**Лабораторная работа №2**

**Оглавление**

[Лабораторная работа № 2. Параллельное программирование и библиотека TPL 3](#_Toc162106655)

[Содержание пояснительной записки 17](#_Toc162106656)

[Используемое программное обеспечение 18](#_Toc162106657)

[Список литературы 19](#_Toc162106658)

Лабораторная работа № 2.  
Параллельное программирование  
и библиотека TPL

**Цель лабораторной работы**

Закрепление теоретических знаний по разработке приложений для параллельных вычислений.

**Постановка задачи**

В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно. Поэтому во фреймворк .NET была добавлена библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен System.Threading.Tasks. Данная библиотека упрощает работу с многопроцессорными, многоядерными системами. Кроме того, она упрощает работу по созданию новых потоков. Поэтому обычно рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом Task, который находится в пространстве имен System.Threading.Tasks. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

**Первый** способ создание объекта **Task** и вызов у него метода **Start**:

Task task = new Task(() => Console.WriteLine("Hello Task!"));

task.Start();

**Второй** способ заключается в использовании статического метода **Task.Factory.StartNew()**. Этот метод также в качестве параметра принимает делегат **Action**, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));

**Третий** способ определения и запуска задач представляет использование статического метода **Task.Run()**:

Task task = Task.Run(() => Console.WriteLine("Hello Task!"));

Метод **Task.Run()** также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

**Ожидание завершения задачи.**

Чтобы приложение ожидало завершения задачи, можно использовать метод **Wait()** объекта Task:

Task task1 = new Task(() => Console.WriteLine("Task1 is executed"));

task1.Start();

Task task2 = Task.Factory.StartNew(() => Console.WriteLine("Task2 is executed"));

Task task3 = Task.Run(() => Console.WriteLine("Task3 is executed"));

task1.Wait(); // ожидаем завершения задачи task1

task2.Wait(); // ожидаем завершения задачи task2

task3.Wait(); // ожидаем завершения задачи task3

Стоит отметить, что метод Wait() блокирует вызывающий поток, в котором запущена задача, пока эта задача не завершит свое выполнение (рисунок 1).

**Например**:

Console.WriteLine("Main Starts");

// создаем задачу

Task task1 = new Task(() =>

{

Console.WriteLine("Task Starts");

Thread.Sleep(1000); // задержка на 1 секунду - имитация долгой работы

Console.WriteLine("Task Ends");

});

task1.Start(); // запускаем задачу

task1.Wait(); // ожидаем выполнения задачи

Console.WriteLine("Main Ends");

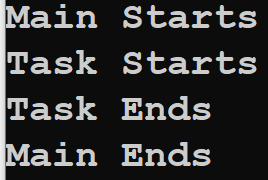


Рисунок 1 – Запуск задачи с ожиданием завершения

Для эмуляции долговременной работы здесь в задаче task1 устанавливается задержка на 1 секунду. В итоге, когда выполнение дойдет до вызова task1.Wait() основной поток остановит свое выполнение и будет ждать завершения задачи.

**Свойства класса Task.**

**Status**: возвращает статус задачи. Представляет перечисление System.Threading.Tasks.TaskStatus, которое имеет следующие значения:

* **Canceled**: задача отменена;
* **Created**: задача создана, но еще не запущена;
* **Faulted**: в процессе работы задачи произошло исключение;
* **RanToCompletion**: задача успешно завершена;
* **Running**: задача запущена, но еще не завершена;
* **WaitingForActivation**: задача ожидает активации и постановки в график выполнения;
* **WaitingForChildrenToComplete**: задача завершена и теперь ожидает завершения прикрепленных к ней дочерних задач;
* **WaitingToRun**: задача поставлена в график выполнения, но еще не начала свое выполнение;

**IsCompleted**: возвращает true, если задача завершена;

**IsCanceled**: возвращает true, если задача была отменена;

**IsFaulted**: возвращает true, если задача завершилась при возникновении исключения;

**IsCompletedSuccessfully**: возвращает true, если задача завершилась успешно.

**Пример:**

Task task1 = new Task(() =>

{

Console.WriteLine(

$"Task{Task.CurrentId} Starts");

