ConcurrencyToolkit

Евгений Пешков

@epeshk

О чём будем говорить?

- Concurrency-примитивы в .NET – особенности и недостатки
- Идеи из других платформ
- Сторонние библиотеки:
  - ConcurrencyToolkit
  - Disruptor.NET
  - NonBlocking

What every CLR developer must know before writing code (2006)

2.6.12 Do not get clever with "lockless" reader-writer data structures

Earlier, we had several hashtable structures that attempted to be "clever" and allow **lockless reading**.

Of course, these structures **didn't take into account multiprocessors** and the **other memory models**.

Even on single-proc x86, stress uncovered exotic race conditions. This wasted **a lot of developer time** debugging stress crashes.

We **finally stopped being clever** and **added proper synchronization**, with **no serious perf degradation**.

So if you are tempted to get clever in this way again, **stop and do something else until the urge passes.**

https://github.com/dotnet/runtime/blob/main/docs/coding-guidelines/clr-code-guide.md#2.6.12

2006 vs 2023: CPU

- 2006: 2 ядра, 2 гига
- 2023:
  - 50% десктопов и ноутбуков – 6 и 8 ядер (Steam)
  - AMD Epyc (128 ядер)
  - ARM: Apple M3, Ampere (80 ядер)

  - vCPU и serverless: время == деньги
  - Single core boost

https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam

2006 vs 2023: Concurrency

- Корректный код писать также сложно, даже сложнее
- Блокировки заметнее влияют на производительность

```
| String_Intern       | 823 ms |
| StringCache_Intern  |  32 ms |
```

https://sergeyteplyakov.github.io/Blog/benchmarking/2023/12/10/Intern_or_Not_Intern.html

Современный .NET

- Избавление от блокировок в BCL
  - TimerQueueTimer (портировано в .NET Framework)
  - string.Intern (пока только в NativeAOT)
- Избавление от аллокаций в ConcurrentQueue
- Эксперименты с асинхронностью:
  - green threads
  - async2

.NET: точки улучшения

- Стандартные concurrent-примитивы хороши в 99% случаев
- Меняя внутреннюю реализацию легко сломать производительность уже написанного кода
- Есть риск занести баги в рантайм/BCL
- Популярные тесты сравнивают не concurrency, а json-сериализацию

.NET: сторонние concurrent-библиотеки

- ConcurrencyToolkit
  - Примитивы синхронизации и concurrent-коллекции, заимствованные из других платформ
  - Аналоги internal-классов из BCL
  - Приёмы из production-опыта
- VSadov/NonBlocking
  - Реализация hash map без блокировок на запись
- Disruptor.NET
  - Порт LMAX Disruptor – алгоритма быстрой передачи сообщений между потоками

ConcurrencyToolkit

- ThreadSafeCounter64
- LocklessPool (Array, Object)
- SingleWriterDictionary
- StripedDictionary
- Semaphores

Счётчик

```
long count;
Interlocked.Increment(ref count);
```

Счётчик

- Задача – сделать счётчик для метрик, минимально влияющий на производительность основного кода

```
long count;

Lock:   lock (syncObj) count++;

Fetch-and-add: Interlocked.Increment(ref count);

Compare-and-swap:
while (true) {
  var c = count;
  if (Interlocked.CompareExchange(ref count, c + 1, c) == c)
    break;
}
```

Scalability

- Запись в одну переменную (кэш-линию) с разных потоков – не масштабируется
- От масштабируемого (scalable) алгоритма ожидается рост пропускной способности с увеличением количества исполнителей (потоков)

Threads	Lock	Atomic
1	110	700
2	18	240
4	9	70
8	8	90
16	9	83

Scalability: решение

- Использовать не одну переменную, а несколько
- Для получения результата считать сумму
- Аналог в Java: LongAdder, Striped64

```
values = new PaddedLong[Environment.ProcessorCount];
Interlocked.Increment(ref values[GetSlotId()].Value);
```

```
[StructLayout(LayoutKind.Explicit, Size = 64)]
internal struct PaddedLong
{
  [FieldOffset(0)]
  public long Value;
}
```

