Основы профилирования и оптимизации .NET приложений

SEP 30 2017

Ключевые темы

- Что такое профилирование
- Утечка памяти
- Оптимизация памяти
- Производительность
- Взаимные блокировки

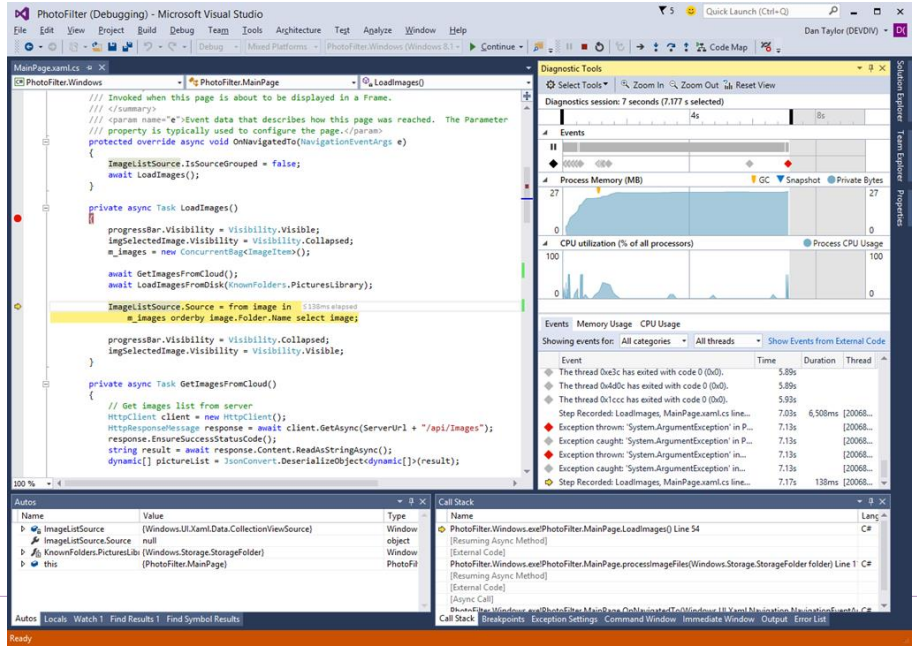
Часть 1:
Что такое профилирование

Что такое отладчик

Отладка – этап разработки программы, на котором проверяется корректность работы программы.

Отладчик

- “Debug” сборка приложения
- Прерывание выполнения программы
- Пошаговое выполнение приложения
- Изменения хода выполнения программы
- Влияние на выполнение программы

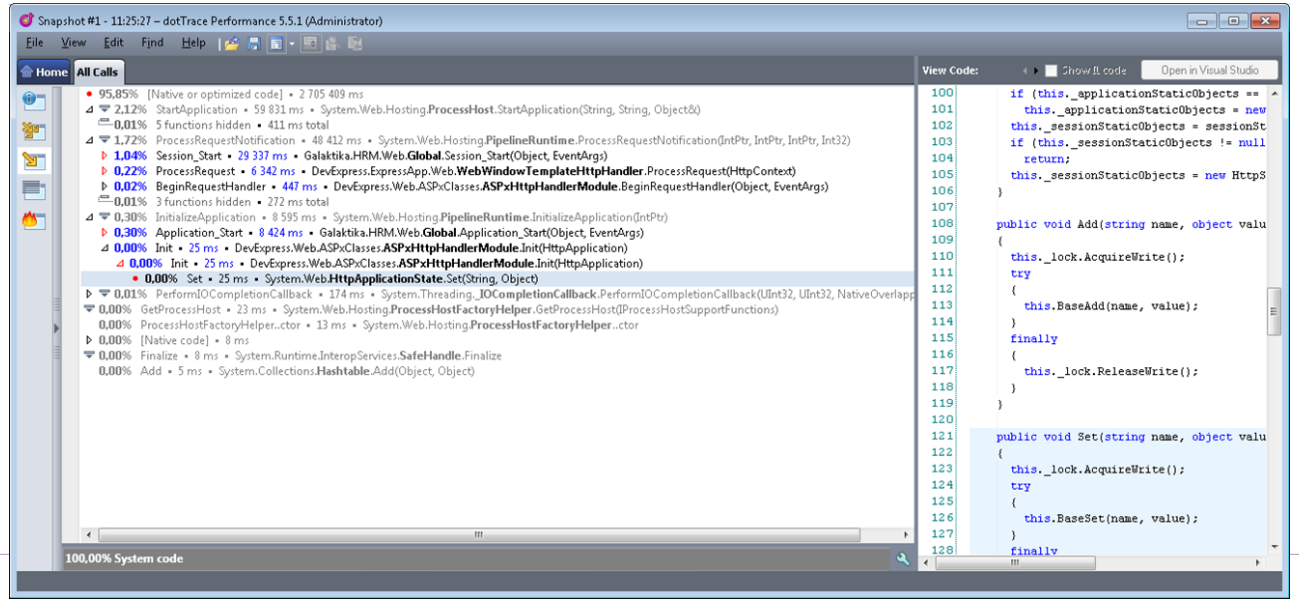


Что такое профилирование

Профилирование – процесс сбора количественных характеристик, во время работы программы.

Профайлер

- “Release” сборка приложения
- Работа программы не прерывается
- Сбор количественных данных во время выполнения
- Минимальное влияние на выполнение программы



Как это работает

Профилирование производительности:

- Информация методе
- Время выполнения метода

Профилирование памяти:

- Количество объектов созданных в теле метода
- Тип объектов которые создаются

Получение результатов:

- ETW Events
- .NET Remoting

```
0 references
class Program
{
    1 reference
    static void Run()
    {
        Stopwatch sw = new Stopwatch();
        sw.Start();
        int n = 10000;
        int j;
        int[] result = new int[n * n];
        for (int i = 0; i < n; i++)
        {
            j = 0;
            int pos = i * n;
            for (j = 0; j < n; j++)
            {
                result[pos + j] = pos + j;
            }
        }
        sw.Stop();

        Console.WriteLine("Time:{0}ms", sw.ElapsedMilliseconds);
        Console.WriteLine("StackTrace:{0}", Environment.StackTrace);

    }

    0 references
    static void Main(string[] args)
    {
        Run();

        Console.WriteLine("Please press any key...");
        Console.ReadKey();
    }
}
```

```
Time:232ms
StackTrace:   at System.Environment.GetStackTrace(Exception e, Boolean needFileInfo)
   at System.Environment.get_StackTrace()
   at Profiling.Program.Run() in C:\Data\Projects\Profiling\Profiling\Program.cs:line 34
   at Profiling.Program.Main(String[] args) in C:\Data\Projects\Profiling\Profiling\Program.cs:line 42
   at System.AppDomain._nExecuteAssembly(RuntimeAssembly assembly, String[] args)
   at System.AppDomain.ExecuteAssembly(String assemblyFile, Evidence assemblySecurity, String[] args)
   at Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
   at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object stat
e, Boolean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state, Boole
an preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
Please press any key...
```

Часть 2:
Утечки памяти

Профилирование памяти

УТЕЧКА ПАМЯТИ

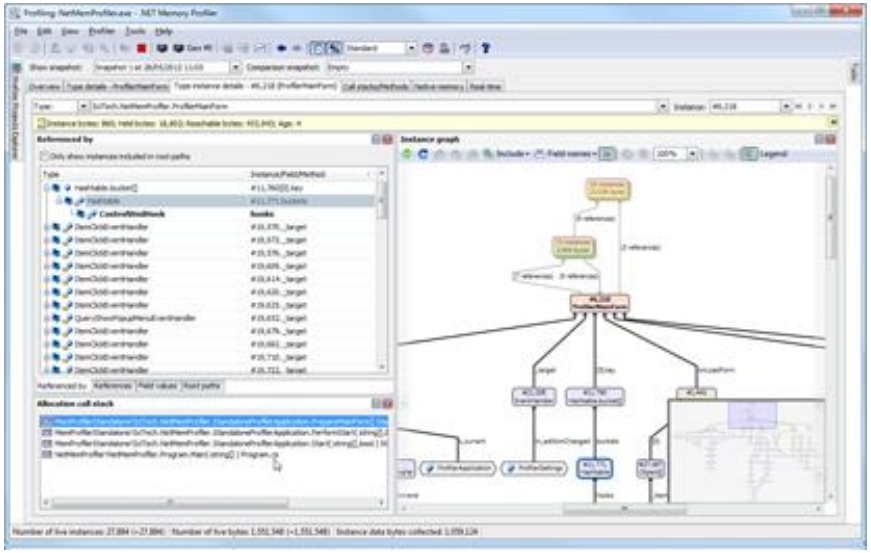
Выявление функционала для профилирования, подготовка и запуск теста

Ключевые параметры:

- Количество выделенной памяти
- Количество освобожденной памяти
- Количество созданных объектов по типам
- Количество памяти занимаемое каждым типом объектов

Действия:

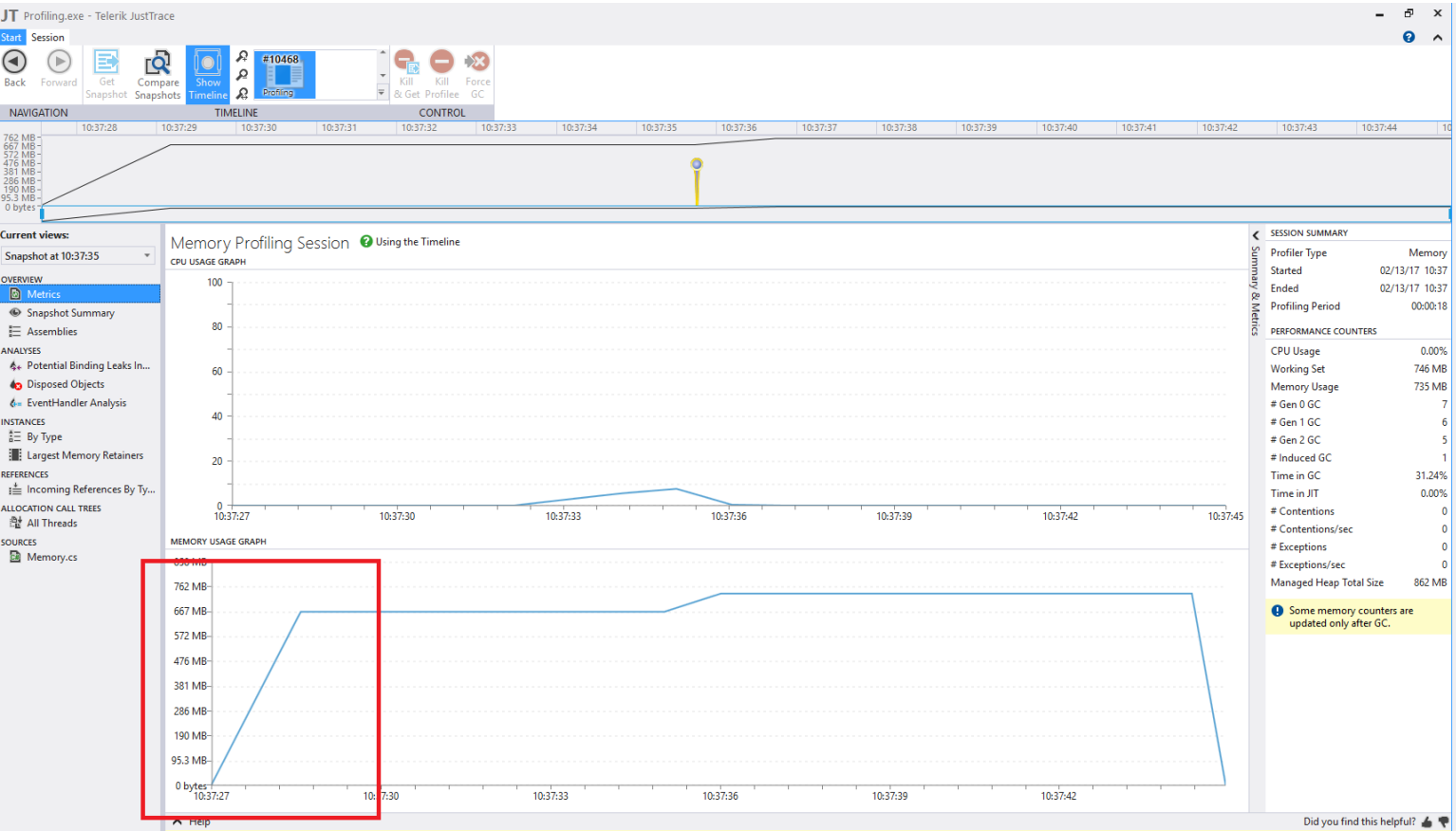
- **Шаг 1:** Выделение объектов, память которых не освобождается
- **Шаг 2:** Выделение участков кода, в которых создаются объекты, память которых не освобождается
- **Шаг 3:** Внесение изменений и повторный запуск теста
- **Шаг 4:** Сравнение результатов, если результат не достигнут, переходим на **Шаг 1**



