МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ИНСТИТУТ НЕПРЕРЫВНОГО И ДИСТАНЦИОННОГО ОБРАЗОВАНИЯ

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

ОЦЕНКА

ПРЕПОДАВАТЕЛЬ

старший преподаватель		Поляк М.Д.
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 2

Синхронизация потоков средствами POSIX

по дисциплине: Операционные системы

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №	Z1432К		Куралесов Ф.А.
	номер группы	подпись, дата	инициалы, фамилия

Студенческий билет № 2021/3806

Санкт-Петербург 2025

Цель работы: Знакомство с многопоточным программированием и методами синхронизации потоков средствами POSIX.

Задание на лабораторную работу:

1. С помощью таблицы вариантов заданий выбрать граф запуска потоков в соответствии с номером варианта. Вершины графа являются точками запуска/завершения потоков, дугами обозначены сами потоки. Длину дуги следует интерпретировать как ориентировочное время выполнения потока. В процессе своей работы каждый поток должен в цикле выполнять два действия:
   i. выводить букву имени потока в консоль;
   ii. вызывать функцию computation() для выполнения вычислений, требующих задействования ЦП на длительное время. Эта функция уже написана и подключается из заголовочного файла lab2.h, изменять ее не следует.
2. В соответствии с вариантом выделить на графе две группы с выполняющимися параллельно потоками. В первой группе потоки не синхронизированы, параллельное выполнение входящих в группу потоков происходит за счет планировщика задач (см. примеры 1 и 2). Вторая группа синхронизирована семафорами и потоки внутри группы выполняются в строго зафиксированном порядке: входящий в групу поток передает управление другому потоку после каждой итерации цикла (см. пример 3 и задачу производителя и потребителя). Таким образом потоки во второй группе выполняются в строгой очередности.
3. С использованием средств POSIX реализовать программу для последовательно-параллельного выполнения потоков в ОС Linux или Mac OS X. Запрещается использовать какие-либо библиотеки и модули, решающие задачу кроссплатформенной разработки многопоточных приложений (std::thread, Qt Thread, Boost Thread и т.п.), а также функции приостановки

выполнения программы за исключением pthread_yield(). Для этого необходимо написать код в файле lab2.cpp:

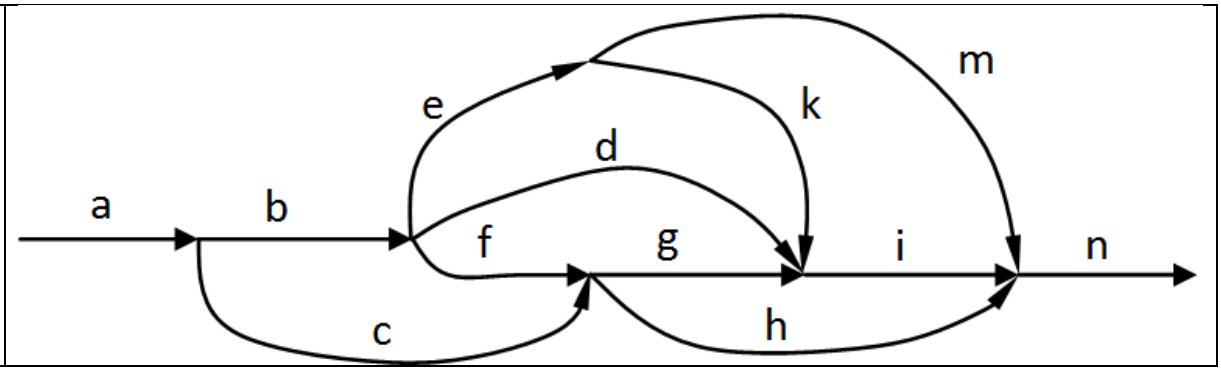
i. Функция unsigned int lab2_thread_graph_id() должна возвращать номер графа запуска потоков, полученный из таблицы вариантов заданий.
ii. Функция const char* lab2_unsynchronized_threads() должна возвращать строку, состоящую из букв потоков, выполняющихся параллельно без синхронизации (см. примеры в файлах lab2.cpp и lab2_ex.cpp).
iii. Функция const char* lab2_sequential_threads() должна возвращать строку, состоящую из букв потоков, выполняющихся параллельно в строгой очередности друг за другом (см. примеры в файлах lab2.cpp и lab2_ex.cpp).
iv. Функция int lab2_init() заменяет собой функцию main(). В ней необходимо реализовать запуск потоков, инициализацию вспомогательных переменных (мьютексов, семафоров и т.п.). Перед выходом из функции lab2_init() необходимо убедиться, что все запущенные потоки завершились. Возвращаемое значение: 0 - работа функции завершилась успешно, любое другое числовое значение - при выполнении функции произошла критическая ошибка.
v. Добавить любые другие необходимые для работы программы функции, переменные и подключаемые файлы.
vi. Создавать функцию main() не нужно. В проекте уже имеется готовая функция main(), изменять ее нельзя. Она выполняет единственное действие: вызывает функцию lab2_init().
vii. Не следует изменять какие-либо файлы, кроме lab2.cpp. Также не следует создавать новые файлы и писать в них код, поскольку код из этих файлов не будет использоваться во время тестирования.

