

Методы повышения производительности .NET-приложения на примере программы поиска дубликатов

Юрий Малич

yurymalich@yandex.ru

О себе

- Програмирую примерно с 1996 года, с MS DOS, i386
- На C# .NET с первых версий
- Закончил ПГУПС по специальности Автоматика и телемеханика на ж/д транспорте
- Писал статьи по архитектуре микропроцессоров для сайтов iXBT.com, fcenter.ru, 3dnews.ru
- Microsoft Certified Professional, Microsoft Specialist: Programming in C#

О программе Duplicate Files Search & Link

Задача

- Расхламление, наведение порядка и освобождение рабочего пространства за счёт удаления лишних файлов
- Ускорение работы системы (больше пространства, меньше фрагментация)
- Поиск дубликатов файлов и жёстких ссылок с максимально возможной скоростью даже на старых компьютерах

О программе Duplicate Files Search & Link

Какие проблемы приходится решать?

- Больше файлов => больше времени нужно на чтение и сравнение
- Антивирусы замедляют доступ к файлам
- Чем больше файлов, тем больше памяти требуется программе

О программе Duplicate Files Search & Link

- Поиск дубликатов файлов в выбранных папках
- Поиск жёстких ссылок, символьных ссылок на файлы (NTFS, ReFS, EXT2,3,4)
- Поиск дубликатов медиафайлов (.mp3, .mp4, .jpg etc) по медиаконтенту с игнорированием метаданных (тэгов и т.п.)
- Замена дубликатов файлов на жёсткие или символьные ссылки
- Удаление дубликатов (в корзину или насовсем)
- Собирается сразу под 4 платформы: .NET4.7.2, .NET 6.0, 7.0, 8.0 (32+64 бит)

Алгоритмическая оптимизация поиска

Алгоритмы поиска дубликатов

- Побайтное сравнение файлов одинакового размера между собой
- Посчитать хэш каждого файла по отдельности и сгруппировать по хэшам

Алгоритмическая оптимизация поиска

Побайтное сравнение файлов между собой

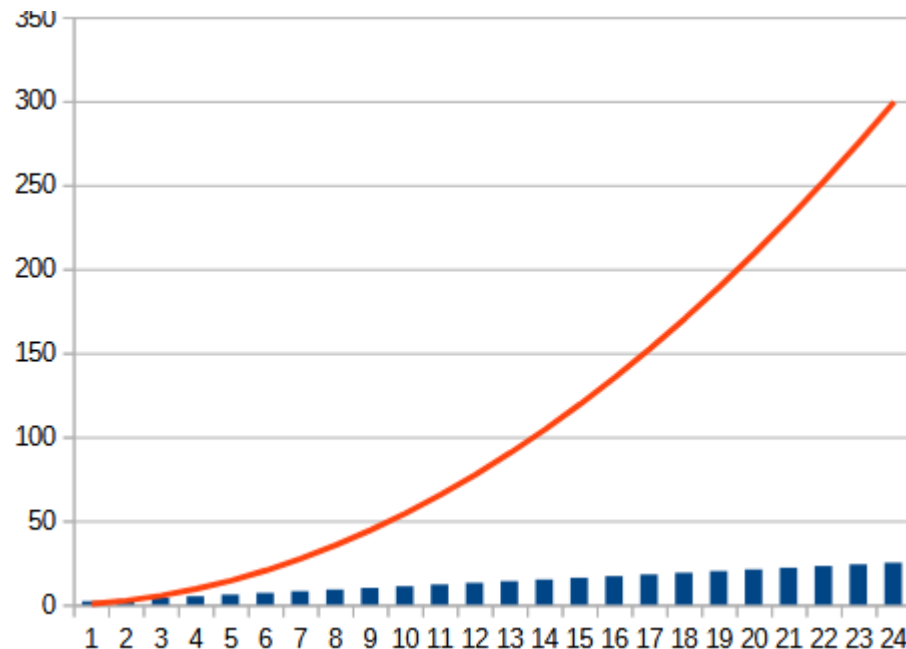
Достоинства:

- Не нужно читать файлы целиком, если они не равны в начале
- Высокая скорость сравнения небольших групп
- Низкая ресурсоёмкость и нагрузка на процессор
- Надёжность

Алгоритмическая оптимизация поиска

Побайтное сравнение файлов между собой

- Недостатки:
 - Квадратичная алгоритмическая сложность в самом худшем сценарии $O((n^2-n)/2)$



Алгоритмическая оптимизация поиска

Хэширование файлов и сравнение хэшей

Достоинства:

- Алгоритмическая простота алгоритма сравнения $O(n)$
- Алгоритмы криптографического хэширования из коробки

Алгоритмическая оптимизация поиска

Хэширование файлов и сравнение хэшей

Недостатки:

- Каждый файл нужно прочитать целиком
- Ресурсоёмкость, высокая нагрузка на процессор
- Скорость хэширования одного файла намного ниже скорости сравнения двух файлов
- Вероятность коллизий (одинаковые хэши разных данных) больше нуля

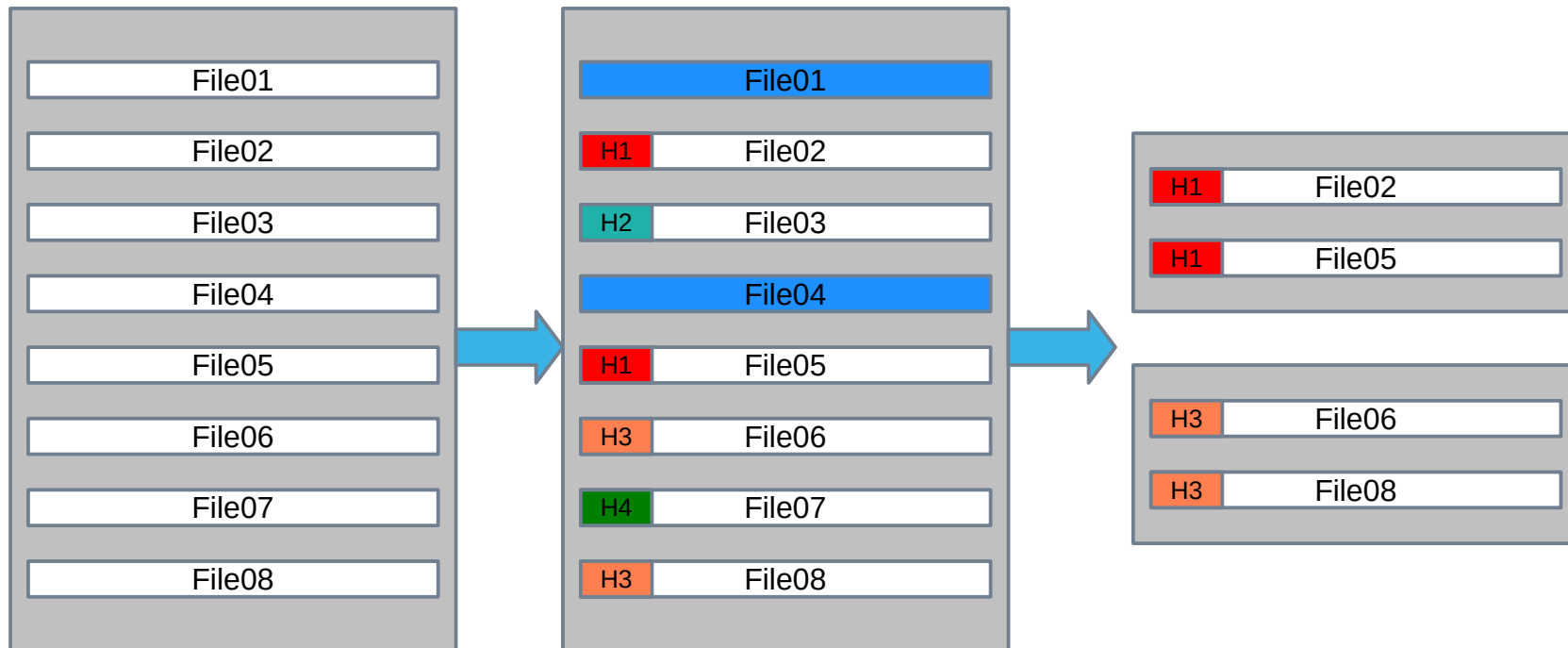
Алгоритмическая оптимизация поиска

Гибридное решение: скомбинировать достоинства, нивелировать недостатки.

