Министерство науки и высшего образования РФ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Курский государственный университет»

Кафедра программного обеспечения и администрирования информационных систем

Направление подготовки математическое обеспечение и администрирование информационных систем

Форма обучения очная

**Отчет**

**по лабораторной работе №13**

«Процессы и потоки в ОС Linux»

дисциплина «Операционные системы и оболочки»

Выполнил:

студент группы 313.1 Козявин М.С.

Проверил:

к.т.н., доцент кафедры ПОиАИС Кривонос А.В.

Курск, 2023

**Цели работы:**

Исследовать методы создания процессов и потоков в ОС Linux, основные функции создания и управления процессами и обмен данными между процессами.

**Задание:**

1. Изучить теоретическую часть лабораторной работы.
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 (для дочерних процессов) 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.
3. В основной программе создать два потока. После этого процесс-отец создает файл, записывает в него строки вида: N pid, текущее время в формате часы: минуты: секунды: миллисекунды (где N – номер выводимой строки) и выводит формируемые строки в левой половине экрана. Количество записываемых в файл строк k = 100. Оба потока читают строки из файла и выводят их в правой части экрана в виде поток id потока текущее время строка из файла.

**Вариант 4**

1. Написать программу поиска одинаковых по содержимому файлов в двух каталогов, например, Dir1 и Dir2. Пользователь задает имена Dir1 и Dir2. В результате работы программы файлы, имеющиеся в Dir1, сравниваются с файлами в Dir2 по их содержимому. Процедуры сравнения должны запускаться с использованием функции fork() в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой pid, имя файла, число просмотренных байт и результаты сравнения. Число запущенных процессов не должно превышать N (вводится пользователем).

**Выполнение:**

1. Ознакомился с теоретический частью работы с процессами и потоками в ОС Linux.
2. Написал программу, создающую два дочерних процесса, с использованием двух вызовов fork(). Родительский и два дочерних процесса выводят на экран свой pid и текущее время. Используя вызов system(), выполнил команду ps -x в родительском процессе и нашёл свои процессы в списке запущенных процессов.

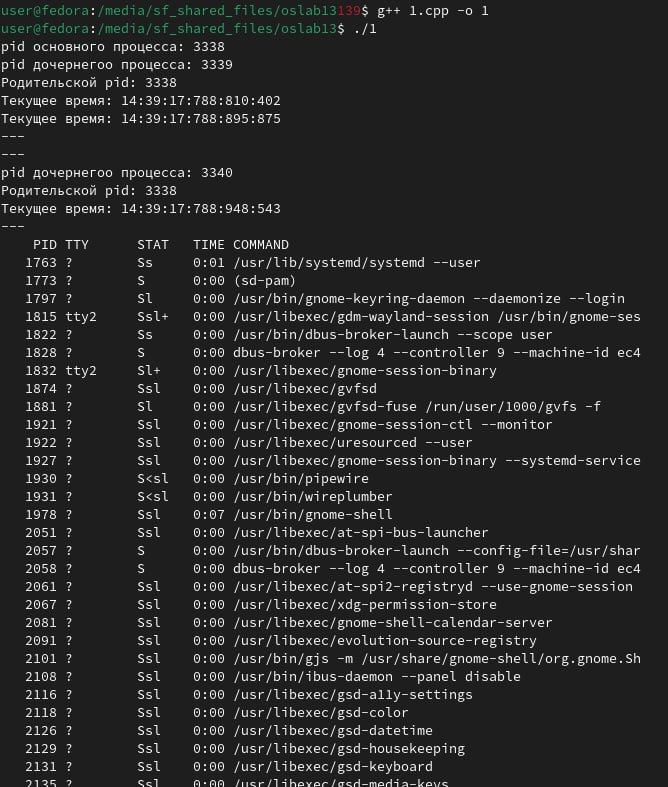


Рисунок 1 – Задание 1, вывод программы

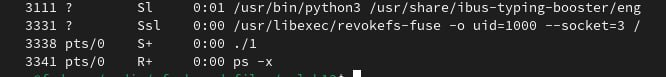


Рисунок 2 – Задание 1, процессы программы ps и задания 1 в списке запущенных процессов