Thread.Sleep(1000);

Console.WriteLine(

$"Task{Task.CurrentId} Ends");

});

task1.Start(); //запускаем задачу

// получаем информацию о задаче

Console.WriteLine($"task1 Id: {task1.Id}");

Console.WriteLine($"task1 is Completed: {task1.IsCompleted}");

Console.WriteLine($"task1 Status: {task1.Status}");

task1.Wait(); // ожидаем завершения задачи

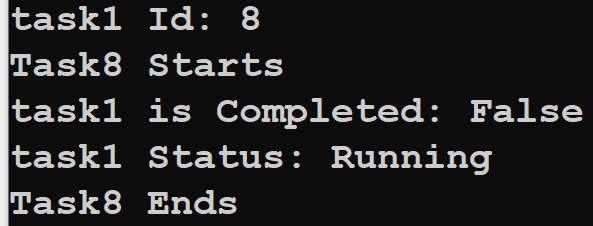


Рисунок 2 – Запуск задачи с отображением свойств

**Вложенные задачи.**

Одна задача может запускать другую - вложенную задачу. При этом эти задачи выполняются независимо друг от друга. **Например:**

var outer = Task.Factory.StartNew(() => // внешняя задача

{

Console.WriteLine("Outer task starting...");

var inner = Task.Factory.StartNew(() => // вложенная задача

{

Console.WriteLine("Inner task starting...");

Thread.Sleep(2000);

Console.WriteLine("Inner task finished.");

});

});

// ожидаем выполнения внешней задачи

outer.Wait();

Console.WriteLine("End of Main");

Вложенная задача может завершиться даже после завершения метода Main.

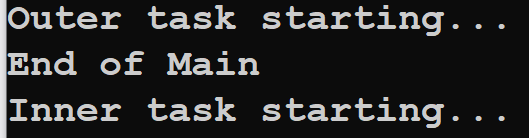


Рисунок 3 – Запуск вложенной задачи

Если необходимо, чтобы вложенная задача выполнялась как часть внешней, необходимо использовать значение TaskCreationOptions.AttachedToParent:

var outer = Task.Factory.StartNew(() => // внешняя задача

{

Console.WriteLine("Outer task starting...");

var inner = Task.Factory.StartNew(() => // вложенная задача

{

Console.WriteLine("Inner task starting...");

Thread.Sleep(2000);

Console.WriteLine("Inner task finished.");

}, TaskCreationOptions.AttachedToParent);

});

outer.Wait(); // ожидаем выполнения

Console.WriteLine("End of Main");

Внешняя задача завершится когда завершатся все прикрепленные к ней вложенные задачи.

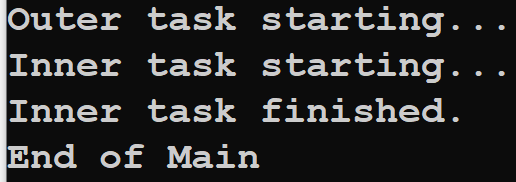


Рисунок 4 – Ожидание завершения вложенной задачи

**Массив задач.**

Task[] tasks = new Task[3];

for (var i = 0; i < tasks.Length; i++)

{

var index = i;

tasks[i] = new Task(() =>

{

Thread.Sleep(1000); // эмуляция долгой работы

Console.WriteLine($"Task{index} finished");

});

tasks[i].Start(); // запускаем задачу

}

Console.WriteLine("Завершение метода Main");

Task.WaitAll(tasks); // ожидаем завершения всех задач

**Получение результата выполнения задачи.**

int n1 = 4, n2 = 5;

Task<int> sumTask = new Task<int>(() => Sum(n1, n2));

sumTask.Start();

int result = sumTask.Result;

Console.WriteLine($"{n1} + {n2} = {result}"); // 4 + 5 = 9

int Sum(int a, int b) => a + b;

При обращении к свойству **Result** текущий поток останавливает выполнение и ждет, когда будет получен результат из выполняемой задачи.

**Задачи-продолжения.**

Задачи продолжения или **continuation task** позволяют определить задачи, которые выполняются после завершения других задач.

Благодаря этому мы можем вызвать после выполнения одной задачи несколько других, определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные.