- В небольших группах файлы сравниваются побайтно
- Для больших групп используется хэширование, совмещённое со сравнением
- Результаты хэширования используются для последующей группировки сравнения по подгруппам

Алгоритмическая оптимизация поиска

Гибридное сравнение



Низкоуровневые оптимизации скорости

1. Простейшее побайтное сравнение буферов

```
for (int i = 0; i < buf1.Length && i < buf2.Length; i++)  
{  
    if (buf1[i] != buf2[i])  
    {  
        return false;  
    }  
}  
  
return true;
```

Слишком медленно, неэффективно

Низкоуровневые оптимизации скорости

2. Сравнение Int64-словами через fixed-pointer в unsafe-коде

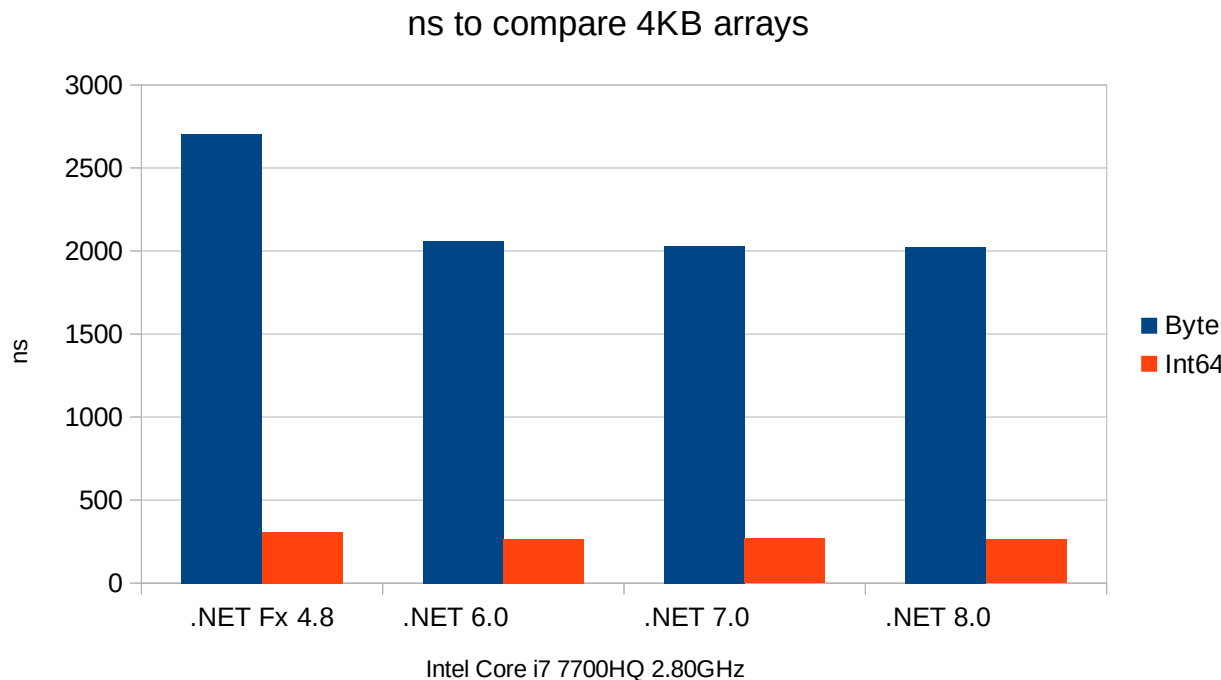
```
fixed (byte* pBuf1 = buf1, pBuf2 = buf2)
{
    int i8 = 0, i = 0;
    int size8 = size / sizeof(long);
    var p1 = (long*)pBuf1;
    var p2 = (long*)pBuf2;

    for (; i8 < size8; i8++)
    {
        if (p1[i8] != p2[i8])
        {
            return i8 * sizeof(long);
        }
    }

    i += i8 * sizeof(long);
    for (; i < size; i++)
    {
        if (pBuf1[i] != pBuf2[i]) { return i; }
    }
}
```

Низкоуровневые оптимизации скорости

Сравнение 4К-массивов побайтно и словами Int64 в BenchmarkDotNet



Ускорение в 9 раз

Низкоуровневые оптимизации скорости

3. AVX-сравнение через 32-байтный Vector<byte>

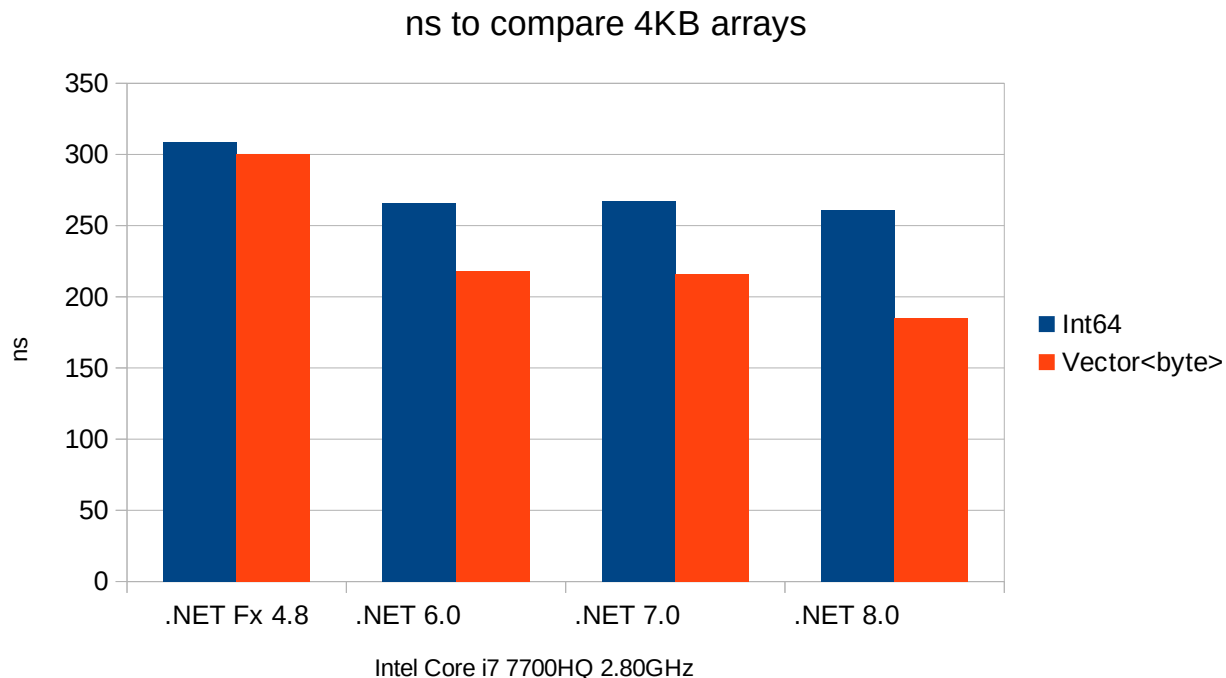
```
var countVectors = size / Vector<byte>.Count;
var sizeAligned = countVectors * Vector<byte>.Count;

for (int i = 0; i < sizeAligned; i += Vector<byte>.Count)
{
    var vector2 = new Vector<byte>(buf2, i);
    var vector1 = new Vector<byte>(buf1, i);

    if (vector1 != vector2)
    {
        return i;
    }
}
```


Низкоуровневые оптимизации скорости

Сравнение 4К-массивов словами Int64 и 256-битными векторами в BenchmarkDotNet



Ускорение всего на
10-20%

Низкоуровневые оптимизации скорости

3. AVX-сравнение через 32-байтный Vector<byte>, дизассемблерный код

M01_L00:

```
    cmp     eax,r10d
    jae     near ptr M01_L09
    lea     esi,[rax+1F]
    cmp     esi,r10d
    jae     near ptr M01_L09
    vmovupd ymm0,[rdx+rax+10]
    cmp     eax,r9d
    jae     near ptr M01_L09
    cmp     esi,r9d
    jae     near ptr M01_L09
    vmovupd ymm1,[rcx+rax+10]
    vpcmpeqb ymm0,ymm1,ymm0
    vpmovmskb esi,ymm0
    cmp     esi,0FFFFFFFF
    jne     near ptr M01_L06
    add     eax,20
    cmp     eax,r11d
    jl      short M01_L00
```

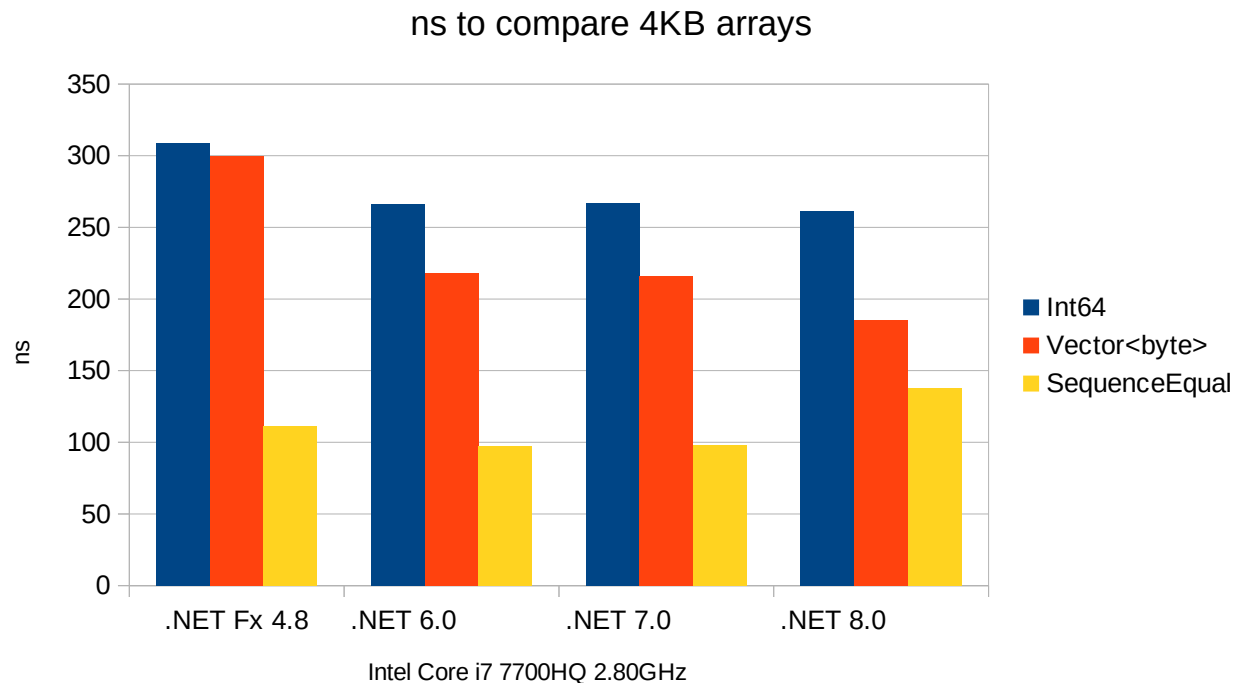
Низкоуровневые оптимизации скорости

4. AVX-сравнение через `MemoryExtensions.SequenceEqual` (Nuget `System.Memory`)

```
var span1 = buf1.AsSpan();  
var span2 = buf2.AsSpan();  
return span1.SequenceEqual(span2);
```