**Исходный код программы задания 2:**

#include <stdlib.h>

#include <signal.h>

#include <stdio.h>

#include <unistd.h>

#include <chrono>

#include <sys/wait.h>

#include <inttypes.h>

#include <iostream>

using namespace std;

void subproc() {

pid\_t ppid = getppid();

pid\_t pid = getpid();

std::chrono::time\_point<std::chrono::system\_clock> now = std::chrono::system\_clock::now();

auto duration = now.time\_since\_epoch();

auto hours = std::chrono::duration\_cast<std::chrono::hours>(duration);

duration -= hours;

auto minutes = std::chrono::duration\_cast<std::chrono::minutes>(duration);

duration -= minutes;

auto seconds = std::chrono::duration\_cast<std::chrono::seconds>(duration);

duration -= seconds;

auto milliseconds = std::chrono::duration\_cast<std::chrono::milliseconds>(duration);

duration -= milliseconds;

auto microseconds = std::chrono::duration\_cast<std::chrono::microseconds>(duration);

duration -= microseconds;

auto nanoseconds = std::chrono::duration\_cast<std::chrono::nanoseconds>(duration);

cout << "pid дочернегоо процесса: " << (int)pid << endl;

cout << "Родительской pid: " << (int)ppid << endl;

cout << "Текущее время: " << hours.count()%24 << ":" << minutes.count() << ":" << seconds.count() << ":" << milliseconds.count() << ":" << microseconds.count() << ":" << nanoseconds.count() << endl;

cout << "---\n";

}

int main(void) {

pid\_t pid1 = fork();

if (pid1 == -1) {

cerr << "Не удалось запустить дочерний процесс\n";

}

if (pid1 != 0) {

pid\_t pid = getpid();

pid\_t pid2 = fork();

if (pid2 != 0) {

cout << "pid основного процесса: " << (int)pid << endl;

std::chrono::time\_point<std::chrono::system\_clock> now = std::chrono::system\_clock::now();

auto duration = now.time\_since\_epoch();

auto hours = std::chrono::duration\_cast<std::chrono::hours>(duration);

duration -= hours;

auto minutes = std::chrono::duration\_cast<std::chrono::minutes>(duration);

duration -= minutes;

auto seconds = std::chrono::duration\_cast<std::chrono::seconds>(duration);

duration -= seconds;

auto milliseconds = std::chrono::duration\_cast<std::chrono::milliseconds>(duration);

duration -= milliseconds;

auto microseconds = std::chrono::duration\_cast<std::chrono::microseconds>(duration);

duration -= microseconds;

auto nanoseconds = std::chrono::duration\_cast<std::chrono::nanoseconds>(duration);

cout << "Текущее время: " << hours.count()%24 << ":" << minutes.count() << ":" << seconds.count() << ":" << milliseconds.count() << ":" << microseconds.count() << ":" << nanoseconds.count() << endl;

cout << "---\n";

waitpid(pid1, NULL, 0);

waitpid(pid2, NULL, 0);

system("ps -x");

} else {

subproc();

}

} else {

subproc();

}

}

1. Написал программу, создающую два потока. После этого процесс-отец создает файл, записывает в него строки вида: N pid, текущее время в формате часы: минуты: секунды: миллисекунды (где N – номер выводимой строки) и выводит формируемые строки в левой половине экрана. Количество записываемых в файл строк k = 100. Оба потока читают строки из файла и выводят их в правой части экрана в виде поток id потока текущее время строка из файла. Доступ к общему файлу и общим массивам, разделяемым между потоками, контролируется с помощью мьютекса.