Задачи продолжения похожи на методы обратного вызова, но фактически являются обычными задачами **Task**.

Посмотрим на **примере**:

Task<int> sumTask = new Task<int>(() => Sum(4, 5));

// задача продолжения

Task printTask = sumTask.ContinueWith(task => PrintResult(task.Result));

sumTask.Start();

// ждем окончания второй задачи

printTask.Wait();

Console.WriteLine("Конец метода Main");

int Sum(int a, int b) => a + b;

void PrintResult(int sum) => Console.WriteLine($"Sum: {sum}");

Рассмотрим **пример параллельного вычисления интеграла**.

double Integrate(double from, double to, int intervals = 100)

{

var integral = 0.0;

var step = (to - from) / intervals;

for (int i = 0; i < intervals; i++) {

var xLeft = from + i \* step;

var xRight = xLeft + step;

integral += (f(xLeft) + f(xRight)) / 2 \* step;

}

return integral;

}

double f(double x) => x \* x;

var from = 0.0;

var to = 1.0;

//Разобьём интеграл на множество интегралов, равное количеству ядер ЦП

var integralTasks = new Task<double>[Environment.ProcessorCount];

var step = (to - from) / integralTasks.Length;

for (var i = 0; i < integralTasks.Length; i++)

{

var xLeft = from + i \* step;

var xRight = xLeft + step;

integralTasks[i] = Task.Run(() => Integrate(xLeft, xRight, 100000000));

}

Task.WaitAll(integralTasks);

Console.WriteLine(integralTasks.Sum(t => t.Result));

**Parallel.For**(int, int, Action<int>)

Метод **Parallel.For** позволяет выполнять итерации цикла параллельно.

Первый параметр метода задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию:

Parallel.For(1, 5, Square);

// вычисляем квадрат числа

void Square(int n)

{

Console.WriteLine($"Выполняется задача {Task.CurrentId}");

Console.WriteLine($"Квадрат числа {n} равен {n \* n}");

Thread.Sleep(2000);

}

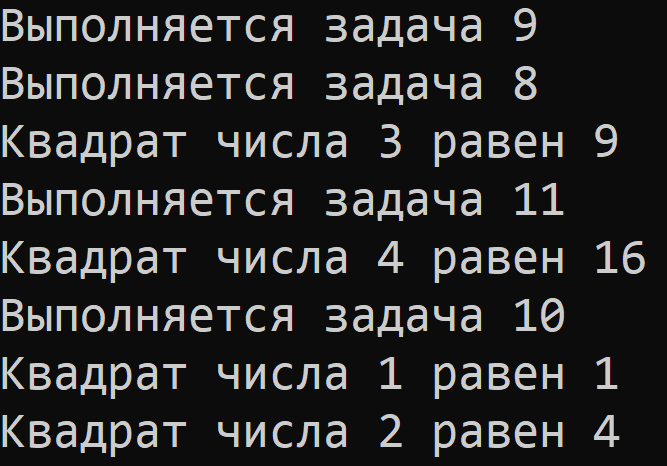


Рисунок 5 – Параллельный расчет квадратов чисел в цикле for

**Parallel.ForEach<TSource>(IEnumerable<TSource> source,Action<TSource> body)**

Метод Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора.

ParallelLoopResult result = Parallel.ForEach<int>(

new List<int>() { 1, 3, 5, 8 },

Square

);

// вычисляем квадрат числа

void Square(int n)

{

Console.WriteLine($"Выполняется задача {Task.CurrentId}");

Console.WriteLine($"Квадрат числа {n} равен {n \* n}");

Thread.Sleep(2000);

