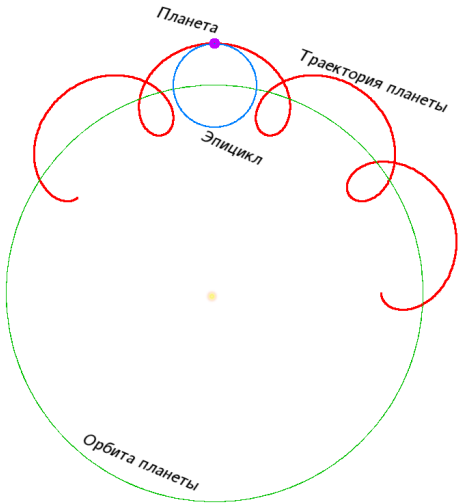
Прагматичное управление памятью

Анатолий Жмур
Senior Technical Architect
Broadridge Financial Solutions

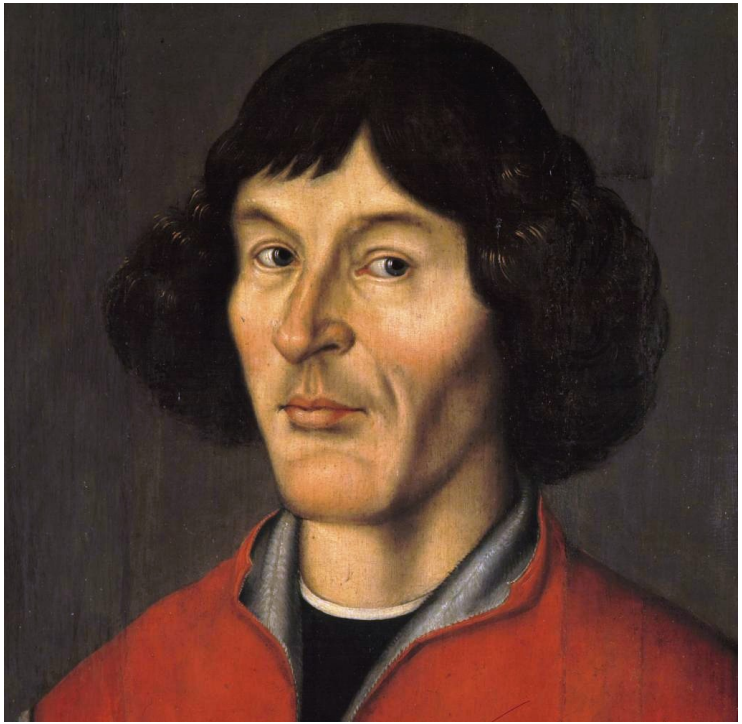
Сначала был Он

Альфонсинские таблицы (эфимерид)

- 1252-1270 создание
- 1485 впервые напечатаны
- Построены на птолемеевой космологии

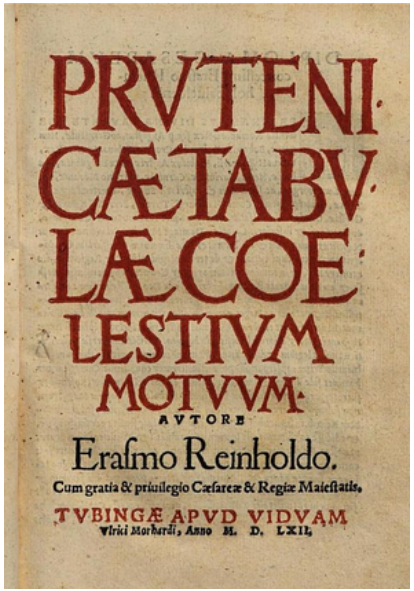


И тут пришел Он



Пру́сские табли́цы

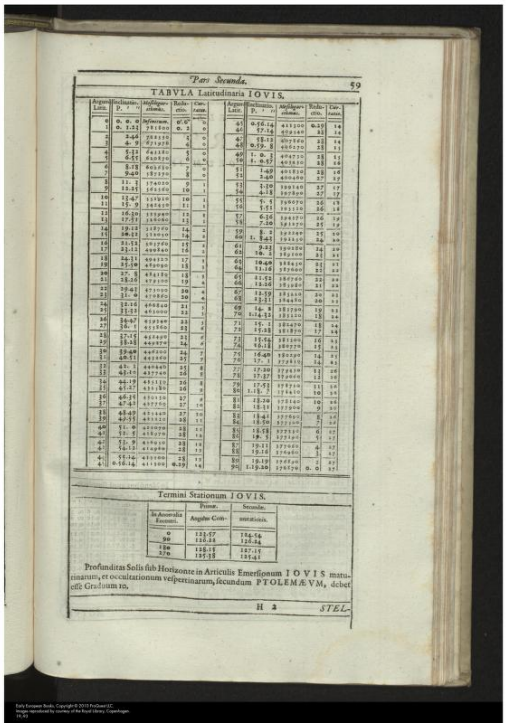
- Изданы в 1553
- Построены на космологии Коперника



И тут пришел Он

Рудольфинские таблицы

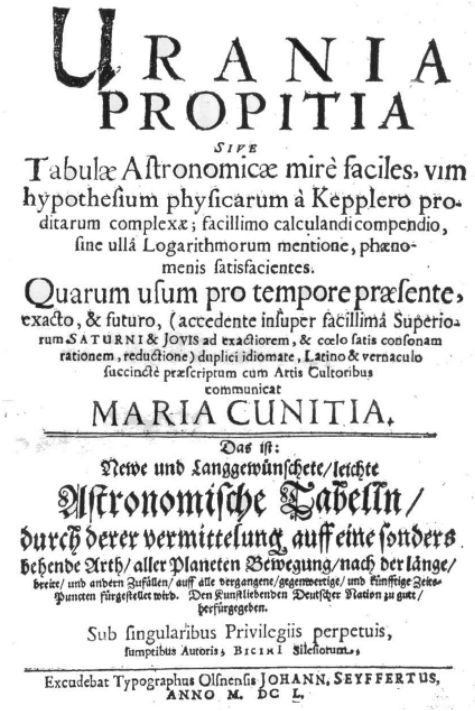
- Напечатаны в 1627
- Включали таблицы логарифмов
- Новые наблюдательные данные Тихо Браге



И тут пришла Она (embrace diversity!)

