

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. №

M011

М.А. Долгова

\_\_\_\_\_  
подпись, дата

\_\_\_\_\_  
инициалы, фамилия

Санкт-Петербург 2023

## Цель работы

Знакомство с многопоточным программированием и методами синхронизации потоков средствами POSIX.

## Задание

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:
  - a. выводить букву имени потока в консоль;
  - b. 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 группу 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`:
  - a. Функция `unsigned int lab2_thread_graph_id()` должна возвращать номер графа запуска потоков, полученный из таблицы вариантов заданий.
  - b. 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`).
  - c. 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`).
  - d. 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.
  - e. Добавить любые другие необходимые для работы программы функции, переменные и подключаемые файлы.

- f. Создавать функцию `main()` не нужно. В проекте уже имеется готовая функция `main()`, изменять ее нельзя. Она выполняет единственное действие: вызывает функцию `lab2_init()`.
- g. 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.

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 а 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 aaaaabbbbb, причем после появления первой буквы b, буква а 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.

## Вариант 9

Номер варианта	Номер графа запуска потоков	Интервалы с несинхронизированными потоками	Интервалы с чередованием потоков
9	13	cdef	him

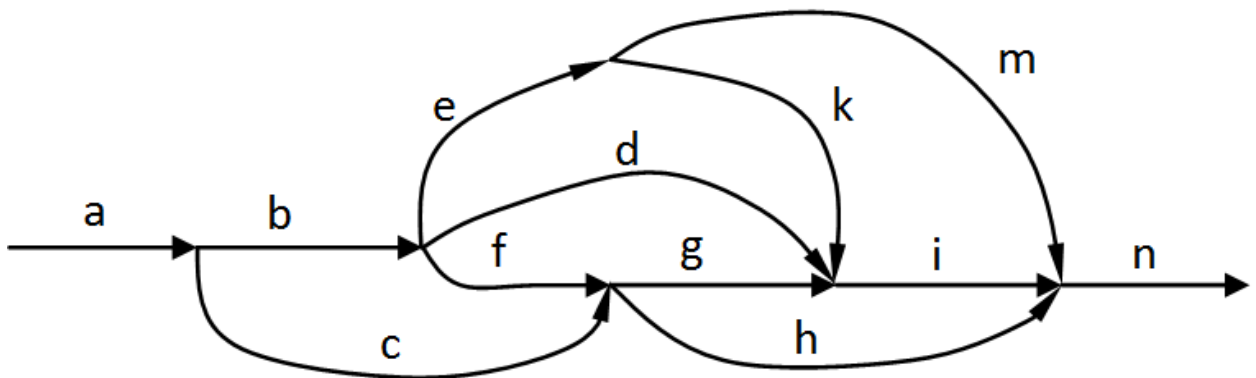
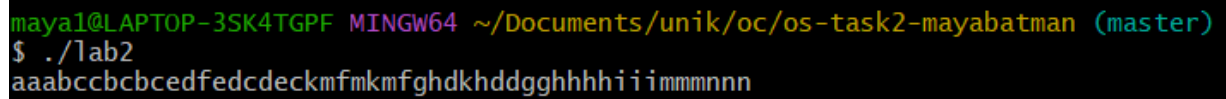


Рис.1. Граф запуска потоков.

## Результат выполнения задания

В разработанной программе потоки запускаются из других потоков, кроме первого и последнего потоков. В результате запуска программы получаем следующий вывод в консоль:



```
maya1@LAPTOP-3SK4TGPF MINGW64 ~/Documents/unik/os-task2-mayabatman (master)
$ ./lab2
aaabccbcbcedfedcdeckmfkmfghdkhddgghhhhiimmnnn
```

Рис.2. Вывод в консоль.

Именно такой результат мы и ожидали.

Можно наблюдать 6 временных интервалов, где одновременно запущены потоки:

1. a (aaa)
2. bc (bccbcb)
3. cdef (cedfedcdec)
4. dgkhm (kmfkmfghdkhddggh)
5. him (hhhiimm)
6. n (nnn)

### Файл lab2.cpp:

```
#include "lab2.h"
#include <cstring>
#include <semaphore.h>
//
// lab2 code should be located here!
//

#define NUMBER_OF_THREADS 12

// thread identifiers
pthread_t tid[NUMBER_OF_THREADS];
// critical section lock
pthread_mutex_t lock;
// semaphores for sequential threads
sem_t semH, semI, semM;

int err;

unsigned int lab2_thread_graph_id()
{
    return 13;
}

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

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

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_h(void* ptr);
void* thread_i(void* ptr);
void* thread_k(void* ptr);
void* thread_m(void* ptr);
void* thread_n(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();
    }
    err = pthread_create(&tid[1], NULL, thread_b, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;
    err = pthread_create(&tid[2], NULL, thread_c, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;

    // wait for thread C to finish
    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();
    }

    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();
    }
    // wait for thread B to finish
    pthread_join(tid[1], NULL);

    err = pthread_create(&tid[3], NULL, thread_e, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;
    err = pthread_create(&tid[4], NULL, thread_d, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;
    err = pthread_create(&tid[5], NULL, thread_f, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;