Низкоуровневые оптимизации скорости

Сравнение 4К-массивов тремя разными функциями BenchmarkDotNet



Ускорение в ~2 раза
от SequenceEqual

Низкоуровневые оптимизации скорости

AVX Vector<byte> vs SequenceEqual в дизассемблере (.NET 7.0)

| SequenceEqual | | CompareBuffersVectorized | |
|---------------|----------------------------|--------------------------|-------------------|
| M01_L01: | | M01_L00: | |
| vmovdqu | ymm0,ymmword ptr [rcx+rax] | cmp | eax,r10d |
| | | jae | near ptr M01_L09 |
| | | lea | esi,[rax+1F] |
| | | cmp | esi,r10d |
| | | jae | near ptr M01_L09 |
| | | vmovupd | ymm0,[rdx+rax+10] |
| | | cmp | eax,r9d |
| | | jae | near ptr M01_L09 |
| | | cmp | esi,r9d |
| | | jae | near ptr M01_L09 |
| | | vmovupd | ymm1,[rcx+rax+10] |
| vpcmpeqb | ymm0,ymm0,[rdx+rax] | vpcmpeqb | ymm0,ymm1,ymm0 |
| vpmovmskb | r9d,ymm0 | vpmovmskb | esi,ymm0 |
| cmp | r9d,0FFFFFFFF | cmp | esi,0FFFFFFFF |
| jne | short M01_L06 | jne | near ptr M01_L06 |
| add | rax,20 | add | eax,20 |
| cmp | r8,rax | cmp | eax,r11d |
| ja | short M01_L01 | j1 | short M01_L00 |

Выбор оптимального алгоритма хэширования

Хэширование файлов

$h = \text{hash}(\text{data})$

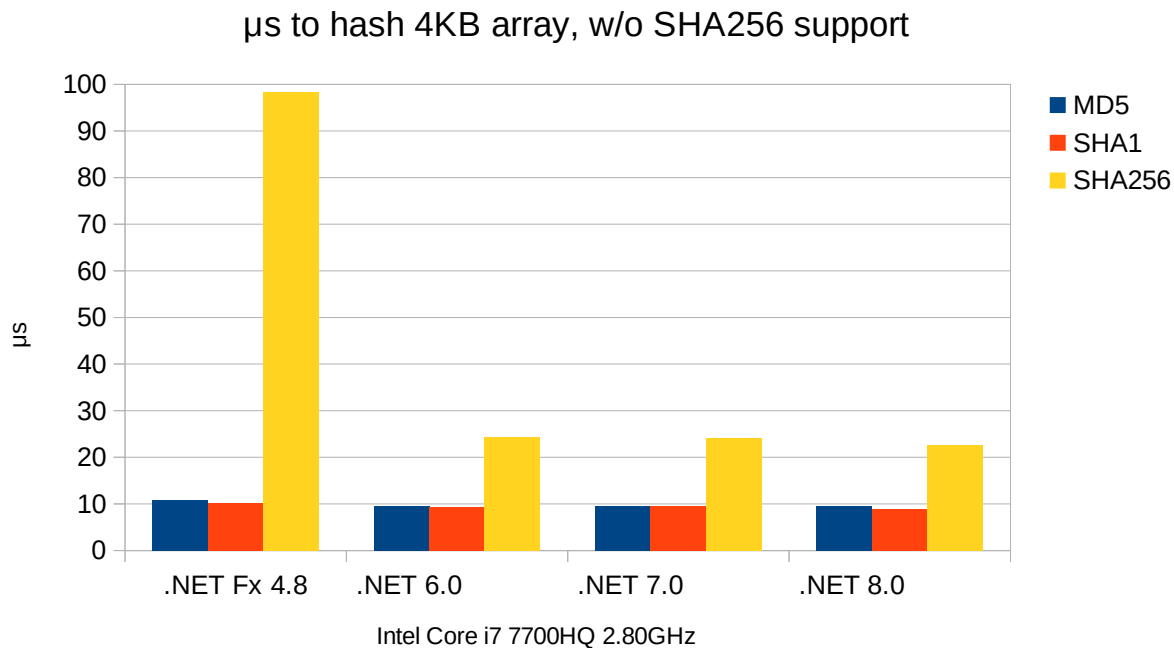


`SHA1("Hello") => f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0`

`SHA1("Hello.") => 9b56d519ccd9e1e5b2a725e186184cdc68de0731`

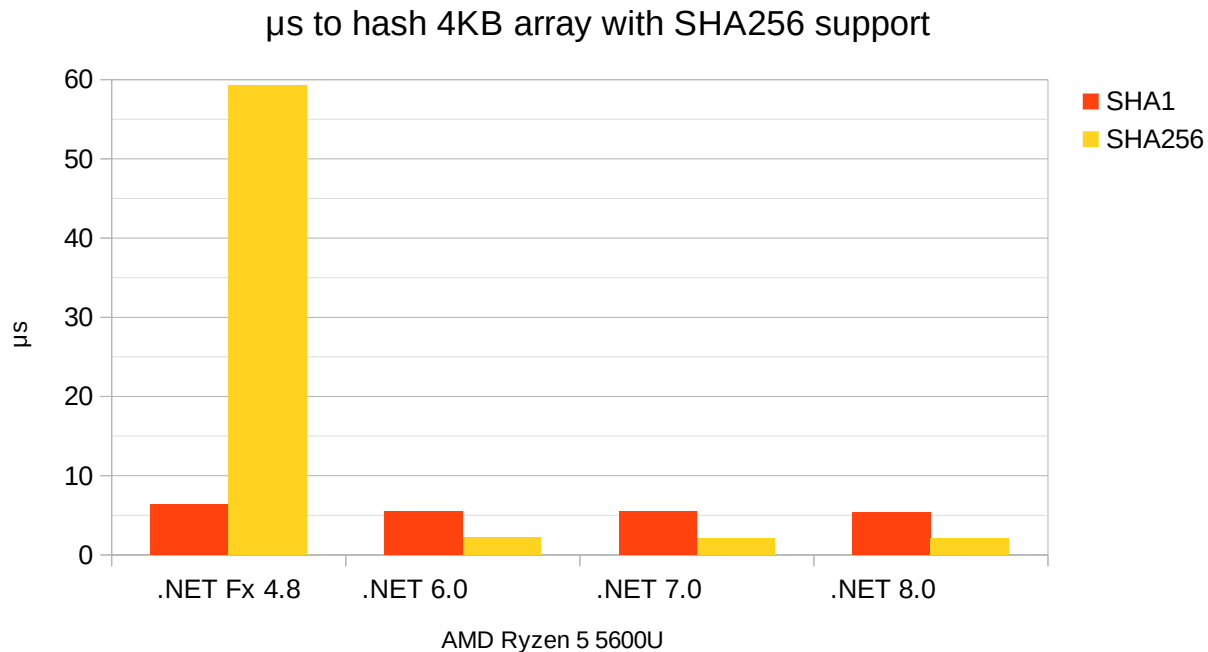
Выбор оптимального алгоритма хэширования

Сравнение SHA1 и SHA256 в BenchmarkDotNet на процессоре без поддержки Intel SHA extensions



Выбор оптимального алгоритма хэширования

Сравнение SHA1 и SHA256 в BenchmarkDotNet на процессоре с поддержкой Intel SHA extensions



| | SHA1 | SHA256 | Ratio |
|-------------|--------|---------|-------|
| .NET Fx 4.8 | 6,4 µs | 59,3 µs | 923% |
| .NET 6.0 | 5,5 µs | 2,1 µs | 39% |
| .NET 7.0 | 5,4 µs | 2,1 µs | 39% |
| .NET 8.0 | 5,4 µs | 2,1 µs | 39% |

Выбор оптимального алгоритма хэширования

Некриптографический хэш-алгоритм XxHash128

Реализован в пакете System.IO.Hashing 8.0

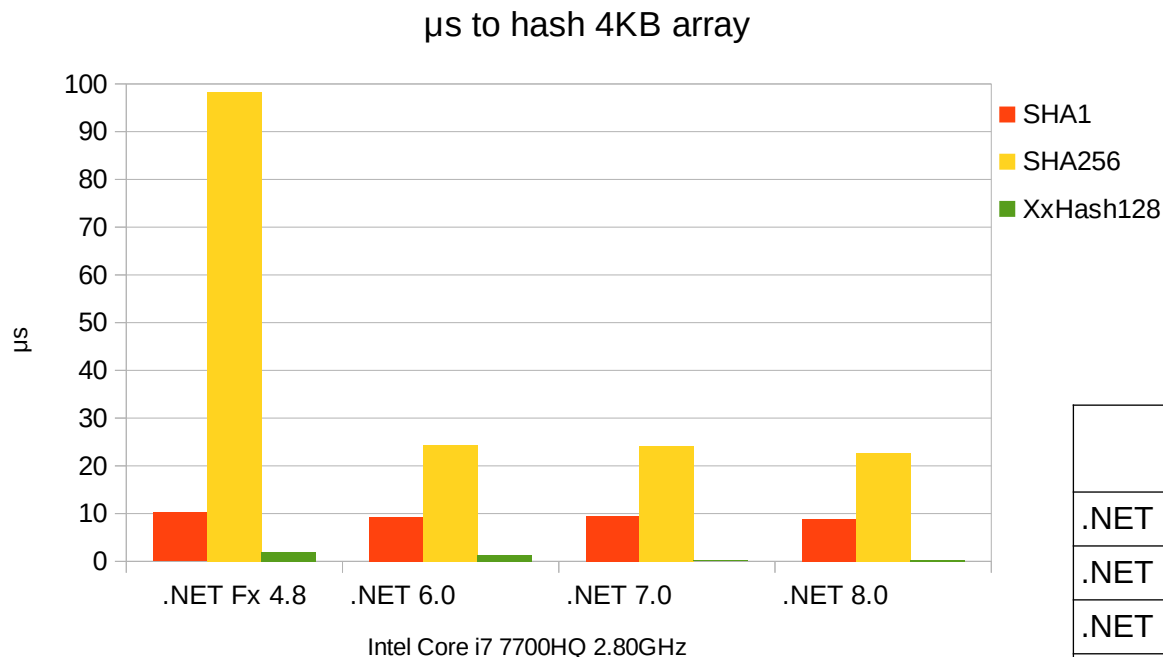
Достоинства:

- Скорость, оптимизирован для SSE2/AVX
- Длина 128 бит = 16 байт
- Низкий рейтинг коллизий

<https://github.com/Cyan4973/xxHash/wiki/Collision-ratio-comparison>

Выбор оптимального алгоритма хэширования

Сравнение SHA1 и SHA256, XxHash128 в BenchmarkDotNet

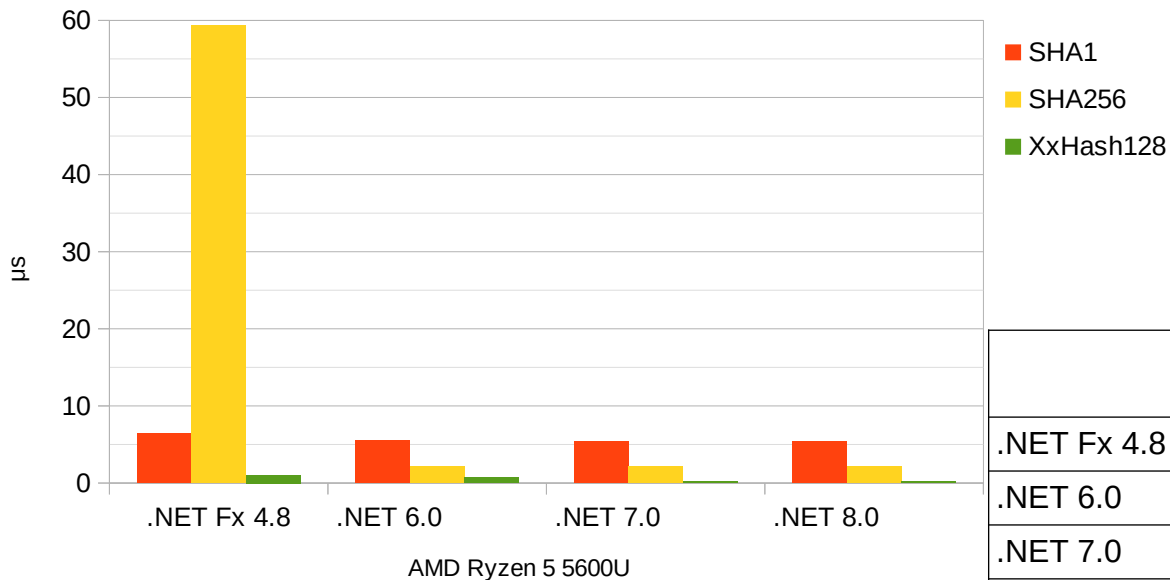


| | SHA1 | SHA256 | XxHash128 | Ratio vs SHA1 |
|-------------|---------|---------|-----------|---------------|
| .NET Fx 4.8 | 10,2 µs | 98,2 µs | 1,90 µs | 5,4x |
| .NET 6.0 | 9,3 µs | 24,4 µs | 1,30 µs | 7,2x |
| .NET 7.0 | 9,4 µs | 24,1 µs | 0,20 µs | 46,3x |
| .NET 8.0 | 8,8 µs | 22,6 µs | 0,27 µs | 32,9x |

Выбор оптимального алгоритма хэширования

Сравнение SHA1 и SHA256, XxHash128 в BenchmarkDotNet

µs to hash 4KB array (CPU with Intel SHA extension)



| | SHA1 | SHA256 | XxHash128 | Ratio vs SHA256 |
|-------------|--------|---------|-----------|-----------------|
| .NET Fx 4.8 | 6,4 µs | 59,3 µs | 1,04 µs | |
| .NET 6.0 | 5,5 µs | 2,1 µs | 0,70 µs | 3,0x |
| .NET 7.0 | 5,4 µs | 2,1 µs | 0,16 µs | 13,2x |
| .NET 8.0 | 5,4 µs | 2,1 µs | 0,19 µs | 10,9x |

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

1. Сравнение 64-битными словам через fixed-pointer *ulong

```
fixed (byte* pBuf1 = buf1, pBuf2 = buf2)
{
    var pLong1 = (ulong*)pBuf1;
    var pLong2 = (ulong*)pBuf2;

    return pLong1[3] == pLong2[3]
        && pLong1[2] == pLong2[2]
        && pLong1[1] == pLong2[1]
        && pLong1[0] == pLong2[0];
}
```

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

2. Сравнение 64-битными словам через Span<ulong>

```
var span1 = buf1.AsSpan();  
var span2 = buf2.AsSpan();  
var intLong1 = MemoryMarshal.Cast<byte, ulong>(span1);  
var intLong2 = MemoryMarshal.Cast<byte, ulong>(span2);  
return intLong1[3] == intLong2[3] &&  
       intLong1[2] == intLong2[2] &&  
       intLong1[1] == intLong2[1] &&  
       intLong1[0] == intLong2[0];
```

Низкоуровневые оптимизации скорости

| Обратный порядок | | Прямой порядок | | |
|------------------|---------------|----------------|---------------|----------|
| shr | ecx,3 | shr | ecx,3 | |
| shr | r10d,3 | shr | r10d,3 | |
| cmp | ecx,3 | test | ecx,ecx | [0]([3]) |
| jbe | short M01_L05 | je | short M01_L05 | |
| mov | rax,[r8+18h] | mov | rax,[r8] | |
| cmp | r10d,3 | test | r10d,r10d | [0]([3]) |
| jbe | short M01_L05 | je | short M01_L05 | |
| cmp | rax,[r9+18h] | cmp | rax,[r9] | |
| jne | short M01_L02 | jne | short M01_L02 | |
| | | cmp | ecx,1 | [1] |
| | | jbe | short M01_L05 | |
| mov | rax,[r8+10h] | mov | rax,[r8+8] | |
| | | cmp | r10d,1 | [1] |
| | | jbe | short M01_L05 | |
| cmp | rax,[r9+10h] | cmp | rax,[r9+8] | |
| jne | short M01_L02 | jne | short M01_L02 | |

| Обратный порядок | | Прямой порядок | | |
|------------------|---------------|----------------|---------------|-----|
| | | cmp | ecx,2 | [2] |
| | | jbe | short M01_L05 | |
| mov | rax,[r8+8] | mov | rax,[r8+10h] | |
| | | cmp | r10d,2 | [2] |
| | | jbe | short M01_L05 | |
| cmp | rax,[r9+8] | cmp | rax,[r9+10h] | |
| jne | short M01_L02 | jne | short M01_L02 | |
| | | cmp | ecx,3 | [3] |
| | | jbe | short M01_L05 | |
| mov | rax,[r8] | mov | rax,[r8+18h] | |
| | | cmp | r10d,3 | [3] |
| | | jbe | short M01_L05 | |
| cmp | rax,[r9] | cmp | rax,[r9+18h] | |
| sete | al | sete | al | |
| movzx | eax,al | movzx | eax,al | |

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

3. AVX- и SSE2-сравнение через Vector<byte>

```
if (Vector<byte>.Count == 32)
{
    var vector2 = new Vector<byte>(buf2, 0);
    var vector1 = new Vector<byte>(buf1, 0);
    return vector1 == vector2;
}

if (Vector<byte>.Count == 16)
{
    var vector12 = new Vector<byte>(buf1, 16);
    var vector22 = new Vector<byte>(buf2, 16);
    return vector12 == vector22
        && new Vector<byte>(buf1, 0) == new Vector<byte>(buf2, 0);
}
```

Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

4. SequenceEqual (const 32)

```
var span1 = buf1.AsSpan(0, 32);  
var span2 = buf2.AsSpan(0, 32);  
return span1.SequenceEqual(span2);
```

