ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

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

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2

Разработка многопоточного приложения средствами POSIX в ОС Linux или Mac OS

по курсу: Операционные системы

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. №	4233К	11.02.2025	С. В. Голанова
		подпись, дата	инициалы, фамилия

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.

Последовательное выполнение потоков может обеспечиваться как за счет использования семафоров, так и с помощью функции pthread_join(). Запускать потоки можно все сразу в функции lab2_init(), а можно и по одному (или группами) из других потоков. Количество запускаемых потоков должно быть равно количеству дуг на графе плюс один (для потока main). Запрещается завершать поток в конце интервала, а затем заново его запускать.

В процессе своей работы каждый поток выводит свою букву в консоль. Оценка правильности выполнения лабораторной работы осуществляется следующим образом. Если потоки **a** и **b** согласно графу должны выполняться одновременно (параллельно), то в консоли должна присутствовать последовательность вида **abababab** (или схожая, например, **aabbba**); если потоки выполняются последовательно, то в консоли присутствует последовательность вида **aaaaabbbbbb**, причем после появления первой буквы **b**, буква **a** больше не должна появиться в консоли.

Количество букв, выводимых каждым потоком в консоль, должно быть пропорционально числу интервалов (длине дуги), соответствующей данному потоку на графе. При этом количество символов, выводимых в консоль каждым из потоков, должно быть не меньше, чем 3Q и не больше, чем 5Q, где Q - количество интервалов на графе, в течении которых выполняется поток. Множитель перед величиной Q следует выбрать одинаковым для всех потоков, задав его равным 3, 4 или 5. Ожидается, что на каждом интервале своей работы поток выводит одинаковое количество символов в консоль.

Граф запуска потоков

Вариант на лабораторную работу 28 представлен в Таблице 1.

Таблица 1 – Вариант на лабораторную работу

Номер варианта	**Номер графа запуска потоков**	**Интервалы с несинхронизированными потоками**	**Интервалы с чередованием потоков**
28	12	ghi	defg

Граф запуска потоков 12, представлен на рисунке 1 ниже.

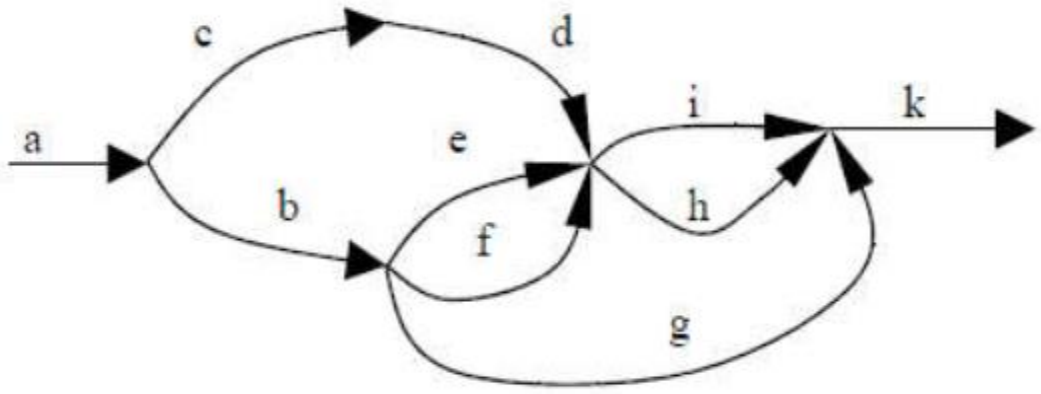

Рисунок 1 – Граф запуска потоков

Результат выполнения работы

В результате выполнения работы была написана программа.

Программа, реализует указанную на графе последовательность запуска потоков, запускает 10 потоков: *a, b, c, d, e, f, g, i, h,* и *k*. Работу программы можно разбить на пять временных интервалов:

1. С момента времени T_0 до T_1 работает только поток *a*.
2. С T_1 до T_2 параллельно работают потоки *b* и *c*.
3. С T_2 до T_3 параллельно работают потоки *d*, *e*, *f* и *g*.
4. С T_3 до T_4 параллельно работают потоки *g*, *i* и *h*.
5. С T_4 до T_5 работает только поток *k*.