GetSlotId

- `Environment.CurrentManagedThreadId`
- `Thread.GetCurrentProcessorId()`

- `ThreadLocal ?`

Scalability: решение

- Использовать не одну переменную, а несколько
- Для получения результата считать сумму

```
var counter = new ThreadSafeCounter64();
counter.Increment();
counter.Add(100);

var result = counter.GetAndReset();
```

Пул объектов: стандартная реализация

Microsoft.Extensions.ObjectPool.DefaultObjectPool

- На основе ConcurrentQueue<T>
- Ограниченный размер – достигается дополнительным счётчиком

```
public T Get() {
  var item = _item;
  if (item == null || Interlocked.CompareExchange(ref _item, null, item) != item) {
    if (!_items.TryDequeue(out item))
      return new T();
    Interlocked.Decrement(ref _numItems);
    return item;
  }

  return item;
}
```

Пул объектов: стандартная реализация

- Работает быстро в отсутствии конкуренции
- В условиях конкуренции:
  - Нагрузка на ConcurrentQueue
  - Нагрузка на счётчик

ConcurrentQueue внутри

```
public class ConcurrentQueue<T>
{
  object _crossSegmentLock;
  volatile ConcurrentQueueSegment<T> _tail;
  volatile ConcurrentQueueSegment<T> _head;
}
```

- `ConcurrentQueueSegment` сам по себе ограниченного размера
- Можно избавиться от внешнего счётчика элементов

- FixedSizeConcurrentQueue в ConcurrencyToolkit

Идеи из ArrayPool<T>.Shared

- Храним один объект на поток в `ThreadLocal<T>`
- Несколько FixedSizeConcurrentQueue

- `LocklessArrayPool<T>`
- `ObjectPool<T>`

Семафоры

- Семафор – примитив синхронизации, пропускающий внутрь не более N воркеров (потоков или асинхронных задач)

```
await semaphore.WaitAsync(token);
try { /* ... */ }
finally {
  semaphore.Release(1);
}
```

Недостатки SemaphoreSlim

- Стандартный SemaphoreSlim основан на блокировке
- Полезен только для длительных активностей, с низкой нагрузкой на Wait/Release
- Иначе не будет большой разницы с блокировкой потоков

- Непригоден для троттлинга сетевых запросов
- В kotlinx-coroutines Semaphore сделан без блокировок

Интерфейс ISemaphore

```
public interface ISemaphore
{
  ValueTask AcquireAsync(CancellationToken token=default);
  ValueTask<bool> TryAcquireAsync(CancellationToken token=default);

  void Acquire(CancellationToken token=default);
  bool TryAcquire(CancellationToken token=default);

  bool TryAcquireImmediately();

  void Release();

  int CurrentCount { get; }
  int CurrentQueue { get; }
}
```

Интерфейс ISemaphore

```
public interface ISemaphore
{
  ValueTask AcquireAsync(CancellationToken token=default);
  ValueTask<bool> TryAcquireAsync(CancellationToken token=default);

  void Acquire(CancellationToken token=default);
  bool TryAcquire(CancellationToken token=default);

  bool TryAcquireImmediately();

  void Release();

  int CurrentCount { get; }
  int CurrentQueue { get; }
}
```

Интерфейс ISemaphore

```
public interface ISemaphore
{
  ValueTask AcquireAsync(CancellationToken token=default);
  ValueTask<bool> TryAcquireAsync(CancellationToken token=default);

  void Acquire(CancellationToken token=default);
  bool TryAcquire(CancellationToken token=default);

  bool TryAcquireImmediately();

  void Release();

  int CurrentCount { get; }
  int CurrentQueue { get; }
}
```

