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 №3

## Синхронизация потоков средствами WinAPI

по курсу: Операционные системы

РАБОТУ ВЫПОЛНИЛ

| СТУДЕНТ гр. № 4233К | 12.02.25 | С. В. Голанова |
|---|---|---|
| | подпись, дата | инициалы, фамилия |

Санкт-Петербург 2025

**Цель работы**

Знакомство с многопоточным программированием и методами синхронизации потоков средствами Windows API.

**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:
   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 lab3.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. 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. задачу производителя и потребителя). Таким образом потоки во второй группе выполняются в строгой очередности.
3. С использованием средств Windows API реализовать программу для последовательно-параллельного выполнения потоков в ОС Windows. 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 (например, Sleep(), SwitchToThread() и подобные). Для этого необходимо написать код в файле lab3.cpp:
   i. Функция unsigned int lab3_thread_graph_id() должна возвращать номер графа запуска потоков, полученный из таблицы вариантов заданий.
   ii. Функция const char* lab3_unsynchronized_threads() должна возвращать строку, состоящую из букв потоков, выполняющихся параллельно без синхронизации.
   iii. Функция const char* lab3_sequential_threads() должна возвращать строку, состоящую из букв потоков, выполняющихся параллельно в строгой

очередности друг за другом.

iv. Функция int lab3_init() заменяет собой функцию main(). В ней необходимо реализовать запуск потоков, инициализацию вспомогательных переменных (мьютексов, семафоров и т.п.). Перед выходом из функции lab3_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 lab3_init().

vii. Не следует изменять какие-либо файлы, кроме lab3.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 WaitForSingleObject(). Запускать потоки можно все сразу в функции lab3_init(), а можно и по одному (или группами) из других потоков.

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.

**Вариант задания**

| Номер варианта | Номер графа запуска потоков | Несинхронизированные потоки | Потоки с чередованием |
|---|---|---|---|
| 28 | 8 | bce | deg |

Вариант запуска потоков приведен на рисунке 1.

8

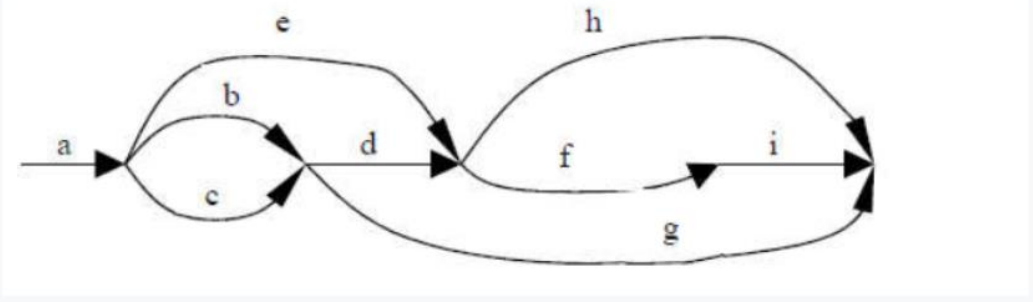


Рисунок 1 – Вариант запуска потоков

**Результат выполнения работы**

В результате выполнения работы была написана программа.

Программа, реализует указанную на графе последовательность запуска потоков, запускает 9 потоков: *a, b, c, d, e, f, g, i,* и *h*. 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, c* и *e*.
3. С $T_2$ до $T_3$ параллельно работают потоки *d, e* и *g*.
4. С $T_3$ до $T_4$ параллельно работают потоки *f, g* и *h*.
5. С $T_4$ до $T_5$ параллельно работают потоки *g, h* и *i*.

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

Возможные выводы программы представлены ниже:

```
aaabeccbcebedegdegdegghfgfhfggihhhgihgi
aaabcebecbecdegdegdegghffghghfhgiihgghi
aaabcecbbceedegdegdegghfghfhgfhgihgihgi
aaabecebcbcedegdegdegghfgfhhgfhgihghgii
```

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

```
▶ Run cp ../test/task*.txt test
[==========] Running 5 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 5 tests from lab3_tests
[ RUN ] lab3_tests.tasknumber
TASKID is 8
[ OK ] lab3_tests.tasknumber (0 ms)
[ RUN ] lab3_tests.unsynchronizedthreads
Unsynchronized threads are bce
[ OK ] lab3_tests.unsynchronizedthreads (0 ms)
[ RUN ] lab3_tests.sequentialthreads
Sequential threads are deg
[ OK ] lab3_tests.sequentialthreads (0 ms)
[ RUN ] lab3_tests.threadsync
Output for graph 8 is: aaabcebcbecedegdegdeggfhhgfhfghgihgihig

Intervals are:
aaa
bcebcbece
degdegdeg
gfhhgfhfg
hgihgihig
[ OK ] lab3_tests.threadsync (1495 ms)
[ RUN ] lab3_tests.concurrency
Completed 0 out of 100 runs.
Completed 20 out of 100 runs.
Completed 40 out of 100 runs.
Completed 60 out of 100 runs.
Completed 80 out of 100 runs.
[ OK ] lab3_tests.concurrency (161832 ms)
[----------] 5 tests from lab3_tests (163327 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test case ran. (163327 ms total)
[ PASSED ] 5 tests.
```

