(child\_pid) в программе с совместной работой fork и exec? Что будет, если эту строку закомментировать?

Функция wait приостанавливает родительский процесс до тех пор, пока не завершится один из его дочерних процессов. Если эту строку закомментировать, родительский процесс не будет приостановлен при работе дочерних процессов.

Вывод

В ходе выполнения данной лабораторной работы я изучила возможности, предоставляемые ОС GNU/Linux для создания и управления процессами.