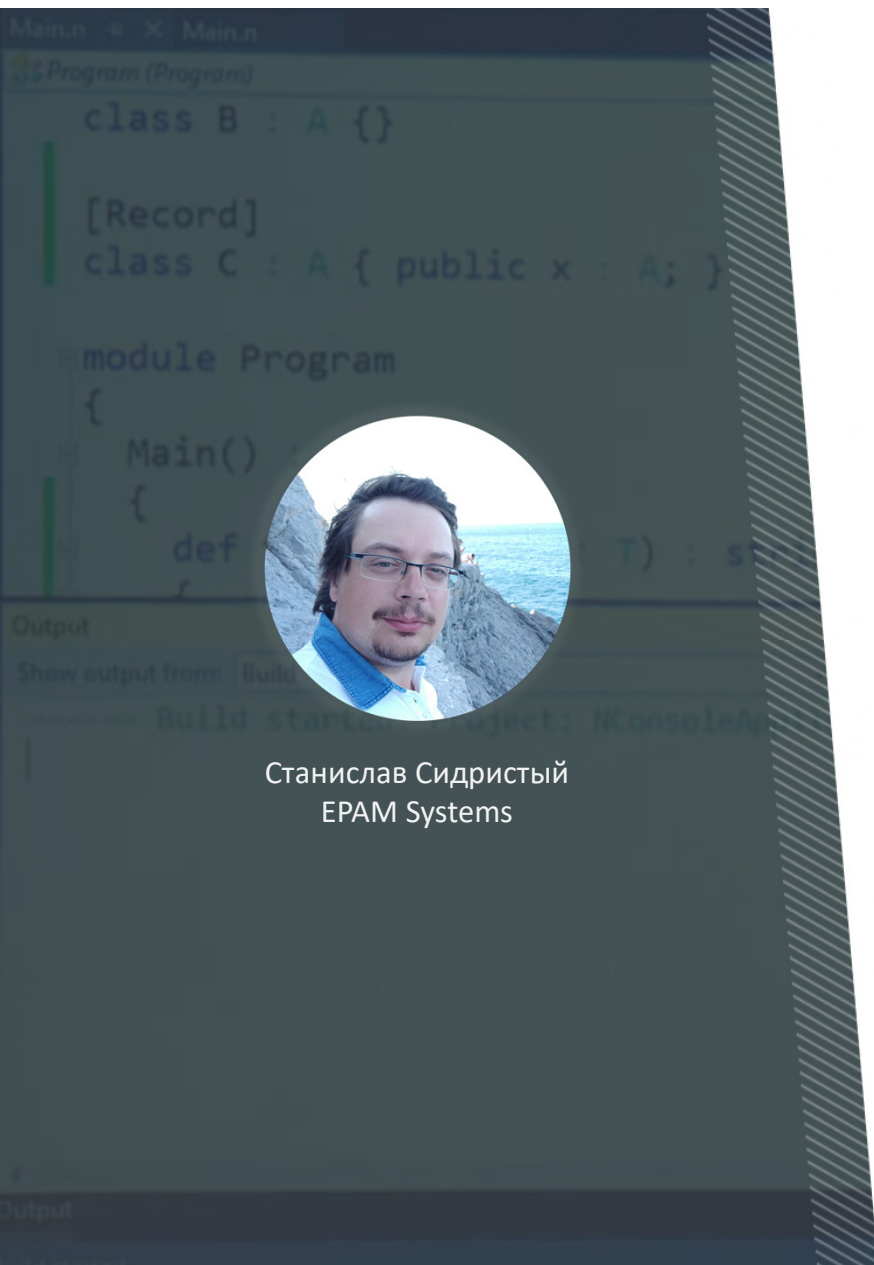


# Garbage Collector


Работа над производительностью,  
вооружившись знаниями о GC

Сидристый Станислав



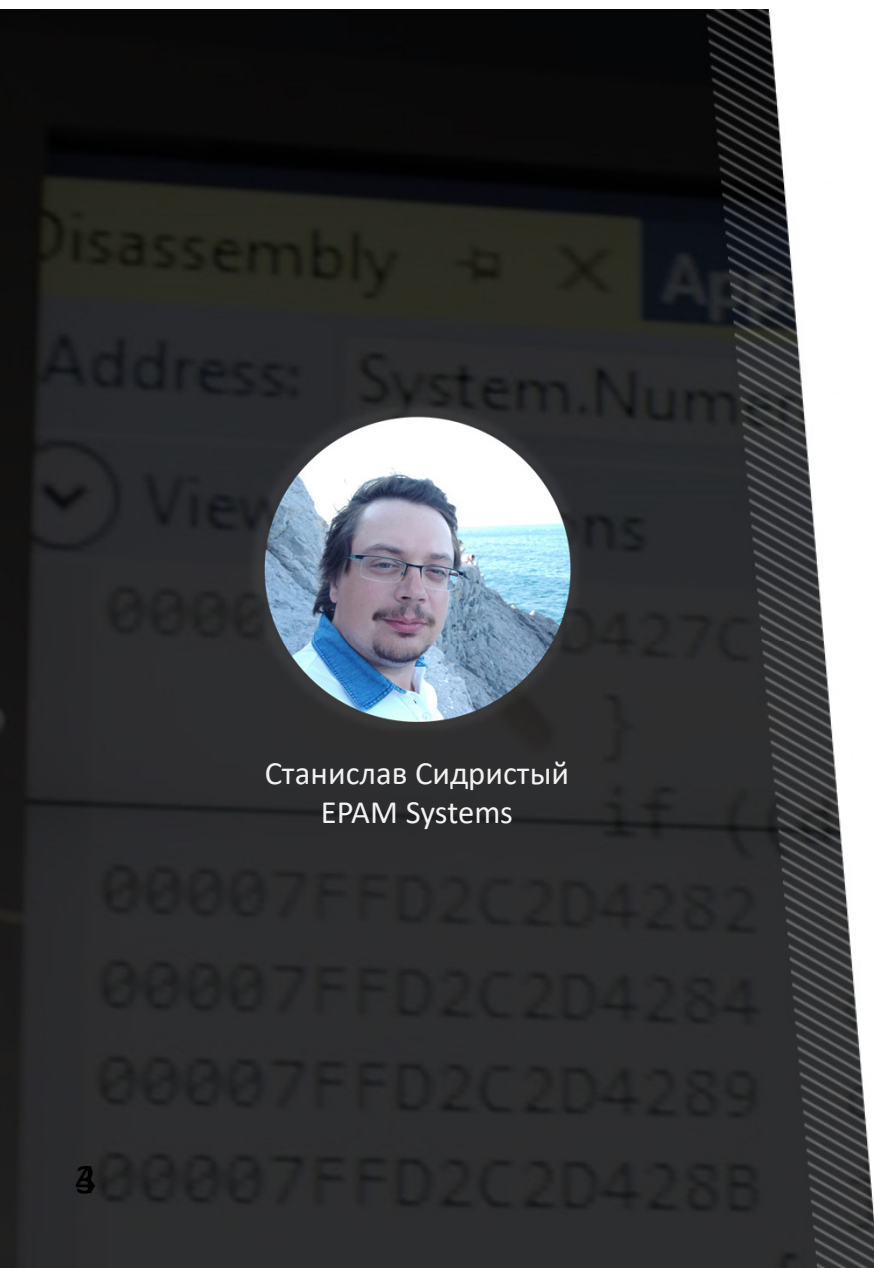


#### Стеки:

- WEB/WPF/WinForms/... стеки
- C/C++, C++/CLI когда необходимо
- EPAM Systems
- Книга:  <https://github.com/sidristij/dotnetbook>

#### Связь:

- telegram: @sidristij
- skype: stanislav.sidristy
- sunex.development@gmail.com



Watch 207 Star 830 Fork 71



 <https://github.com/sidristij/dotnetbook>



# Снижайте кросспоколенческую связность

## Проблема

Для оптимизации скорости сборки мусора GC собирает по-возможности младшее поколение. Но чтобы сделать это, ему необходима информация о ссылках со старших: **карточного стола**.

Одна ссылка заставляет покрывать область:

- 4 байта перекрывает 4 Кб или макс. 320 объектов – для x86 архитектуры
- 8 байт перекрывает 8 Кб или макс. 320 объектов – для x64 архитектуры

Разреженные ссылки в младшее поколение сделают GC более трудоёмким

## Решение

1. Располагать объекты со связями в младшее поколение – рядом
2. Аллоцировать их группами, выдавая пользовательскому коду по запросу
3. Избегать ссылок в младшее поколение



# Не допускайте сильной связности

## Проблема

Как следует из алгоритмов фазы сжатия объектов в SOH:

1. Для сжатия кучи необходимо обойти дерево и проверить все ссылки, исправляя их на новые значения
2. При этом ссылки с Карточного стола затрагивают целые группы объектов

Поэтому общая сильная связность объектов может привести к проседаниям при GC.