Urania Propitia

- Напечатаны в 1650
- Космология Тихо Браге: солнце и луна двигаются вокруг земли, остальные планеты вокруг солнца
- Упрощение вычислений ценой потери точности
- It was described by Noel Swerdlow as "the earliest surviving scientific work by a woman on the highest technical level of its age."



URANIA PROPITIA

SIVE

Tabulæ Aſtronomicæ mirè faciles, vim hypotheſium phyſicarum à Kepplero proditarum complexæ; facillimo calculandi compendio, ſine ullâ Logarithmorum mentione, phænomenis ſatisfacientes.

Quarum uſum pro tempore præſente, exacto, & futuro, (accedente inſuper facillimâ Superiorum SATURNI & JOVIS ad exactiorem, & cœlo ſatis conſonam rationem, reductione) duplici idiomate, Latino & vernaculo ſuccinctè præſcriptum cum Artis Cultoribus communicat

MARIA CUNITIA.

Das iſt:

Newe und Langgewünſchete/leichte

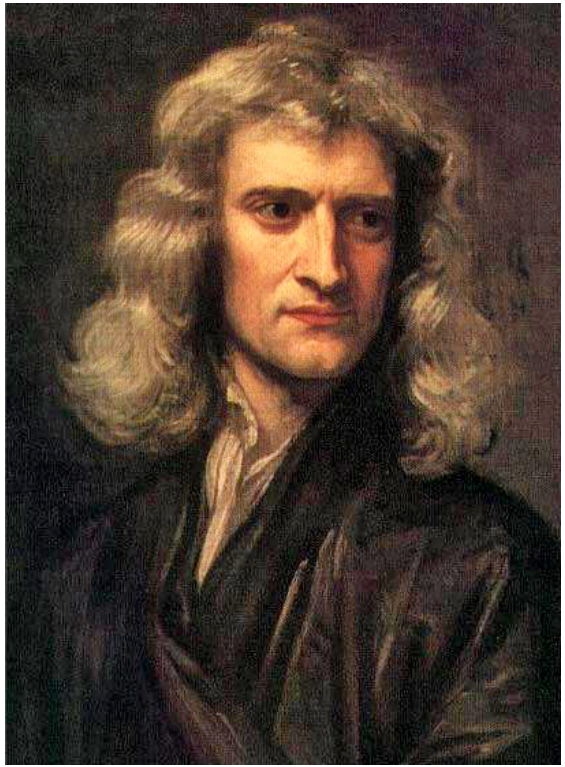
Aſtronomiſche Tabellen/

durch derer vermittelung auff eine ſonders behende Arth/ aller Planeten Bewegung/nach der Länge/ breite/ und andern Zufällen/ auff alle vergangene/gegenwertige/ und künfftige Zeits-Punncten fürgeſtellet wird. Den Kunſtliebenden Deutſcher Nation zu gute/ herfürgegeben.

Sub ſingularibus Privilegiis perpetuis, ſumptibus Autoris, BICINI Silefiorum.

Excudebat Typographus Olſnenſis JOHANN. SEYFFERTUS, ANNO M. DC L.

И тут пришел Он

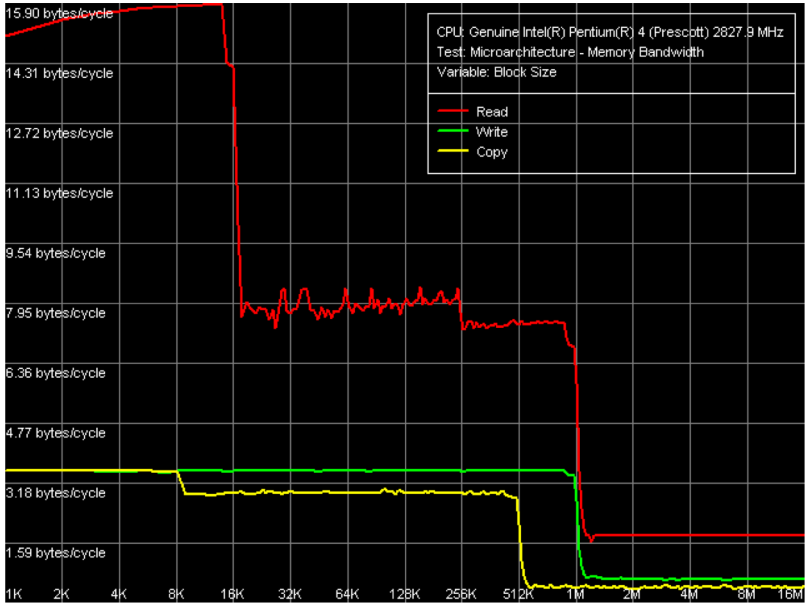


Philosophiæ Naturalis Principia Mathematica

- Издана на латыни в 1687
- Законы механики и закон всемирного тяготения
- Основы матанализа
- Доказательство того что законы Кеплера это хорошее приближение

Законы Ньютона для оперативной памяти

- Последовательный доступ быстрее произвольного, несмотря на название Random Access Memory
- Большие регистры быстрее маленьких регистров (SSE до 512 бит)
- Невыравненный доступ медленее (x86) или запрещен (ARM)
- Выход за пределы кэша меняет скорость в разы
- Линии на графике далеки от идеальных