Шаг №1: Выделение памяти

- Количество выделенной памяти
- Количество освобожденной памяти

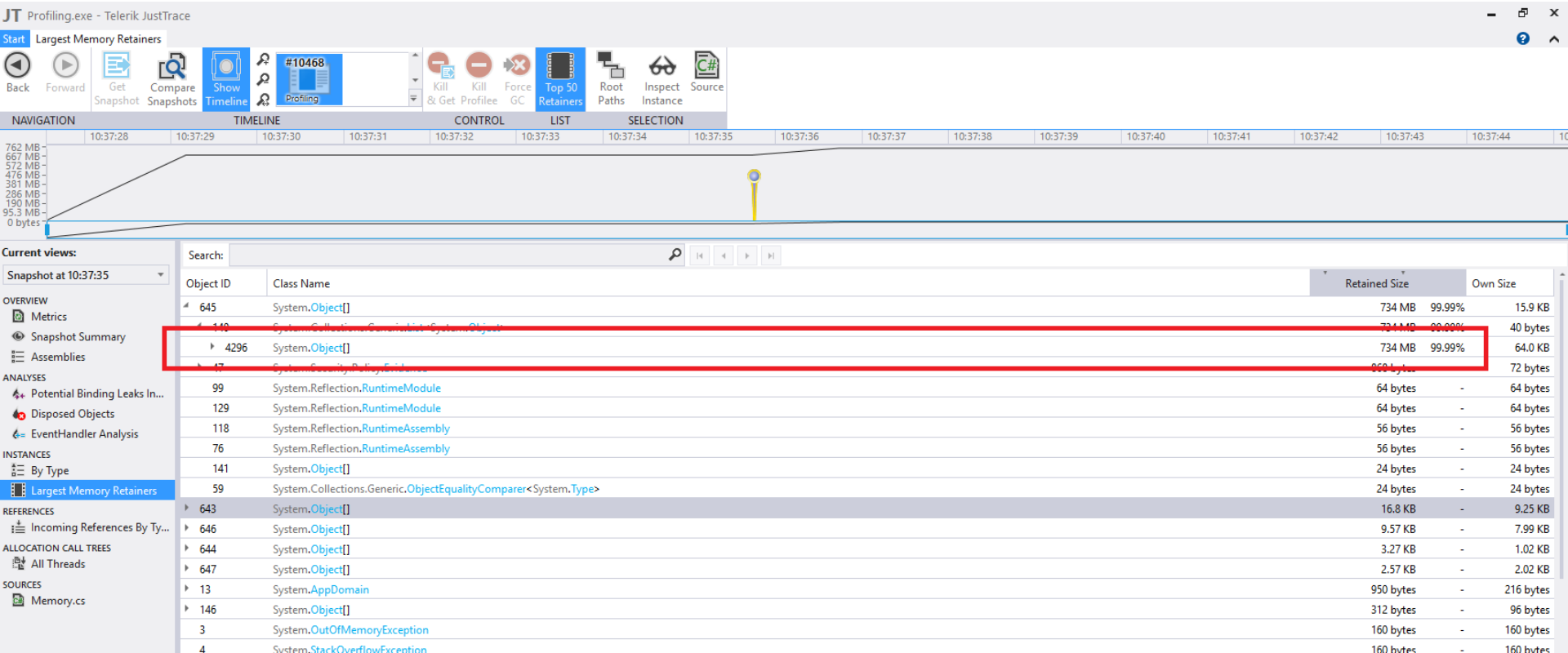
```
public static List<object> _objects = new List<object>();

1 reference
public static void MemoryLeak()
{
    string s = String.Format("test string");
    for (int i = 0; i < 7000; i++)
    {
        s = s +String.Format("test string {0}", i);
        _objects.Add(s);
    }
}
```



Шаг №1: Типы созданных объектов

- Количество созданных объектов по типам

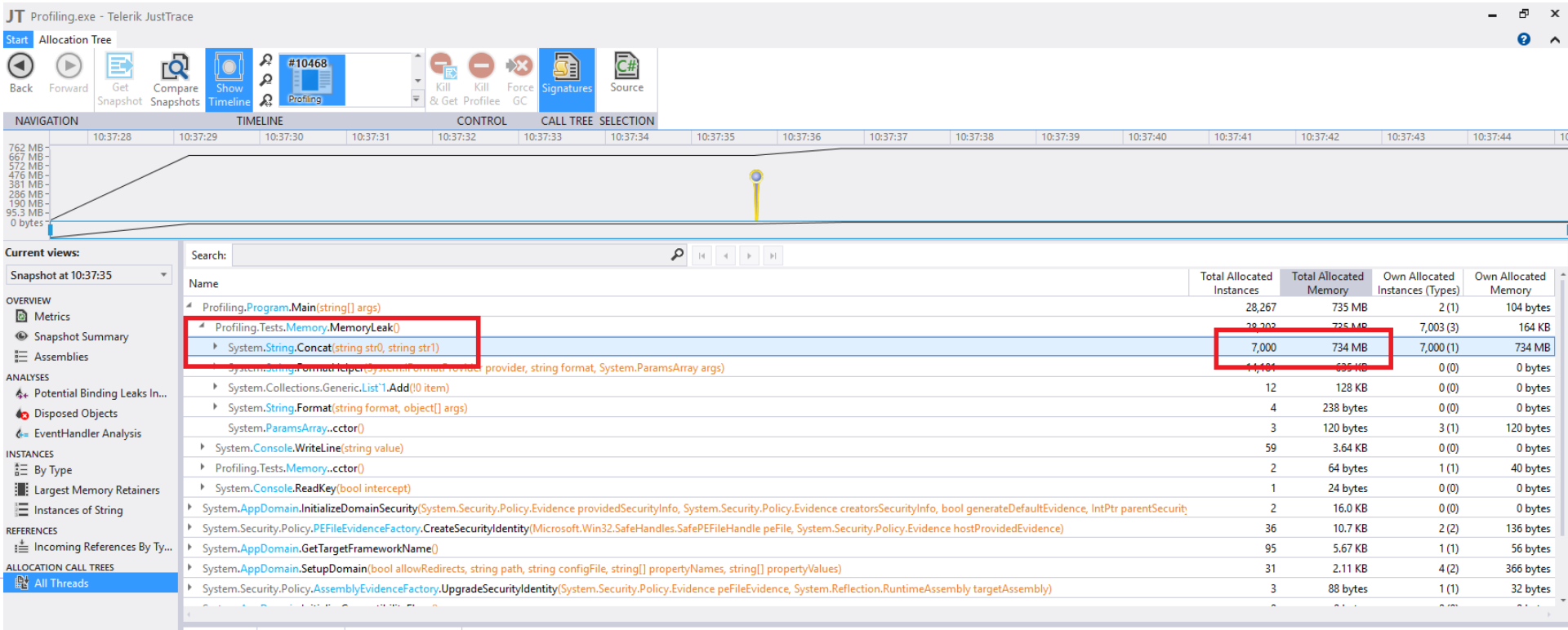
```
public static List<object> _objects = new List<object>();

1 reference
public static void MemoryLeak()
{
    string s = String.Format("test string");
    for (int i = 0; i < 7000; i++)
    {
        s = s +String.Format("test string {0}", i);
        _objects.Add(s);
    }
}
```