Интерфейс ISemaphore

```
public interface ISemaphore
{
  ValueTask AcquireAsync(CancellationToken token=default);
  ValueTask<bool> TryAcquireAsync(CancellationToken token=default);

  void Acquire(CancellationToken token=default);
  bool TryAcquire(CancellationToken token=default);

  bool TryAcquireImmediately();

  void Release();

  int CurrentCount { get; }
  int CurrentQueue { get; }
}
```

Интерфейс ISemaphore

```
public interface ISemaphore
{
  ValueTask AcquireAsync(CancellationToken token=default);
  ValueTask<bool> TryAcquireAsync(CancellationToken token=default);

  void Acquire(CancellationToken token=default);
  bool TryAcquire(CancellationToken token=default);

  bool TryAcquireImmediately();

  void Release();

  int CurrentCount { get; }
  int CurrentQueue { get; }
}
```

Реализации ISemaphore

С использованием concurrent-коллекций:

- ConcurrentQueueSemaphore
- ConcurrentStackSemaphore

На основе kotlinx-coroutines:

- SimpleSegmentSemaphore
- SegmentSemaphore

На основе блокировки:

- PrioritySemaphore

Benchmarks

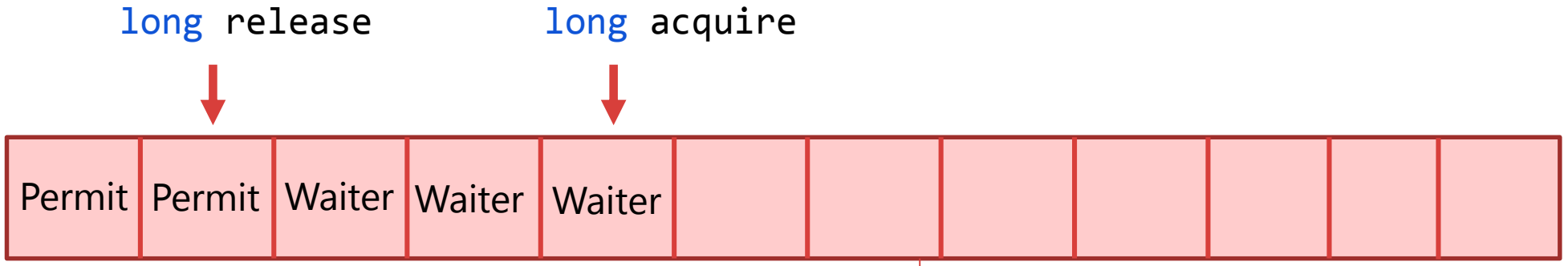
Without cancellation token

Semaphore	Mean	Allocated
Queue	201.8 ms	1.54 KB
Segment	176.4 ms	11222.83 KB
SimpleSegment	170.3 ms	7944.15 KB
Slim	344.7 ms	82379.76 KB
Stack	195.5 ms	30991.4 KB

With cancellation token

Mean	Allocated
215.1 ms	117.87 MB
205.4 ms	11.33 MB
200.9 ms	8.03 MB
1,004.0 ms	347.75 MB
212.7 ms	144.25 MB