Рисунок 2 – Результат прохождения тестов

**Исходный код программы с комментариями**

```
#include "lab3.h"
#include <iostream>
#include <string>
#include <windows.h>
```

```cpp
unsigned int lab3_thread_graph_id() { return 8; }

const char* lab3_unsynchronized_threads() { return "bce"; }

const char* lab3_sequential_threads() { return "deg"; }

CRITICAL_SECTION g_coutLock;

//Семафоры
HANDLE semD, semE, semG;
HANDLE sem_g_done, sem_c_done, sem_m_done, sem_k_done;

//Потоки
HANDLE a_thread;
HANDLE b_thread;
HANDLE c_thread;
HANDLE d_thread;
HANDLE e_thread;
HANDLE f_thread;
HANDLE g_thread;
HANDLE h_thread;
HANDLE i_thread;

// Объявление функций (прототипы)
DWORD WINAPI thread_a(LPVOID);
DWORD WINAPI thread_b(LPVOID);
DWORD WINAPI thread_c(LPVOID);
DWORD WINAPI thread_e(LPVOID);
DWORD WINAPI thread_f(LPVOID);
DWORD WINAPI thread_g(LPVOID);
DWORD WINAPI thread_i(LPVOID);
DWORD WINAPI thread_h(LPVOID);

//Безопасное выведение
void print_safe(const char* str) {
    EnterCriticalSection(&g_coutLock);
    std::cout << str << std::flush;
    LeaveCriticalSection(&g_coutLock);
}

DWORD WINAPI thread_a(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        print_safe("a");
        computation();
    }

    b_thread = CreateThread(NULL, 0, thread_b, NULL, 0, NULL);
    if (b_thread == NULL) {
        std::cerr << "CreateThread_b error: " << GetLastError() << std::endl;
        return 1;
    }

    c_thread = CreateThread(NULL, 0, thread_c, NULL, 0, NULL);
    if (c_thread == NULL) {
        std::cerr << "CreateThread_c error: " << GetLastError() << std::endl;
        return 1;
    }

    e_thread = CreateThread(NULL, 0, thread_e, NULL, 0, NULL);
    if (e_thread == NULL) {
        std::cerr << "CreateThread_e error: " << GetLastError() << std::endl;
        return 1;
    }
    //Ожидание потоков
```

```
    WaitForSingleObject(b_thread, INFINITE);
    WaitForSingleObject(c_thread, INFINITE);
    WaitForSingleObject(e_thread, INFINITE);
    return 0;
}

DWORD WINAPI thread_b(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        print_safe("b");
        computation();
    }
    return 0;
}

DWORD WINAPI thread_c(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        print_safe("c");
        computation();
    }
    return 0;
}

DWORD WINAPI thread_d(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        WaitForSingleObject(semD, INFINITE);// Ждем сигнала от потока D
        print_safe("d");
        computation();
        ReleaseSemaphore(semE, 1, NULL); // Сигнализируем потоку E
    }
    return 0;
}

DWORD WINAPI thread_e(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        print_safe("e");
        computation();
    }

    WaitForSingleObject(b_thread, INFINITE);
    WaitForSingleObject(c_thread, INFINITE);

    // создаем дочерние потоки
    d_thread = CreateThread(NULL, 0, thread_d, NULL, 0, NULL);
    if (d_thread == NULL) {
        std::cerr << "CreateThread_d error: " << GetLastError() << std::endl;
        return 1;
    }
    // создаем дочерние потоки
    g_thread = CreateThread(NULL, 0, thread_g, NULL, 0, NULL);
    if (g_thread == NULL) {
        std::cerr << "CreateThread_g error: " << GetLastError() << std::endl;
        return 1;
    }

    for (int i = 0; i < 3; ++i) {
        WaitForSingleObject(semE, INFINITE);// Ждем сигнала от потока E
        print_safe("e");
        computation();
        ReleaseSemaphore(semG, 1, NULL); // Сигнализируем потоку G
    }

    //Ожидание потоков
    WaitForSingleObject(d_thread, INFINITE);
    WaitForSingleObject(g_thread, INFINITE);
```

```
    return 0;
}

DWORD WINAPI thread_f(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        print_safe("f");
        computation();
    }
    return 0;
}

DWORD WINAPI thread_g(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        WaitForSingleObject(semG, INFINITE);// Ждем сигнала от потока G
        print_safe("g");
        computation();
