



Асинхронные примитивы синхронизации: алгоритмы троттлинга запросов

Евгений Пешков

@epeshk

О чём будем говорить?

- Асинхронные примитивы синхронизации
- `System.Threading.RateLimiting` и троттлинг запросов
- `SemaphoreSlim`
- Другие реализации семафоров

Asp.Net Core 7 RateLimiting

- System.Threading.RateLimiting
- Microsoft.AspNetCore.RateLimiting

```
builder.Services.AddRateLimiter(options => options.AddConcurrencyLimiter(...))  
app.UseRateLimiter();
```

- Rate Limiters:
 - Token Bucket Rate Limiter
 - Fixed Window Rate Limiter
 - Sliding Window Rate Limiter
- Concurrency Limiters (Semaphores):
 - Concurrency Limiter (FIFO + LIFO)

Выживание под нагрузкой

Without graceful degradation:

- Рост нагрузки на сервис, упор в неявные лимиты
- Рост latency, ошибок, таймауты на клиентах
- Падение сервиса с OutOfMemoryException
- Деградация производительности после спада нагрузки
- Необходимость ручного вмешательства

Выживание под нагрузкой

With graceful degradation:

- Нагрузочное тестирование
- Рост нагрузки на сервис, упор в заранее определённые лимиты
- Сглаживание пиков нагрузки с помощью очереди запросов
- Отброс запросов со статусом Too Many Requests
- Обработка части запросов без роста latency
- Быстрое восстановление при спаде нагрузки

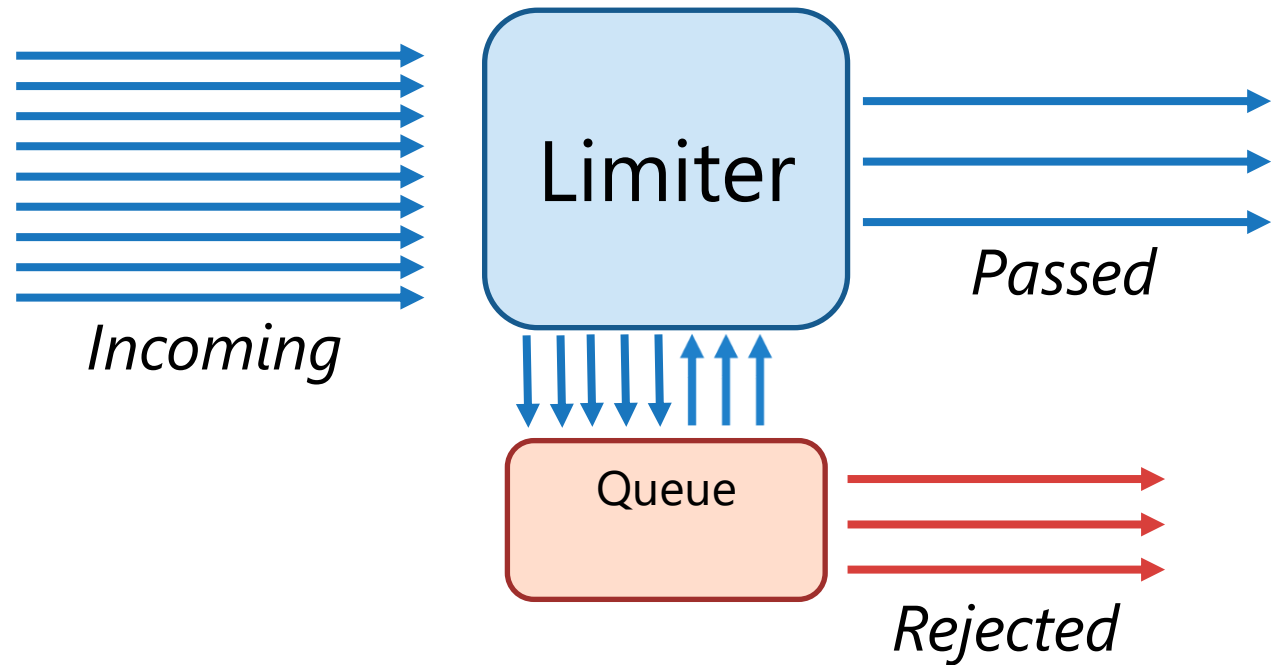
<https://sre.google/sre-book/handling-overload/>

