출처: http://ikorin2.hatenablog.jp/entry/2020/05/13/042007

메모리 확보 (+ 삭제) 방법을 6가지 준비했다.

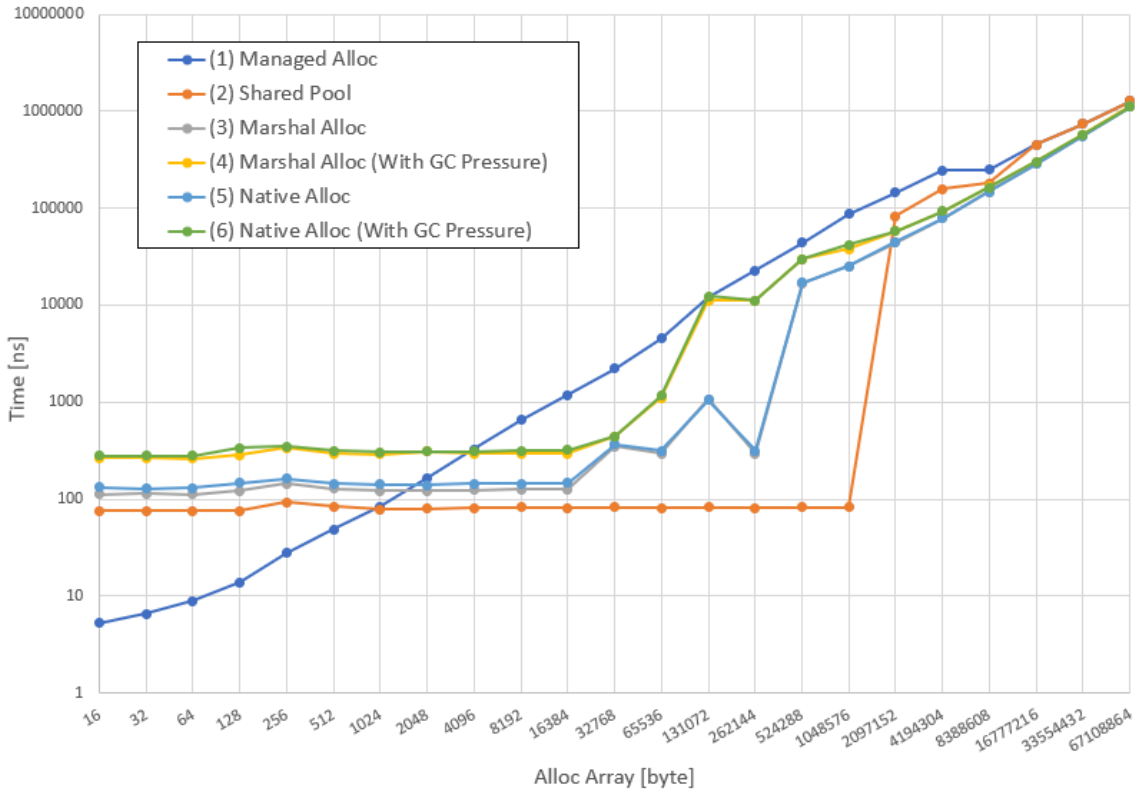
```
// (1)  new
byte[] MarshalAlloc()
{
    return new byte[N];
}

// (2)  ArrayPool
void SharedPool()
{
    var array = ArrayPool<byte>.Shared.Rent(N);
    ArrayPool<byte>.Shared.Return(array);
}

// (3)  Marshal로 확보
void MarshalAlloc()
{
    var array = Marshal.AllocHGlobal(N);
    Marshal.FreeHGlobal(array);
}

// (4)  (3) + Memory Pressure
void MarshalAlloc_WithGCPressure()
{
    var array = Marshal.AllocHGlobal(N);
    GC.AddMemoryPressure(N);
    Marshal.FreeHGlobal(array);
    GC.RemoveMemoryPressure(N);
}

// (5)  C++의 "std::malloc", "std::free" 호출
void NativeAlloc()
{
    var array = NativeAllocator.Alloc(N);
    NativeAllocator.Free(array);
}

// (6)  (5) + Memory Pressure
void NativeAlloc_WithGCPressure()
{
    var array = NativeAllocator.Alloc(N);
    GC.AddMemoryPressure(N);
    NativeAllocator.Free(array);
    GC.RemoveMemoryPressure(N);
}
```