Потоки b и c выполняются параллельно без использования средств синхронизации. Потоки *d*, *e*, *f* и *g* выполняются параллельно в строгой очередности друг за другом за счет использования семафоров, а потоки *g*, *i* и *h* выполняются параллельно без использования средств синхронизации.

Возможные выводы программы представлены ниже:

```
vboxuser@LinuxOS:~/lab2/os-task2-linesofia14$ ./lab2_test
aaabcbbccdefgdefgdefgihgihighgkkk
vboxuser@LinuxOS:~/lab2/os-task2-linesofia14$ ./lab2_test
aaabcbbccdefgdefgdefggihhihgigkkk
vboxuser@LinuxOS:~/lab2/os-task2-linesofia14$ ./lab2_test
aaabcbcbcdefgdefgdefggihgihighkkk
vboxuser@LinuxOS:~/lab2/os-task2-linesofia14$ ./lab2_test
aaabcbbccdefgdefgdefggihghhiigkkk
vboxuser@LinuxOS:~/lab2/os-task2-linesofia14$ ./lab2_test
aaabcbbccdefgdefgdefggihgigihhkkk
```

Данные полученные в результате тестирования программы приведены ниже на рисунке 2.

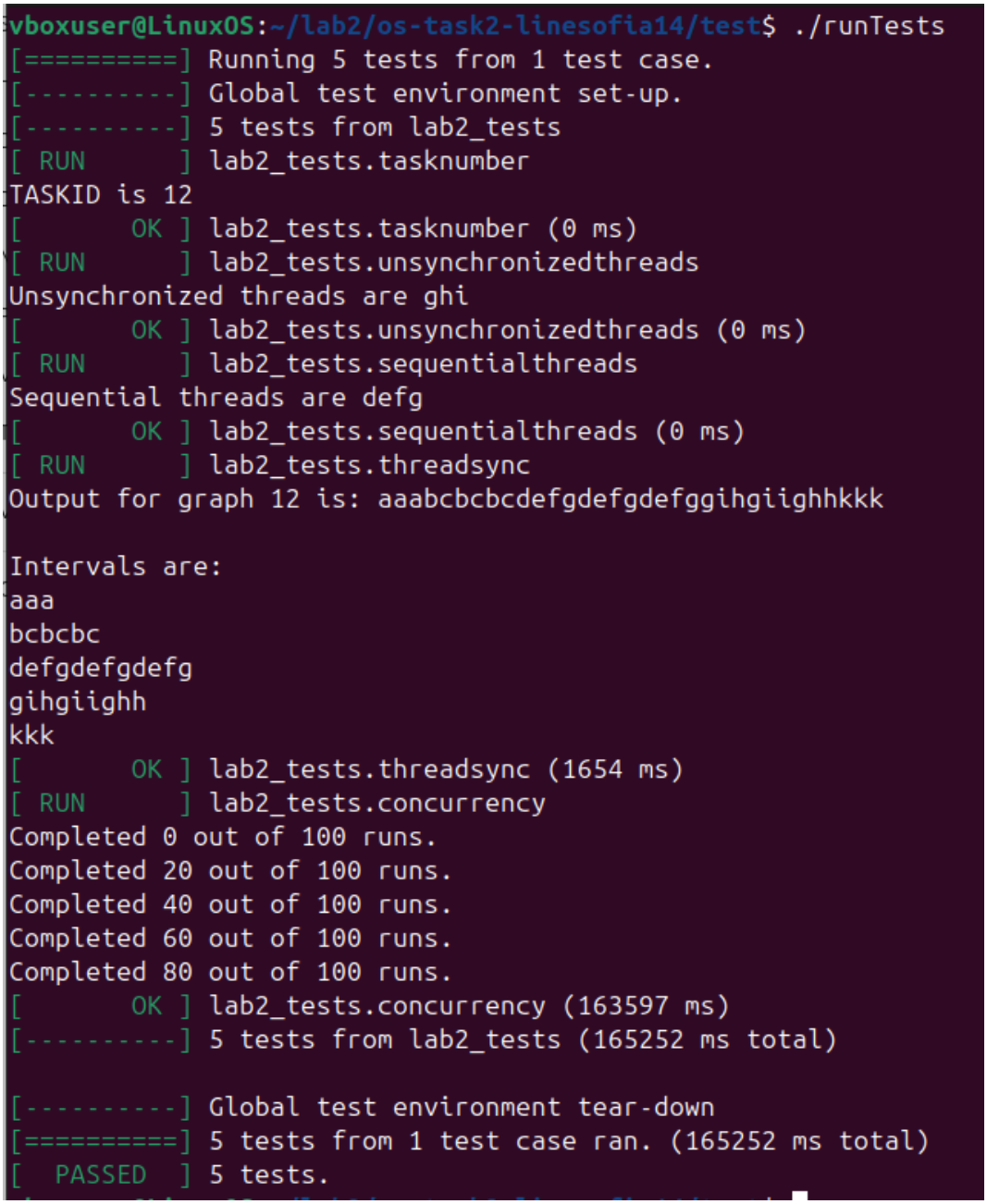

Рисунок 2 – Результат прохождения тестов

Исходный код программы с комментариями

Ниже приведено содержание файла lab2.cpp.

```
#include "lab2.h"
#include <cstring>
#include <semaphore.h>
#define NUMBER_OF_THREADS 10

// thread identifiers
pthread_t tid[NUMBER_OF_THREADS];
// critical section lock
pthread_mutex_t lock;
```

```
// semaphores for sequential threads
sem_t semD, semE, semF, semG, semBC;

unsigned int lab2_thread_graph_id()
{
    return 12;
}

const char* lab2_unsynchronized_threads()
{
    return "ghi";
}

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

// Функции потоков
void *thread_a(void *ptr);
void *thread_b(void *ptr);
void *thread_c(void *ptr);
void *thread_d(void *ptr);
void *thread_e(void *ptr);
void *thread_f(void *ptr);
void *thread_g(void *ptr);
void *thread_i(void *ptr);
void *thread_h(void *ptr);
void *thread_k(void *ptr);

void *thread_a(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        pthread_mutex_lock(&lock);
        std::cout << "a" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }

    int err = pthread_create(&tid[1], NULL, thread_b, NULL);
    if (err != 0)
        std::cerr << "Не удается создать поток. Ошибка: " << strerror(err) <<
std::endl;

    err = pthread_create(&tid[2], NULL, thread_c, NULL);
    if (err != 0)
        std::cerr << "Не удается создать поток. Ошибка: " << strerror(err) <<
std::endl;
```

```
    // Ждём завершения обоих потоков
    pthread_join(tid[1], NULL);
    pthread_join(tid[2], NULL);

    return ptr;
}

void *thread_b(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        pthread_mutex_lock(&lock);
        std::cout << "b" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }

    int err = pthread_create(&tid[4], NULL, thread_e, NULL);
    if (err != 0)
        std::cerr << "Не удаётся создать поток. Ошибка: " << strerror(err) <<
std::endl;

    err = pthread_create(&tid[5], NULL, thread_f, NULL);