Клиентский троттлинг

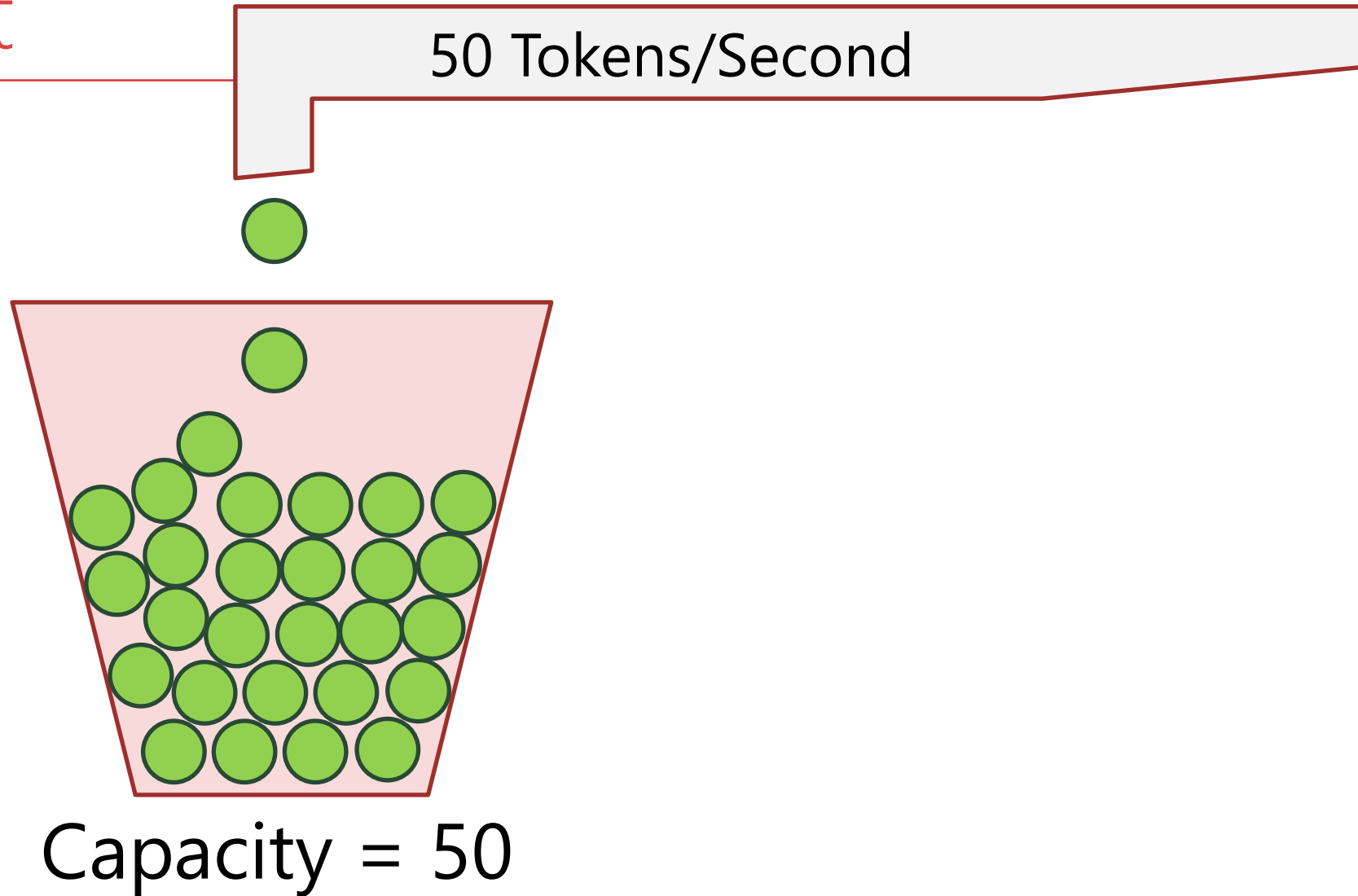
- Если сервер начал троттлить запросы – клиент может помочь ему
- Получив Too Many Request, клиент может:
 - Не отправлять новый запрос, сразу вернув Too Many Requests
 - Ограничить количество retry
 - Увеличить задержку перед retry (exponential backoff)

