**МИНОБРНАУКИ РОССИИ**

**Федеральное государственное бюджетное образовательное  
 учреждение высшего образования   
«Ярославский государственный университет им. П.Г. Демидова»**

Кафедра компьютерной безопасности и математических   
методов обработки информации

Сдано на кафедру

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_ г.

Заведующий кафедрой

д. ф.-м. н., профессор

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Дурнев В.Г.

Курсовая работа по теме:

**“Изучение и реализация различных механизмов распределения ресурсов в многопоточном приложении на примере задачи Штейнера”**

специальность 10.05.01 Компьютерная безопасность

Научный руководитель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(степень, звание)*

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_

*(подпись) (ФИО)*

«\_\_\_» \_\_\_\_\_\_\_\_\_ 20\_\_ г.

Студент группы \_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(подпись) (ФИО)*

«\_\_\_» \_\_\_\_\_\_\_\_\_ 20\_\_ г.

**Оглавление**

Введение………………………………………………………………………….....3

Глава 1. Что такое многопоточность……………………………………………….4

Глава 2. Задача Штейнера, алгоритм Мелзака…………………………………...9

Глава 3. Реаизация алгоритма Мелзака в многопоточном приложении……….14

Глава 4. Тестирование приложения и анализ полученных данных…………...20

Заключение…………………………………………………………………………23

Список использованных источников литературы…………………………...…..24

Приложения……………………………………………………………………….25

**Введение**

Ещё недавно, почти все создавали только однопоточные приложения. Однако в мире уже довольно давно существуют технология многопоточности. Она подразумевает под собой выполнение параллельно нескольких поставленных задач компьютеру. Одной из главных причин, почему многопоточность в производственном программировании стала востребована лишь относительно недавно, является возросшая потребность в скорости обработки информации. Самый простой пример, это обращение к серверу сотен тысяч пользователей одновременно. Выполнение всех этих запросов поочередно занимает очень много времени, а пользователь не любит ждать. И со временем таких пользователей становится всё больше и больше и вместе с тем растёт потребность в написании многопоточных приложений.

Актуальность данной курсовой работы обусловлена тем, что с каждым днем всё больше и больше становятся востребованными многопоточные приложения для ускорения выполнения задач данных компьютеру.

Объектом исследования курсовой работы является изучение и способы реализации механизмов распределения ресурсов в многопоточном приложении.

Предмет исследования курсовой работы — многопоточное приложение написанное на C#, с использованием платформы .NET 5, для решения задачи Штейнера алгоритмом Мелзака и анализ полученных временных данных.

Цель работы: выяснить, как можно реализовать многопоточное приложение на C# для решения задачи Штейнера и как лучше распределить ресурсы компьютера.

Задачи курсовой работы:

1. изучить способы реализации многопоточного приложения;
2. выяснить, какие способы есть для распределения ресурсов компьютера;
3. реализовать многопоточное приложение для решения задачи Штейнера на C#;
4. проанализировать полученные временные данные.

**Глава 1. Что такое многопоточность**

Многопоточность (на англ. multithreading) — свойство платформы, позволяющее выполнять задачу сразу в нескольких “потоках”, которая была предварительно разбита на несколько подзадач, которые и будут выполняться в этих “потоках” параллельно. Это значит, что эти подзадачи будут выполняться без определенного порядка.

Также существует такое понятие как многозадачность, это понятие уже относится к операционной системе и подразумевает под собой способность системы обеспечивать возможность параллельной обработки задач.

И ещё одним значимым термином можно обозначить понятие псевдопараллельности. Оно подразумевает под собой, что когда по какой то причине выполнение той или иной задачи останавливается на время, то компьютер берёт другую задачу и выполняет её, чтобы не стоять в ожидании и не растрачивать ресурсы, до тех пор, пока не придёт команда на возобновление выполнения задачи, которая была остановлена.