4. Подготовить отчет о выполнении лабораторной работы и загрузить его под именем report.pdf в репозиторий. В случае использования системы компьютерной верстки LaTeX также загрузить исходный файл report.tex.

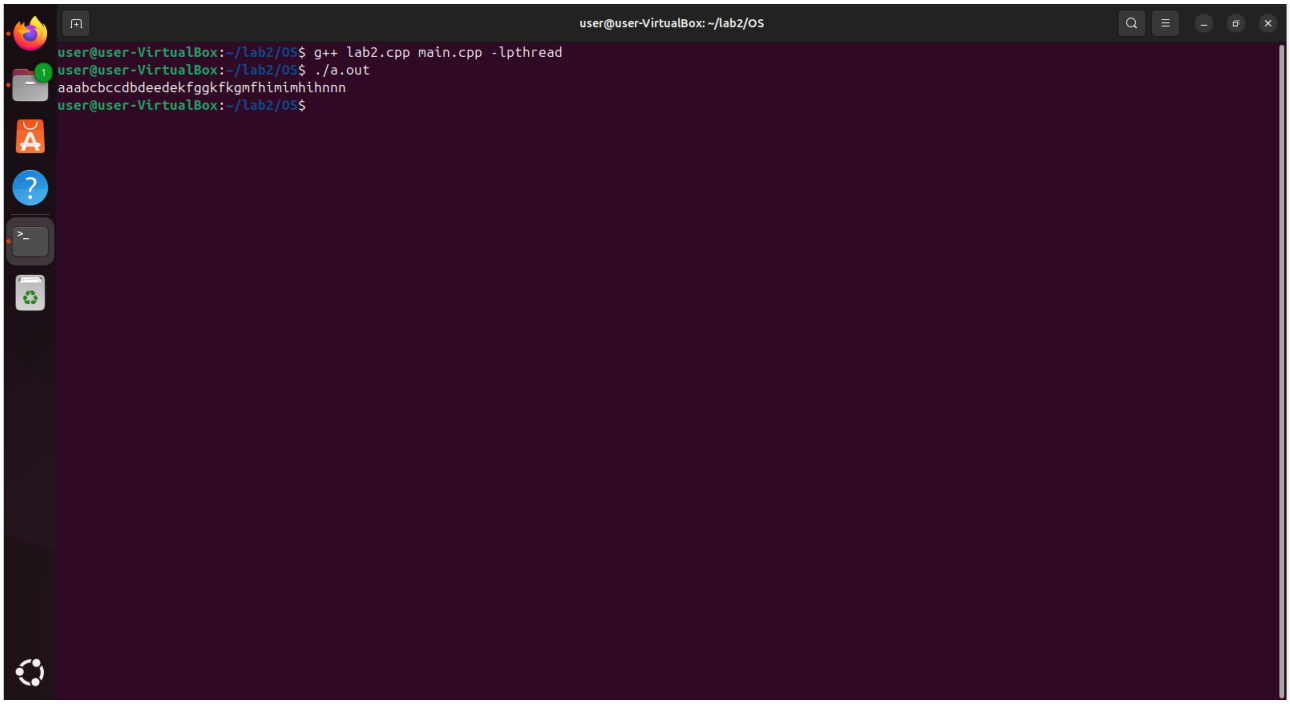
Вариант задания:

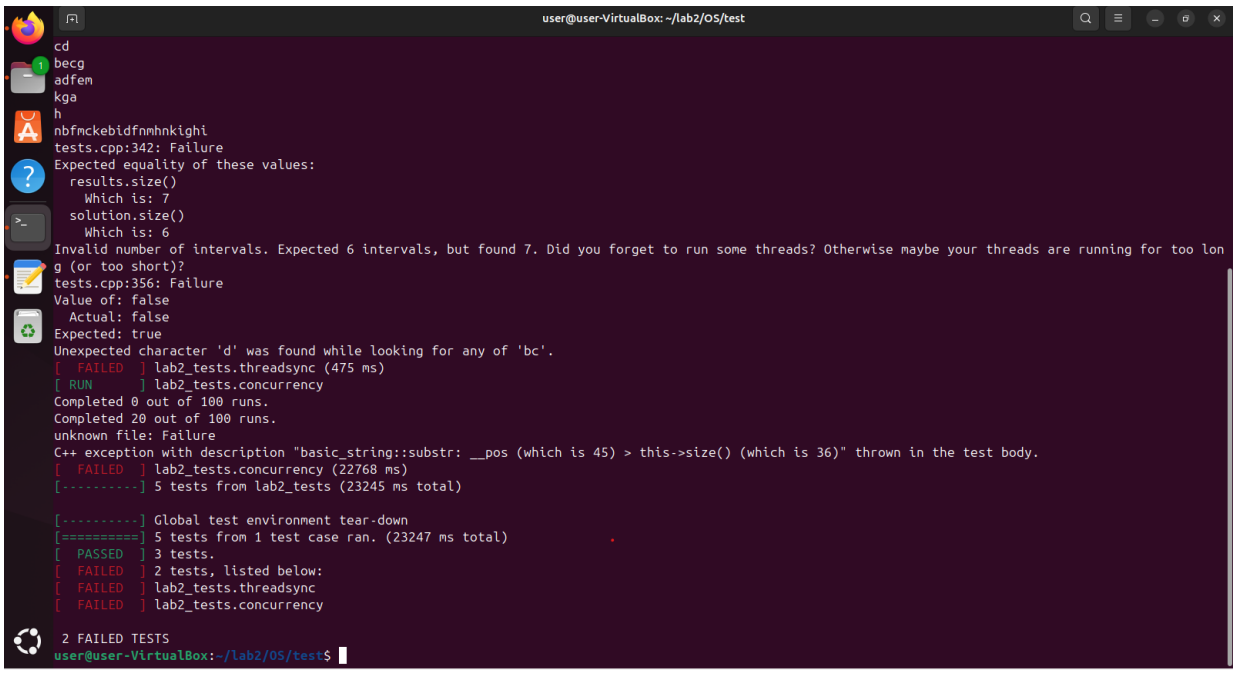
Номер варианта	Номер графа запуска потоков	Несинхронизированные потоки	Потоки с чередованием
9	13	cdef	him

Варианты графов запуска потоков:

9	

1. Результат выполнения работы:





Исходный код программы:

```
#include "lab2.h"
#include <pthread.h>
#include "lab2.h"
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <vector>
#include <string>

// Кол-во итераций каждого потока
const int NUM_ITERATIONS = 3;

// Семафоры для синхронизации
sem_t sem_b, sem_c, sem_d, sem_e, sem_f, sem_g, sem_h, sem_i, sem_n;

// Буфер для вывода
std::string output;
pthread_mutex_t output_mutex = PTHREAD_MUTEX_INITIALIZER;

void write_char(char c) {
    pthread_mutex_lock(&output_mutex);
    std::cout << c << std::flush;
    output += c;
    pthread_mutex_unlock(&output_mutex);
}

// Потоки A и B — независимые
void* thread_a(void*) {
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        computation();
```

```
        write_char('a');
    }
    return nullptr;
}

void* thread_b(void*) {
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        computation();
        write_char('b');
    }
    sem_post(&sem_b); // запускаем C
    sem_post(&sem_b); // C требует 2 поста
    return nullptr;
}

// Поток C — после B
void* thread_c(void*) {
    sem_wait(&sem_b);
    sem_wait(&sem_b);
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        computation();
        write_char('c');
    }
    sem_post(&sem_c); // запускаем D
    sem_post(&sem_c);
    sem_post(&sem_c);
    return nullptr;
}

// Поток D — после C
void* thread_d(void*) {
    sem_wait(&sem_c);
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        computation();
        write_char('d');
    }
    sem_post(&sem_d); // запускаем E
    sem_post(&sem_d);
    sem_post(&sem_d);
    return nullptr;
}

// Поток E — после D
void* thread_e(void*) {
    sem_wait(&sem_d);
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        computation();
        write_char('e');
    }
    sem_post(&sem_e); // запускаем F и G
    sem_post(&sem_e);
    return nullptr;
```

```
}

// Поток F — после E
void* thread_f(void*) {
    sem_wait(&sem_e);
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        computation();
        write_char('f');
    }
    sem_post(&sem_f); // завершение
    return nullptr;
}

// Поток G — после E
void* thread_g(void*) {
    sem_wait(&sem_e);
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        computation();
        write_char('g');
    }
    sem_post(&sem_g); // запускает H
    return nullptr;
}

// Поток H — после G, запускает I
void* thread_h(void*) {
    sem_wait(&sem_g);
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        computation();
        write_char('h');
        sem_post(&sem_h); // запускает I
        sem_wait(&sem_i); // ждёт I, чтобы продолжить
    }
    return nullptr;
}

// Поток I — запускается после H, запускает M
void* thread_i(void*) {
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        sem_wait(&sem_h);
        computation();
        write_char('i');
        sem_post(&sem_i); // отпускает H
        sem_post(&sem_i); // запускает M
    }
    return nullptr;
}

// Поток M — после I, запускает N
void* thread_m(void*) {
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        sem_wait(&sem_i);
```

```
        computation();
        write_char('m');
        sem_post(&sem_n); // запускает N
    }
    return nullptr;
}

// Поток N — после M
void* thread_n(void*) {
    for (int i = 0; i < NUM_ITERATIONS; ++i) {
        sem_wait(&sem_n);
        computation();
        write_char('n');
    }
    return nullptr;
}

// Реализация API
const char* lab2_unsynchronized_threads() {
    return "ab";
}

const char* lab2_sequential_threads() {
    return "cdefghimn";
}

unsigned int lab2_thread_graph_id() {
    return 123; // Любой уникальный номер
}

int lab2_init() {
    pthread_t threads[11];

    sem_init(&sem_b, 0, 0);
    sem_init(&sem_c, 0, 0);
    sem_init(&sem_d, 0, 0);
    sem_init(&sem_e, 0, 0);
    sem_init(&sem_f, 0, 0);
    sem_init(&sem_g, 0, 0);
    sem_init(&sem_h, 0, 0);
    sem_init(&sem_i, 0, 0);
    sem_init(&sem_n, 0, 0);

    pthread_create(&threads[0], nullptr, thread_a, nullptr);
    pthread_create(&threads[1], nullptr, thread_b, nullptr);
    pthread_create(&threads[2], nullptr, thread_c, nullptr);
    pthread_create(&threads[3], nullptr, thread_d, nullptr);
    pthread_create(&threads[4], nullptr, thread_e, nullptr);
    pthread_create(&threads[5], nullptr, thread_f, nullptr);
    pthread_create(&threads[6], nullptr, thread_g, nullptr);
    pthread_create(&threads[7], nullptr, thread_h, nullptr);
    pthread_create(&threads[8], nullptr, thread_i, nullptr);
```

```
    pthread_create(&threads[9], nullptr, thread_m, nullptr);
    pthread_create(&threads[10], nullptr, thread_n, nullptr);

    for (auto& t : threads) {
        pthread_join(t, nullptr);
    }

    std::cout << std::endl;
    return 0;
}
```

Выводы

В ходе выполнения лабораторной работы была реализована многопоточная программа. Согласно заданному графу потоков, выполнена синхронизация потоков при помощи семафоров и других примитивов синхронизации, обеспечивающая как независимое выполнение некоторых потоков, так и строгую последовательность в группах потоков.