Побайтовый и векторизированный цикл

```
[Benchmark()]
public void CopyByteLoop()
{
    for (int i = 0; i < this.Length; ++i)
    {
        this.data2[i] = this.data[i];
    }
}

[Benchmark(Baseline = true)]
public void Copy()
{
    Buffer.BlockCopy(this.data, 0, this.data2, 0, this.Length);
}
```

Побайтовый и векторизированный цикл

```
[Benchmark()]
public void CopyByteLoop()
{
    for (int i = 0; i < this.Length; ++i)
    {
        this.data2[i] = this.data[i];
    }
}
```

Method	Length	Mean	Error	StdDev
CopyByteLoop	20	13.90 ns	0.2654 ns	0.2353 ns
CopyByteLoop	10000	6,679.13 ns	391.6955 ns	366.3922 ns
CopyByteLoop	100000	65,049.09 ns	668.5089 ns	625.3237 ns

```
[Benchmark(Baseline = true)]
public void Copy()
{
    Buffer.BlockCopy(this.data, 0, this.data2, 0, this.Length);
}
```

Method	Length	Mean	Error	StdDev
Copy	20	6.920 ns	0.7605 ns	0.7114 ns
Copy	10000	133.854 ns	18.9302 ns	17.7073 ns
Copy	100000	2,354.645 ns	208.5673 ns	195.0940 ns

Управление памятью как компромис

- Выбираем между удобством написания/поддержки и memory traffic
- Сложно придумать пользовательский код, который использует память быстрее чем за O(N)

Конвертация между кодировками (наивная версия)

```
[Benchmark()]
public void Recode()
{
    var str = this.source.GetString(this.data);
    this.data2 = this.target.GetBytes(str);
}
```

Я сделяль!

```
[Benchmark()]
public void RecodeOptimized()
{
    var maxlength = this.source.GetMaxCharCount(this.data.Length);
    var charData = ArrayPool<char>.Shared.Rent(maxlength);
    var charLength = this.source.GetChars(this.data, charData);
    var charSlice = charData.AsSpan(0, charLength);
    var targetData = ArrayPool<byte>.Shared.Rent(this.target.GetMaxByteCount(charLength));
    var targetLength = this.target.GetBytes(charSlice, targetData);
    ArrayPool<char>.Shared.Return(charData);
    var result = targetData.AsMemory(0, targetLength);

    // have to be done at caller
    ArrayPool<byte>.Shared.Return(targetData);
}
```

Результаты ASCII -> UTF8

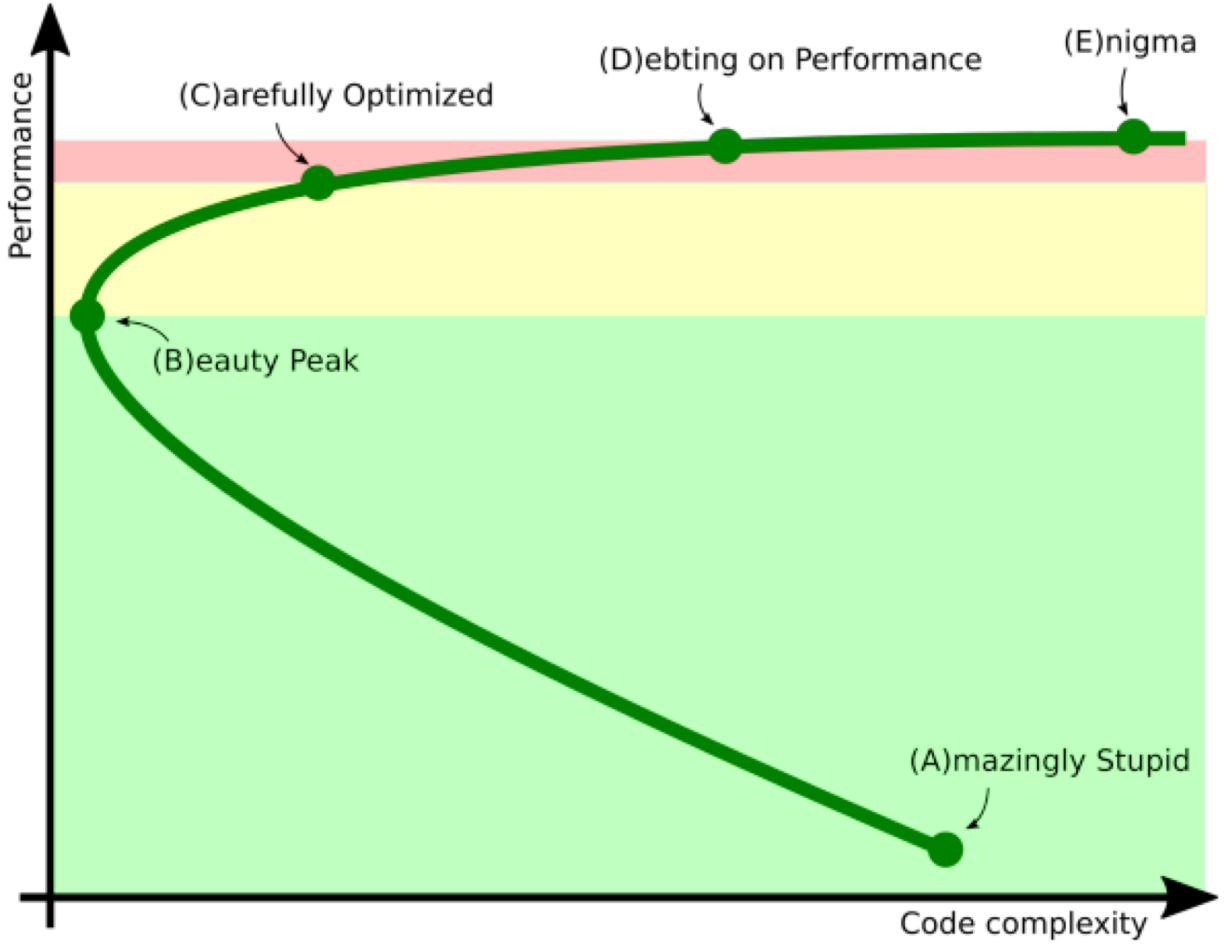
```
private readonly Encoding source = Encoding.ASCII;
private readonly Encoding target = Encoding.UTF8;
```