우선 (1)은 일반 관리 배열이다. 이를 바탕으로 다른 방법의 속도를 비교하는 것이다. (2)는 ArrayPool<T>.Shared로 배열 획득 및 반환. (3)과 (4)는 Marshal.AllocHGlobal(int)로 관리되지 않는 메모리의 취득과 해방, (4)는 GC에 MemoryPressure를 더 하고 있다. (5)와 (6)은 관리 되지 않는 메모리의 확보를 C++ 에서 std::malloc과 std::free만 하는 dll을 준비하고 P/Invoke 하고 있다. (3)과 (4)의 속도 비교를 위해 준비했다.

벤치 마크 결과

벤치 마크는 BenchmarkDotNet에서 측정. 환경은 .Net Framework 4.8(x86)이다.
왜 .NET Core 3.1 (x64) 이 아니냐고 하면, 우연히 이미 있던 과거의 벤치 마크용 프로그램을 적당히 재 사용해서 측정한 것이다. Core 3.1로 측정하면 전체적으로 10% 정도는 빨라질 것이라고 생각하지만 각 방법의 속도 비율은 그다지 변하지 않으리라 생각한다.

```
BenchmarkDotNet=v0.12.1, OS=Windows 10.0.18362.778 (1903/May2019Update/19H1)
Intel Core i5-4300U CPU 1.90GHz (Haswell), 1 CPU, 4 logical and 2 physical cores
  [Host]     : .NET Framework 4.8 (4.8.4121.0), X86 LegacyJIT
  DefaultJob : .NET Framework 4.8 (4.8.4121.0), X86 LegacyJIT
```



(1)이 선형, (2)이 N=2^20 때까지는 정수라는 것은 예상대로.
(2)의 ArrayPool<T>은 그렇게까지 빠른 처리가 아니므로 N=1024 정도까지는 new byte[N]에 단순한 속도만으로 진다는 것을 알 수 있다. 그러나 쓰레기는 0.
ArrayPool<T>이 N=2^20 때까지 pool 되어 있지만 N=2^20+1에서는 풀링 되지 않은 배열을 새로 생성하고 있기 때문에 속도가 약 천 배 감소하고 있는 것을 알 수 있다. (pool 되지 않으므로 가비지 되는데다 85000 byte 이상이므로 LOH (Large Object Heap)에 들어가 gen2가 되는 최악의 시나리오를 밟고 있다)

(3)과 (5) 및 (4)와 (6)은 거의 같은 동작을 하고 있으며, Marshal.AllocHGlobal의 내용이 거의 std::malloc과 동일하다고 추측 할 수 있다. Marshal.AllocHGlobal 내용을 따라 가면 windows 의 경우 P/Invoke 에 kernel32.dll의 LocalAlloc을 호출 하고 있다. 하나 std::malloc는 컴파일 될 때 어떻게 될지 솔직히 자세히 모르겠지만, 벤치 마크의 그래프가 딱 싱크로하고 있는 것을 보는 한 거의 같은 일을 하고 있다고 생각한다.

(4)와 (6)이 (3)과 (5)에 비해 분명 늦은 것은 단순히 처리가 늘어난 것과 MemoryPressure가 GC 를 유발하고 있기 때문이다.

C#의 레이어에서만 보면 관리 되지 않는 메모리의 확보도 (1)과 같은 선형이 될 것 같은 생각이 들지만, 관리되지 않는 메모리는 OS의 메모리 관리의 영향을 직접적으로 받기 때문에 그렇게 단순하지 없는 모양. 벤치 마크를 보는 한 N=2^14 정도는 상수 N=2^19 정도 이상으로는 거의 선형이 되는 느낌이 든다.

최적의 솔루션

그런 것은 없다.

최적의 솔루션이 있다면 모두 메모리 관리로 고생하지 않는다. 각종 방법의 특성을 제대로 기억하고 유연하게 구사하할 수 밖에 없다. 그러나 너무 지나치면 목적에 따라 자작 GC에 파묻혀 버리므로 유연하게 대응하자. 대부분의 목적이라면 .NET의 GC로 충분히 고성능이고 gen0의 회수는 매우 빠르다. 그러나 LOH에 배열을 확보하는 것은 그만두자.