Шаг №1: Количество созданных объектов

- Точка создания объектов
- Количество памяти занимаемое каждым типом объектов

```
public static List<object> _objects = new List<object>();

1 reference
public static void MemoryLeak()
{
    string s = String.Format("test string");
    for (int i = 0; i < 7000; i++)
    {
        s = s +String.Format("test string {0}", i);
        _objects.Add(s);
    }
}
```



Часть 3:
Потребление памяти

Профилирование памяти

ОПТИМИЗАЦИЯ ПАМЯТИ

Выявление функционала для профилирования, подготовка и запуск теста

Ключевые параметры:

- Количество выделенной памяти
- Количество освобожденной памяти
- Количество созданных объектов по типам
- Количество памяти занимаемое каждым типом объектов

Действия:

- **Шаг 1:**Выделение объектов которые активней всего используют оперативную память
- **Шаг 2:**Выделение участков кода, в которых создаются объекты.
- **Шаг 3:**Внесение изменений и повторный запуск кода
- **Шаг 4:** Сравнение результатов, если результат не достигнут переходим на **Шаг 1**

Оптимизация памяти

Легкий уровень сложности:

- Изменение стратегии работы GC
- Определение избыточного набора данных, поиск случаев когда можно обойтись ограниченным набором данных
- Использование Cache которые расположены не в оперативной памяти

Средний уровень сложности:

- Искусственное ограничение, времени жизни объекта
- Принудительное удаление объекта, не дожидаясь когда сработает GC

Тяжелый уровень сложности:

- Оптимизация размера данных
- Компрессия/декомпрессия данных
- Использование нескольких процессов, для реализации распределенного in memory cache, для случая x86 платформы

Часть 4:
Производительность

Профилирование производительности

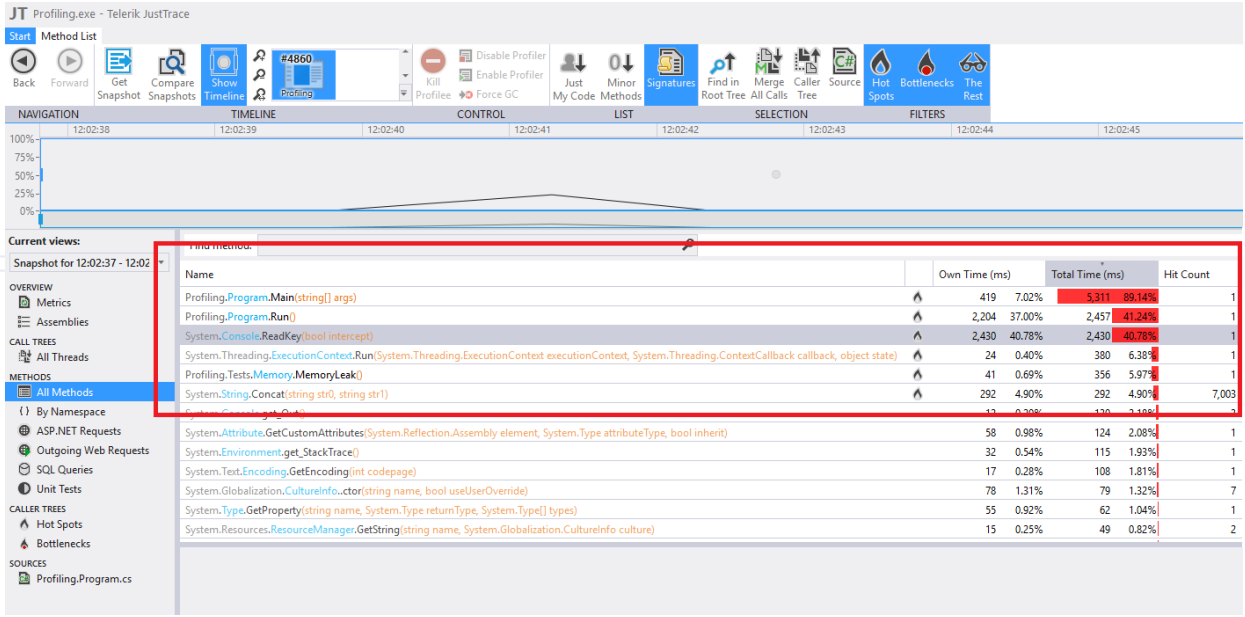
Ключевые параметры:

- Общее время выполнения функции
- Количество вызовов функции
- Потоки в которых вызываются функции

```
public static List<object> _objects = new List<object>();

1 reference
public static void MemoryLeak()
{
    string s = String.Format("test string");
    for (int i = 0; i < 7000; i++)
    {
        s = s +String.Format("test string {0}", i);
        _objects.Add(s);
    }
}
```