Method	Length	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Recode	24	91.41 ns	0.4887 ns	0.4572 ns	0.0035	-	-	120 B
RecodeOptimized	24	114.90 ns	1.5215 ns	1.4232 ns	-	-	-	-
RecodeOptimized	10000	41,845.34 ns	118.7460 ns	105.2652 ns	-	-	-	-
Recode	10000	44,302.43 ns	44.4862 ns	34.7319 ns	0.9155	-	-	30048 B
RecodeOptimized	100000	444,931.91 ns	7,371.7567 ns	6,895.5458 ns	-	-	-	-
Recode	100000	613,659.51 ns	7,377.0585 ns	6,900.5051 ns	23.4375	23.4375	23.4375	200024 B

Результаты UTF8 -> UTF7

```
private readonly Encoding source = Encoding.UTF8;
private readonly Encoding target = Encoding.UTF7;
```

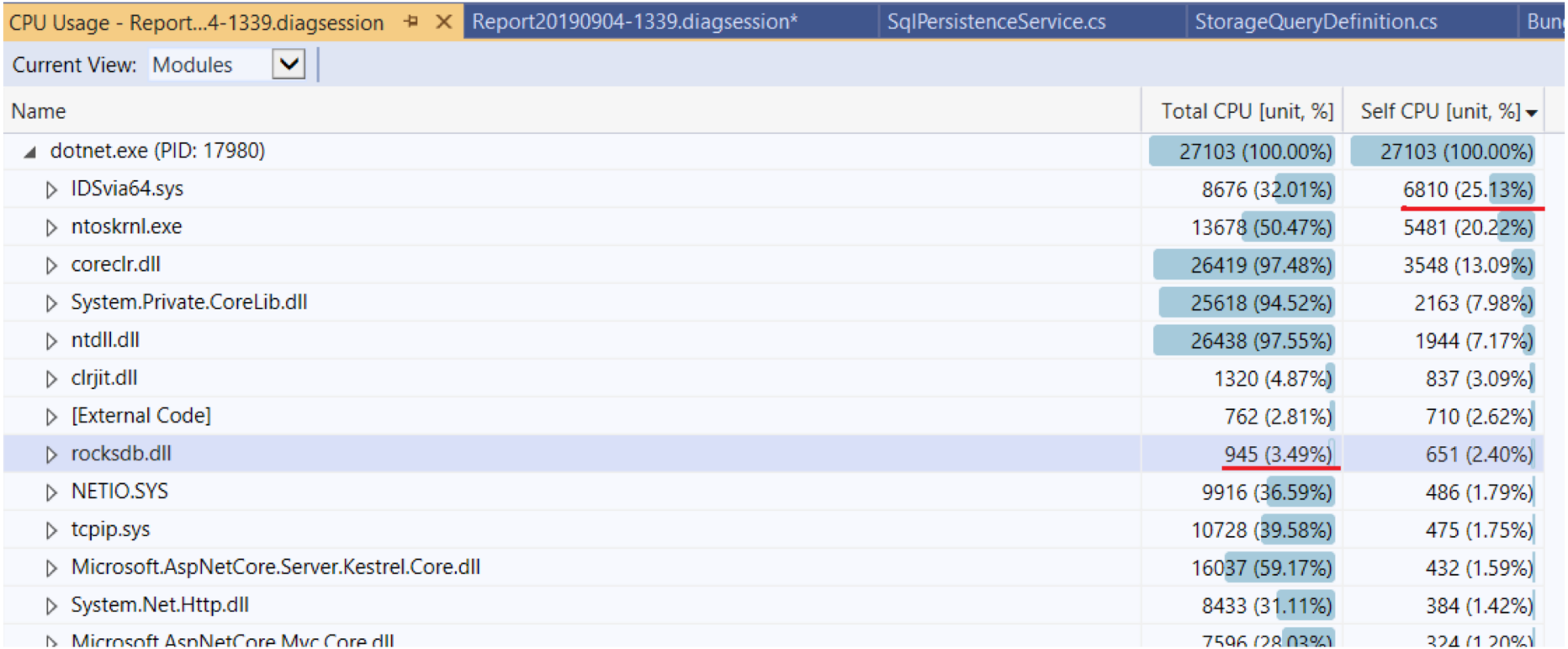
Method	Length	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
RecodeOptimized	24	515.9 ns	0.7914 ns	0.6608 ns	0.0048	-	-	176 B
Recode	24	1,034.1 ns	9.6923 ns	9.0661 ns	0.0267	-	-	928 B
RecodeOptimized	10000	225,737.3 ns	1,088.4524 ns	908.9070 ns	-	-	-	176 B
Recode	10000	523,466.0 ns	515.4715 ns	402.4462 ns	4.8828	-	-	177616 B
RecodeOptimized	100000	2,307,439.2 ns	18,017.4863 ns	16,853.5679 ns	-	-	-	176 B
Recode	100000	5,338,476.9 ns	52,392.7133 ns	49,008.1765 ns	70.3125	31.2500	31.2500	1524046 B


Performance
(C)arefully Optimized
(D)ebting on Performance
(E)nigma
(B)eauty Peak
(A)mazingly Stupid
Code complexity