Изначально платформа могла выполнять команды только по порядку. То есть компьютер принимал команду и выполнял её, если команды не поступало, то компьютер просто не работал. Так как компьютер не мог сам хранить поступающие ему команды, то он в те моменты, когда ему ничего не поступало на обработку, просто находился в состоянии ожидания и не использовал свои ресурсы. Позже пришло такое понятие как операционная система и её конкретная реализация. Пользователь мог уже записать множество команд в память компьютера, и он их выполнял по порядку. Таким образом, уже не приходилось в ручную вводить команду компьютеру каждый раз, как компьютер закончил выполнять предыдущую, что значительно улучшило использование ресурсов компьютера.

Не смотря такие улучшения, всё равно нельзя было назвать эффективным использование ресурсов компьютера. Например, в процессе выполнения набора команд, какая то их системных инструкций была заблокирована по какой то причине и процессор не сможет перейти к следующей инструкции и будет в режиме ожидания простаивать. И чтобы решить эту проблему, а именно растрачивание процессорного времени на ожидание, была придумана концепция процессов и потоков.

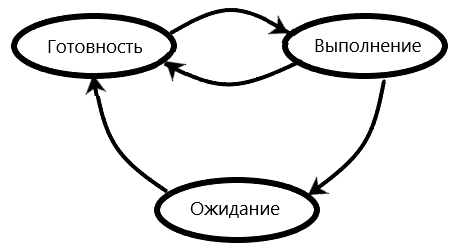
С точки зрения пользователя и операционной системы потоки и процессы, несколько разные понятия.

С точки зрения пользователя: -Процесс — экземпляр выполняемой программы; -Поток — ветви кода, выполняющиеся параллельно.

С точки зрения операционной системы: -Процесс — абстракция, реализованная на уровне операционной системы, созданная для организации данных, необходимых для программы; -Поток — абстракция, реализованная на уровне операционной системы, созданная для контроля выполнения кода программы.

Как уже было сказано ранее, данная концепция была придумана для решения проблемы распределения ресурсов центрального процессора компьютера. Согласно этой концепции поток может находится в трёх состояниях: выполняемый, ожидающий, готовый.

Выполняемый — поток, который выполняется в данный момент; Готовый — поток ожидает команду на старт выполнения заданных инструкций; Ожидаемый — поток был заблокирован в процессе выполнения по какой либо причине.

Ниже представлена схема “конвейера” потоков. 

Также у потоков в современных операционных системах существует такое понятие как приоритет. В реальных задачах некоторые действия необходимо выполнить раньше других, поэтому при планировании, какой поток будет поставлен на выполнение следующим, обязательно определяется приоритет и соответственно, тот поток, у которого приоритет выше, будет выполнен раньше тех, у которых приоритет ниже, если же приоритет совпадает, то выбор потока происходит поочередно.

Одной из главных проблем многопоточности является их синхронизация. Например, когда два потока пытаются перезаписать одну и ту же ячейку памяти, то возникает конфликт, так называемое состояние гонки.

Ожидаемый результат выполнения потоков будет такой:

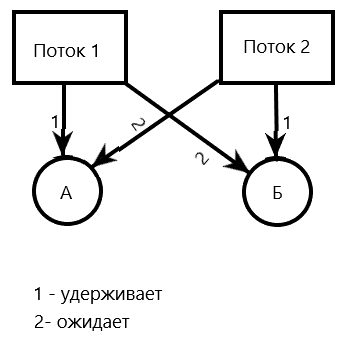
|  |  |  |  |
| --- | --- | --- | --- |
| **Thread 1** | **Thread 2** |  | **Counter** |
|  |  |  | 0 |
| read |  | < | 0 |
| increase |  |  | 0 |
| write |  | > | 1 |
|  | read | < | 1 |
|  | increase |  | 1 |
|  | write | > | 2 |

А в реальности может получиться так:

|  |  |  |  |
| --- | --- | --- | --- |
| **Thread 1** | **Thread 2** |  | **Counter** |
|  |  |  | 0 |
| read |  | < | 0 |
|  | read | < | 0 |
| increase |  |  | 0 |
|  | increase |  | 0 |
| write |  | > | 1 |
|  | write | > | 1 |

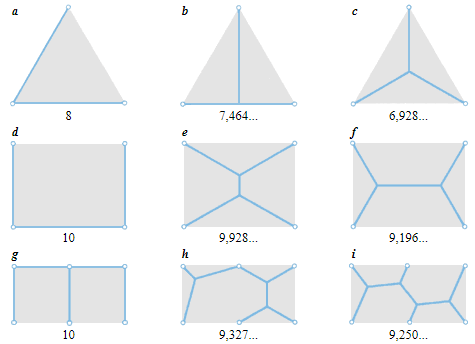
И подобное очень сложно обнаружить в процессе отладки программы.