RateLimiting в коде

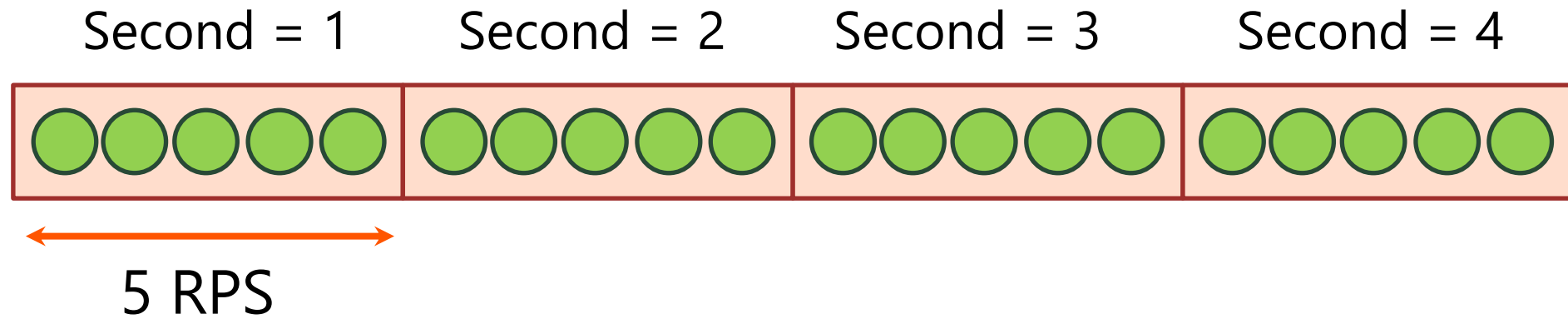
```
// Incoming  
using (await limiter.AcquireAsync())  
{  
    // Passed  
}
```