    if (err != 0)
        std::cerr << "Не удаётся создать поток. Ошибка: " << strerror(err) <<
std::endl;

    err = pthread_create(&tid[6], NULL, thread_g, NULL);
    if (err != 0)
        std::cerr << "Не удаётся создать поток. Ошибка: " << strerror(err) <<
std::endl;

    // Сигнализируем о завершении потока b
    sem_post(&semBC);

    // Ждём завершения потоков e, f и g
    pthread_join(tid[4], NULL);
    pthread_join(tid[5], NULL);
    pthread_join(tid[6], NULL);

    return ptr;
}

void *thread_c(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        pthread_mutex_lock(&lock);
```

```
        std::cout << "c" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }

    // Сигнализируем о завершении потока c
    sem_post(&semBC);

    // Ждём завершения потоков b и c
    sem_wait(&semBC);
    sem_wait(&semBC);

    // Создаём поток d
    int err = pthread_create(&tid[3], NULL, thread_d, NULL);
    if (err != 0)
        std::cerr << "Не удаётся создать поток. Ошибка: " << strerror(err) <<
std::endl;

    return ptr;
}

void *thread_d(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        sem_wait(&semD);
        pthread_mutex_lock(&lock);
        std::cout << "d" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
        sem_post(&semE);
    }

    return ptr;
}

void *thread_e(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        sem_wait(&semE);
        pthread_mutex_lock(&lock);
        std::cout << "e" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
        sem_post(&semF);
    }

    return ptr;
```

```
}

void *thread_f(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        sem_wait(&semF);
        pthread_mutex_lock(&lock);
        std::cout << "f" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
        sem_post(&semG);
    }

    return ptr;
}

void *thread_g(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        sem_wait(&semG);
        pthread_mutex_lock(&lock);
        std::cout << "g" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
        sem_post(&semD);
    }

    int err = pthread_create(&tid[7], NULL, thread_i, NULL);
    if (err != 0)
        std::cerr << "Не удаётся создать поток. Ошибка: " << strerror(err) <<
std::endl;

    err = pthread_create(&tid[8], NULL, thread_h, NULL);
    if (err != 0)
        std::cerr << "Не удаётся создать поток. Ошибка: " << strerror(err) <<
std::endl;

    for (int i = 0; i < 3; ++i) {
        pthread_mutex_lock(&lock);
        std::cout << "g" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }
        // Ждём завершения потоков I и H
    pthread_join(tid[7], NULL);
    pthread_join(tid[8], NULL);
```

```
    err = pthread_create(&tid[9], NULL, thread_k, NULL);
    if (err != 0)
        std::cerr << "Не удаётся создать поток. Ошибка: " << strerror(err) <<
std::endl;