## Решение

1. Располагать сильно-связные объекты рядом, в одном поколении
2. Избегать лишних связей в целом
3. Избегайте кода со скрытой связностью. Например, замыканий





# Мониторьте использование сегментов

## Проблема

При интенсивной работе может возникнуть ситуация, когда выделение новых объектов приводит к задержкам: выделению новых сегментов под кучу и дальнейшему их декоммиту при очистке мусора

## Решение

1. При помощи PerfMon / Sysinternal Utilities проконтролировать точки выделения новых сегментов и их декоммитинг и освобождение
2. Если речь идет о LOH, в котором идёт плотный трафик буферов, воспользоваться ArrayPool
3. Если речь идет о SOH, убедиться что объекты одного времени жизни выделяются рядом, обеспечивая срабатывание Sweep вместо Collect
4. SOH: использовать пулы объектов



# Не выделяйте память в нагруженных участках кода

## Проблема

Нагруженный участок кода выделяет память:

- Как результат, GC выбирает окно аллокации не 1Кб, а 8Кб.
- Если окну не хватает места, это приводит к GC и расширению закоммиченной зоны
- Плотный поток новых объектов заставит короткоживущие объекты с других потоков быстро уйти в старшее поколение с худшими условиями сборки мусора
- Что приведет к расширению времени сборки мусора
- Что приведет к более длительным Stop the World даже в Concurrent режиме

## Решение

1. Полный запрет на использование замыканий в критичных участках кода
2. Полный запрет боксинга на критичных участках кода (можно использовать эмуляцию через пуллинг если необходимо)
3. Там где необходимо создать временный объект под хранение данных, использовать структуры. Лучше – ***ref struct***. При количестве полей более 2-х передавать по ***ref***



# Не выделяйте память в нагруженных участках кода

## Проблема

Нагруженный участок кода выделяет память:

- Как результат, GC выбирает окно аллокации не 1Кб, а 8Кб.
- Если окну не хватает места, это приводит к GC и расширению закоммиченной зоны
- Плотный поток новых объектов заставит короткоживущие объекты с других потоков быстро уйти в старшее поколение с худшими условиями сборки мусора
- Что приведет к расширению времени сборки мусора
- Что приведет к более длительным Stop the World даже в Concurrent режиме

## Решение

1. Полный запрет на использование замыканий в критичных участках кода
2. Полный запрет боксинга на критичных участках кода (можно использовать эмуляцию через пуллинг если необходимо)
3. Там где необходимо создать временный объект под хранение данных, использовать структуры. Лучше – ***ref struct***. При количестве полей более 2-х передавать по ***ref***





# Избегайте излишних выделений памяти в LON

## Проблема

Размещение массивов в LON приводит либо к его фрагментации либо к утяжелению процедуры GC

## Решение

1. Использовать разделение массивов на подмассивы и класса, инкапсулирующего логику работы с такими массивами.
  1. Массивы уйдут в SON
  2. После пары сборок мусора лягут рядом с вечноживущими объектами и перестанут влиять на сборку мусора
2. Контролировать использования `double[n < 1000]`

# Где оправдано и возможно, использовать thread stack



## Проблема

Есть ряд сверхкороткоживущих объектов либо объектов, живущих в рамках вызова метода (включая внутренние вызовы). Они создают трафик объектов

## Решение

1. Использование выделения памяти на стеке, где возможно:
  1. Оно не нагружает кучу
  2. Не нагружает GC
  3. Освобождение памяти - моментальное
2. Использовать `Span<T> x = stackalloc T[]`; вместо `new T[]` где возможно
3. Использовать `Span/Memory` где это возможно
4. Перевести алгоритмы на ref stack типы (`StackList : struct, ValueStringBuilder`)



# Освобождайте объекты как можно раньше

## Проблема

Задуманные как короткоживущие, объекты попадают в gen1, а иногда и в gen2. Это приводит к утяжеленному GC, который работает дольше.

## Решение

1. Необходимо освобождать ссылку на объект как можно раньше.
2. Если длительный алгоритм содержит код, который работает с какими-либо объектами, разнесенный по коду. Но который может быть сгруппирован в одном месте, необходимо его сгруппировать, разрешая тем самым собрать их раньше.
  - Например, на строке 10 достали коллекцию, а на строке 120 – отфильтровали.

# Вызывать GC.Collect() не нужно



## Проблема

Часто кажется что если вызвать GC.Collect(), то это исправит ситуацию

## Решение

1. Гораздо корректнее выучить алгоритмы работы GC, посмотреть на приложение под ETW и другими средствами диагностики (JetBrains dotMemory, ...)
2. Оптимизировать наиболее проблемные участки

# Избегайте Pinning



## Проблема

Pinning создает целый ряд проблем:

1. Усложняет сборку мусора
2. Создает пробелы свободной памяти (ноды free-list items, bricks table, buckets)
3. Может оставить некоторые объекты в более младшем поколении, образуя при этом ссылки с карточного стола

## Решение

1. Не надо так.