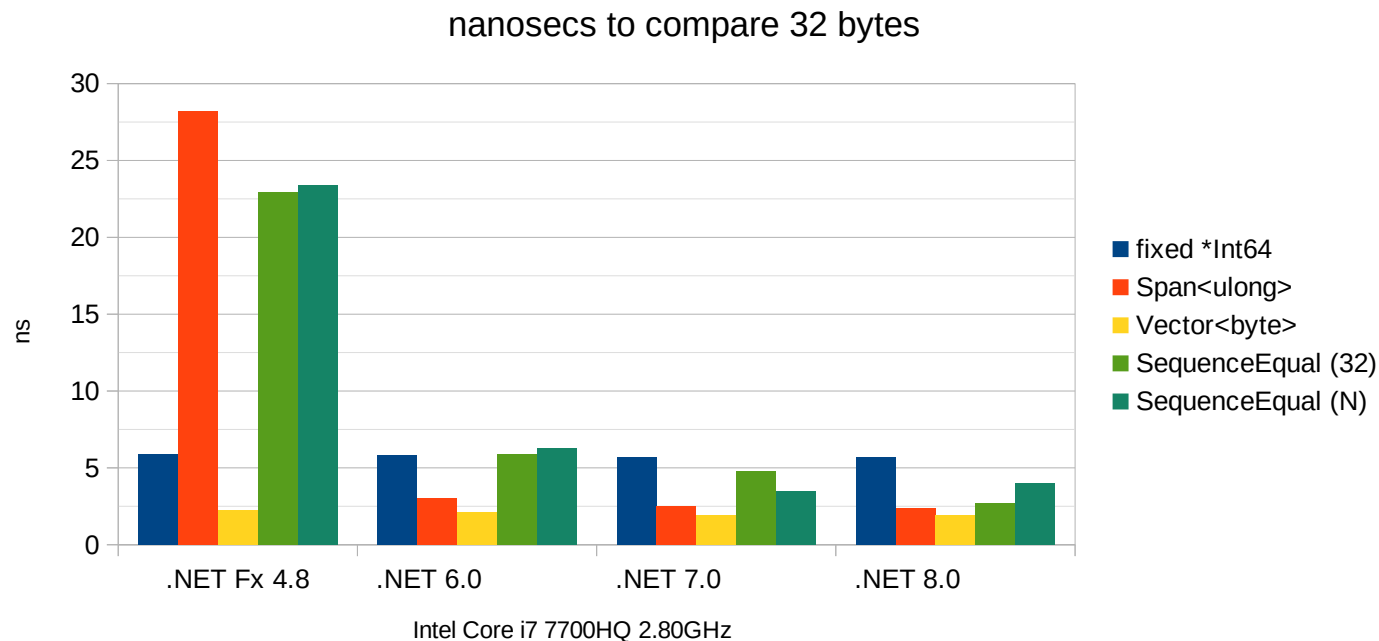
5. SequenceEqual (N)

```
var buf1Span = buf1.AsSpan(0, size);  
var buf2Span = buf2.AsSpan(0, size);  
var res = buf1Span.SequenceEqual(buf2Span);  
return res;
```


Низкоуровневые оптимизации скорости

Сравнение хэш-значений размером 32 байта между собой

Сравнение всех реализаций по скорости в BenchmarkDotNet



Файловые операции и многопоточность

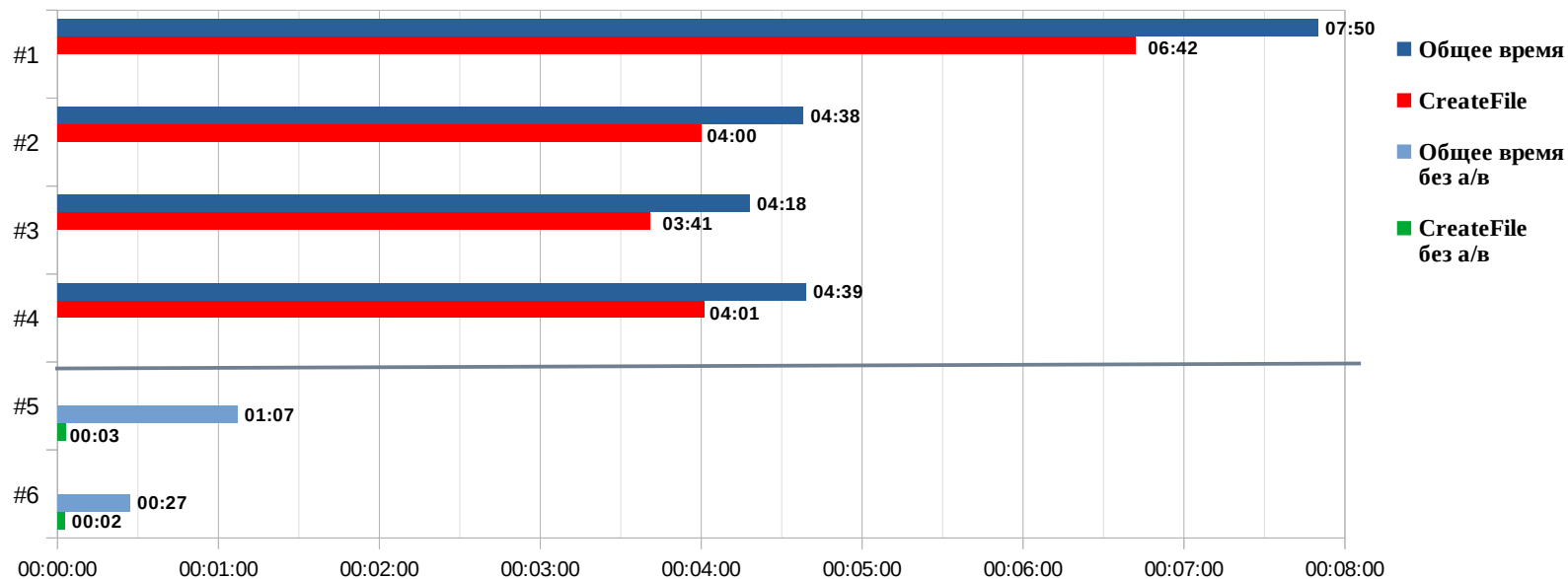
Какая самая «дорогая»/медленная системная функция в программе?

CreateFile(..)!

- Чтение с диска файловых атрибутов метаданных из таблиц
- Проверка прав доступа пользователя
- Самая ресурсоёмкая операция: проверка антивирусом(!)

Файловые операции и многопоточность

Сравнение 63 000 файлов в 1 поток с антивирусом и без



4 минуты = в среднем ~4 мс на 1 файл

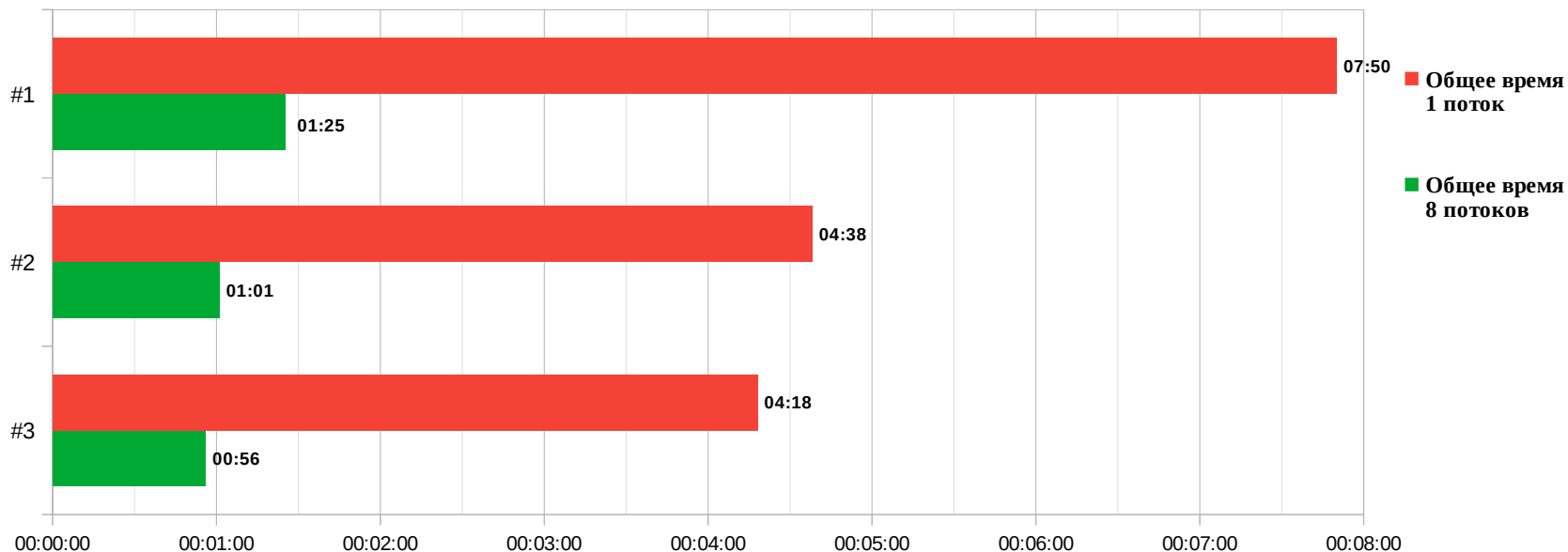
Файловые операции и многопоточность

Оптимизации

1. Сокращение количества открытий файлов
2. Совмещение сравнения с хэшированием
3. Заккрытие файловых стримов происходит в фоновом потоке
4. Многопоточность для SSD и/или независимых HDD

Файловые операции и многопоточность

Прирост от многопоточности при поиске дубликатов на SSD



Файловые операции и многопоточность

Задачи при синхронизации доступа потоков к дискам

- Определить, какие физические диски в системе являются SSD, какие HDD, а какие прочими носителями (через WinAPI, WMI)
- Определить разделы носителей и назначенные им буквы (через WMI)
- Обеспечить синхронизацию одновременного доступа с учётом типа носителя

Файловые операции и многопоточность

Кто знает, для чего эти металлические стержни?