    // Ждём завершения потока K
    pthread_join(tid[9], NULL);

    return ptr;
}

void *thread_i(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        pthread_mutex_lock(&lock);
        std::cout << "i" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }

    return ptr;
}

void *thread_h(void *ptr)
{
    for (int i = 0; i < 3; ++i)
    {
        pthread_mutex_lock(&lock);
        std::cout << "h" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }

    return ptr;
}

void *thread_k(void *ptr)
{
    pthread_mutex_lock(&lock);
    for (int i = 0; i < 3; ++i)
    {
        std::cout << "k" << std::flush;
        computation();
    }
    pthread_mutex_unlock(&lock);

    return ptr;
}
```

```
int lab2_init()
{
    // инициализация мьютекса
    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        std::cerr << "Ошибка инициализации мьютекса" << std::endl;
        return 1;
    }

    // инициализация семафоров
    if (sem_init(&semD, 0, 1) != 0)
    {
        std::cerr << "Ошибка инициализации семафора #1" << std::endl;
        return 1;
    }
    if (sem_init(&semE, 0, 0) != 0)
    {
        std::cerr << "Ошибка инициализации семафора #2" << std::endl;
        return 1;
    }
    if (sem_init(&semF, 0, 0) != 0)
    {
        std::cerr << "Ошибка инициализации семафора #3" << std::endl;
        return 1;
    }
    if (sem_init(&semG, 0, 0) != 0)
    {
        std::cerr << "Ошибка инициализации семафора #4" << std::endl;
        return 1;
    }
    if (sem_init(&semBC, 0, 0) != 0)
    {
        std::cerr << "Ошибка инициализации семафора semBC" << std::endl;
        return 1;
    }

    //    // Запускаем основной поток
    int err = pthread_create(&tid[0], NULL, thread_a, NULL);
    if (err != 0)
    {
        std::cerr << "Не удалось создать поток. Ошибка: " << strerror(err) <<
std::endl;
        return 1;
    }

    // Ожидаем завершения основного потока
    pthread_join(tid[0], NULL);
```

```
    // Освобождаем ресурсы
    pthread_mutex_destroy(&lock);
    sem_destroy(&semD);
    sem_destroy(&semE);
    sem_destroy(&semF);
    sem_destroy(&semG);
    sem_destroy(&semBC);

    std::cout << std::endl;

    // Завершаем программу успешно
    return 0;
}
```

Выводы

В ходе выполнения лабораторной работы были получены знания о принципах многопоточного программирования и методах синхронизации потоков с использованием средств POSIX. В частности, приобретён практический опыт создания и управления потоками посредством мьютексов и семафоров, обеспечивающих координацию выполнения потоков и защиту общих ресурсов. Разработана программа для последовательного и параллельного исполнения потоков в операционной системе Linux. Также был изучен практические подходы к выявлению и устранению распространенных проблем многопоточности, что способствует написанию эффективного и потокобезопасного кода.