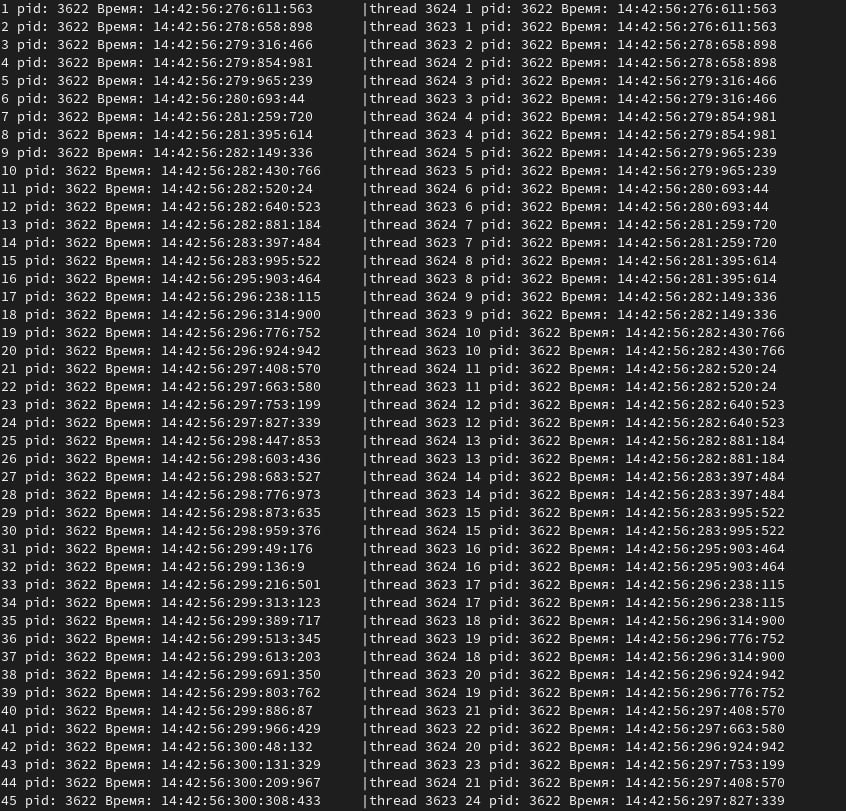


Рисунок 3 – Задание 2, вывод программы

**Исходный код программы задания 3:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <chrono>

#include <iostream>

#include <pthread.h>

#include <fstream>

#include <vector>

#include <string>

#include <iomanip>

using namespace std;

vector<string> left\_side;

vector<string> right\_side;

pthread\_mutex\_t mutex;

fstream file;

void\* print(void\* param) {

bool fl = true;

while (fl) {

system("clear");

pthread\_mutex\_lock(&mutex);

if (right\_side.size() >= 200)

fl = false;

for (int i = 0; i < 200; i++) {

if (i < left\_side.size()) {

cout << setw(50) << setfill(' ') << left << left\_side[i] << '|';

} else {

cout << setw(46) << setfill(' ') << right << '|' ;

}

if (i < right\_side.size()) {

cout << right\_side[i] << endl;

} else {

cout << endl;

}

}

pthread\_mutex\_unlock(&mutex);

if (fl)

usleep(1000000);

}

pthread\_exit(0);

}

void\* thread(void\* param) {

pid\_t id = gettid();

string line;

int i = 1;

while (i <= 100) {

usleep(10000);

line = "";

pthread\_mutex\_lock(&mutex);

file.seekg(51 \* (i-1), file.beg);

getline(file, line);

if (line != "") {

right\_side.push\_back("thread " + to\_string(id) + " " + line);

i++;

}

pthread\_mutex\_unlock(&mutex);

}

pthread\_exit(0);

}

int main(void) {

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_t thread1\_id;

pthread\_t thread2\_id;

pthread\_t print\_thread;

file.open("text.txt", ios::in | ios::out | ios::trunc | ios::ate);

file.seekg(0);

file.seekp(0);

if (pthread\_create(&thread1\_id, &attr, thread, NULL)) {

cerr << "Не удалось создать поток\n";

}

if (pthread\_create(&thread2\_id, &attr, thread, NULL)) {

cerr << "Не удалось создать поток\n";

}

if (pthread\_create(&print\_thread, &attr, print, NULL)) {

cerr << "Не удалось создать поток\n";

}

for (int i = 1; i <= 100; i++) {

std::chrono::time\_point<std::chrono::system\_clock> now = std::chrono::system\_clock::now();

auto duration = now.time\_since\_epoch();

auto hours = std::chrono::duration\_cast<std::chrono::hours>(duration);

duration -= hours;

auto minutes = std::chrono::duration\_cast<std::chrono::minutes>(duration);

duration -= minutes;

auto seconds = std::chrono::duration\_cast<std::chrono::seconds>(duration);

duration -= seconds;

auto milliseconds = std::chrono::duration\_cast<std::chrono::milliseconds>(duration);

duration -= milliseconds;

auto microseconds = std::chrono::duration\_cast<std::chrono::microseconds>(duration);

duration -= microseconds;

auto nanoseconds = std::chrono::duration\_cast<std::chrono::nanoseconds>(duration);

pthread\_mutex\_lock(&mutex);

left\_side.push\_back(to\_string(i) + " pid: " + to\_string(getpid()) + " Время: " + to\_string(hours.count()%24) + ":" + to\_string(minutes.count()) +

