# Управление Памятью: День 4

## Оптимизация Large Object Heap и Паттерн Object Pooling

### 1. Введение: Почему Span<T> недостаточно?

В третий день мы научились использовать Span<T>, чтобы "нарезать" существующие массивы и строки без аллокаций. Это решает проблему *обработки* данных. Но откуда берутся сами данные?

В сетевом программировании (ASP.NET Core Kestrel, gRPC, сокеты) или при чтении файлов мы постоянно нуждаемся в буферах для приема данных.

* Если мы будем создавать new byte на каждый запрос, при нагрузке 10 000 RPS мы будем аллоцировать ~160 МБ в секунду.
* stackalloc не подходит для больших буферов (риск Stack Overflow) и для данных, которые должны жить дольше одного метода (например, в async методах до C# 13).

Здесь мы сталкиваемся с главным врагом долгоживущих серверных приложений — **Large Object Heap (LOH)**.

### 2. Теория: Анатомия Кучи и Проблема LOH

CLR (Common Language Runtime) разделяет управляемую кучу на несколько сегментов. Понимание их поведения критично для Senior-разработчика.

#### 2.1. Пороговое значение 85,000 байт

Любой объект размером **>= 85,000 байт** автоматически попадает в **LOH** (Large Object Heap). Для массива double это примерно 10 625 элементов.

* **Small Object Heap (SOH):** Поколения 0, 1, 2. Здесь происходит компактификация (дефрагментация) памяти. Ссылки сдвигаются, "дыры" закрываются.
* **Large Object Heap (LOH):** Объекты здесь *никогда не копируются* (по умолчанию) во время сборки мусора.

#### 2.2. Почему LOH не компактифицируется?

Копирование 100 МБ памяти — это дорогая операция (блокада шины памяти, загрузка CPU). Если бы GC перемещал большие объекты каждый раз, паузы ("Stop-The-World") длились бы сотни миллисекунд. Поэтому Microsoft приняла архитектурное решение: **LOH работает по принципу свободного списка (Free List)**, как куча в C/C++ (malloc/free).

#### 2.3. Проблема: Фрагментация LOH

Представьте LOH как ряд пустых стульев.

1. Вы аллоцируете массив A (100 КБ).
2. Вы аллоцируете массив B (100 КБ).
3. Массив A удаляется (освобождается). Образуется "дыра" в 100 КБ.
4. Вы хотите аллоцировать массив C (150 КБ). Он не влезает в "дыру" от A. GC вынужден выделить новую память в конце кучи.

Со временем LOH превращается в "швейцарский сыр". Процесс потребляет все больше RAM, хотя физически свободного места много (но оно в мелких "дырах"). Это приводит к OutOfMemoryException даже при наличии свободной памяти.

**Важно:** Начиная с.NET Core 3.0, существует режим GCLargeObjectHeapCompactionMode.CompactOnce, который позволяет принудительно сжать LOH, но это очень дорогая операция, которую нельзя использовать в цикле обработки запросов.

### 3. Решение: ArrayPool<T>

Чтобы не создавать и не выбрасывать большие массивы, мы должны их **переиспользовать**. В.NET для этого существует класс System.Buffers.ArrayPool<T>.

#### 3.1. Механика работы

ArrayPool<T> — это потокобезопасная реализация паттерна "Пул объектов".

* **Rent(int minSize):** Вы просите массив *минимум* такого размера. Пул может вернуть массив большего размера (и часто делает это).
* **Return(T array, bool clearArray):** Вы возвращаете массив обратно в пул.

C#

// Берем массив минимум на 4000 элементов  
int buffer = ArrayPool<int>.Shared.Rent(4000);  
  
try   
{  
 // Важно: buffer.Length может быть 4096 или 8192 (степень двойки)  
 // Всегда используйте переменную с реальным количеством данных!  
 int itemsWritten = ProcessData(buffer);  
   
 // Работаем только с заполненной частью  
 var validData = buffer.AsSpan(0, itemsWritten);   
}  
finally   
{  
 // Обязательно возвращаем в пул в блоке finally  
 ArrayPool<int>.Shared.Return(buffer);  
}

#### 3.2. Стратегия "Buckets" (Ведра)

ArrayPool внутри разбит на "ведра" (buckets). Каждое ведро хранит массивы определенной степени двойки (16, 32,..., 1024, 2048...).

Когда вы просите Rent(1000), пул ищет массив в ведре "1024". Если там пусто, он аллоцирует новый. Если вы просите Rent(1025), пул выдаст массив на 2048.