Segment queue synchronizer

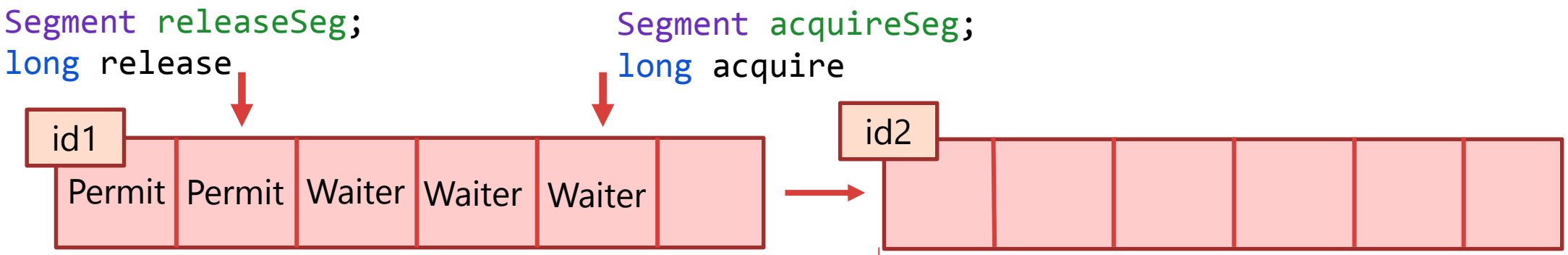



```
bool Acquire(Waiter waiter)
{
  int i = Increment(ref acquire);
  var prev = CAS(ref arr[i], waiter, if: null);
  return prev == null;
}
```

```
void Release()
{
  int i = Increment(ref release);
  var prev = Swap(ref arr[i], permit);
  prev?.SetResult();
}
```

Hydra 2020. Nikita Koval

Segment queue synchronizer

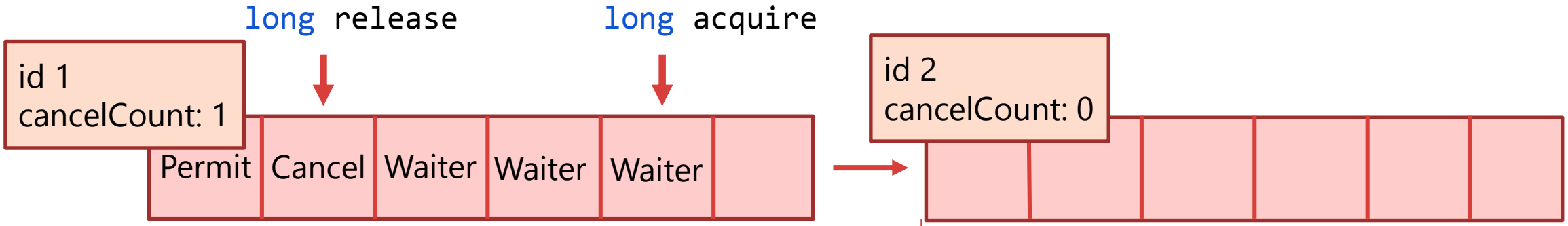



```
bool Acquire(Waiter waiter)
{
  Segment segment = acquireSegment;
  int i = Increment(ref acquire);
  // iterate over links
  // move release index
  // try install waiter
}
```

```
void ReleaseLinked()
{
  Segment segment = releaseSegment;
  int i = Increment(ref release);
  // iterate over links
  // move release index
  // try install waiter
}
```

Hydra 2020. Nikita Koval

Segment queue synchronizer




```
bool Acquire(Waiter waiter)
{
  Segment segment = acquireSegment;
  int i = Increment(ref acquire);
  // iterate over links
  // move release index
  // try install waiter
}
```

```
void Release()
{
  while (true)
  {
    ...
    var prev = Swap(..., permit);
    if (prev == canceled)
      countinue;
    prev?.SetResult();
    return;
   }
}
```

Hydra 2020. Nikita Koval

Передача сообщений между потоками

- Задача: сделать быстрый async-wrapper для логов

- Буфер для данных (лог-сообщений)
- Сигнализация background-потока о появлении новых логов
- Режимы AddOrBlock, TryAdd

Передача сообщений между потоками

- Стандартные решения:
  - BlockingCollection: комбинация ConcurrentQueue и SemaphoreSlim
  - Bounded Channel: ThreadPool, блокировка + AsyncOperation

- Сторонние решения:
  - HellBrick/AsyncCollections
  - Disruptor.NET

AsyncCollections: история одного велосипеда
https://habr.com/ru/articles/240891/comments/