# Избегайте финализации

## Проблема

Финализация вызывается не детерминированно:

1. Невызванный `Dispose()` приводит к финализации со всеми исходящими ссылками из объекта
2. Зависимые объекты задерживаются дольше запланированного
3. Стареют, перемещаясь в более старые поколения
4. Если они при этом содержат ссылки на более младшие, порождают ссылки с карточного стола
5. Усложняя сборку старших поколений, фрагментируя их и приводя к Compacting вместо Sweep

## Решение

1. Аккуратно вызывать `Dispose()`



# Избегайте большого количества потоков

## Проблема

При большом количестве потоков растет количество allocation context, т.к. они выделяются каждому потоку:

1. Как следствие – быстрее наступает GC.Collect.
2. Вследствие нехватки места в эфимерном сегменте вслед за Sweep наступит Collect

## Решение

1. Контролировать количество потоков по количеству ядер



# Избегайте траффика объектов разного размера

## Проблема

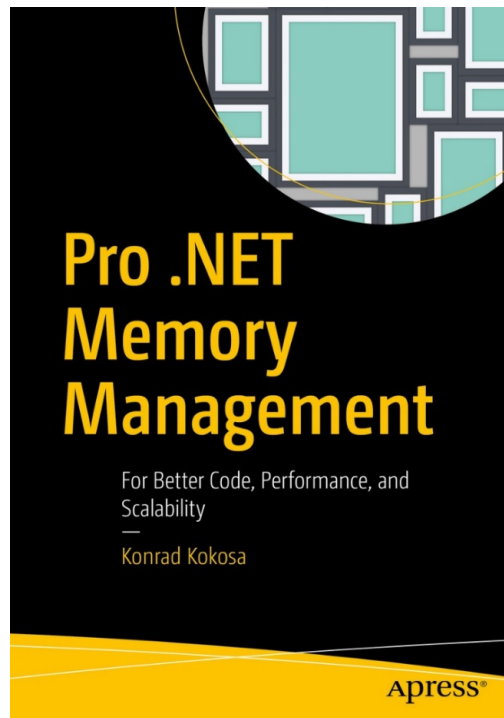
При траффике объектов разного размера и времени жизни возникает фрагментация:

1. Повышение Fragmentation ratio
2. Срабатывание Collection с фазой изменения адресов во всех ссылающихся объектах

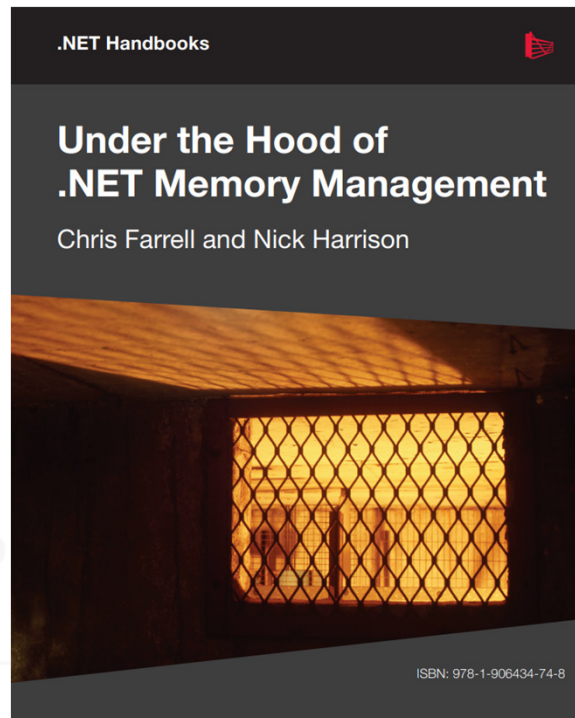
## Решение

1. Если предполагается траффик объектов:
  1. Проконтролировать наличие лишних полей, приблизив размеры
  2. Проконтролировать отсутствие манипуляций со строками: там, где возможно, заменить на ReadOnlySpan/ReadOnlyMemory
  3. Освобождать ссылку как можно раньше
    1. Не обязательно обнулять. В методах достаточно «поднять» использование как можно выше

<https://prodotnetmemory.com/>



<https://www.red-gate.com/hub/books/#dotnet>



<https://github.com/sidristij/dotnetbook>



QA