Направление: Оптимизируйте функции, которые вызываются наибольшее количество раз.

```
static void Run()
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    int n = 10000;
    int j;
    int[] result = new int[n * n];
    for (int i = 0; i < n; i++)
    {
        j = 0;
        int pos = i * n;
        for (j = 0; j < n; j++)
        {
            lock (_sync3)
            {
                result[pos + j] = pos + j;
            }
        }
    }
    sw.Stop();

    Console.WriteLine("Time:{0}ms", sw.ElapsedMilliseconds);
    Console.WriteLine("StackTrace:{0}", Environment.StackTrace);

}

0 references
static void Main(string[] args)
{
    Run();

    DeadLock.MakeDeadlock();

    System.Threading.Thread t = new System.Threading.Thread(() => { Memory.MemoryLeak(); });
    t.Start();
    t.Join();

    Console.WriteLine("Please press any key...");
    Console.ReadKey();
}
```

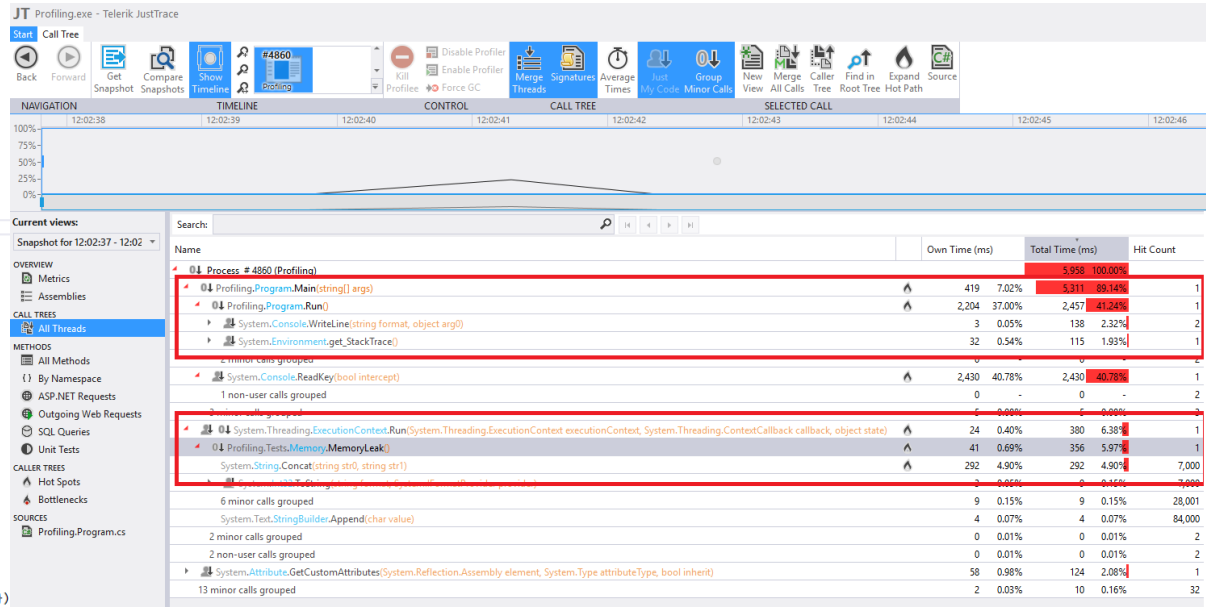


Профилирование производительности: Потоки

Потоки в которых вызываются функции

```
public static List<object> _objects = new List<object>();

1 reference
public static void MemoryLeak()
{
    string s = String.Format("test string");
    for (int i = 0; i < 7000; i++)
    {
        s = s +String.Format("test string {0}", i);
        _objects.Add(s);
    }
}
```

```
static void Run()
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    int n = 10000;
    int j;
    int[] result = new int[n * n];
    for (int i = 0; i < n; i++)
    {
        j = 0;
        int pos = i * n;
        for (j = 0; j < n; j++)
        {
            lock (_sync3)
            {
                result[pos + j] = pos + j;
            }
        }
    }
    sw.Stop();

    Console.WriteLine("Time:{0}ms", sw.ElapsedMilliseconds);
    Console.WriteLine("StackTrace:{0}", Environment.StackTrace);

}

0 references
static void Main(string[] args)
{
    Run();

    DeadLock.MakeDeadlock();

    System.Threading.Thread t = new System.Threading.Thread(() => { Memory.MemoryLeak(); })
    t.Start();
    t.Join();

    Console.WriteLine("Please press any key...");
    Console.ReadKey();
}
```



Оптимизации производительности

Легкий уровень сложности:

- Оптимизация ввода/вывода
- Выбор оптимального алгоритма
- Пакетная обработка данных
- Параллельная обработка

Средний уровень сложности:

- Оптимизация блокировок, за счет более точного разделения множеств данных
- Отложенное освобождение памяти, выделение памяти заранее
- Отказ от использования динамических конструкций
- Использование массивов
- Использование struct вместо class
- Использование цикла for
- Отказ от использования volatile
- Отказ от использования ThreadPool

Тяжелый уровень сложности:

- Векторизация кода (SSE, AVX)
- Unsafe код
- Inline методы
- Выравнивание данных
- Lock-free алгоритмы и структуры данных

Часть 5:
Взаимные блокировки

Поиск взаимных блокировок

Шаг	Процесс 1	Процесс 2
1	Хочет захватить A и B, начинает с A	Хочет захватить A и B, начинает с B
2	Захватывает ресурс A	Захватывает ресурс B
3	Ожидает освобождения ресурса B	Ожидает освобождения ресурса A
4	Взаимная блокировка	

Ключевые параметры:

- Общее время выполнения функции
- Количество вызовов функции

Направление: Поиск функций время выполнения которых, сильно изменяется после возникновения блокировки.

Действия:

- **Шаг 1:** Поиск функций, с наибольшим изменением времени.
- **Шаг 2:** Анализ времени жизни потоков
- **Шаг 3:** Внесение изменений в код
- **Шаг 4:** Повторный запуск теста
- **Шаг 5:** Сравнение результатов тестов.