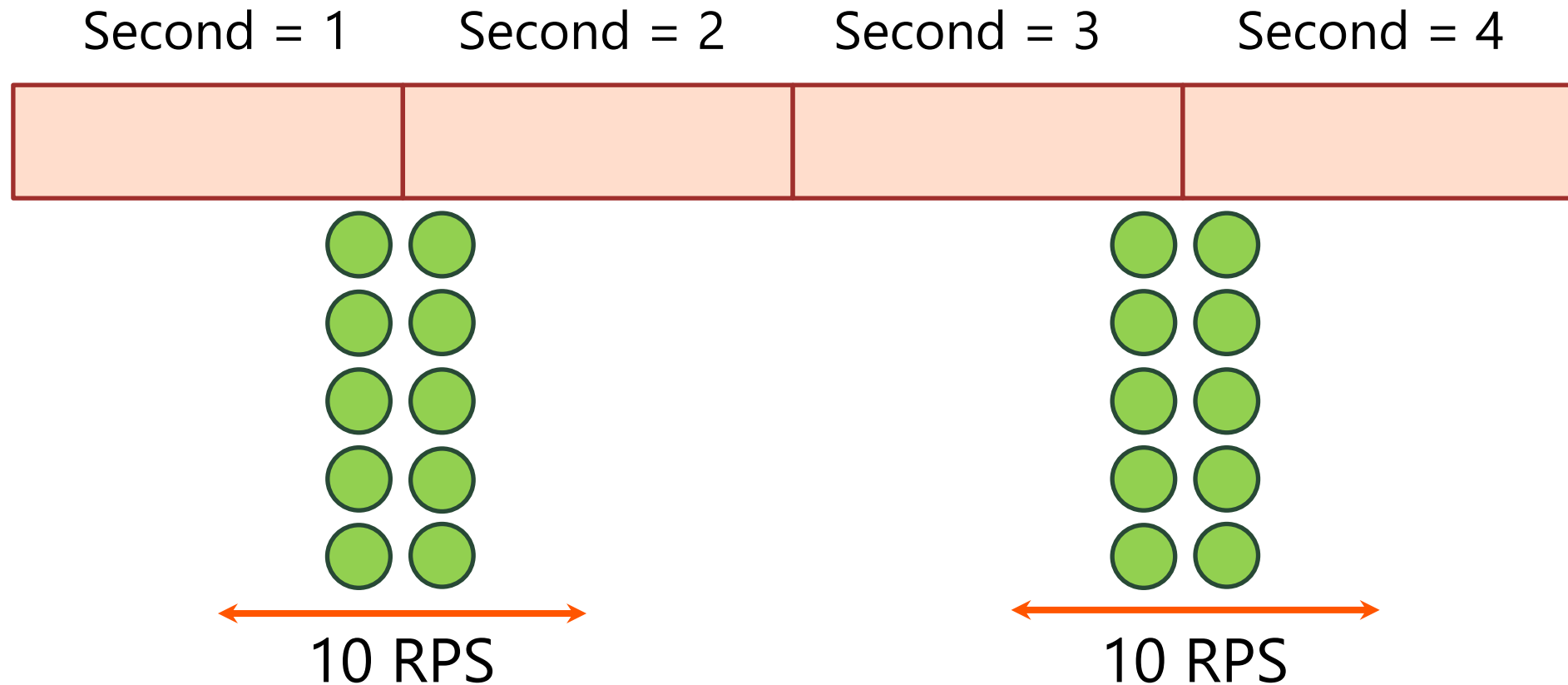
Token Bucket



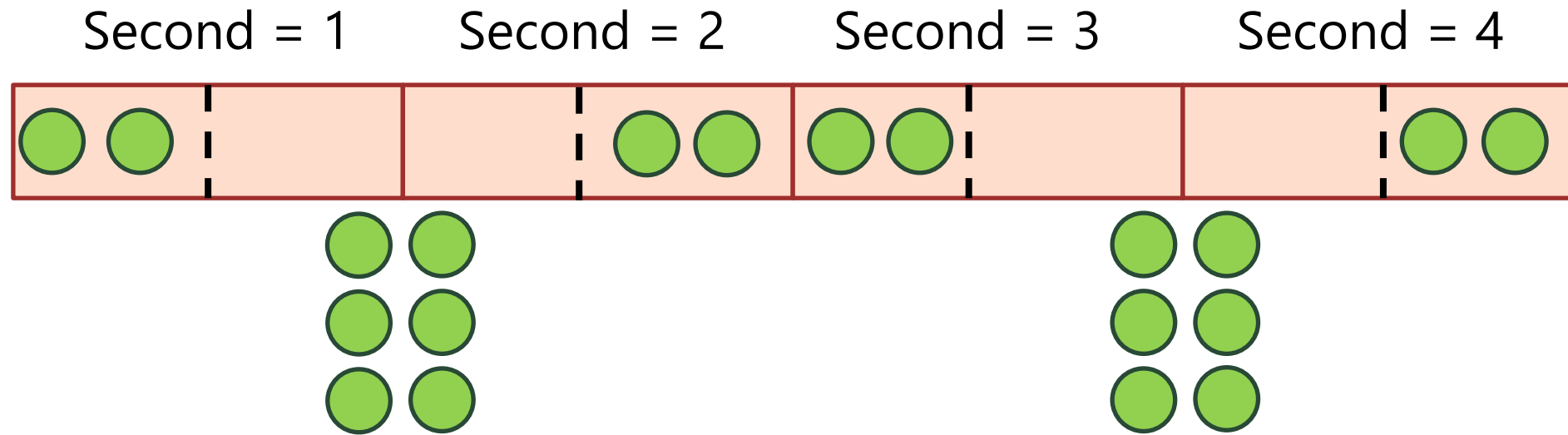
Fixed Window



Fixed Window



Sliding Window



ConcurrencyLimiter

- Позволяет ограничить число одновременно выполняющихся задач
- Неужели наконец-то замена SemaphoreSlim?

Limiter vs Semaphore: constructor

```
new SemaphoreSlim(  
    initialCount: 8,  
    maxCount: int.MaxValue);
```

```
new ConcurrencyLimiter(new ConcurrencyLimiterOptions  
{  
    PermitLimit = 8,  
    QueueLimit = 200,  
    QueueProcessingOrder = QueueProcessingOrder.NewestFirst  
});
```

Limiter vs Semaphore: constructor

```
new SemaphoreSlim(  
    initialCount: 8,  
    maxCount: int.MaxValue);
```

**Можно изменить capacity
пустыми Wait и Release**

```
new ConcurrencyLimiter(new ConcurrencyLimiterOptions  
{  
    PermitLimit = 8,  
    QueueLimit = 200,  
    QueueProcessingOrder = QueueProcessingOrder.NewestFirst  
});
```

Limiter vs Semaphore: constructor

```
new SemaphoreSlim(  
    initialCount: 8,  
    maxCount: int.MaxValue);
```

Reject не поместившихся в очередь
SemaphoreSlim не поддерживает

```
new ConcurrencyLimiter(new ConcurrencyLimiterOptions  
{  
    PermitLimit = 8,  
    QueueLimit = 200,  
    QueueProcessingOrder = QueueProcessingOrder.NewestFirst  
});
```

Limiter vs Semaphore: constructor

```
new SemaphoreSlim(  
    initialCount: 8,  
    maxCount: int.MaxValue);
```

**Можно задать порядок разбора:
FIFO – очередь или LIFO – стек**

```
new ConcurrencyLimiter(new ConcurrencyLimiterOptions  
{  
    PermitLimit = 8,  
    QueueLimit = 200,  
    QueueProcessingOrder = QueueProcessingOrder.NewestFirst  
});
```


Limiter vs Semaphore: usage

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

Limiter vs Semaphore: usage

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

```
using (var lease = await limiter.AcquireAsync(permitCount: 1, token)) {  
    if (!lease.IsAcquired)  
        return;  
    // ...  
}
```