Также ярким примером проблемы многопоточности является взаимная блокировка (на англ. deadlock). Суть проблемы заключается, когда первый поток пытается получить доступ к объекту, который удерживает второй поток, а второй поток пытается получить доступ к объекту, который удерживает первый поток. И в результате ни один поток не может перейти к следующей инструкции. Схематично это можно изобразить так:



В .NET основной единицей для работы многопоточности является класс Task. Официальное определение класса Task в документации: представляет асинхронную операцию. Если описывать простыми словами, что такое Task, то это инкапсулированная задача, которую необходимо выполнить. Если запустить её на выполнение, то она попадает в планировщик задач и там уже операционная система сама решит, как и когда её выполнить. Также относительно недавно в платформу .NET добавили ключевые конструкции async/await. Они позволяют легко и изящно использовать многопоточность в C#. До Task основной единицей для создания многопоточного приложения использовался класс Thread. Официальное определение класса Thread в документации: создаёт и контролирует поток. Проще говоря это инкапсуляция для потока. Если выбирать между Task и Thread, то лучше по возможности выбирать класс Task, так как с ним работать легче. Также для создания многопоточных приложений в C# существует класс Parallel и концепция PLINQ. Официальное определение класса Parallel в документации: предоставляет поддержку параллельных циклов и областей. Проще говоря, позволяет легко создавать циклы, которые будут выполняться параллельно, причем параллельное выполнение этих циклов распланирует сама система. Официальное определение PLINQ в документации: Parallel LINQ является параллельной реализацией шаблона PLINQ. Проще говоря PLINQ даёт возможность выполнять LINQ запросы с использованием параллельных операций. Кроме того Parallel и PLINQ аналогичные понять, однако PLINQ имеет более широкое применение, и всё же внутри они устроены по разному и поэтому они подходят для разных задач по разному. Платформа .NET устроена так, чтоб программист мог максимально комфортно использовать многопоточность и при этом у него возникали минимальные трудности с этим.

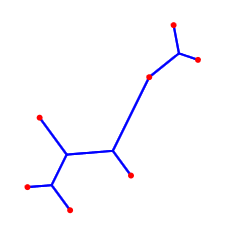
**Глава 2. Задача Штейнера, алгоритм Мелзака**

Задача Штейнера состоит в поиске кратчайшей сети, соединяющей заданный набор точек плоскости. Наиболее тривиальный вариант для построения кратчайшей сети, это построение минимального остовного дерева. Однако, что если имеется возможность использовать дополнительные точки? Используя такие точки, можно получить дерево меньше чем остовное. Пример (взят из открытых источников): 

В случаях a, d и g точки соединяются без дополнительных промежуточных точек и такое решение называется минимальным остовным деревом. Деревья Штейнера, полученные путём добавления узловых точек, показаны для случаев b, c, e, f, h и i. Только c, f и i являются кратчайшими деревьями Штейнера, или кратчайшими сетями. Числа под каждым решением указывают примерную суммарную длину отрезков сети.