История про оптимизацию ядра ОС

- Ранние 70ые
- IBM OS/360 первая ОС где ядро написано на языке высокого уровня (специльный тип PL/I)
- Человек написать лучшие код чем компилятор в те времена
- В результате профайлинга обнаружили место где операционная система проводит 60% времени
- Создали микробенчмарк и смогли оптимизировать его в 3 раза
- Обновили ядро и ожидали примерно двухкратного прироста производительности
- В линуксе подобная работа была сделана в 4.17
- Мораль: Всегда проверяйте полный тест, а не отчитывайтесь по результатам микробенчмарка

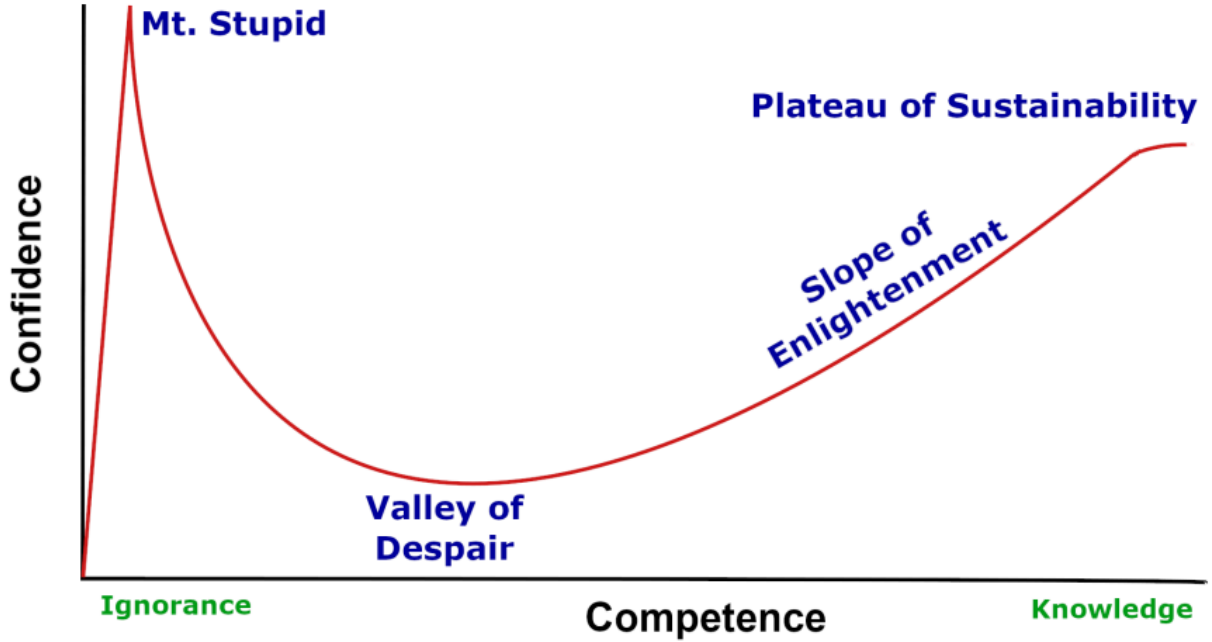
Производительность БД в реальности



GC по сравнению с современным Си-хипом

- GC сложнее
- GC требует пауз для перемещения объектов
- Jemalloc содежит большинство оптимизаций который обычно приписывают GC: регионы привязанные к потокам, фоновые потоки и т.п.
- Перемещение объектов очень дорогостоящий процесс, но он ускоряет последующие аллокации и повышает эффективность использования памяти
- Таким образом GC пауза является амортизированной стоимостью деаллокаций
- LOH ближе к Си-хипу

Эффект Даннинга-Крюгера



https://agile-mercurial.com/2019/07/12/the-dunning-kruger-effect/

Идеальная аллокация

- Без блокировок (в регионе привязанном к потоку)
- Увеличиваем указатель свободного места на длину объекта
- В идеале это O(1)
- В реальность из-за очистки памяти стоимость O(N)
- ОС чистит страницы перед тем как отдать процессу
- GC чистит блоки памяти перед тем как вернуть из аллокатора

Альтернативы New[]

Method	Length	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated
StackAlloc	24	1.046 ns	0.0037 ns	0.0034 ns	0.32	0.00	-	-	-	-
StackAllocWithoutLocalsInit	24	1.114 ns	0.0013 ns	0.0012 ns	0.34	0.00	-	-	-	-
Fill	24	3.302 ns	0.0117 ns	0.0098 ns	1.00	0.00	-	-	-	-
NewArray	24	4.257 ns	0.0680 ns	0.0636 ns	1.29	0.02	0.0014	-	-	48 B
ArrayPool	24	23.901 ns	0.7139 ns	0.5962 ns	7.24	0.18	-	-	-	-
Native	24	62.108 ns	0.7310 ns	0.6838 ns	18.83	0.24	-	-	-	-
StackAllocWithoutLocalsInit	10000	1.940 ns	0.0321 ns	0.0300 ns	0.02	0.00	-	-	-	-
ArrayPool	10000	22.243 ns	0.2489 ns	0.2329 ns	0.18	0.00	-	-	-	-
Native	10000	64.903 ns	0.6466 ns	0.6048 ns	0.52	0.01	-	-	-	-
Fill	10000	124.164 ns	0.0956 ns	0.0848 ns	1.00	0.00	-	-	-	-
StackAlloc	10000	280.988 ns	2.4779 ns	2.3179 ns	2.26	0.02	-	-	-	-
NewArray	10000	551.524 ns	10.2993 ns	9.6340 ns	4.43	0.07	0.2966	-	-	10024 B
StackAllocWithoutLocalsInit	100000	12.247 ns	0.0164 ns	0.0154 ns	0.008	0.00	-	-	-	-
ArrayPool	100000	21.531 ns	0.2242 ns	0.2098 ns	0.014	0.00	-	-	-	-
Native	100000	139.627 ns	30.6061 ns	28.6289 ns	0.088	0.02	-	-	-	-
Fill	100000	1,581.974 ns	25.1339 ns	23.5103 ns	1.000	0.00	-	-	-	-
StackAlloc	100000	3,144.020 ns	1.7905 ns	1.6749 ns	1.988	0.03	-	-	-	-
NewArray	100000	45,511.712 ns	1,177.6962 ns	1,101.6177 ns	28.770	0.62	8.3008	8.3008	8.3008	-

Disable localsInit: https://github.com/ltrzesniewski/LocalsInit.Fody

Большой блок за раз или много маленьких?