**Это объясняет, почему нельзя полагаться на .Length арендованного массива.** Вы просили 1000, вам дали 2048. Остальные 1048 байт могут содержать "мусор" от предыдущего использования.

### 4. Критические ошибки (Anti-Patterns) при работе с пулами

Работа с ручным управлением памятью (а ArrayPool — это именно оно) возвращает нас к классу проблем C++.

#### 4.1. Use-After-Return (Использование после возврата)

Самая страшная ошибка безопасности.

1. Поток А берет массив arr из пула.
2. Поток А возвращает arr в пул.
3. Поток Б берет этот же массив arr из пула.
4. Поток А (по ошибке) продолжает писать в arr.

**Результат:** Поток А коррумпирует данные Потока Б. Это может привести к тому, что один пользователь увидит данные другого (например, в веб-сервере). Отладка таких багов чрезвычайно сложна.

#### 4.2. Double-Return (Двойной возврат)

Если вы вернете один и тот же массив в пул дважды, пул может выдать его двум разным потребителям одновременно.

Результат: Та же гонка данных (Race Condition) и порча памяти.

#### 4.3. Невозврат (Memory Leak)

Если вы взяли массив и забыли вернуть (не использовали try/finally), он просто будет удален GC. Это не утечка памяти в классическом смысле (память освободится), но это **утечка производительности**. Пул будет вынужден аллоцировать новые массивы, нагружая GC, вместо переиспользования старых.

#### 4.4. Очистка массива (clearArray = true)

Метод Return принимает параметр bool clearArray.

* false (по умолчанию): Массив возвращается "грязным". Это очень быстро.
* true: Пул зануляет массив (Array.Clear). Это стоит CPU. Используйте только если массив содержал чувствительные данные (пароли, ключи шифрования).

### 5. Advanced: Pinned Object Heap (POH) и Unsafe

В.NET 5 был введен **Pinned Object Heap (POH)**.

#### 5.1. Проблема Pinning (Закрепления)

Когда мы передаем массив в нативный код (ОС, драйвер сети), мы должны "закрепить" (pin) его, чтобы GC не переместил его во время дефрагментации. Закрепленный объект в SOH (Gen 0/1/2) мешает компактификации, создавая фрагментацию.

#### 5.2. POH

Теперь мы можем аллоцировать массивы сразу в Pinned Heap:

C#

// Аллокация в Pinned Object Heap  
int pinnedArray = GC.AllocateUninitializedArray<int>(10000, pinned: true);

Такие массивы гарантированно не перемещаются. Это полезно для создания долгоживущих буферов для сетевых сокетов (SocketAsyncEventArgs).

#### 5.3. Введение в Unsafe и Memory<T>

ArrayPool<T> возвращает массив T. Но что если нам нужна абстракция, которая может указывать и на массив, и на нативную память?

**Memory<T>** — это универсальный контейнер. В отличие от Span<T> (который живет только на стеке), Memory<T> может жить в куче (например, быть полем класса).

* Memory<T> можно легко превратить в Span<T> методом .Span (но это стоит дешевой проверки границ).
* Memory<T> идеально подходит для async/await методов, куда нельзя передать Span.

**Сценарий:**

1. Rent массива из пула.
2. Оборачивание в Memory<byte>.
3. Передача в Stream.ReadAsync(Memory<byte>).
4. После await, получение Span из Memory и парсинг данных.
5. Return массива в пул.

### 6. Практическая памятка (Cheat Sheet)

| **Операция** | **Инструмент** | **Риски** |
| --- | --- | --- |
| Нужен буфер < 1 КБ внутри метода | stackalloc byte[N] | Stack Overflow при больших N или в рекурсии |
| Нужен буфер > 1 КБ или для async | ArrayPool<byte>.Shared | Забыть вернуть, вернуть дважды, использовать после возврата |
| Нужна подстрока без аллокации | Span<T>.Slice | Нельзя хранить в полях класса (ref struct) |
| Нужен буфер для async метода | Memory<T> | Чуть медленнее Span при доступе к элементам |
| Нужен буфер для Interop/OS | GC.AllocateArray(pinned: true) | Создает нагрузку на POH, нужно использовать разумно |

### 7. Итог

Четвертый день трансформирует ваше понимание стоимости создания объектов.

* **Junior:** new byte — "просто массив".
* Senior: new byte — давление на GC. ArrayPool — риск use-after-free.  
  Выбор правильного инструмента требует понимания жизненного цикла данных:

1. Живет только в синхронном методе? -> Span / stackalloc.
2. Живет в async методе или между запросами? -> Memory + ArrayPool.
3. Нужно передавать в Native API? -> Pinned Heap или fixed блок.