Limiter vs Semaphore: usage

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

**Нет timeout, но token его заменяет
new CancellationTokenSource(timeout)**

```
using (var lease = await limiter.AcquireAsync(permitCount: 1, token)) {  
    if (!lease.IsAcquired)  
        return;  
    // ...  
}
```

Limiter vs Semaphore: usage

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

1) Не нужен try/finally или extension

2) ValueTask<> не IDisposable

```
using (var lease = await limiter.AcquireAsync(permitCount: 1, token)) {  
    if (!lease.IsAcquired)  
        return;  
    // ...  
}
```

Limiter vs Semaphore: usage

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

**Можно сделать Wait
сразу нескольких единиц параллельности**

```
using (var lease = await limiter.AcquireAsync(permitCount: 1, token)) {
    if (!lease.IsAcquired)
        return;
    // ...
}
```

Limiter vs Semaphore: usage

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

**Не нужно ожидать
OperationCanceledException**

**SemaphoreSlim хоть и возвращает bool,
но только при таймауте**

```
using (var lease = await limiter.AcquireAsync(permitCount: 1, token)) {  
    if (!lease.IsAcquired)  
        return;  
    // ...  
}
```

Await без exception

```
try
{
    await task.WaitAsync();
}
catch {
}
```

Support await'ing a Task without throwing
<https://github.com/dotnet/runtime/issues/22144>

Await без exception

```
try
{
    await task.WaitAsync();
}
catch {
}
```

```
if (!await task.WaitAsync().NoThrow())
    return;
```

Support await'ing a Task without throwing
<https://github.com/dotnet/runtime/issues/22144>

Await без exception

```
try
{
    await task.WaitAsync();
}
catch {
}
```

Mean	Allocated
5300 ns	552 B

```
if (!await task.WaitAsync().NoThrow())
    return;
```

Mean	Allocated
10 ns	-

Support await'ing a Task without throwing

<https://github.com/dotnet/runtime/issues/22144>

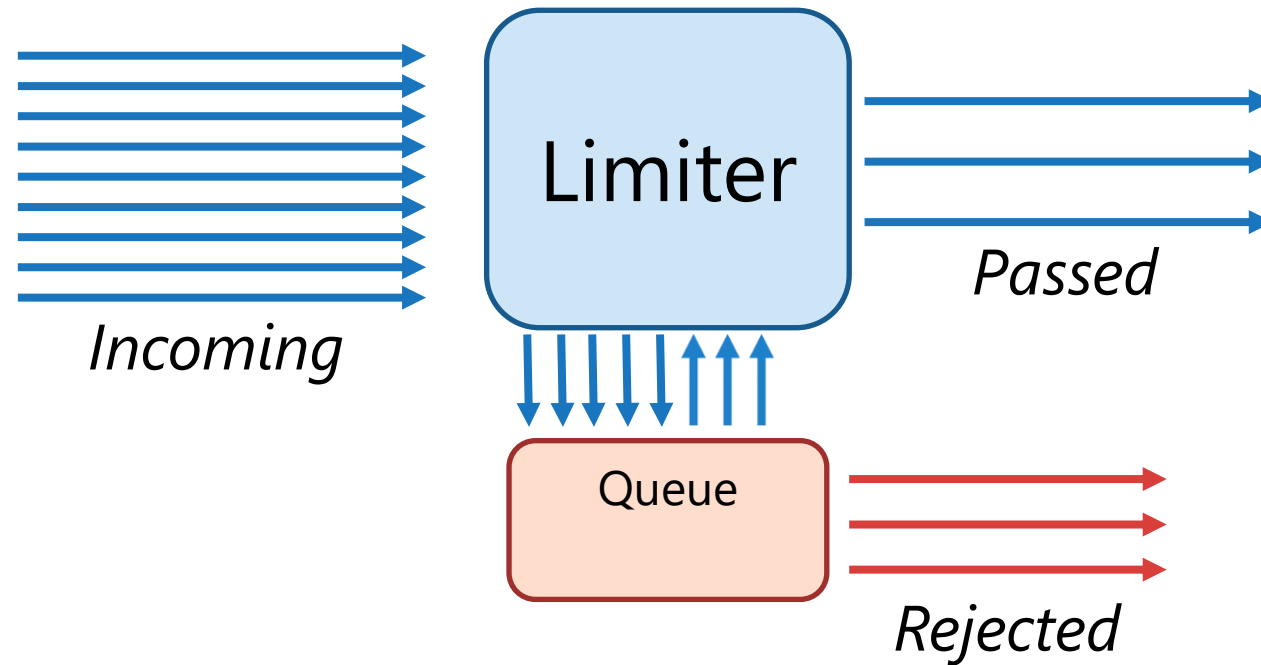
Await без exception

```
public readonly struct ConfiguredNoThrowAwaiter : ICriticalNotifyCompletion {  
    private readonly Task task;  
    public ConfiguredNoThrowAwaiter(Task task) => this.task = task;  
    public ConfiguredNoThrowAwaiter GetAwaiter() => this;  
    public bool IsCompleted => task.IsCompleted;
```

```
    public bool GetResult() {  
        task.MarkExceptionsAsHandled();  
        return task.IsCompletedSuccessfully;  
    }
```

```
    public void UnsafeOnCompleted(Action continuation)  
        => task.ConfigureAwait(false).GetAwaiter().UnsafeOnCompleted(continuation);  
    public void OnCompleted(Action continuation)  
        => task.ConfigureAwait(false).GetAwaiter().OnCompleted(continuation);  
}
```