":" + to\_string(seconds.count()) + ":" + to\_string(milliseconds.count()) + ":" + to\_string(microseconds.count()) + ":" + to\_string(nanoseconds.count()));

file << setw(50) << setfill(' ') << left << left\_side[left\_side.size()-1] << endl;

pthread\_mutex\_unlock(&mutex);

}

file.clear();

pthread\_join(thread1\_id, NULL);

pthread\_join(thread2\_id, NULL);

pthread\_join(print\_thread, NULL);

pthread\_mutex\_destroy(&mutex);

file.close();

}

1. Написал программу поиска одинаковых по содержимому файлов в двух каталогов. Пользователь задает имена файлов. В результате работы программы файлы сравниваются по их содержимому. Если размер файлов не совпадает проверки не будет и файлы обработаются, прочитав 0 байт. Процедуры сравнения запускаются с использованием функции fork() в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой pid, имя файла, число просмотренных байт и результаты сравнения. Число запущенных процессов не должно превышать N (вводится пользователем).

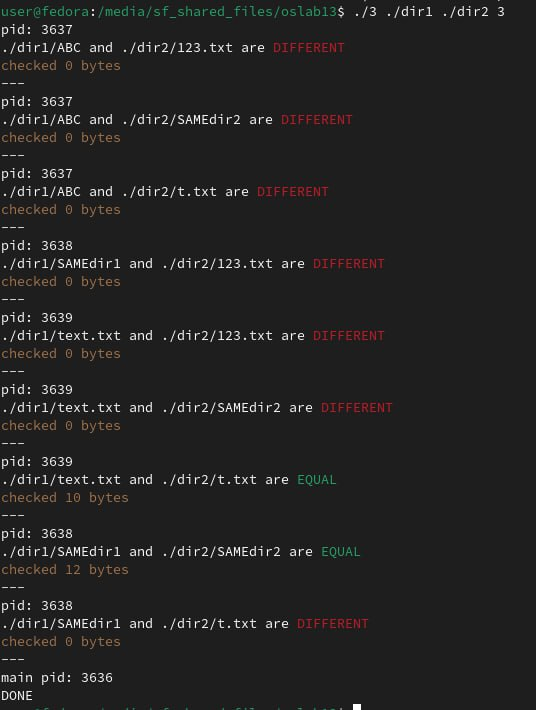


Рисунок 4 – Задание 4, вывод программы при N = 3

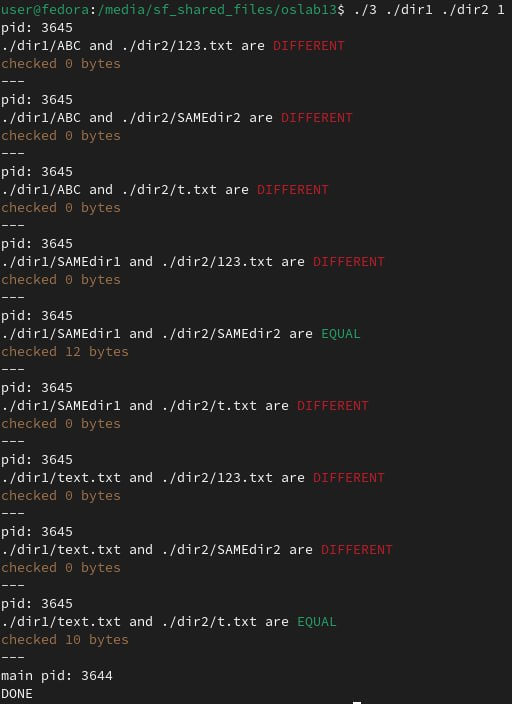


Рисунок 5 – Задание 4, вывод программы при N = 1

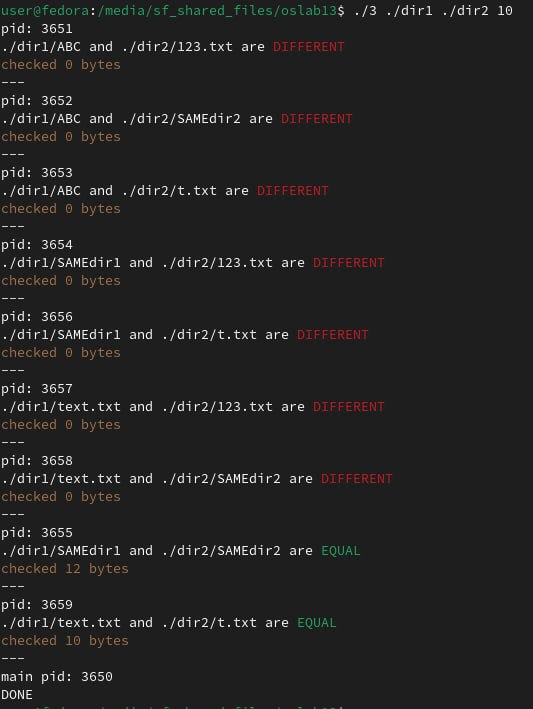


Рисунок 6 – Задание 4, вывод программы при N = 10

**Исходный код программы задания 4:**

#include <dirent.h>

#include <sys/stat.h>

#include <vector>

#include <iostream>

#include <fstream>

#include <cstring>

#include <string>

#include <sys/wait.h>

#include <cmath>

using namespace std;

#define RED "\033[31m"

#define YELLOW "\033[33m"

#define GREEN "\033[32m"

#define RESET "\033[0m"

struct Data {

string path;

string name;

long long size;

char\* time;

unsigned int rights;

long unsigned int index;

};

vector<Data> dir1\_files;

vector<Data> dir2\_files;

int dir\_i = 0;

int dir\_j = 0;