Файловые операции и многопоточность

Это жезлы (**tokens**) системы блокировки ж/д перегона



Файловые операции и многопоточность

Распараллеливание сравнения файлов

Классы управления доступом

- DriveAccessManager (управляет семафорами для носителей, раздаёт токены)
- DriveAccessSemaphore (регулирует доступ к носителю)
- SingleEnterToken (токен одиночной блокировки для одного файла)
- DoubleEnterToken (токен двойной блокировки для файлов на разных носителях)

Файловые операции и многопоточность

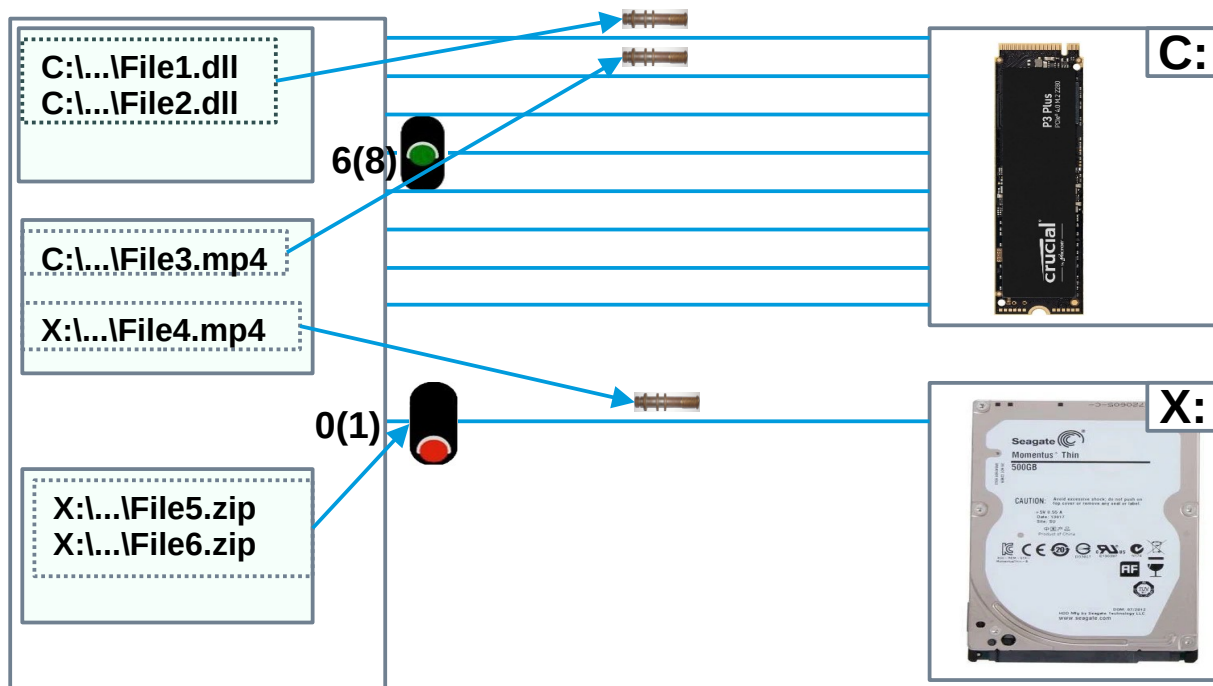
Распараллеливание сравнения файлов

- Файлы группируются по размеру
- Группы файлов одинакового размера сравниваются параллельно через **Parallel.ForEach()**
- Ограничение доступа к дискам регулируется через семафоры, выдаваемые `DriveAccessManager`
- На каждое сравнение файлов выдаётся токен, освобождается после сравнения

Файловые операции и многопоточность

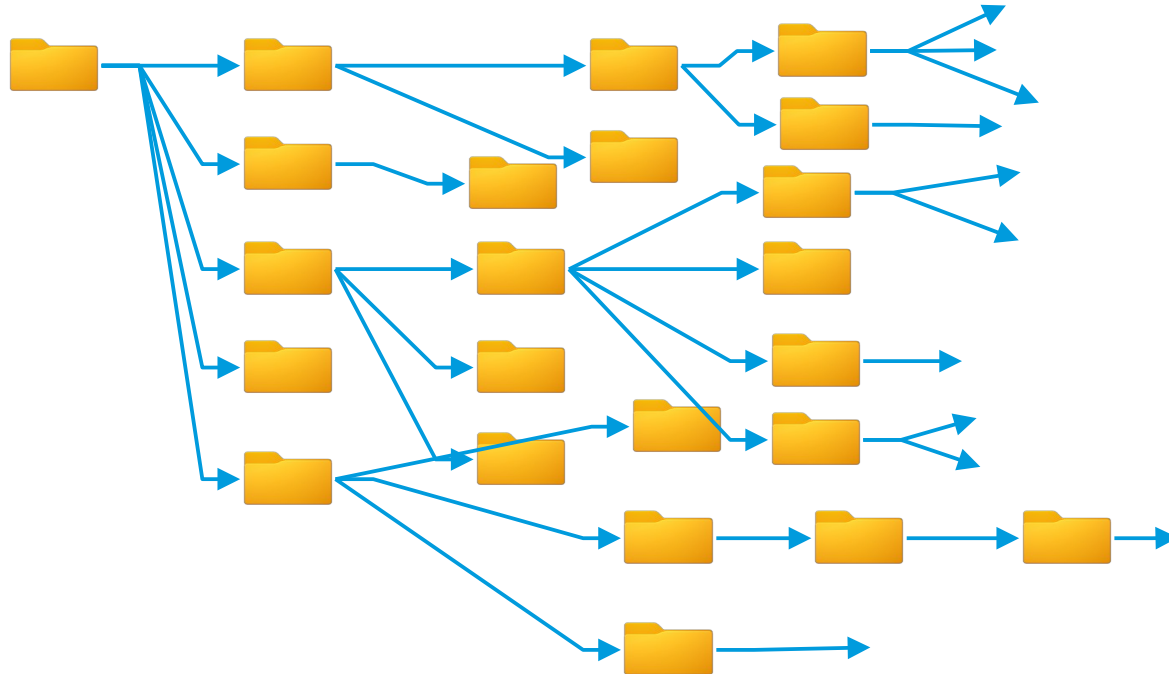
Распараллеливание при сравнении файлов

О семафорах и токенах в программе.