Limiter vs Semaphore: internals



Примитивы синхронизации

Синхронные:

- EventWaitHandle
 - Semaphore
 - Monitor (lock)
 - BlockingCollection
 - ThreadLocal<T>
-
- Возвращают Void
 - Блокируют поток на всё время ожидания

Асинхронные:

- TaskCompletionSource
 - SemaphoreSlim
 - AsyncLock
 - Channel<T>
 - AsyncLocal<T>
-
- Возвращают Awaitable (Task...)
 - Работают на уровне асинхронных задач, а не потоков

<https://dotnet.github.io/dotNext/features/threading/index.html>
<https://github.com/microsoft/vs-threading>

TaskCompletionSource

Сигнал. Почти что CancellationToken,
только на нём можно сделать await

```
var tcs = new TaskCompletionSource();  
Task task = tcs.Task;  
  
tcs.SetResult();  
tcs.TrySetResult();  
tcs.TrySetCanceled();
```

TaskCompletionSource: thread theft

Сигнал. Почти что CancellationToken,
только на нём можно сделать await


```
var tcs = new TaskCompletionSource();  
Task task = tcs.Task;
```

// Task 1

```
tcs.SetResult();  
Console.WriteLine(1);
```

// Task 2

```
await task;  
Thread.Sleep(-1);
```

A blue arrow originates from the `tcs.SetResult();` line in Task 1 and points to the `Thread.Sleep(-1);` line in Task 2. The `Thread.Sleep(-1);` line is enclosed in a red dashed rectangular box.

TaskCompletionSource: thread theft

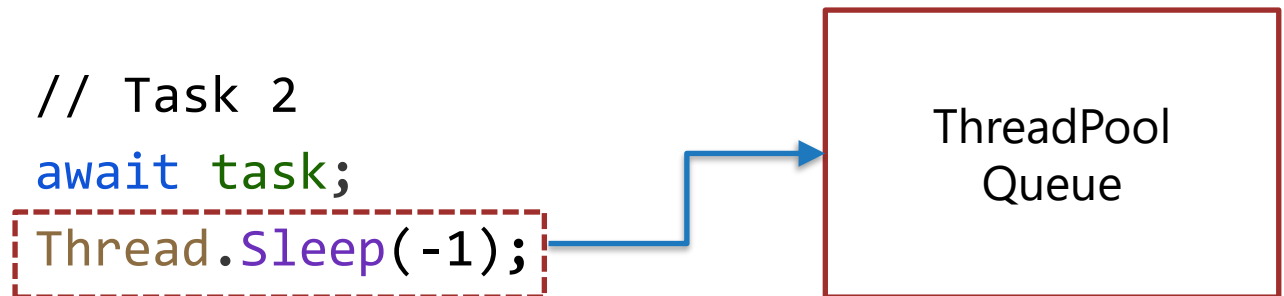
Сигнал. Почти что CancellationToken,
только на нём можно сделать await

```
var tcs = new TaskCompletionSource(RunContinuationsAsynchronously);  
Task task = tcs.Task;
```

```
// Task 1  
tcs.SetResult();  
Console.Write(1);
```

```
// Task 2  
await task;
```

```
Thread.Sleep(-1);
```



Название не обмануло

```
public sealed class ConcurrencyLimiter : RateLimiter {  
    private int _permitCount;  
    private readonly Deque<TaskCompletionSource> _queue = new();  
    private object Lock => _queue;  
}
```


Название не обмануло

```
public sealed class ConcurrencyLimiter : RateLimiter {  
    private int _permitCount;  
    private readonly Deque<TaskCompletionSource> _queue = new();  
    private object Lock => _queue;  
}
```

```
ValueTask Acquire(...) {  
    // Perf: Check SemaphoreSlim implementation instead of locking  
    lock (Lock) { /* ... */ }  
}
```

Поможет ли переход на SemaphoreSlim против лока?