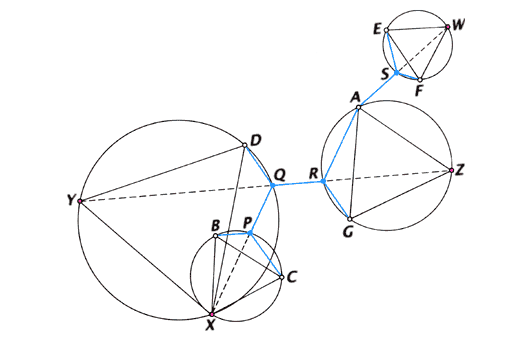
В решении задачи Штейнера для соединения позволяется использовать дополнительные точки, которые находятся в процессе нахождения этой сети. Они называются точки Штейнера. Количество точек Штейнера *n* должно удовлетворять условию *n ≤ m* – *2,* где *m* — количество исходных точек. Также все точки, будь это исходные точки или же точки Штейнера, должны удовлетворять условию, что угол между смежными рёбрами должен быть не менее *120о.* Это значит, что у точки может быть максимум 3 ребра. Дерево Штейнера — это кратчайшая сеть, соединяющая заданный набор точек. Поиск дерева Штейнера является NP-полной задачей. И поэтому на данный момент не существует эффективного полиномиального алгоритма для решения этой задачи. Эта задача нашла огромное применение в реальной жизни, например, как соединить города электросетью, при этом использовав минимально ресурсов. Из-за того, что это NP-полная задача, для решения реальных задач, используют алгоритмы, которые дают приближенный результат.

Пример дерева Штейнера:



Первый алгоритм, решающий эту задачу, является алгоритмом Мелзака. Он является экспоненциальным алгоритмом. Смысл этого алгоритма заключается в том, что исходный набор точек разбивается на все возможные подмножества точек, не включая подмножества из одной точки, так как для них не имеет смысла в нахождении кратчайшей сети, и для них строится дерево Штейнера. Само дерево Штейнера строится путём “стягивания” множества точек к двум точкам путём их замещения одной из двух возможных замещающих точек, которые получаются при построении равностороннего треугольника. Так как для двух точек существует два равносторонних треугольника, то соответственно и вариантов замещающих точек — два. Это продолжается до тех пор, пока не останется две точки. Данное построение обусловлено тем, что вместо двух исходных точек из множества можно поставить замещающую точку, при этом, не изменяя длины предполагаемого локально минимального дерева. После того как исходное множество точек было “стянуто” к двум точкам, начинается обратный ход алгоритма. Эти две точки (будем называть точка A и точка B) соединяются линией. Одну из двух точек (предположим, что это точка B), которая является замещающей (в получившемся множестве из двух точек, одна из них может оказаться не замещающей, а исходной), замещается обратно на замещаемые точки (точка D и точка C). На точках B, C и D строится окружность и точка пересечения между окружностью и линией, которой были соединены точки A и B, будет являться точкой Штейнера (точно также происходит нахождение точки Ферма-Торричели). Если же линия не пересекла окружность или точка пересечения совпадает с точкой A, значит точка A является точкой Штейнера. Однако на это построение накладываются ограничения. Первое ограничение: получившаяся точка Штейнера должна лежать на малой дуге CD окружности, если линия имеет пересечение с окружностью. Второе ограничение: точки A и B должны лежать в разных полуплоскостях относительно прямой CD. Если одно из этих условий не выполнено, то дальнейшее построение дерева Штейнера прерывается, так как оно приведёт к заведомо ложному результату. Если же расчёт привёл к конечному результату, то получается локально минимальное дерево. Так как вариантов построения таких деревьев очень много (для подмножества из восьми точек существует более двух миллионов замещающих последовательностей), алгоритм Мелзака может потребовать колоссального количества времени. Далее все получившиеся локально минимальные деревья сравниваются, и среди них выбирается самое минимальное. Это будет являться глобально минимальным деревом или же деревом Штейнера. После того, как для всех подмножеств точек было найдено дерево Штейнера, они комбинируются между собой всеми возможными способами, путём нахождения одной общей точки между исходными подмножествами, и сравнивается их длина. Соответственно минимальная по длине комбинация будет деревом Штейнера для исходного множества точек.

Пример построения дерева Штейнера (взято из открытых источников):



Точка A подходит для разбиения задачи на подзадачи из 3 и 5 точек. Чтобы построить возможные деревья Штейнера для 5 точек, пару точек (например, B и C) можно заменить одной (здесь X), построив равносторонний треугольник с основанием BC. Теперь задача сведена к 4 точкам: X, D, G и A. Пару точек опять можно заменить — сначала D и X на Y, а потом G и A на Z. Вокруг каждого из полученных равносторонних треугольников (XDY, AGZ и BCX) описываем окружности. Точки Q и R, в которых прямая YZ пересекает две окружности, — это точки Штейнера, а пересечение прямой XQ с третьей окружностью определяет точку Штейнера P. Поскольку невозможно заранее предугадать наилучшее разбиение на подзадачи и группировки на пары, необходимо рассмотреть все варианты, чтобы найти кратчайшее дерево.