Шаг №1: Методы

Без блокировки

```
private static readonly object _sync1 = new object();
private static readonly object _sync2 = new object();

1 reference
public static void MakeDeadlock()
{
    System.Threading.Thread t1 = new System.Threading.Thread(() => { Memory.MemoryLeak(); });
    t1.Start();

    System.Threading.Thread t2 = new System.Threading.Thread(() => { lock (_sync1)
        {
            System.Threading.Thread.Sleep(1000);
            Console.WriteLine("============Deadlock Enter: Finish");
            lock (_sync2)
            {
                System.Threading.Thread.Sleep(1000);
            }
            } });
    t2.Start();

    System.Threading.Thread t3 = new System.Threading.Thread(() => {
        lock (_sync2)
        {
            System.Threading.Thread.Sleep(500);
            Console.WriteLine("============Deadlock Enter: Start");
            lock (_sync1)
            {
                System.Threading.Thread.Sleep(500);
            }
        }
    });
    t3.Start();


    t1.Join();
}
```

JT Profiling.exe - Telerik JustTrace

Start | Method List

Back | Forward | Get Snapshot | Compare Snapshots | Show Timeline | #21420 Profiling | Kill Profilee | Disable Profiler | Enable Profiler | Force GC | Just My Code | Minor Methods | Signatures

NAVIGATION | TIMELINE | CONTROL | LIST

2:46:16 | 12:46:17 | 12:46:18 | 12:46:19 | 12:46:20 | 12:46:21 | 12:46:22 | 12:46:23 | 12:46:24 | 12:46:25

100% 75% 50% 25% 0%

Current views:

Snapshot for 12:46:16 - 12:46

OVERVIEW
- Metrics
- Assemblies

CALL TREES
- All Threads

METHODS
- All Methods
- By Namespace
- ASP.NET Requests
- Outgoing Web Requests
- SQL Queries
- Unit Tests

CALLER TREES
- Hot Spots
- Bottlenecks

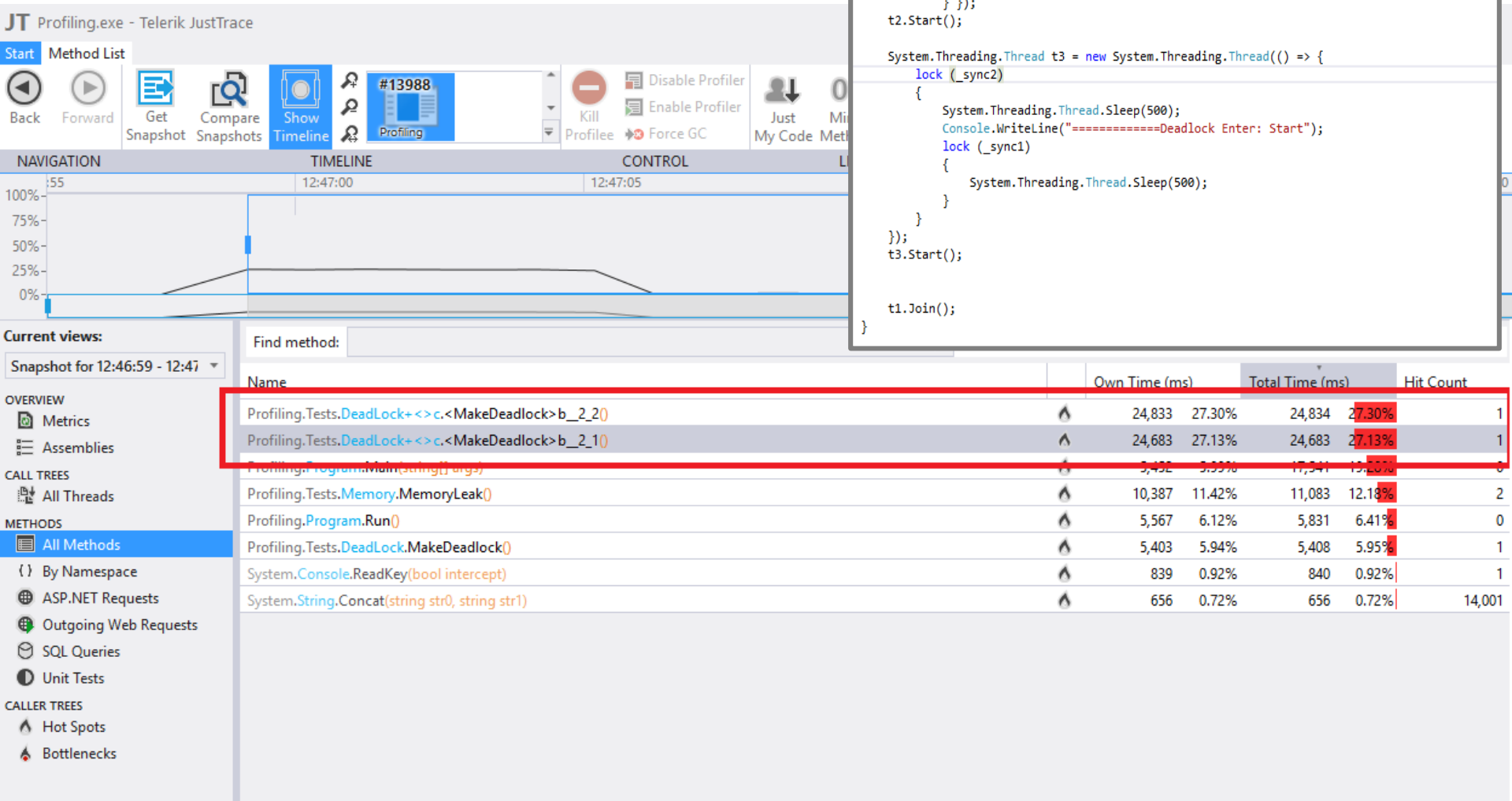
Find method:

Name	Own Time (ms)		Total Time (ms)		Hit Count
Profiling.Program.Main(string[] args)	5,397	14.78%	20,596	56.42%	1
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext executionContext, System.Threading.ContextCallback callback, object state)	64	0.18%	15,533	42.55%	4
Profiling.Tests.Memory.MemoryLeak()	10,259	28.10%	10,898	29.86%	2
Profiling.Program.Run()	8,263	22.64%	8,515	23.33%	1
Profiling.Tests.DeadLock+<>c.<MakeDeadlock>b__2_2()	2,548	6.98%	2,548	6.98%	1
Profiling.Tests.DeadLock+<>c.<MakeDeadlock>b__2_1()	2,022	5.54%	2,022	5.54%	1
System.String.Concat(string str0, string str1)	598	1.64%	598	1.64%	14,003
System.Attribute.GetCustomAttributes(System.Reflection.Assembly element, System.Type attributeType, bool inherit)	122	0.34%	191	0.52%	1
System.Globalization.CultureInfo..ctor(string name, bool useUserOverride)	75	0.20%	75	0.21%	7
System.Type.GetProperty(string name, System.Type returnType, System.Type[] types)	55	0.15%	62	0.17%	1

Шаг №1: Методы

С блокировкой

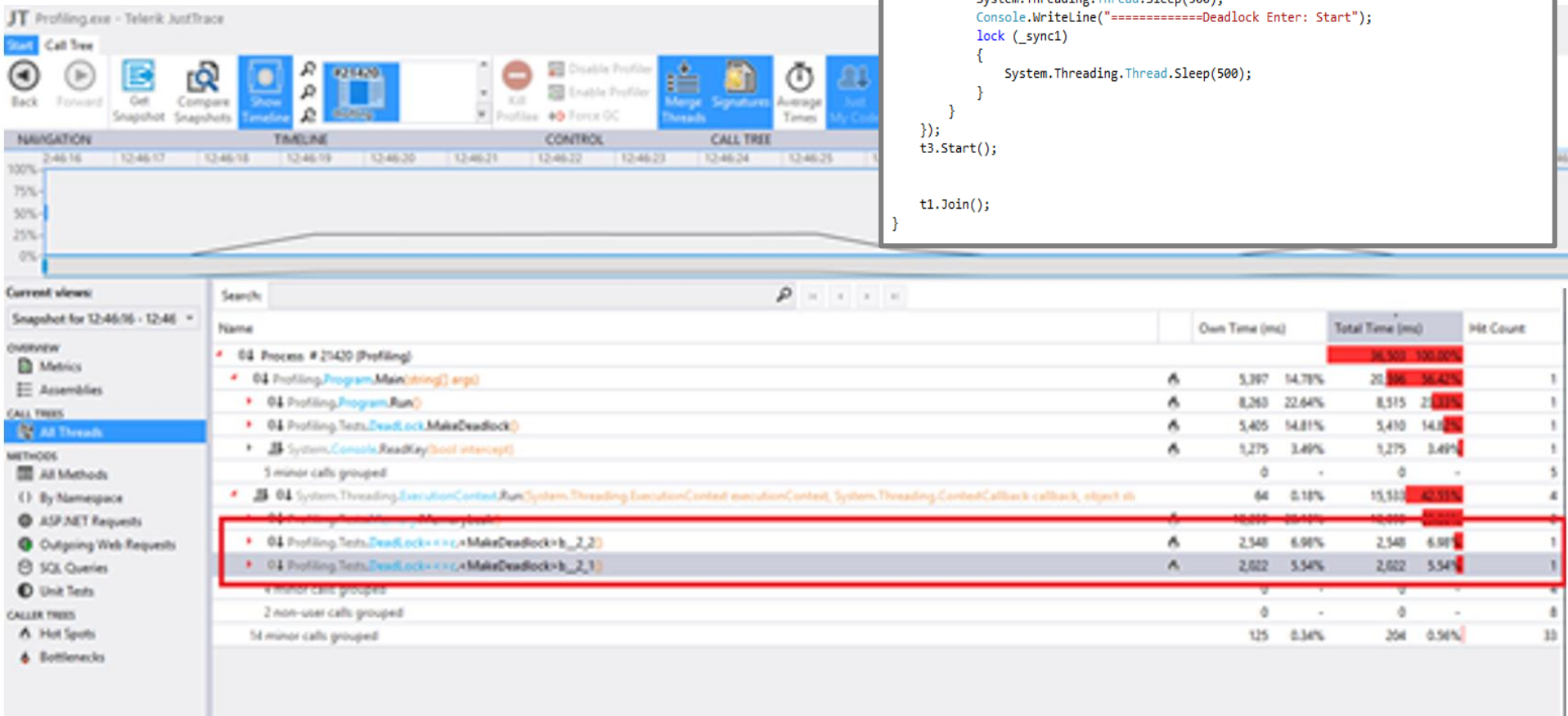
```
private static readonly object _sync1 = new object();
private static readonly object _sync2 = new object();

1 reference
public static void MakeDeadlock()
{
    System.Threading.Thread t1 = new System.Threading.Thread(() => { Memory.MemoryLeak(); });
    t1.Start();

    System.Threading.Thread t2 = new System.Threading.Thread(() => { lock (_sync1)
        {
            System.Threading.Thread.Sleep(1000);
            Console.WriteLine("=============Deadlock Enter: Finish");
            lock (_sync2)
            {
                System.Threading.Thread.Sleep(1000);
            }
            } });
    t2.Start();

    System.Threading.Thread t3 = new System.Threading.Thread(() => {
        lock (_sync2)
        {
            System.Threading.Thread.Sleep(500);
            Console.WriteLine("=============Deadlock Enter: Start");
            lock (_sync1)
            {
                System.Threading.Thread.Sleep(500);
            }
        }
    });
    t3.Start();


    t1.Join();
}
```



