

Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина «Операционные среды и системное программирование»

ОТЧЕТ
к лабораторной работе №1
на тему:
«УПРАВЛЕНИЕ ПРОЦЕССАМИ, ПОТОКАМИ, НИТЯМИ»
БГУИР 6-05-0612-02 67

Выполнил студент группы 353503
КОХАН Артём Игоревич

(дата, подпись студента)

Проверил ассистент каф. информатики
Гриценко Никита Юрьевич

(дата, подпись преподавателя)

Минск 2025

СОДЕРЖАНИЕ

1 Индивидуальное задание.....	3
2 Краткие теоритические сведения	4
3 Описание функций программы.....	5
3.1 Функция ThreadFunction.....	5
3.2 Функция MultiplyMatrices	5
3.3 Функция UpdateProgressBar	5
3.4 Функция main	5
4 Пример выполнения программы	6
4.1 Запуск программы и процесс выполнения	6
4.2 Описание работы и результатов	7
Вывод.....	8
Список использованных источников	9
Приложение А (справочное) Исходный код	10

1 ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Целью выполнения лабораторной работы является закрепление и развитие навыков программирования приложений для операционной системы Windows с использованием многопоточности. В ходе выполнения работы необходимо изучить основные этапы жизненного цикла потоков: их порождение, завершение, получение и изменение состояния. Также важно понять, как приоритеты потоков влияют на их выполнение и производительность системы.

Задача лабораторной работы заключается в разработке многопоточного приложения, которое демонстрирует основные принципы управления потоками, а также оценку их производительности в зависимости от установленных приоритетов. Программа должна выполнять следующие функции:

1 Создание потоков с разными приоритетами: запуск нескольких потоков с заранее заданными приоритетами (CRITICAL, HIGHEST, NORMAL, BELOW_NORMAL, LOWEST, IDLE).

2 Выполнение вычислительной задачи: каждый поток должен выполнять вычислительную сложную задачу (умножение матриц), что обеспечит достаточную нагрузку для демонстрации работы потоков.

3 Отображение процесса выполнения: в консоли приложения необходимо в реальном времени отображать прогресс выполнения каждого потока с помощью прогресс-бара.

4 Измерение времени выполнения: для каждого потока необходимо измерять время выполнения задачи и выводить его после завершения потока, чтобы оценить влияние приоритетов на производительность.

5 Закрытие потоков и завершение программы: приложение должно завершать все потоки, корректно закрывая их дескрипторы, и выводить итоговое сообщение о завершении работы всех потоков.

Приложение должно демонстрировать влияние приоритетов на производительность потоков, наглядно показывая, как операционная система распределяет ресурсы между потоками с различными приоритетами.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Приложение может состоять из одного или нескольких процессов. Процесс – это выполнение программы, включающее в себя системные ресурсы и выделенную область памяти. Процессы изолированы друг от друга, что предотвращает прямое взаимодействие и возможные ошибки. В рамках одного процесса могут функционировать несколько потоков. Поток является основной единицей исполнения кода в процессе, которой операционная система выделяет процессорное время. Он может выполнять любой участок кода процесса, в том числе те, которые уже используются другими потоками.

Используя объекты заданий, можно управлять группами процессов как единым целым. Эти объекты – именуемые, защищаемые и могут делиться между процессами, управляя атрибутами связанных с ними процессов. Взаимодействие с объектом задания влияет на все процессы, ассоциированные с этим объектом [1].

Пул потоков представляет собой коллекцию рабочих потоков, которые осуществляют асинхронные вызовы от имени приложения. Это средство позволяет сократить количество создаваемых потоков и облегчить управление ими, повышая тем самым эффективность приложения. Пул потоков используется в основном для уменьшения нагрузки на процессор и улучшения общей производительности.

Нить – это единица выполнения, которую приложение должно запланировать вручную. Нити выполняются в контексте потоков, которые планируют их, и обычно используются для задач, требующих быстрого отклика системы [2].

Критические секции – это механизм синхронизации, который обеспечивает эксклюзивный доступ к коду или ресурсам, предотвращая одновременное выполнение участка кода несколькими потоками. Они важны для предотвращения условий гонки и обеспечения целостности данных в многопоточных приложениях. Критическая секция гарантирует, что только один поток может выполнять защищённый участок кода в любой момент времени, что предотвращает возникновение ошибок и несогласованности данных, которые могут произойти при одновременном доступе к общим данным [3].