Benchmark

```
var iter = 0;
while (iter < n)
{
    using (var lease = await limiter.AcquireAsync())
    {
        if (!lease.IsAcquired)
            continue;
        iter++;
        DoRandomWork(random);
    }
}
```

Benchmark

Method	Permits	Threads	Mean	Contentions	Allocated
-----	-----	-----	-----:	-----:	-----:
NoSemaphore	16	16	44 ms	-	10 KB
ConcurrencyLimiter	16	16	146 ms	9960	10250 KB
SemaphoreSlim	16	16	134 ms	9452	10 KB
ConcurrencyLimiter	16	32	331 ms	3805	10258 KB
SemaphoreSlim	16	32	245 ms	10660	21645 KB

SemaphoreSlim

```
public class SemaphoreSlim : IDisposable {  
    private volatile int m_currentCount; // <- счётчик мест  
  
    private int m_waitCount;             // <- счётчик синхронных waiter  
    private int m_countOfWaitersPulsedToWake;  
  
    private TaskNode? m_asyncHead;       // <- linked list  
    private TaskNode? m_asyncTail;  
  
    private readonly object m_lockObj;    // <- sync object  
}
```

Deque vs LinkedList

- LinkedList поддерживает удаление из середины за $O(1)$
- Deque – поддерживает удаление только с концов
- В итоге:
 - Удаление отменённых waiter происходит в Release
 - Deque может разрастись относительно логического размера
- Добавление в LinkedList происходит за $O(1)$, в Deque – можно попасть на удвоение массива

Оптимизации SemaphoreSlim

```
private sealed class TaskNode : Task<bool>
{
    internal TaskNode Prev, Next;
    internal TaskNode() : base(
        null,
        TaskCreationOptions.RunContinuationsAsynchronously) { }
}
```

- Вместо TaskCompletionSource используется Task напрямую. Активация происходит через internal метод (-1 объект)
- LinkedList Node унаследована напрямую от Task (-1 объект)

Кастомизация SemaphoreSlim

- Что можно кастомизировать:
 - Изменить порядок с очереди (FIFO) на стек (LIFO)
 - Добавить поддержку приоритетов
 - Добавить поддержку единиц конкурентности
 - Добавить ограничение на размер очереди
 - Добавить метрики
 - Убрать `OperationCancelledException`

Изменение порядка очереди

Release:

// Get the next async waiter to release and queue it to be completed

- TaskNode waiterTask = m_asyncHead;

+ TaskNode waiterTask = m_asyncTail;

Поддержка приоритетов

```
public class SemaphoreSlim : IDisposable {  
    ...  
    private TaskNode[] m_asyncHeads; // <- linked lists  
    private TaskNode[] m_asyncTails;  
    ...  
}
```

- Для низкоприоритетных задач можно делать TryWait: .Wait(0);

Поддержка единиц конкурентности

```
private sealed class TaskNode : TaskCompletionSource<bool>
{
    internal TaskNode Prev, Next;

    internal int Weight;

    internal TaskNode() : base(
        null,
        TaskCreationOptions.RunContinuationsAsynchronously) { }
}
```

Выводы

- Плоха ли реализация с локом?
- Нет, в большинстве случаев она подходит
- Корректность такой реализации не вызывает сомнений
- Легко модифицировать под конкретный сценарий

Lock-Free семафоры

- Стандартные .NET-семафоры основаны на блокировке:
 - Применимы для длительных активностей, с низкой нагрузкой на Wait/Release
 - Иначе не будет большой разницы с блокировкой потоков

Счётчик

- Semaphore на основе `int`
- Только `TryAcquire` – пропускает если есть свободное место
- Счётчик содержит количество свободных мест
- Изменяется атомарно: `Swap`, `Compare and Swap`, `Fetch and Add`
- Zero-overhead, если мест хватает: одна `atomic` операция

Счётчик

```
int count;
```

```
bool TryAcquire()  
{  
    int c = Decrement(ref count);  
    if (c >= 0)  
        return true;  
    ...;  
}
```

```
void Release()  
{  
    Increment(ref count);  
}
```

СЧЁТЧИК

```
int count;
```

```
bool TryAcquire()  
{  
    while (true)  
    {  
        int c = count;  
        if (c <= 0)  
            return false;  
        if (CAS(ref count, c-1, if: c))  
            return true;  
    }  
}
```

```
void Release()  
{  
    Increment(ref count);  
}
```