Со временем были найдены множества улучшений для алгоритма Мелзака. Например, одно из улучшений подразумевает под собой рассматривать не только геометрическое расположение точек, а также и их топологию. Это позволяет на начальном этапе прервать ветви вычислений, которые заведомо приведут к ложному результату. Однако подобные улучшения носят непостоянный характер, так как сильно зависят от исходного расположения точек. И подобные улучшения всё также оставляют алгоритм Мелзака экспоненциальным алгоритмом.

Данная задачи хорошо подходит для демонстрации быстроты выполнения задачи в многопоточном режиме по сравнению с выполнением её в однопоточном режиме, так как этот алгоритм является экспоненциальным, и уже на более семи точках выполняется за большое количество времени. Это позволяет сравнить разницу во времени выполнения задачи в разных режимах даже на небольшом количестве точек.

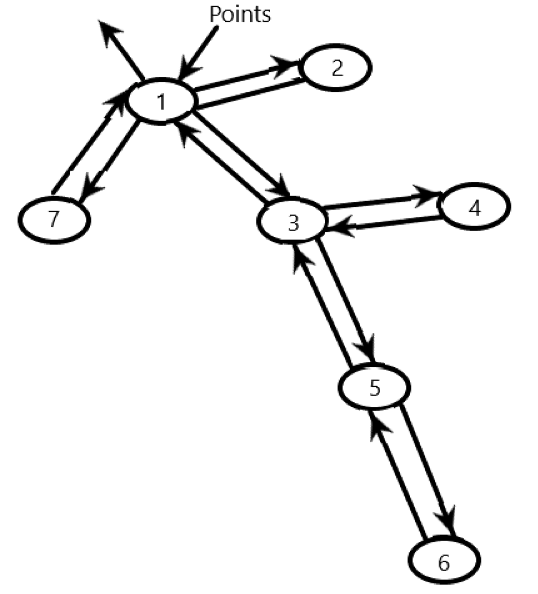
**Глава 3. Реализация алгоритма Мелзака в многопоточном приложении и анализ полученных данных**

Конфигурация компьютера:

* Центральный процессор: AMD Ryzen 5 1600 Six core, 6 ядер, 12 потоков, частота 3200 МГц;
* ОЗУ: 16 ГБ, частота 3000 МГц, пропускная способность PC2400, CL 15, tRCD 17, tRP 17.

В реализации приложения использовались стандартные библиотеки предоставляемые платформой .NET 5, собственные классы структур данных Dequeue и Stack (стандартная не использовалась по причине того, что нужен был собственная реализация конструктора этой структуры данных) и сторонняя библиотека OxyPlot для WPF (использовалась для создания интерфейса координатной плоскости).

Проектировка алгоритма:



1 — точка входа, на вход даётся исходный набор точек, возвращает дерево Штейнера для этого набора точек;

2 — метод, возвращающий все возможные подмножества точек исходного множества, на вход даётся исходное множество;

3 — точка входа для построения дерева Штейнера для подмножества точек, на вход даётся подмножество точек, возвращается дерево Штейнера для подмножества точек;

4 — метод, возвращающий все возможные замещающие последовательности для подмножества точек, на вход даётся подмножетсво точек;

5 — метод, “стягивающий” исходное подмножество точек до двух точек и вычисляющий дерево Штейнера для входного набора точек, возвращает дерево Штейнера для подмножества точек и его длину;

6 — метод обратного хода алгоритма, на вход принимает, “стянутое” до двух точек, множество;

7 — метод, комбинирующий полученные деревьев Штейнера для всех подмножетсв, на вход получает все деревья Штейнера для подмножеств, возвращает дерево Штейнера для исходного множества точек.

Далее этот алгоритм был разбит на слабо связанные подзадачи, и было решено реализовать в отдельных потоках ветвь кода выполняющее задачу под номером 5 (она включает в себя и задачу под номером 6).