}

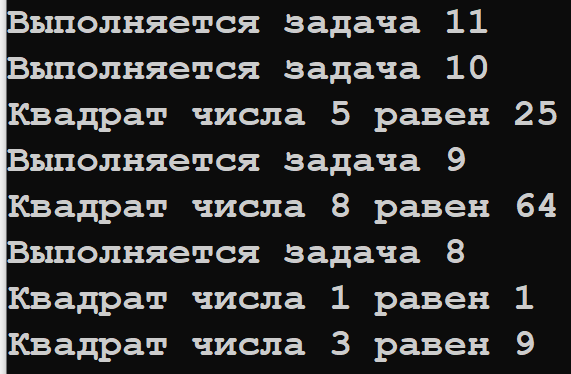


Рисунок 6 – Параллельный расчет квадратов чисел  
в цикле foreach

**Выход из цикла.**

В стандартных циклах for и foreach предусмотрен преждевременный выход из цикла с помощью оператора **break**. В методах Parallel.ForEach и Parallel.For мы также можем, не дожидаясь окончания цикла, выйти из него:

ParallelLoopResult result = Parallel.For(1, 25, Square);

if (!result.IsCompleted)

Console.WriteLine($"Выполнение цикла завершено на итерации {result.LowestBreakIteration}");

// вычисляем квадрат числа

void Square(int n, ParallelLoopState pls)

{

// если передано 5, выходим из цикла

if (n == 5) pls.Break();

Console.WriteLine($"Квадрат числа {n} равен {n \* n}");

Thread.Sleep(2000);

}

**Отмена задач.**

Параллельное выполнение задач может занимать много времени. И иногда может возникнуть необходимость прервать выполняемую задачу. Для этого платформа .NET предоставляет структуру **CancellationToken** из пространства имен System.Threading.

**Общий алгоритм отмены задачи** обычно предусматривает следующий порядок действий:

1. Создание объекта **CancellationTokenSource**, который управляет и посылает уведомление об отмене токену.
2. С помощью свойства **CancellationTokenSource.Token** получаем токен и передаем его в задачу, которая может быть отменена.
3. Определяем в задаче действия на случай ее отмены.
4. Вызываем метод **CancellationTokenSource.Cancel**(), который устанавливает для свойства **CancellationToken.IsCancellationRequested** значение true.
5. когда работа с объектом **CancellationTokenSource** завершена, у него следует вызвать метод **Dispose** для освобождения всех связанных с ним используемых ресурсов.

**Пример**.

CancellationTokenSource cancelTokenSource = new

CancellationTokenSource();

CancellationToken token = cancelTokenSource.Token;

Task task = new Task(() => // задача вычисляет квадраты чисел

{

for (int i = 1; i <= 10; i++)

{

if (token.IsCancellationRequested) // проверяем наличие сигнала отмены

{

Console.WriteLine("Операция прервана");

return; // выходим из метода и тем самым завершаем задачу

}

Console.WriteLine($"Квадрат числа {i} равен {i \* i}");

Thread.Sleep(200);

}

}, token);

task.Start();

Thread.Sleep(1000); // после задержки по времени отменяем выполнение задачи

cancelTokenSource.Cancel();

task.Wait(); // ожидаем завершения задачи

Console.WriteLine($"Task Status: {task.Status}"); // проверяем статус задачи

cancelTokenSource.Dispose(); // освобождаем ресурсы

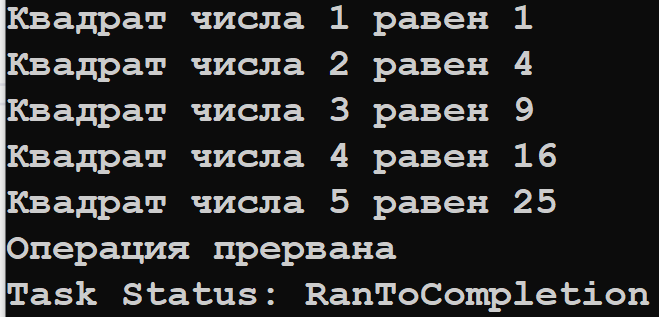


Рисунок 6 – Отмена параллельной задачи

**Задание на лабораторную работу**

1. Модифицировать программу из лабораторной работы 1. Выделить независимые участки кода для параллельного запуска.
2. Добавить форму авторизации и регистрации пользователя при запуске игры. Данные пользователей хранятся в JSON файле.
3. Добавить сохранение игровой статистики (например, набранные очки, побежденные противники, выбранные варианты действий и т.д. в зависимости от игры) в JSON файл в виде задачи (Task), которая выполняется параллельно игре.
4. Модифицировать алгоритм игры с помощью задач для ускорения работы.
5. Защита работы включает демонстрацию работы программы.
6. Для успешной защиты:

* программа должна быть задокументирована с помощью комментариев,
* код программы не должен содержать не используемые блоки, должен быть «чистым» и стилистически верно оформлен,
* интерфейс программы должен быть гибким и удобным.