Method	Length	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
NewObject	24	3.987 ns	0.0416 ns	0.0348 ns	0.0007	-	-	24 B
NewArray	24	4.140 ns	0.0760 ns	0.0635 ns	0.0014	-	-	48 B
NewComplexObject	24	38.693 ns	1.0454 ns	0.9779 ns	0.0007	-	-	24 B
NewArray	10000	567.238 ns	11.7187 ns	10.9616 ns	0.2966	-	-	10024 B
NewObject	10000	1,616.874 ns	15.5231 ns	13.7608 ns	0.2995	-	-	9984 B
NewComplexObject	10000	15,081.706 ns	72.9726 ns	68.2587 ns	0.2899	-	-	9984 B
NewObject	100000	16,407.386 ns	301.3395 ns	281.8732 ns	2.9907	-	-	99984 B
NewArray	100000	40,168.345 ns	6,421.2814 ns	6,006.4706 ns	8.3618	8.3618	8.3618	-
NewComplexObject	100000	148,111.563 ns	560.5506 ns	468.0852 ns	2.9297	-	-	99984 B

LOH был преждевременной оптимизацией

- Объекты больше чем 85k попадают в gen3/LOH
- 8k если double и x86
- В LOH объекты не перемещаются, он работает как обычный Heap с free list
- За годы было очень много проблем с неэффективностью использования памяти в LOH
- Вы скорее всего никогда не аллоцируете в LOH напрямую
- Это делают List, Dictionary and MemoryStream, и Kestrel.
- В Java Hotspot тоже было особое поведение по отношению к большим объектам, но все же они компактились

Method	Length	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
NewArray	84900	4.193 us	0.0070 us	0.0066 us	2.5558	-	-	84928 B
NewArray	85000	40.847 us	1.7640 us	1.6501 us	7.3242	7.3242	7.3242	-

Секретные настройки GC

- Изначальная позиция была в том что ничего настраивать не надо (в отличие от джавы)
- Сейчас в .net core более 50 ключей для управления GC
- GCLOHThreshold – недокументированный ключ для увеличения порога больших объектов сверх 85k. Так же доступен в .Net framework 4.8
- Соответственно переменная окружения COMPlus_GCLOHThreshold
- Помогает, когда вы аллоцируете короткоживущие большие буферы, которые иначе бы попали в LOH
- Вы можете практический отключить LOH, но это не заменит то “мягкое” отношение к большим объектам в остальных GC
- https://devblogs.microsoft.com/dotnet/the-history-of-the-gc-configs/
- https://github.com/dotnet/coreclr/blob/ed5dc831b09a0bfed76ddad684008bebc86ab2f0/src/gc/gcconfig.h

Что еще может быть полезно из настроек

- GCHeapCount – когда у вас много ядер, что бы уменьшить неиспользуемую память
- GCLargePages – хороший способ улучшить перформанс нагруженного приложения ценой использования большего объема невыгружаемой памяти. Требует установленного GCHardLimit или контейнерного окружения!
- GCHardLimit – проверить как работает ваше приложения при недостатке памяти. Или уменьшить футпринт процесса, который не делает так уж много работы.
- Если вам очень важно максимальная продолжительность GC паузы: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/latency

Когда нужно начинать оптимизировать аллокации

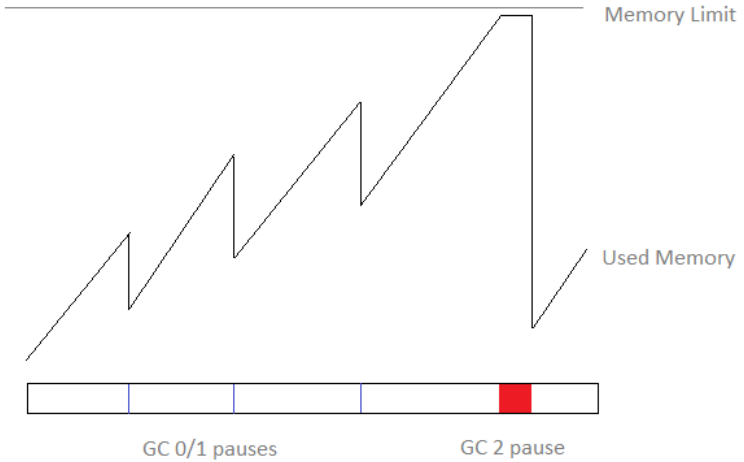
- Мало доступной памяти + много LOH аллокаций
- Критичность к паузам и много выживших во втором поколении
- Счетчик "Time Spent in GC%” больше 10%
- Page Faults

Эмпирические законы .Net GC

- Продолжительность GC Pause зависит линейно от объема выживших, память занимаемая процессом не имеет значения
- Concurrent GC не так уж и помогает. Не рассчитывайте на него.
- Максимальный траффик около 2 GiB/s
- GC Pause занимает 0.2-0.5 на 1 GiB выживших
- Если у вас много LOH аллокаций возможно вам придется об этом задуматься
- Накладные расходы на аллокацию в куче 16 байт. Минимальный размер объекта 24 байта (x64).