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Согласно формулировке задачи, были спроектированы следующие функции программы:

- ThreadFunction;
- MultiplyMatrices;
- UpdateProgressBar;
- main.

3.1 Функция ThreadFunction

Данная функция принимает указатель на индекс потока. Функция совершает следующие действия:

- 1 Устанавливает приоритет потока;
- 2 Выполняет инициализацию и запускает матричное умножение;
- 3 Измеряет время выполнения операции;
- 4 Выводит в консоль результаты выполнения, включая время и приоритет потока.

3.2 Функция MultiplyMatrices

Функция принимает матрицы для умножения и индекс потока. Функция совершает следующие действия:

- 1 Производит матричное умножение;
- 2 Регулярно обновляет прогресс выполнения с помощью функции UpdateProgressBar.

3.3 Функция UpdateProgressBar

Данная функция принимает индекс потока и текущий прогресс выполнения. Функция совершает следующие действия:

- 1 Блокирует вывод в консоль для синхронизации;
- 2 Обновляет строку прогресса в консоли;
- 3 Снимает блокировку вывода после обновления.

3.4 Функция main

Функция совершает следующие действия:

- 1 Инициализирует необходимые механизмы синхронизации;
- 2 Создает потоки для выполнения задачи умножения матриц;
- 3 Ожидает завершение всех потоков;
- 4 Закрывает все потоки и выводит сообщение о завершении работы.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

На рисунке 4.1 показана консоль после запуска программы на одном ядре и завершения всех потоков. Каждый поток отображает свой прогресс выполнения и окончательные результаты, включая время выполнения и приоритет.

```
Test: 1 core/cores

Поток 1 [TIME_CRITICAL] : [=====] 100%
Поток 2 [HIGHEST (2)] : [=====] 100%
Поток 3 [NORMAL (0)] : [=====] 100%
Поток 4 [BELOW_NORMAL] : [=====] 100%
Поток 5 [LOWEST (-2)] : [=====] 100%
Поток 6 [IDLE (-15)] : [=====] 100%
Поток 1 [TIME_CRITICAL (15)] завершен за 2.74132 сек.
Поток 2 [HIGHEST (2)] завершен за 4.93491 сек.
Поток 3 [NORMAL (0)] завершен за 7.96617 сек.
Поток 4 [BELOW_NORMAL (-1)] завершен за 11.0033 сек.
Поток 5 [LOWEST (-2)] завершен за 13.8525 сек.
Поток 6 [IDLE (-15)] завершен за 16.9445 сек.
```

Рисунок 4.1 – Результат выполнения на 1 ядре

На рисунке 4.2 показана консоль после запуска программы на трёх ядрах и завершения всех потоков.

```
Test: 3 core/cores

Поток 1 [TIME_CRITICAL] : [=====] 100%
Поток 2 [HIGHEST (2)] : [=====] 100%
Поток 3 [NORMAL (0)] : [=====] 100%
Поток 4 [BELOW_NORMAL] : [=====] 100%
Поток 5 [LOWEST (-2)] : [=====] 100%
Поток 6 [IDLE (-15)] : [=====] 100%
Поток 1 [TIME_CRITICAL (15)] завершен за 4.24075 сек.
Поток 2 [HIGHEST (2)] завершен за 4.17306 сек.
Поток 3 [NORMAL (0)] завершен за 4.28313 сек.
Поток 4 [BELOW_NORMAL (-1)] завершен за 5.00645 сек.
Поток 5 [LOWEST (-2)] завершен за 5.65965 сек.
Поток 6 [IDLE (-15)] завершен за 6.15839 сек.
```

Рисунок 4.2 – Результат выполнения на 3 ядрах

На рисунке 4.3 показана консоль после запуска программы на шестнадцати ядрах и завершения всех потоков.