Наиболее тривиальным вариантом было бы использование класса Parallel или концепции PLINQ, однако данные способы могут оказаться медленнее (что в дальнейшем подтвердилось), чем использование Task ил Thread, по причине того, что задача сама простая, но операционная система использует много ресурсов для её планирования. Далее стоял выбор использовать Task или Thread. И выбор был сделан в пользу Thread для демонстрации механизма распределения ресурсов, так как при использовании Task, подобные задачи, как распределение ресурсов, берет на себя операционная система. Наиболее тривиальным решением было в каждой итерации цикла создавать экземпляр класса Thread и запускать в нём задачу, однако сразу же возникают две проблемы. Первая проблема заключается в том, что Thread представляет собой инкапсуляцию физического системного потока и соответственно число таких потоков ограничено физически. Эта проблема решается просто ограничением существующих потоков в приложении. Если число потоков равно ограничению, то ожидается, пока один из потоков не закончит своё вычисление. Вторая проблема заключается в том, что создавать объекты Thread в каждой итерации цикла очень накладно и занимает существенное количество времени. Поэтому этот вариант крайне неэффективен. После было решено создать отдельный класс под названием ThreadController, который будет контролировать выполнение потоков для задачи. Концепция этого класса заключается в том, что при старте вычисления будет сразу запущено несколько объектов Thread (количество задаётся пользователем), которые будут находится в бесконечном цикле, пока не будет сделан вызов метода этого класса, останавливающий работу потоков (делает это непринудительно). В классе есть приватное поле LinkedList<Action> — это очередь задач. В бесконечном цикле каждый поток проверяет есть задача на данный момент в очереди, если есть, то поток берёт эту задачу из очереди и выполняет её.

Ниже представлена реализация этого класса:

**public** **class** ThreadController

{

**private** Thread[] \_threads;

**private** LinkedList<Action> \_queueAction;

**private** **bool** \_stop;

**private** **object** \_lock;

**public** **void** Start(**int** ThreadsCount)

{

\_stop = **false**;

\_threads = **new** Thread[ThreadsCount];

\_queueAction = **new** LinkedList<Action>();

\_lock = **new**();

**for** (**int** i = 0; i < ThreadsCount; i++)

{

\_threads[i] = **new** Thread(DoWork) { IsBackground = **true** };

\_threads[i].Start();

}

}

**public** **void** AddNewTask(Action action)

{

**lock** (\_lock)

{

\_queueAction.AddLast(action);

}

}

**public** **void** Wait()

{

**while** (\_queueAction.Count != 0) ;

}

**public** **void** Stop()

{

\_stop = **true**;

\_Wait();

\_threads = **null**;

\_queueAction = **null**;

\_lock = **null**;

}

**private** **void** \_Wait()

{

**for** (**int** i = 0; i < \_threads.Length; i++)

{

\_threads[i].Join();

}

}

**private** **void** DoWork()

{

**while** (!\_stop || \_queueAction.Count > 0)

{

Action a = **null**;

**lock** (\_lock)

{

**if** (\_queueAction.Count > 0)

{

a = \_queueAction.First.Value;

\_queueAction.RemoveFirst();

}

}

a?.Invoke();

}

}

}

Хотелось бы обратить внимание на метод DoWork. Это тот самый метод с бесконечным циклом, который обрабатывается в потоке. В нём используется инструкция lock. Официальное определение инструкции lock из документации: оператор lock получает взаимоисключающую блокировку заданного объекта перед выполнением определённых операторов, а затем снимает блокировку; во время блокировки поток, удерживающий блокировку, может снова поставить и снять блокировку; любой другой поток не может получить блокировку и ожидает ее снятия. Данная инструкция была использована для исключения состояния гонки потоков. Например:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Thread 1** | **Thread 2** |  | **Queue.Count** | **Exception** |
| Read Count |  | < | 1 |  |
|  | Read Count | < | 1 |  |
| Take Action |  |  | 1 |  |
| Decrease count |  |  | 1 |  |
| Write count |  | > | 0 |  |
|  | Take Action |  | 0 | throw Queue is empty excpetion |

Как видно из этого примера, если два потока делают обращение к очереди, то может возникнуть фатальная ошибка, которая приведёт к экстренной остановке приложения.