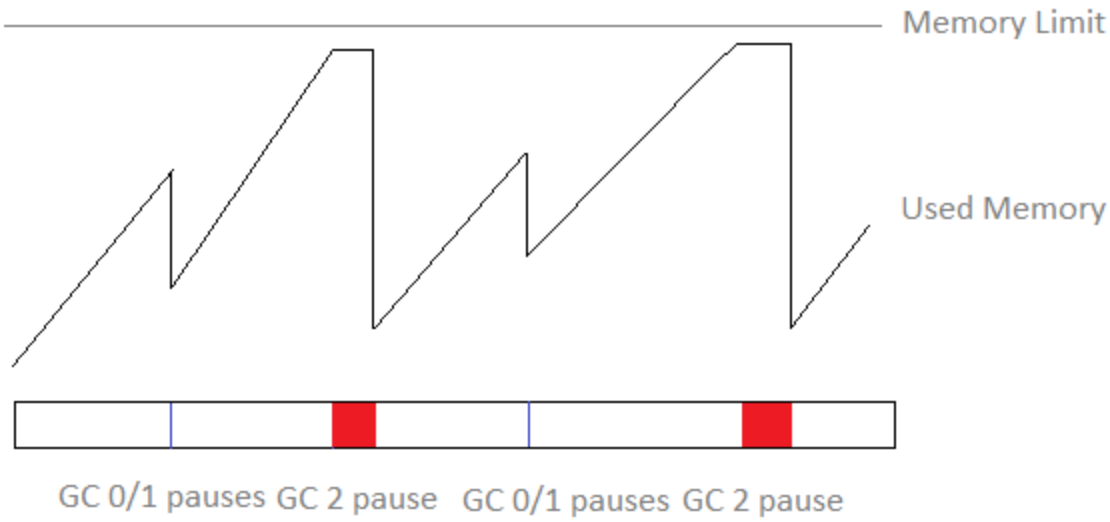
Экспоненциальное замедление от недостатка памяти 1

- Пусть у нас есть 0.5 GiB/s мемори трафика
- Выживших на 1 GiB
- Ограничение на память процесса 10 GiB
- В худшем случае Gen2 сборка каждые 18 секунд, которая займет примерно 0.5 секунды
- Время в GC = 3%