```

```

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

    // wait for thread D to finish
    pthread_join(tid[4], NULL);

    return ptr;
}

void* thread_d(void* ptr)
{
    for (int i = 0; i < 3; ++i) {
        pthread_mutex_lock(&lock);
        std::cout << "d" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }
    // wait for thread E to finish
    pthread_join(tid[3], NULL);
    // wait for thread F to finish
    pthread_join(tid[5], NULL);

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

    err = pthread_create(&tid[10], NULL, thread_i, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;
    err = pthread_create(&tid[8], NULL, thread_g, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;
    // wait for thread G to finish
    pthread_join(tid[8], NULL);
    // start thread G

    // wait for thread G to finish
    //pthread_join(tid[8], NULL);
    // wait for thread k to finish
    //pthread_join(tid[6], NULL);
    // wait for thread M to finish
    pthread_join(tid[7], NULL);
    //wait for thread H to finish
    pthread_join(tid[9], NULL);
    //wait for thread I to finish
    pthread_join(tid[10], NULL);

    //sem_post(&semH);
    return ptr;
}

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

```

```

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

    err = pthread_create(&tid[6], NULL, thread_k, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;
    err = pthread_create(&tid[7], NULL, thread_m, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;

    // wait for thread F to finish
    //pthread_join(tid[5], NULL);

    return ptr;
}

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

    err = pthread_create(&tid[9], NULL, thread_h, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;

    return ptr;
}

void* thread_g(void* ptr)
{
    for (int i = 0; i < 3; ++i) {
        pthread_mutex_lock(&lock);
        std::cout << "g" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }
    // wait for thread K to finish
    pthread_join(tid[6], NULL);

    sem_post(&semH);
    return ptr;
}

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

    sem_wait(&semH);
    // perform computations
    for (int i = 0; i < 3; ++i) {
        pthread_mutex_lock(&lock);
        std::cout << "h" << std::flush;
        pthread_mutex_unlock(&lock);
    }
}

```

```

        computation();
    }
    sem_post(&semI);
    //sem_post(&semI);
    return ptr;
}

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

    sem_wait(&semM);
    // perform computations
    for (int i = 0; i < 3; ++i) {
        pthread_mutex_lock(&lock);
        std::cout << "m" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }

    return ptr;
}

void* thread_i(void* ptr)
{
    sem_wait(&semI);
    // perform computations
    for (int i = 0; i < 3; ++i) {
        pthread_mutex_lock(&lock);
        std::cout << "i" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }
    sem_post(&semM);
    return ptr;
}

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

void* thread_n(void* ptr)
{
    // perform computations
    for (int i = 0; i < 3; ++i) {
        pthread_mutex_lock(&lock);
        std::cout << "n" << std::flush;
        pthread_mutex_unlock(&lock);
        computation();
    }
    return ptr;
}

```



```

}

int lab2_init()
{
    // initilize mutex
    if (pthread_mutex_init(&lock, NULL) != 0) {
        std::cerr << "Mutex init failed" << std::endl;
        return 1;
    }
    // initialize semaphores
    // THIS CODE WILL NOT RUN ON MacOS!
    // MacOS doesn't support unnamed semaphores, so named semaphores should be used
    instead
    if (sem_init(&semH, 0, 0) != 0) {
        std::cerr << "Semaphore #1 init failed" << std::endl;
        return 1;
    }
    if (sem_init(&semI, 0, 0) != 0) {
        std::cerr << "Semaphore #2 init failed" << std::endl;
        return 1;
    }
    if (sem_init(&semM, 0, 0) != 0) {
        std::cerr << "Semaphore #2 init failed" << std::endl;
        return 1;
    }

    // start the first thread
    err = pthread_create(&tid[0], NULL, thread_a, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;

    // ... and wait for it to finish
    pthread_join(tid[0], NULL);

    err = pthread_create(&tid[11], NULL, thread_n, NULL);
    if (err != 0)
        std::cerr << "Can't create thread. Error: " << strerror(err) << std::endl;

    // wait for thread N to finish
    pthread_join(tid[11], NULL);

    // free resources
    pthread_mutex_destroy(&lock);
    sem_destroy(&semH);
    sem_destroy(&semI);
    sem_destroy(&semM);
    std::cout << std::endl;
    // success
    return 0;
}

```

## Выводы

В результате выполнения лабораторной работы был получен опыт работы с потоками и запуск их с помощью мьютексов и семафоров.