        ReleaseSemaphore(semD, 1, NULL); // Сигнализируем потоку D
    }

    WaitForSingleObject(d_thread, INFINITE);

    // создаем дочерние потоки
    f_thread = CreateThread(NULL, 0, thread_f, NULL, 0, NULL);
    if (f_thread == NULL) {
        std::cerr << "CreateThread_d error: " << GetLastError() << std::endl;
        return 1;
    }
    // создаем дочерние потоки
    h_thread = CreateThread(NULL, 0, thread_h, NULL, 0, NULL);
    if (h_thread == NULL) {
        std::cerr << "CreateThread_g error: " << GetLastError() << std::endl;
        return 1;
    }

    for (int i = 0; i < 3; ++i) {
        print_safe("g");
        computation();
    }

    WaitForSingleObject(f_thread, INFINITE);

    // создаем дочерние потоки
    i_thread = CreateThread(NULL, 0, thread_i, NULL, 0, NULL);
    if (i_thread == NULL) {
        std::cerr << "CreateThread_g error: " << GetLastError() << std::endl;
        return 1;
    }

    for (int i = 0; i < 3; ++i) {
        print_safe("g");
        computation();
    }

    WaitForSingleObject(i_thread, INFINITE);
    WaitForSingleObject(h_thread, INFINITE);

    return 0;
}

DWORD WINAPI thread_i(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        print_safe("i");
        computation();
```

```
    }
    return 0;
}

DWORD WINAPI thread_h(LPVOID) {
    for (int i = 0; i < 3; ++i) {
        print_safe("h");
        computation();
    }

    WaitForSingleObject(f_thread, INFINITE);

    for (int i = 0; i < 3; ++i) {
        print_safe("h");
        computation();
    }

    return 0;
}

void clean_up() {
    if (semD != NULL) {
        CloseHandle(semD);
    }
    if (semE != NULL) {
        CloseHandle(semE);
    }
    if (semG != NULL) {
        CloseHandle(semG);
    }

    DeleteCriticalSection(&g_coutLock);

    if (a_thread != NULL) {
        WaitForSingleObject(a_thread, INFINITE);
        CloseHandle(a_thread);
    }
    if (b_thread != NULL) {
        WaitForSingleObject(b_thread, INFINITE);
        CloseHandle(b_thread);
    }
    if (c_thread != NULL) {
        WaitForSingleObject(c_thread, INFINITE);
        CloseHandle(c_thread);
    }
    if (d_thread != NULL) {
        WaitForSingleObject(d_thread, INFINITE);
        CloseHandle(d_thread);
    }
    if (e_thread != NULL) {
        WaitForSingleObject(e_thread, INFINITE);
        CloseHandle(e_thread);
    }
    if (f_thread != NULL) {
        WaitForSingleObject(f_thread, INFINITE);
        CloseHandle(f_thread);
    }
    if (g_thread != NULL) {
        WaitForSingleObject(g_thread, INFINITE);
        CloseHandle(g_thread);
    }
    if (h_thread != NULL) {
        WaitForSingleObject(h_thread, INFINITE);
        CloseHandle(h_thread);
```

```
    }
    if (i_thread != NULL) {
        WaitForSingleObject(i_thread, INFINITE);
        CloseHandle(i_thread);
    }
}

int lab3_init() {
    InitializeCriticalSection(&g_coutLock);

    semD = CreateSemaphore(NULL, 1, 1, NULL);
    if (semD == NULL) {
        std::cerr << "CreateSemaphore_D error: " << GetLastError() << std::endl;
        clean_up();
        return 1;
    }
    semE = CreateSemaphore(NULL, 0, 1, NULL);
    if (semE == NULL) {
        std::cerr << "CreateSemaphore_E error: " << GetLastError() << std::endl;
        clean_up();
        return 1;
    }
    semG = CreateSemaphore(NULL, 0, 1, NULL);
    if (semG == NULL) {
        std::cerr << "CreateSemaphore_G error: " << GetLastError() << std::endl;
        clean_up();
        return 1;
    }

    // Создаем поток a
    a_thread = CreateThread(NULL, 0, thread_a, NULL, 0, NULL);
    if (a_thread == NULL) {
        std::cerr << "CreateThread_a error: " << GetLastError() << std::endl;
        clean_up();
        return 1;
    }

    // ждем завершения потока
    WaitForSingleObject(a_thread, INFINITE);

    clean_up();
    std::cout << std::endl;
    return 0;
      }
```

**Выводы**

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