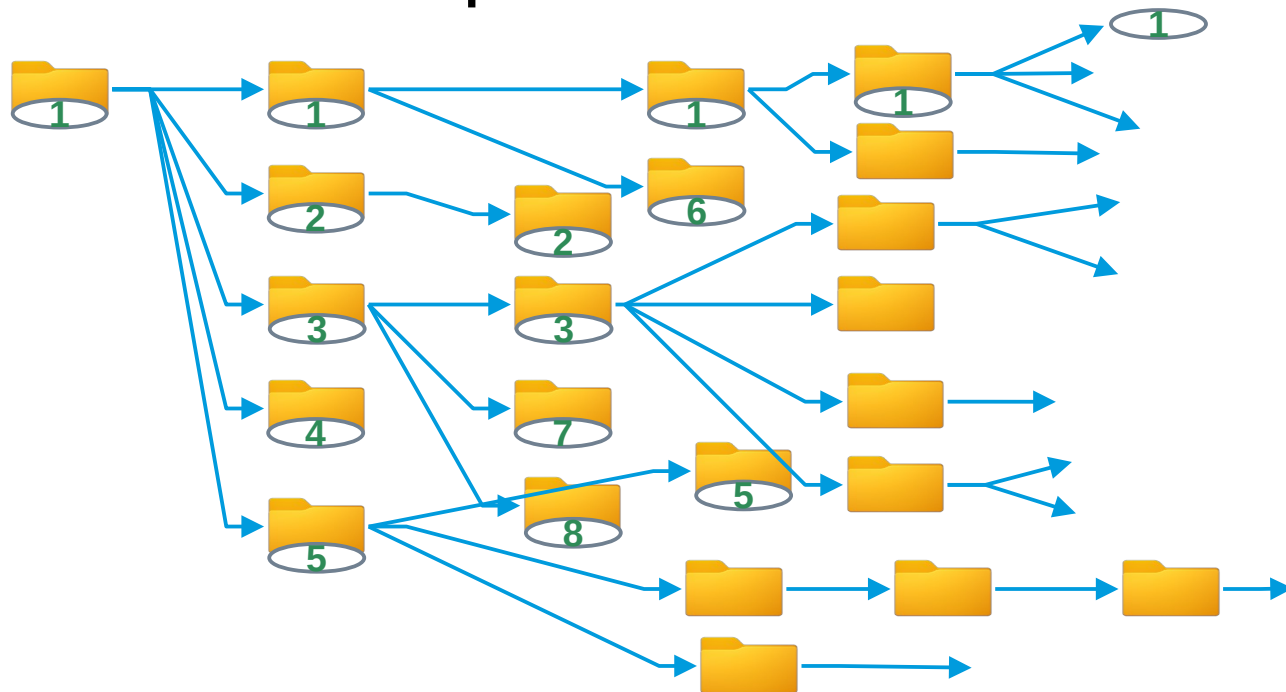
Файловые операции и многопоточность

Параллельное сканирование каталогов



Файловые операции и многопоточность

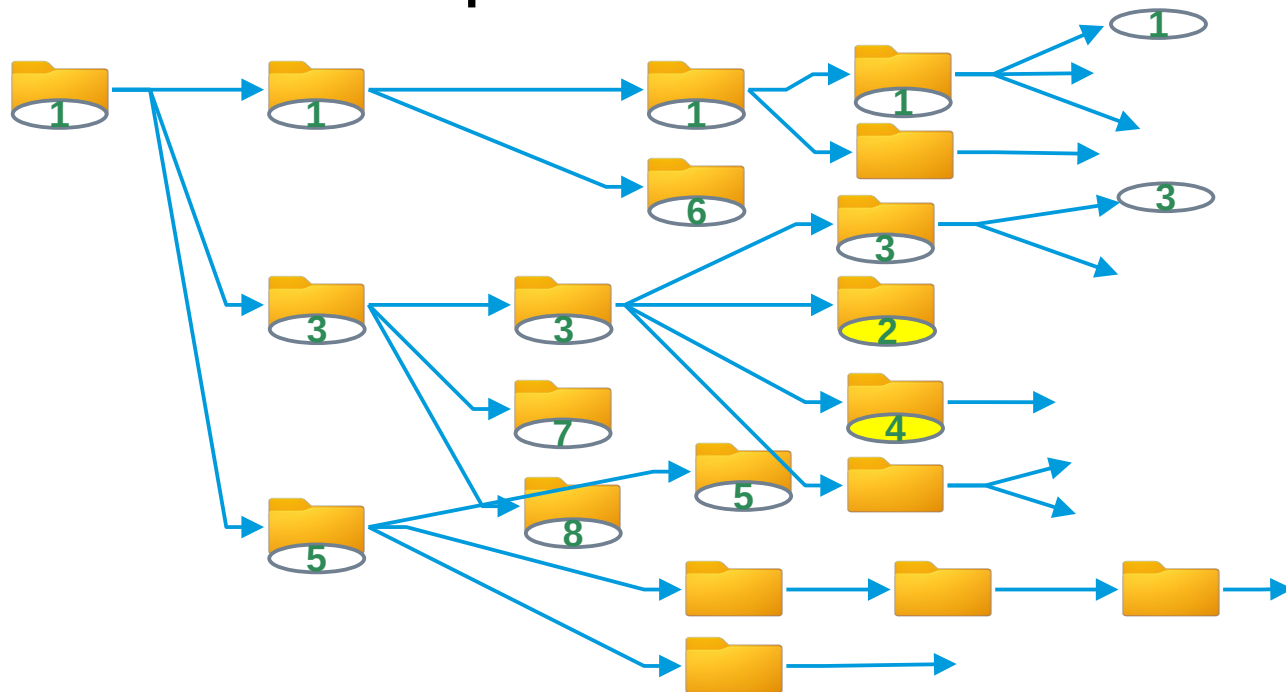
Параллельное сканирование каталогов



Parallel.ForEach + MaxDegreeOfParallelism + Interlocked-счётчик

Файловые операции и многопоточность

Параллельное сканирование каталогов



Parallel.ForEach + MaxDegreeOfParallelism + Interlocked-счётчик

Методы экономии памяти

Что общего в этих строках?

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\ProjectTemplates"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\PublicAssemblies"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\ReferenceAssemblies"

Методы экономии памяти

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\ProjectTemplates"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\PublicAssemblies"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\ReferenceAssemblies"

Методы экономии памяти

Решение — класс `AbsolutePath`

```
internal class AbsolutePath : IEquatable<AbsolutePath>, IComparable<AbsolutePath>
{
    public AbsolutePath? Parent { get; }

    public string Name { get; }

    public int Length => GetLength();

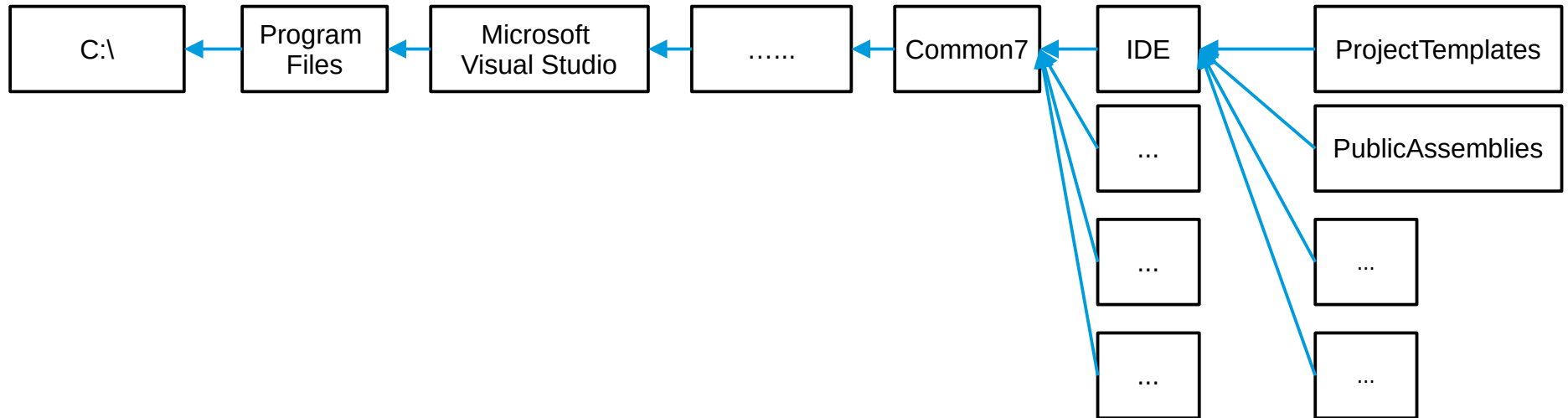
    public int Levels => GetLevels();

    public AbsolutePath(string name, AbsolutePath? parent)
    {
        Parent = parent;
        Name = GetSharedName(name ?? string.Empty);
    }
}
```

Методы экономии памяти

Класс `AbsolutePath`

Связный список узлов для одного пути



Методы экономии памяти

— Некоторые недостатки **AbsolutePath** против строк

- Нужно выделять временную строку для WinApi-функций
- Динамическое вычисление длины строк

Методы экономии памяти

+ Преимущества **AbsolutePath** против строк

- Строгая типизация
- Использует значительно меньше памяти в целом
- Ссылки на имя, родительский и корневой каталоги без выделения памяти
- Одинаковые имена каталогов и файлов не дублируются
- Пути между собой сравниваются намного быстрее, чем строки

Методы экономии памяти

+ Преимущества **AbsolutePath** против строк

Сравнение на равенство двух таких путей

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\Packages\00098"

"c:\Program Files\Microsoft Visual Studio\2022\Community\Common7\Packages\00099"

в BenchmarkDotNet на .NET 7.0

- `string.Equals(OrdinalIgnoreCase)`: **48 нс**
- `AbsolutePath.EqualsIgnoreCase`: **2 нс**

Методы экономии памяти

+ Преимущества **AbsolutePath** против строк

- AbsolutePathDriveRoot - быстрое получение информации о корневой папке и томе по пути.

```
internal sealed class AbsolutePathDriveRoot : AbsolutePath
{
    public override string FileSystemName { get; }

    public override Win32.FileSystemFlags FileSystemFlags { get; }

    public override DriveType DriveType { get; }

    public override string VolumeName { get; }

    public override bool IsOnSsd { get; set; }

    public override uint SectorSize { get; }
```

Методы экономии памяти

+ Преимущества **AbsolutePath** против строк

- **AbsolutePathSymlinkPoint** - встроенная поддержка символьных ссылок

```
internal sealed class AbsolutePathSymlinkPoint : AbsolutePath
{
    public AbsolutePath SymlinkTarget { get; }

    public override AbsolutePath FinalPath => SymlinkTarget;

    public override bool IsSymlink => true;

    public override bool IsSymlinkPoint => true;

    public AbsolutePathSymlinkPoint(string name, AbsolutePath? parent, AbsolutePath symlinkTarget) :
        base(name, parent)
    {
        SymlinkTarget = symlinkTarget;
    }
}
```

Методы экономии памяти

ArrayPool<T>

Вместо библиотечного свойства **ArrayPool<byte>.Shared** программа использует разделяемый экземпляр, созданный **ArrayPool<byte>.Create()**

Преимущества

- Меньше суммарный объём памяти, выделенной на массивы
- Позволяет освободить ссылку на выделенную память после использования
- В моих сценариях скорость одинакова на .NET 7.0, но реализация через Create() быстрее на .NET Fx 4.8

Методы экономии памяти

ListPool<T>

Хранит List<T> и, подобно ArrayPool<T>, используется для уменьшения трафика памяти

```
internal sealed class ListPool<T>
{
    public static int MaxCount { get; set; } = Math.Max(32, Environment.ProcessorCount * 2);

    public static ListPool<T> Shared { get; } = new ();

    private ConcurrentBag<List<T>> _listContainer = new ();
    private int _countInContainer = 0;

    [ThreadStatic] private static Box<List<T>>? _box;

    private static Box<List<T>> BoxedList { ... }

    public List<T> Rent() { ... }

    public void Return(ref List<T>? listRef) { ... }
```

Методы экономии памяти

ListPool<T>

Rent() - аренда

```
var box = BoxedList;
var list = box.Item;
if (list is not null)
{
    box.Item = null;
    return list;
}

if (_countInContainer > 0 && _listContainer.TryTake(out list))
{
    Interlocked.Decrement(ref _countInContainer);
    return list;
}

return new List<T>();
```

Методы экономии памяти

ListPool<T>

Return() - возврат

```
var list = listRef;
listRef = null;
if (list is null || _countInContainer >= MaxCount) { return; }

list.Clear();

var box = BoxedList;
if (box.Item is null)
{
    box.Item = list;
    return;
}

if (_countInContainer < MaxCount)
{
    _listContainer.Add(list);
    Interlocked.Increment(ref _countInContainer);
}
```