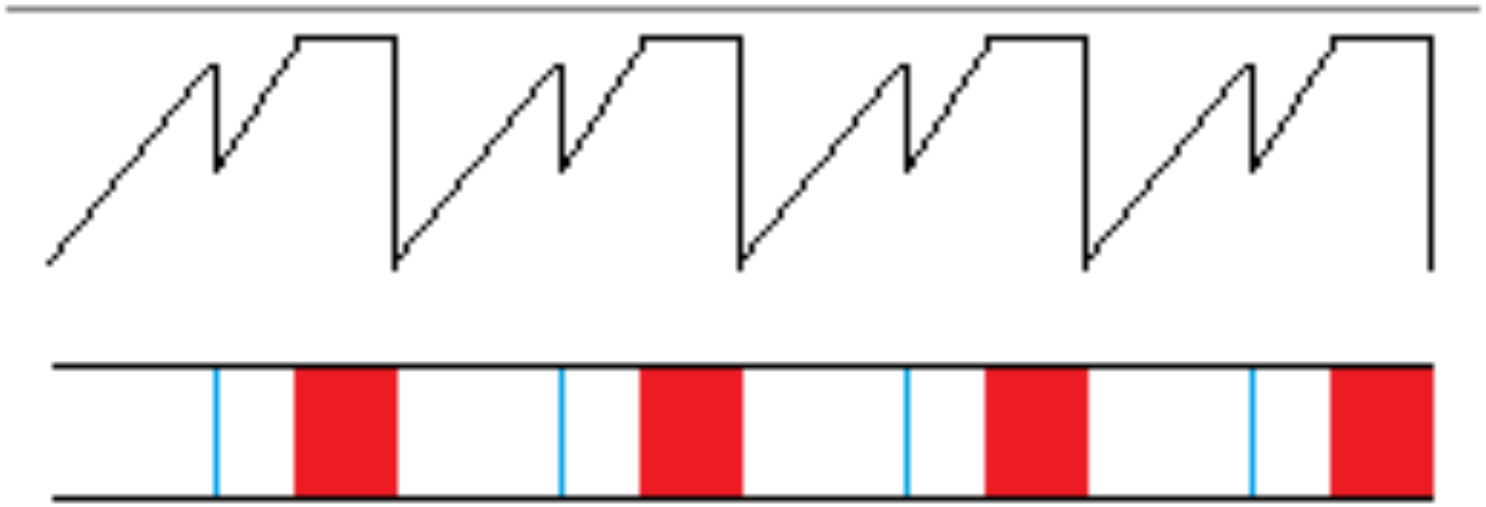


Экспоненциальное замедление от недостатка памяти 2

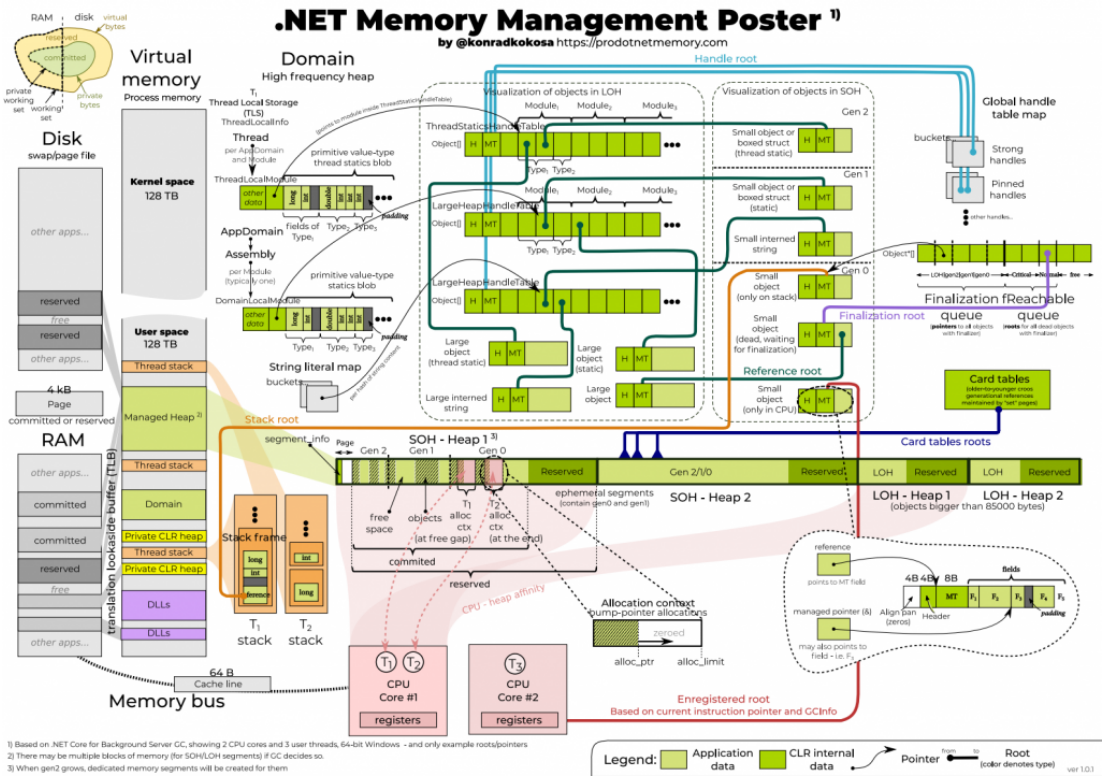
- Пусть у нас есть лимит на память процесса в 3 GiB
- Gen2 сборка каждые 6 секунды так же занимает полсекунды
- Время в GC = 8%



Экспоненциальное замедление от недостатка памяти 3



Постеры от Конрада Кокосы



https://prodotnetmemory.com/

Streams

Method	Length	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated
Fill	24	3.361 ns	0.0474 ns	0.0443 ns	1.00	0.00	-	-	-	-
NewArray	24	4.495 ns	0.0123 ns	0.0115 ns	1.34	0.02	0.0014	-	-	48 B
PoolStream	24	46.841 ns	0.8957 ns	0.7940 ns	13.94	0.28	0.0021	-	-	72 B
PooledMemoryStream	24	47.839 ns	0.4394 ns	0.4110 ns	14.24	0.25	0.0018	-	-	64 B
MemoryStream	24	49.079 ns	2.5654 ns	2.3997 ns	14.60	0.72	0.0104	-	-	352 B
SmallBlockMemoryStream	24	58.385 ns	0.4220 ns	0.3948 ns	17.37	0.22	0.0139	-	-	472 B
RecyclableMemoryStream	24	1,245.351 ns	8.1506 ns	7.6241 ns	370.60	6.10	0.0114	-	-	440 B
Fill	10000	126.599 ns	0.3522 ns	0.2941 ns	1.00	0.00	-	-	-	-
PoolStream	10000	289.478 ns	1.3739 ns	1.2852 ns	2.29	0.01	0.0019	-	-	72 B
PooledMemoryStream	10000	379.710 ns	6.0470 ns	5.6564 ns	3.00	0.04	0.0019	-	-	64 B
NewArray	10000	651.348 ns	20.7916 ns	19.4485 ns	5.13	0.16	0.2966	-	-	10024 B
SmallBlockMemoryStream	10000	1,418.169 ns	28.7968 ns	26.9366 ns	11.18	0.18	0.4959	0.0057	-	16648 B
RecyclableMemoryStream	10000	1,503.184 ns	6.2382 ns	5.5300 ns	11.87	0.06	0.0114	-	-	440 B
MemoryStream	10000	2,431.794 ns	8.1038 ns	6.3269 ns	19.21	0.07	0.8926	0.0267	-	28816 B
Fill	100000	1,648.183 ns	29.4716 ns	24.6101 ns	1.00	0.00	-	-	-	-
PooledMemoryStream	100000	3,290.826 ns	25.9700 ns	20.2757 ns	1.99	0.03	-	-	-	64 B
RecyclableMemoryStream	100000	4,523.710 ns	14.1285 ns	11.7980 ns	2.75	0.05	0.0076	-	-	440 B
PoolStream	100000	4,671.126 ns	108.0121 ns	101.0346 ns	2.84	0.05	-	-	-	72 B
SmallBlockMemoryStream	100000	11,008.327 ns	439.0806 ns	389.2335 ns	6.66	0.27	4.1504	0.8087	-	131408 B
NewArray	100000	43,382.037 ns	2,135.4853 ns	1,997.5343 ns	26.33	1.46	8.2397	8.2397	8.2397	-
MemoryStream	100000	93,852.651 ns	1,347.6407 ns	1,260.5840 ns	56.97	1.27	14.1602	10.7422	10.7422	127204 B

Stream links

- http://www.philosophicalgeek.com/2015/02/06/announcing-microsoft-io-recyclablememorystream/
- https://github.com/azhmur/PooledMemoryStream
- https://github.com/Aethon/SmallBlockMemoryStream
- https://github.com/microsoft/Microsoft.IO.RecyclableMemoryStream
- https://github.com/itn3000/PooledStream/

MemoryStream

- Не используйте ToArray (обычно хватает TryGetBuffer)
- Не используйте MemoryStream как промежуточный буфер, обычно можно писать в целевой поток напрямую.
- Вы можете попробовать использовать аналоги Memory Stream с пулингом буферов.
- Через пару лет должно появится в корке. https://github.com/dotnet/corefx/issues/21380

Links

- https://medium.com/servicetitan-engineering/go-vs-c-part-2-garbage-collection-9384677f86f1
- https://mattwarren.org/2017/01/13/Analysing-Pause-times-in-the-.NET-GC/