int n = 1;

vector<int> pids;

ofstream out\_file;

void find1(const char \*dir1\_name) {

DIR \*dir1 = opendir(dir1\_name);

struct dirent\* item1;

while ((item1 = readdir(dir1))!= NULL) {

if (strcmp(item1->d\_name, ".") == 0 || strcmp(item1->d\_name, "..") == 0) {

continue;

}

struct stat sb;

char path[PATH\_MAX];

snprintf(path, sizeof(path), "%s/%s", dir1\_name, item1->d\_name);

stat(path, &sb);

if (S\_ISDIR(sb.st\_mode)) {

find1(path);

} else {

Data s = {};

s.name = string(item1->d\_name);

s.path = path;

s.size = sb.st\_size;

s.time = ctime(&sb.st\_ctime);

s.rights = sb.st\_mode & 0777;

s.index = sb.st\_ino;

dir1\_files.push\_back(s);

}

}

}

void find2(const char \*dir2\_name) {

DIR \*dir2 = opendir(dir2\_name);

struct dirent\* item2;

while ((item2 = readdir(dir2))!= NULL) {

if (strcmp(item2->d\_name, ".") == 0 || strcmp(item2->d\_name, "..") == 0) {

continue;

}

struct stat sb;

char path[PATH\_MAX];

snprintf(path, sizeof(path), "%s/%s", dir2\_name, item2->d\_name);

stat(path, &sb);

if (S\_ISDIR(sb.st\_mode)) {

find1(path);

} else {

Data s = {};

s.name = string(item2->d\_name);

s.path = path;

s.size = sb.st\_size;

s.time = ctime(&sb.st\_ctime);

s.rights = sb.st\_mode & 0777;

s.index = sb.st\_ino;

dir2\_files.push\_back(s);

}

}

}

void find(const char \*dir1\_name, const char \*dir2\_name) {

DIR \*dir1 = opendir(dir1\_name);

DIR \*dir2 = opendir(dir2\_name);

if (dir1 == NULL) {

fprintf(stderr, "Ошибка открытия каталога %s: %s\n", dir1\_name, strerror(errno));

return;

}

if (dir2 == NULL) {

fprintf(stderr, "Ошибка открытия каталога %s: %s\n", dir2\_name, strerror(errno));

return;

}

struct dirent\* item;

while ((item = readdir(dir1))!= NULL) {

if (strcmp(item->d\_name, ".") == 0 || strcmp(item->d\_name, "..") == 0) {

continue;