Аналогичная ситуация происходит в методе AddNewTask. Там также поставлена инструкция lock, однако фатальной ошибки, если этой инструкции там не будет стоять, не произойдет и приложение продолжить работу. Но будет нарушена структура очереди:

|  |  |  |  |
| --- | --- | --- | --- |
| **Main Thread** | **Thread 1** |  | **Counter** |
|  | Read Count | < | 1 |
| Add Action |  |  | 1 |
|  | Take Action |  | 1 |
| Read Count |  | < | 1 |
|  | Decrease Count |  | 1 |
| Increase Count |  |  | 1 |
|  | Write Count | > | 0 |
| Write Count |  | > | 2 |

Вместо инструкции lock можно использовать аналогичные приёмы, например класс ReaderWriterLockSlim. Официальное определение класса ReaderWriterLockSlim из документации: предоставляет блокировку, используемую для управления доступом к ресурсу, которая позволяет нескольким потокам производить считывание или получать монопольный доступ на запись.

Ниже представлен код цикла в котором происходит добавление задачи в объект класса ThreadController:

**object** o = **new** **object**();

**foreach** (Dequeue<Pair<**int**, **int**>> deq **in** GetReplacementSequences(subset))

{

Action action = **new** Action(() =>

{

**double** length = ContractionSet(deq, **new** List<Point>(subset),

**out** List<Pair<Point, List<Point>>> tmpResult);

**lock** (o)

{

**if** (length < minLength)

{

minLength = length;

result = tmpResult;

}

}

});

tc.AddNewTask(action);

}

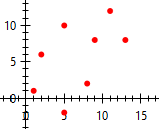
В этом цикле создается делегат Action. Официальное определение делегата Action из документации: инкапсулирует метод, который не принимает параметров и не возвращает значений. В конструктор данного делегата передается анонимный метод, в котором происходит вызов метода “стягивания” входного множества точек до двух точек. Возвращает он длину получившегося дерева Штейнера, а в параметре, помеченным out, находится список смежности этого дерева. Если же в процессе расчёта было не выполнено какое то условие и расчёт был прерван, то этот метод возвращает в качестве длины double.MaxValue, а в параметре tmpResult — null. После происходит сравнение полученной длины с минимальным значением, которое существует на данный момент (изначально оно равно double.MaxValue). Если оно оказывается меньше, то минимальное значение заменяется на полученное значение из метода, и ссылка на дерево заменяется на полученную ссылку из метода. Можно заметить, что в анонимном методе также присутствует инструкция lock. Она необходима, чтобы также избежать состояния гонки потоков, и значения минимальной длины и ссылки на дерево всегда хранили актуальные значения.

**Глава 4. Тестирование приложения и анализ полученных данных.**

Так как центральный процессор, указанный в конфигурации в предыдущей главе, имеет 12 логических потоков, то тестирование будет проходить от 1 до 10 потоков. Два других потока выделены для интерфейса и главного потока выделенного для алгоритма Мелзака. Список точек выбранные для тестирования:

|  |  |  |
| --- | --- | --- |
| № | X | Y |
| 1 | 1 | 1 |
| 2 | 5 | -2 |
| 3 | 8 | 2 |
| 4 | 2 | 6 |
| 5 | 5 | 10 |
| 6 | 9 | 8 |
| 7 | 11 | 12 |
| 8 | 13 | 8 |

Геометрическое расположение точек:



Полученные результаты:

|  |  |
| --- | --- |
| Кол-во потоков | Время, мс |
| 1 | 118515 |
| 2 | 74670 |
| 3 | 57712 |
| 4 | 47175 |
| 5 | 42486 |
| 6 | 41795 |
| 7 | 40937 |
| 8 | 40117 |
| 9 | 40021 |
| 10 | 40035 |

Как можно заметить зависимость времени от количества потоков имеет предположительно обратную зависимость: , где *t*  — время, *C* — некоторая константа, *n* — количество потоков. Однако если попытаться увеличить количество потоков в приложении и их станет больше, чем логических потоков процессора, то производительность незначительно упадёт. Это связано с тем, что все потоки не могут одновременно использовать логические потоки процессора, и они переключаются между собой, что занимает некоторое время.