Методы экономии памяти

ListPool<T>

Box<List<T>>

```
public class Box<T>
{
    public T? Item { get; set; }
}
```

Методы экономии памяти

ListPool<T>

RentedListWrapper<T>

```
internal struct RentedListWrapper<T> : IDisposable
{
    private List<T>? _rentedList;

    public List<T> List => _rentedList ?? new List<T>();

    public RentedListWrapper(List<T> cachedList)
    {
        _rentedList = cachedList;
    }

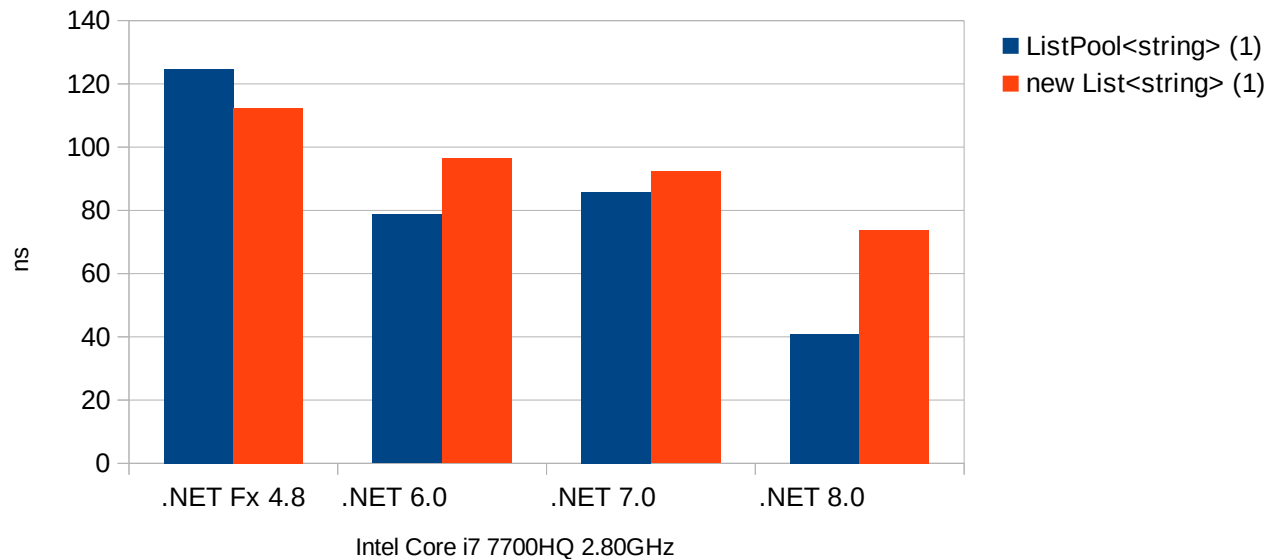
    public void Dispose()
    {
        ListPool<T>.Shared.Return(ref _rentedList);
    }
}

using var wrappedList = ListPool<FileNameSize>.Shared.RentWrapped();
```

Методы экономии памяти

ListPool<T>

- Сравнение по скорости ListPool<string>.Rent() и new List<string>() для одного списка с добавлением 6-ти объектов в список

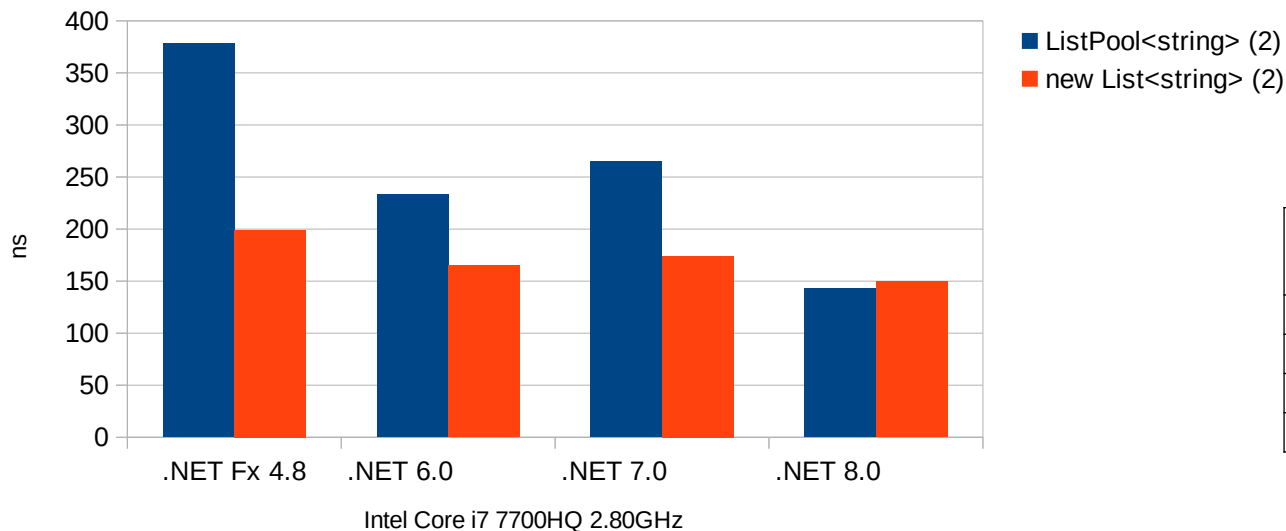


| | ListPool<T> (1) | Alloc | new List<T> (1) | Alloc |
|-------------|--------------------|-------|--------------------|-------|
| .NET Fx 4.8 | 124,8 ns | 0 | 112,3 ns | 185 Б |
| .NET 6.0 | 78,7 ns | 0 | 96,4 ns | 176 Б |
| .NET 7.0 | 85,9 ns | 0 | 92,3 ns | 176 Б |
| .NET 8.0 | 40,9 ns | 0 | 73,7 ns | 176 Б |

Методы экономии памяти

ListPool<T>

- Сравнение по скорости ListPool<string>.Rent() и new List<string>() для 2-х списков с добавлением 6-ти объектов в каждый список



| | ListPool<T> (2) | Alloc | new List<T> (2) | Alloc |
|-------------|-----------------|-------|-----------------|-------|
| .NET Fx 4.8 | 378,3 ns | 0 | 199,1 ns | 369 B |
| .NET 6.0 | 233,1 ns | 0 | 164,9 ns | 352 B |
| .NET 7.0 | 264,8 ns | 0 | 173,9 ns | 352 B |
| .NET 8.0 | 142,7 ns | 0 | 149,7 ns | 352 B |

Методы экономии памяти

StringBuilderPool

Хранит StringBuilder, аналогичен ListPool<T>

```
internal sealed class StringBuilderPool
{
    public static int MaxCount { get; set; } = 32;
    public static StringBuilderPool Shared { get; private set; } = new StringBuilderPool();

    private int _countInContainer;

    private ConcurrentBag<StringBuilder> _stringBuilders = new ();

    [ThreadStatic] private static Box<StringBuilder>? _box;

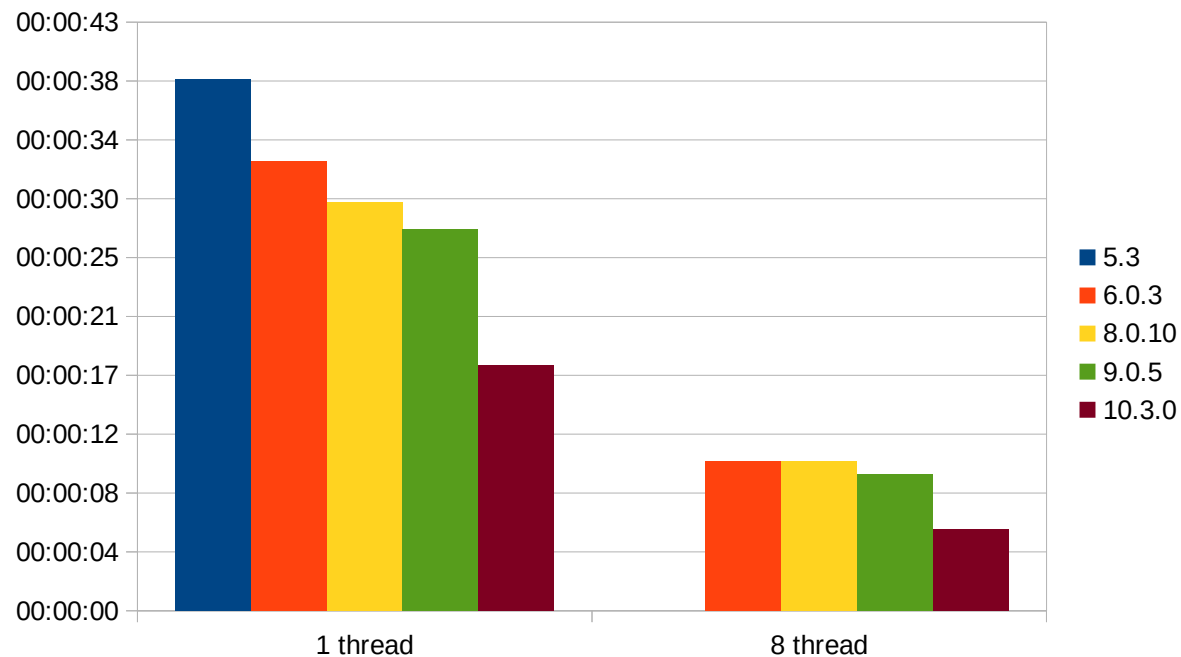
    private static Box<StringBuilder> BoxedStringBuilder{ ... }

    public StringBuilder Rent() { ... }

    public void Return(StringBuilder? sb){ ... }
```



Заключение

Комплексный прирост от всех оптимизаций разных версий



Заключение

Отзывы :)

| | |
|---|--|
| Всего записей: 67 Зарегистр. 26-02-2012 <u>Отправлено:</u> 13:34 03-04-2023 | |
| la_tangram | Редактировать Профиль Сообщение Цитировать Сообщить модератору |
|  Junior Member | rj12 Версией 10.2.1 просканировал 90 Гиг данных на SSD за 1 минуту, в несколько потоков - это фантастика!)) Ранее, чем мельче файлы проверялись, тем медленнее шёл процесс. Спасибо! |
| Всего записей: 141 Зарегистр. 15-03-2005 <u>Отправлено:</u> 07:47 04-04-2023 | |

Спасибо за внимание!

Вопросы?

Контакты:

email: yurymalich@yandex.ru

 @Yury_Malich

web: http://www.malich.org/duplicate_searcher

XING: https://www.xing.com/profile/Yury_Malich