}

struct stat sb;

char path[PATH\_MAX];

snprintf(path, sizeof(path), "%s/%s", dir1\_name, item->d\_name);

stat(path, &sb);

if (S\_ISDIR(sb.st\_mode)) {

find1(path);

} else {

Data s = {};

s.name = string(item->d\_name);

s.path = path;

s.size = sb.st\_size;

s.time = ctime(&sb.st\_ctime);

s.rights = sb.st\_mode & 0777;

s.index = sb.st\_ino;

dir1\_files.push\_back(s);

}

}

closedir(dir1);

while ((item = readdir(dir2))!= NULL) {

if (strcmp(item->d\_name, ".") == 0 || strcmp(item->d\_name, "..") == 0) {

continue;

}

struct stat sb;

char path[PATH\_MAX];

snprintf(path, sizeof(path), "%s/%s", dir2\_name, item->d\_name);

stat(path, &sb);

if (S\_ISDIR(sb.st\_mode)) {

find2(path);

} else {

Data s = {};

s.name = string(item->d\_name);

s.path = path;

s.size = sb.st\_size;

s.time = ctime(&sb.st\_ctime);

s.rights = sb.st\_mode & 0777;

s.index = sb.st\_ino;

dir2\_files.push\_back(s);

}

}

closedir(dir2);

}

void check(int x) {

Data d1;

Data d2;

if (dir\_j+x < dir2\_files.size() && dir\_i < dir1\_files.size()) {

d1 = dir1\_files[dir\_i];

d2 = dir2\_files[dir\_j+x];

} else if (dir\_j+x >= dir2\_files.size() && dir\_i + ((dir\_j+x) / dir2\_files.size()) < dir1\_files.size()) {

d1 = dir1\_files[dir\_i + ((dir\_j+x) / dir2\_files.size())];

d2 = dir2\_files[(dir\_j+x) % dir2\_files.size()];

} else {

return;

}

int bytes\_count = 0;

if (d1.size != d2.size) {

cout << "pid: " << getpid() << endl;

cout << d1.path << " and " << d2.path << " are " << RED << "DIFFERENT" << RESET << endl;

cout << YELLOW << "checked " << bytes\_count << " bytes" << RESET << endl;

cout << "---\n";

return;

}

ifstream fin1(d1.path);

ifstream fin2(d2.path);

if (fin2 && fin2) {

char ch1, ch2;

bool result = true;

while (fin1.get(ch1) && fin2.get(ch2))

{

bytes\_count++;

if (ch1 != ch2)

{

result = false;

break;

}

}

if (result) {

cout << "pid: " << getpid() << endl;

cout << d1.path << " and " << d2.path << " are " << GREEN << "EQUAL" << RESET << endl;

cout << YELLOW << "checked " << bytes\_count << " bytes" << RESET << endl;out\_file << "---\n";

} else {

cout << "pid: " << getpid() << endl;

cout << d1.path << " and " << d2.path << " are " << RED << "DIFFERENT" << RESET << endl;

cout << YELLOW << "checked " << bytes\_count << " bytes" << RESET << endl;

}

cout << "---\n";

}

else

cout << "Error opening file!" << endl;

}

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

if (argc < 4) {

fprintf(stderr, "Использование: <директория> <директория> <число потоков>\n");

return 1;

}

int checked = 0;

const char \*dir1 = argv[1];

const char \*dir2 = argv[2];

find(dir1, dir2);

int threads\_count = stoi(string(argv[3]));

n = ceil((dir1\_files.size() \* dir2\_files.size()) / (double)threads\_count);

int main\_pid = getpid();

for (int i = 0; i < threads\_count; i++) {

pids.push\_back(fork());

if (getpid() != main\_pid) {

break;

}

dir\_j += n;

if (dir\_j >= dir2\_files.size()) {

dir\_i += dir\_j / dir2\_files.size();

dir\_j %= dir2\_files.size();

}

}

if (getpid() == main\_pid) {

for (int i = 0; i < pids.size(); i++) {

waitpid(pids[i], NULL, 0);

}

cout << "main pid: " << getpid() << endl;

cout << "DONE" << endl;

} else {

for (int x = 0; x < n; x++) {

check(x);

}

exit(0);

}

return 0;

}

**Вывод**

В ходе лабораторной работы исследовал методы создания процессов и потоков в ОС Linux, изучил основные функции создания и управления процессами и потоками и обмен данными между процессами.