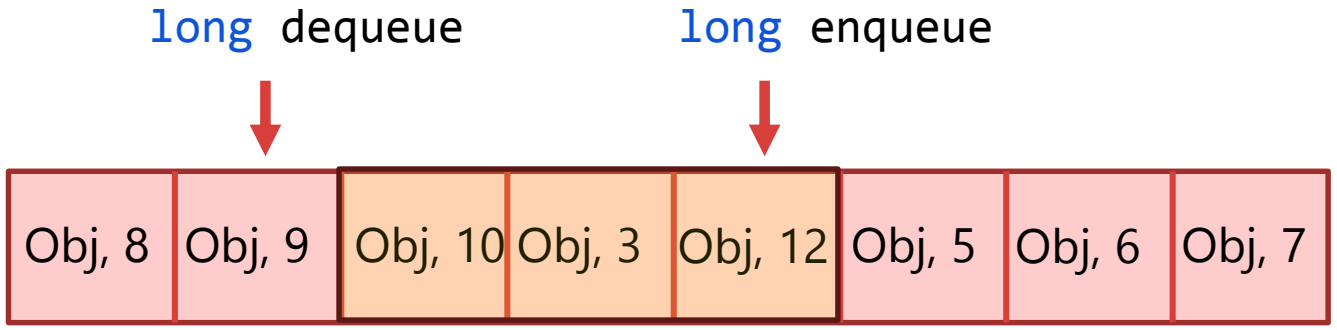
Передача сообщений между потоками

Serilog.Sinks.Async (BlockingCollection)

Serilog.Sinks.Background (Disruptor)

Threads	Async	Background	(K log events / s)
1	2200	3500
2	900	3400
4	730	3300
8	690	3100
16	640	2600

Disruptor.NET




```
void Enqueue(T o)
{
  int i = Increment(ref enqueue) % N;
  WaitAvailability(i);
  arr[i].Value = o;
  arr[i].Index = i;
}
```

Другой поток ещё не успел обновить индекс!

Disruptor.NET: подвохи

- TryAdd использует compare-and-swap вместо fetch-and-add
- Waiting strategy:
  - SpinWait
  - AggressiveSpinWait
  - BlockingWaitingStrategy
  - Custom

ConcurrentDictionary: недостатки

- Использует блокировки на запись
  - VSadov/NonBlocking
- Аллоцирует объект на каждый элемент
  - ConcurrencyToolkit

ConcurrentDictionary: GC overhead

```
ConcurrentDictionary<Guid, Guid>
GC.Collect(2)
```

Count	Mean
100_000	4.9 ms
1_000_000	61.9 ms
10_000_000	474.1 ms

```
Dictionary<Guid, Guid>
GC.Collect(2)
```

Count	Mean
100_000	69.15 us
1_000_000	66.75 us
10_000_000	67.59 us

Tricks from .net Framework 1.1

`System.Collections.Hashtable`

"Hashtable is thread safe for use by **multiple reader threads** and a **single writing thread**. It is thread safe for multi-thread use when only one of the threads perform write (update) operations" – MSDN

Clean reading: writer

```
bool writing;
int version;

this.writing = true;
buckets[index] = ...;
this.version++;
this.writing = false;
```

Clean reading: reader

```
bool writing;
int version;


while (true)
{
    int version = this.version;
    bucket = buckets[index];
    if (this.writing || version != this.version)
        continue;
    break;
}
```

Dictionaries benchmark

Method	Mean	Allocated
SingleWriter_WritingTime	79.23 ms	709 B
Concurrent_WritingTime	264.94 ms	268436184 B
NonBlocking_WritingTime	171.32 ms	167772893 B
SingleWriter_ReadingTime	151.13 ms	1052 B
Concurrent_ReadingTime	90.11 ms	86550712 B
NonBlocking_ReadingTime	98.78 ms	90740858 B

Дальнейшие планы

- Библиотеки логирования:
  - Serilog.Sinks.Background
  - Serilog.Sinks.RawConsole
  - Serilog.Sinks.RawFile
- Оптимизировать SingleWritereDictionary