Test: 16 core/cores

```
Поток 1 [TIME_CRITICAL] : [=====] 100%
Поток 2 [HIGHEST (2) : [=====] 100%
Поток 3 [NORMAL (0)] : [=====] 100%
Поток 4 [BELOW_NORM] : [=====] 100%
Поток 5 [LOWEST (-2) : [=====] 100%
Поток 6 [IDLE (-15)] : [=====] 100%
Поток 1 [TIME_CRITICAL (15)] завершен за 4.26716 сек.
Поток 2 [HIGHEST (2)] завершен за 4.28303 сек.
Поток 3 [NORMAL (0)] завершен за 4.28774 сек.
Поток 4 [BELOW_NORMAL (-1)] завершен за 4.30504 сек.
Поток 5 [LOWEST (-2)] завершен за 4.2799 сек.
Поток 6 [IDLE (-15)] завершен за 4.31876 сек.
```

Рисунок 4.3 – Результат выполнения на 16 ядрах

4.2 Описание работы и результатов

В процессе выполнения, программа инициализировала шесть потоков, каждому из которых был задан различный уровень приоритета. Эти потоки выполняли матричные вычисления, результаты которых отображены на скриншоте. Как видно из данных, время выполнения потоков различается в зависимости от их приоритета, но когда количество выполняемых потоков значительно меньше, чем количество ядер, на которых запущен процесс, то отличия будут минимальные. Высокий приоритет гарантирует, что поток получит процессорное время в конкурентной борьбе за него. Но если ресурсов (ядер) хватает на всех, и потоки не конкурируют за них, их приоритеты отходят на второй план, и на первый план выходит простое параллельное выполнение.

ВЫВОД

В ходе выполнения лабораторной работы была успешно разработана многопоточная консольная программа на языке C++, предназначенная для демонстрации управления потоками, нитями и их приоритетами в среде Windows. Программа эффективно реализует процесс умножения матриц с использованием нескольких потоков, каждый из которых имеет различные уровни приоритета. Это позволило оценить влияние приоритетов потоков на производительность вычислений.

Результаты экспериментов показали, что потоки с более высоким приоритетом завершают задачи быстрее, чем потоки с более низким приоритетом, подтверждая теоретические сведения о механизме планирования и управления потоками в операционных системах. Реализация прогресс-бара для каждого потока в консоли также обеспечила наглядное отображение процесса выполнения задач, что добавило практическую значимость и образовательную ценность к работе.

Таким образом, лабораторная работа не только способствовала закреплению теоретических знаний по управлению процессами и потоками в операционной системе Windows, но и позволила получить практический опыт разработки многопоточных приложений, что является важным навыком в области системного программирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Build desktop Windows apps using the Win32 API [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/>. – Дата доступа: 06.09.2025.

[2] Разработка приложений с помощью WinAPI [Электронный ресурс]. – Режим доступа: <https://shorturl.at/BDJW8>. – Дата доступа: 06.09.2025.

[3] Multithreading Tutorial - CodeProject [Электронный ресурс]. – Режим доступа: <https://www.codeproject.com/Articles/14746/Multithreading-Tutorial>. – Дата доступа: 06.09.2025.

ПРИЛОЖЕНИЕ А

(справочное)