Шаг №2: Потоки

Без блокировки

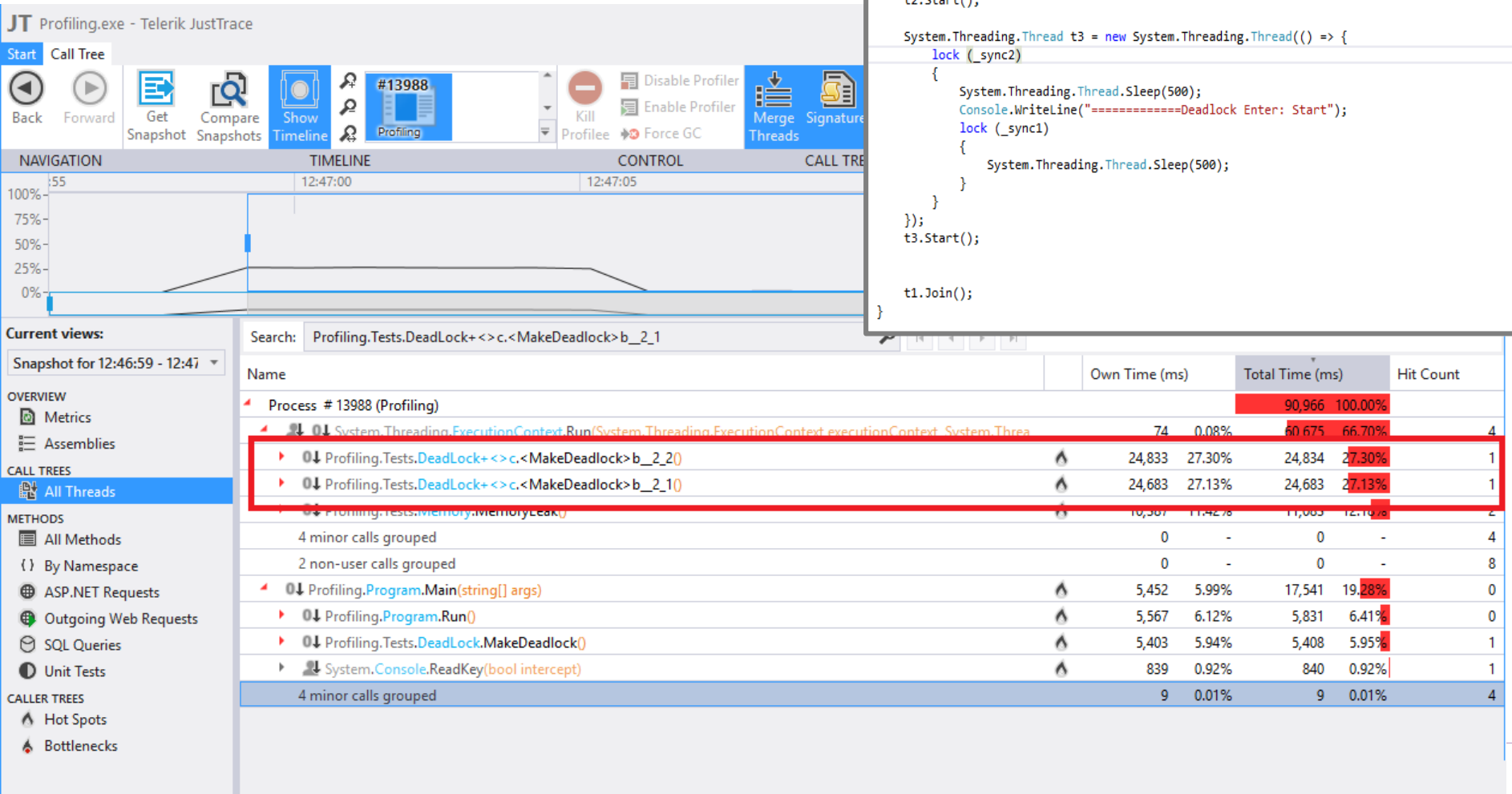
```
private static readonly object _sync1 = new object();
private static readonly object _sync2 = new object();

1 reference
public static void MakeDeadlock()
{
    System.Threading.Thread t1 = new System.Threading.Thread(() => { Memory.MemoryLeak(); });
    t1.Start();

    System.Threading.Thread t2 = new System.Threading.Thread(() => { lock (_sync1)
        {
            System.Threading.Thread.Sleep(1000);
            Console.WriteLine("=============Deadlock Enter: Finish");
            lock (_sync2)
            {
                System.Threading.Thread.Sleep(1000);
            }
            } });
    t2.Start();

    System.Threading.Thread t3 = new System.Threading.Thread(() => {
        lock (_sync2)
        {
            System.Threading.Thread.Sleep(500);
            Console.WriteLine("=============Deadlock Enter: Start");
            lock (_sync1)
            {
                System.Threading.Thread.Sleep(500);
            }
        }
    });
    t3.Start();


    t1.Join();
}
```



Шаг №2: Потоки

С блокировкой

```
private static readonly object _sync1 = new object();
private static readonly object _sync2 = new object();

1 reference
public static void MakeDeadlock()
{
    System.Threading.Thread t1 = new System.Threading.Thread(() => { Memory.MemoryLeak(); });
    t1.Start();

    System.Threading.Thread t2 = new System.Threading.Thread(() => { lock (_sync1)
        {
            System.Threading.Thread.Sleep(1000);
            Console.WriteLine("============Deadlock Enter: Finish");
            lock (_sync2)
            {
                System.Threading.Thread.Sleep(1000);
            }
            } });
    t2.Start();

    System.Threading.Thread t3 = new System.Threading.Thread(() => {
        lock (_sync2)
        {
            System.Threading.Thread.Sleep(500);
            Console.WriteLine("============Deadlock Enter: Start");
            lock (_sync1)
            {
                System.Threading.Thread.Sleep(500);
            }
        }
    });
    t3.Start();


    t1.Join();
}
```



Оптимизация блокировок

Ремарка: Хорошим тоном является, не вызывать блокировки
из кода который стоит под блокировкой.

Легкий уровень сложности:

- Область применения блокировки, должна быть минимальной.

Средний уровень сложности:

- Разделение блокировок за счет разбиения кода на две функции до блокировки и после блокировки. **Есть вероятность создания Livelock.**

Тяжелый уровень сложности:

- Применение изменений к объектам через копии.

Оптимизация блокировок

Livelock – зацикливание ожидания блокировок.

Система, продолжает работу, ее состояние постоянно меняется
за счет ожидания цикличного ожидания освобождения блокировки.

Направление: Построение графа блокировок.

В завершении

Решения из которых складывается производительность:

40%	**Архитектура**	• Определение целей • Выбор архитектуры которая наилучшим образом позволяет достигать целей
30%	**Данные**	• Хранение данных • Доступ к данным • Представление данных
20%	**Алгоритмы**	• Алгоритм, который позволяет достигать целей • Подготовка данных • Работа с результатом
10%	**Оптимизация кода**	• Оптимизация производительности • Оптимизация памяти • Накладные расходы, при параллельной обработке

Спасибо!

Вопросы?

Контакты:

mail: AlexandrSaitov@gmail.com
skype: LeonInc

Всем спасибо за участие и до новых встреч!