# Содержание пояснительной записки

1. Постановка задачи. Приводится теоретический материал, использованный при написании приложения.

2. Формулировка задания и вариант. Приводится задание на лабораторную работу и вариант этого задания.

3. Описание выполняемых действий. Необходимо привести описание последовательности разработки программы, реализации используемых методов, алгоритмов, блок-схем.

4. Анализ результатов. Привести анализ входных и выходных данных. Показать результаты выполнения программного кода. Предоставить скриншоты обработки тестовых примеров. Сделать выводы.

5. Листинг программы. Привести листинг разработанного программного кода, содержание файлов входных и выходных данных.

# Используемое программное обеспечение

1. Среда программирования MS Visual Studio Community 2022 (Свободно распространяемое программное обеспечение (в учебных целях));
2. Microsoft Office Standard 2007 (Open License: 42267924);
3. Open Office (Свободно распространяемое программное обеспечение).
4. Браузер (Свободно распространяемое программное обеспечение).

# Список литературы

* + - 1. Мейер Б. Объектно-ориентированное программирование и программная инженерия [Электронный ресурс]/ Мейер Б. – Электрон. текстовые данные. – М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 285 c.
      2. Биллиг, В. A. Основы объектного программирования на С# (C# 3.0, Visual Studio 2008) [Электронный ресурс] : учебное пособие / В. A. Биллиг. — Электрон. текстовые данные. — Москва, Саратов : Интернет-Университет Информационных Технологий (ИНТУИТ), Вузовское образование, 2017. — 583 c. — 978-5-4487-0145-0. — Режим доступа: http://www.iprbookshop.ru/72339.html
      3. Павловская, Т. А. Программирование на языке высокого уровня C# [Электронный ресурс] / Т. А. Павловская. — 2-е изд. — Электрон. текстовые данные. — М. : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 245 c. — 2227-8397. — Режим доступа: http://www.iprbookshop.ru/73713.html
      4. Агапов, В. П. Основы программирования на языке С# [Электронный ресурс] : учебное пособие / В. П. Агапов. — Электрон. текстовые данные. — М. : Московский государственный строительный университет, ЭБС АСВ, 2012. — 128 c. — 978-5-7264-0576-6. — Режим доступа: http://www.iprbookshop.ru/16366.html
      5. Медведев, М. А. Программирование на СИ# [Электронный ресурс] : учебное пособие / М. А. Медведев, А. Н. Медведев ; под ред. А. В. Присяжный. — Электрон. текстовые данные. — Екатеринбург : Уральский федеральный университет, ЭБС АСВ, 2015. — 64 c. — 978-5-7996-1561-1. — Режим доступа: http://www.iprbookshop.ru/69667.html
      6. Казанский А.А. Объектно-ориентированное программирование на языке Microsoft Visual С# в среде разработки Microsoft Visual Studio 2008 и .NET Framework. 4.3 [Электронный ресурс]: учебное пособие и практикум/ Казанский А.А.— Электрон. текстовые данные.— М.: Московский государственный строительный университет, ЭБС АСВ, 2011.— 180 c
      7. Уйманова Н.А. Основы объектно-ориентированного программирования [Электронный ресурс]: практикум/ Уйманова Н.А., Таспаева М.Г.— Электрон. текстовые данные.— Оренбург: Оренбургский государственный университет, ЭБС АСВ, 2017.— 156 c.— Режим доступа: http://www.iprbookshop.ru/78808.html.— ЭБС «IPRbooks»
      8. Новиков П.В. Объектно-ориентированное программирование [Электронный ресурс]: учебно-методическое пособие к лабораторным работам/ Новиков П.В.— Электрон. текстовые данные.— Саратов: Вузовское образование, 2017.— 124 c.— Режим доступа: http://www.iprbookshop.ru/64650.html.— ЭБС «IPRbooks»