Исходный код

```
#include <iostream>
#include <windows.h>
#include <chrono>
#include <string>
#include <vector>

using namespace std;
using namespace chrono;

CRITICAL_SECTION coutCS;
HANDLE hConsole;
const long long numThreads = 6;

int threadPriorities[numThreads] = {
    THREAD_PRIORITY_CRITICAL,           // 15
    THREAD_PRIORITY_HIGHEST,           // 2
    THREAD_PRIORITY_NORMAL,            // 0
    THREAD_PRIORITY_BELOW_NORMAL,      // -1
    THREAD_PRIORITY_LOWEST,            // -2
    THREAD_PRIORITY_IDLE              // -15
};

string priorityNames[numThreads] = {
    "TIME_CRITICAL (15)",
    "HIGHEST (2)",
    "NORMAL (0)",
    "BELOW_NORMAL (-1)",
    "LOWEST (-2)",
    "IDLE (-15)"
};

WORD threadColors[numThreads] = {
    FOREGROUND_RED | FOREGROUND_INTENSITY,
    FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_INTENSITY,
    FOREGROUND_GREEN | FOREGROUND_INTENSITY,
    FOREGROUND_BLUE | FOREGROUND_INTENSITY,
    FOREGROUND_RED | FOREGROUND_BLUE | FOREGROUND_INTENSITY,
    FOREGROUND_GREEN | FOREGROUND_BLUE | FOREGROUND_INTENSITY
};

void UpdateProgressBar(long long index, long long progress)
{
    EnterCriticalSection(&coutCS);
    SetConsoleTextAttribute(hConsole, threadColors[index]);

    COORD cursorPosition;
    cursorPosition.X = 0;
    cursorPosition.Y = (SHORT)(index);
    SetConsoleCursorPosition(hConsole, cursorPosition);

    string progressBar = "Поток " + to_string(index + 1) + " [" +
priorityNames[index].substr(0, 10) + "] : [";
    long long barWidth = 40;
    long long pos = barWidth * progress / 100;
    for (long long i = 0; i < barWidth; ++i) {
        if (i < pos) progressBar += "=";
        else progressBar += " ";
    }
}
```

```

progressBar += "] " + to_string(progress) + "%";

cout << progressBar;
LeaveCriticalSection(&coutCS);
}

void MultiplyMatrices(const vector<vector<long double>>& A,
                      const vector<vector<long double>>& B,
                      vector<vector<long double>>& C, long long threadIndex)
{
    long long N = A.size();
    for (long long i = 0; i < N; ++i) {
        for (long long j = 0; j < N; ++j) {
            C[i][j] = 0;
            for (long long k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
        long long progress = (100 * (i + 1)) / N;
        UpdateProgressBar(threadIndex, progress);
    }
}

DWORD WINAPI ThreadFunction(LPVOID lpParam)
{
    long long threadIndex = *(long long*)lpParam;
    SetThreadPriority(GetCurrentThread(), threadPriorities[threadIndex]);

    auto startTime = high_resolution_clock::now();

    // Увеличим размер матриц для более заметной разницы
    vector<vector<long double>> A(300, vector<long double>(300, 3.0));
    vector<vector<long double>> B(300, vector<long double>(300, 3.0));
    vector<vector<long double>> C(300, vector<long double>(300, 0.0));

    MultiplyMatrices(A, B, C, threadIndex);

    auto endTime = high_resolution_clock::now();
    duration<double> elapsed = endTime - startTime;

    EnterCriticalSection(&coutCS);
    SetConsoleTextAttribute(hConsole, threadColors[threadIndex]);

    COORD cursorPosition;
    cursorPosition.X = 0;
    cursorPosition.Y = (SHORT)(numThreads + threadIndex);
    SetConsoleCursorPosition(hConsole, cursorPosition);

    cout << "Поток " << threadIndex + 1 << " [" << priorityNames[threadIndex]
        << "] завершен за " << elapsed.count() << " сек.";

    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN |
    FOREGROUND_BLUE);
    LeaveCriticalSection(&coutCS);

    return 0;
}

void RunTest(DWORD_PTR affinityMask)
{
    system("cls");

    int activeCores = 0;
    DWORD_PTR mask = affinityMask;
}

```

```

        while (mask) {
            activeCores += (mask & 1);
            mask >>= 1;
        }

        string title = "Test: " + to_string(activeCores) + " core/cores";
        SetConsoleTitleA(title.c_str());

        // Устанавливаем affinity для процесса
        SetProcessAffinityMask(GetCurrentProcess(), affinityMask);

        InitializeCriticalSection(&coutCS);
        hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

        HANDLE hThreads[numThreads];
        DWORD threadIDs[numThreads];
        long long threadIndices[numThreads];

        // Создаем потоки
        for (long long i = 0; i < numThreads; ++i) {
            threadIndices[i] = i;
            hThreads[i] = CreateThread(NULL, 0, ThreadFunction, &threadIndices[i],
0, &threadIDs[i]);
        }

        // Ждем завершения
        WaitForMultipleObjects(numThreads, hThreads, TRUE, INFINITE);

        // Закрываем handles
        for (long long i = 0; i < numThreads; ++i) {
            CloseHandle(hThreads[i]);
        }

        DeleteCriticalSection(&coutCS);

        if (affinityMask != 0xFFFF) {
            cout << endl << "Тест завершен. Нажмите Enter для продолжения...";
            cin.get();
        }
    }

int main()
{
    setlocale(LC_ALL, "RU");
    system("color 07");

    // Тест 1: 1 ядро
    RunTest(0x01);

    // Тест 2: 3 ядра
    RunTest(0x07);

    // Тест 3: 16 ядер
    RunTest(0xFFFF);

    return 0;
}

```