Снижение прироста производительности по мере увеличения потоков обусловлено несколькими причинами.

Первая причина заключается в том, что задача была лишь частично сделана в параллельном расчёте, однако в этом параллельном расчёте находится наибольшая часть вычислений алгоритма.

Вторая причина, по которой прирост производительности снижается, это то, что для этой задачи оказывается излишне большое количество потоков. Данная причина была выявлена с помощью инструментов отладки, предоставляемые программным обеспечением Visual Studio. Например, допустим, что в текущий момент расчёт ведут 3 потока. В класс ThreadController поступает новая задача, и при этом имеется четвертый поток, который на данный момент не занят. И можно подумать, что новая задача достанется именно ему, но в этот же момент свой расчёт заканчивает первый поток и задача достаётся ему, а тем временем четвертый поток продолжает быть в режиме ожидания.

Из-за этих причин, даже если использовать максимально возможное количество потоков, этот алгоритм, с таким разбиением на подзадачи, смог нагрузить центральный процессор в пике на 54%. Данную проблему можно решить при помощи разбиения задачи на ещё более мелкие независимые подзадачи, однако это потребует перестроения принципа распределения задач между потоками, а также создания возможности перехода потока от одной задачи к другой, до её завершения, с сохранением состояния начальной задачи. Это необходимо по той причине, что если разбивать на ещё более мелкие подзадачи, то придётся добавлять систему приоритетов, и если поступила задача с более высоким приоритетом, то поток должен переключиться на неё. Иначе может возникнуть Deadlock (задача с более низким приоритетом ожидает выполнения задачи с более высоким приоритетом, но если не сможет на неё переключиться, то будет ожидать её бесконечно).

**Заключение**

В ходе работы над этой курсовой работой, были разобраны основные принципы современной многопоточности, а также было написано приложение, которое реализует простой механизм распределения ресурсов между потоками. Анализ полученных данных показал, что многопоточность даёт значительный прирост в производительности, однако чем больше потоков используется в приложении, тем более сложное планирование задачи необходимо. Также в такой ситуации нужно улучшать механизм распределения ресурсов между потоками. В платформе .NET компания Microsoft проделывает огромную работу по улучшению механизмов многопоточности, чтобы программисты могли максимально комфортно использовать её в своем коде без необходимости сложного планирования своих задач.

**Список использованных источников литературы**

1. Троелсен, Джепикс: Язык программирования C# 9 и платформа .NET 5: основные принципы и практики программирования.

2. МГТУ им. Н. Э. Баумана: https://bmstu.ru/

URL: https://mail.bmstu.ru:9100/~sark/Pdf/MDO\_sem\_12\_LMD\_primer.pdf

Дата обращения: 29.04.2022

3. Математика: http://ega-math.narod.ru/

URL: http://ega-math.narod.ru/Nquant/Network.htm

Дата обращения: 29.04.2022

4. Википедия: https://en.wikipedia.org/wiki/Main\_Page

URL: https://ru.wikipedia.org/wiki/Задача\_Штейнера\_о\_минимальном\_дереве

Дата обращения: 29.04.2022

5. Habr @MaxRokatansky: https://habr.com.

URL: https://habr.com/ru/company/otus/blog/549814/

Дата обращения: 10.05.2022

6. Microsoft: https://www.microsoft.com

URL: https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.thread?view=net-5.0

URL: https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.tasks.task?view=net-5.0

URL: https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/async/

URL: https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.tasks.parallel?view=net-5.0

URL: https://docs.microsoft.com/ru-ru/dotnet/standard/parallel-programming/introduction-to-plinq

URL: https://docs.microsoft.com/ru-ru/dotnet/api/system.action?view=net-5.0

URL: https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/lock

URL: https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.readerwriterlockslim?view=net-5.0

Дата обращения: 11.05.2022

**Приложения**

1. https://github.com/GOoppy13/SteinersTreeBuild