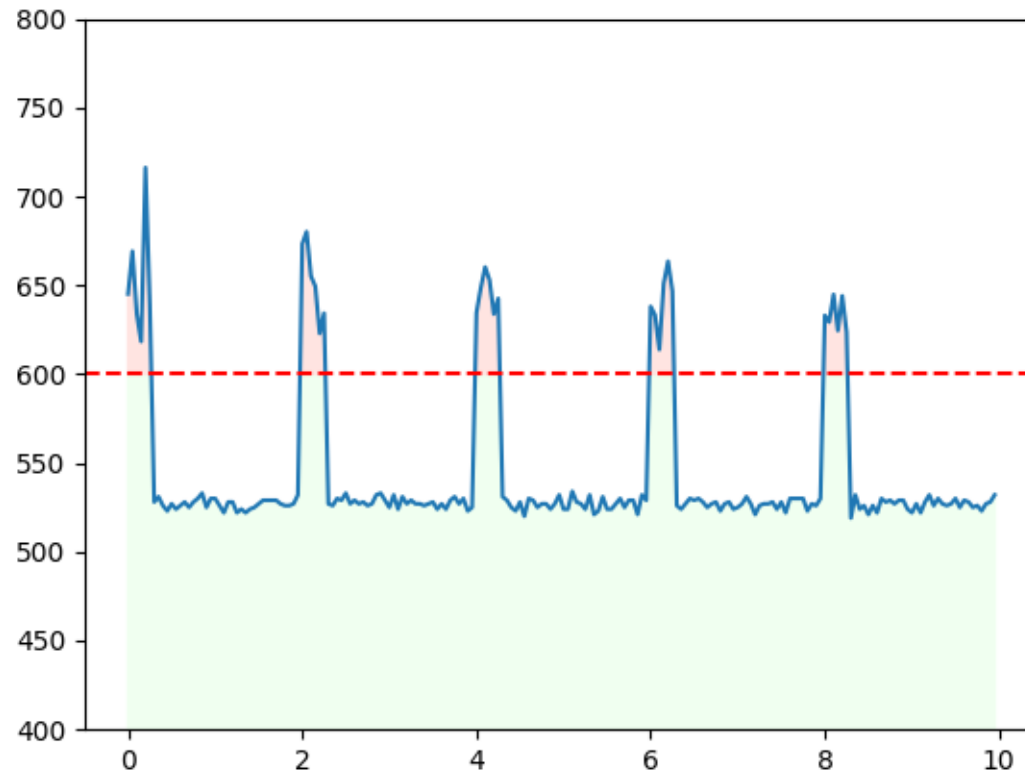
Счётчик

- Kestrel: `FiniteCounter`
- Используется для ограничения количества одновременных соединений

```
builder.WebHost.ConfigureKestrel(options =>
{
    options.Limits.MaxConcurrentConnections = N;
});
```


Доработка: очередь задач

- TryAcquire уже позволяет отклонять запросы при превышении лимита
- Проблема: всплески нагрузки



Доработка: очередь задач

```
int count;
```

```
bool TryAcquire()  
{  
    int c = Decrement(ref count);  
    if (c >= 0)  
        return true;  
    ...;  
}
```

```
void Release()  
{  
    Increment(ref count);  
}
```

Доработка: очередь задач

```
int count; ... waiters = ...;
```

```
Task Acquire()  
{  
    int c = Decrement(ref count);  
    if (c >= 0)  
        return true;  
  
    Waiter w = CreateWaiter();  
    waiters.Add(w);  
    return w.Task;  
}
```

```
void Release()  
{  
    int c = Increment(ref count);  
    if (c > 0)  
        return;  
  
    Waiter w = waiters.Get(waiter);  
    w.SetResult();  
}
```

Доработка: очередь задач

```
int count; ... waiters = ...;
```

```
Task Acquire()  
{  
    int c = Decrement(ref count);  
    if (c >= 0)  
        return true;  
  
    Waiter w = CreateWaiter();  
    waiters.Add(w);  
    return w.Task;  
}
```

```
void Release()  
{  
    int c = Increment(ref count);  
    if (c > 0)  
        return;  
  
    Waiter w;  
    while (!waiters.TryGet(out w)) ;  
    w.SetResult();  
}
```

Доработка: очередь задач

```
int count;
```

Waiter

- TaskCompletionSource
- ManualResetValueTaskSource

waiters

- ConcurrentQueue
- ConcurrentStack

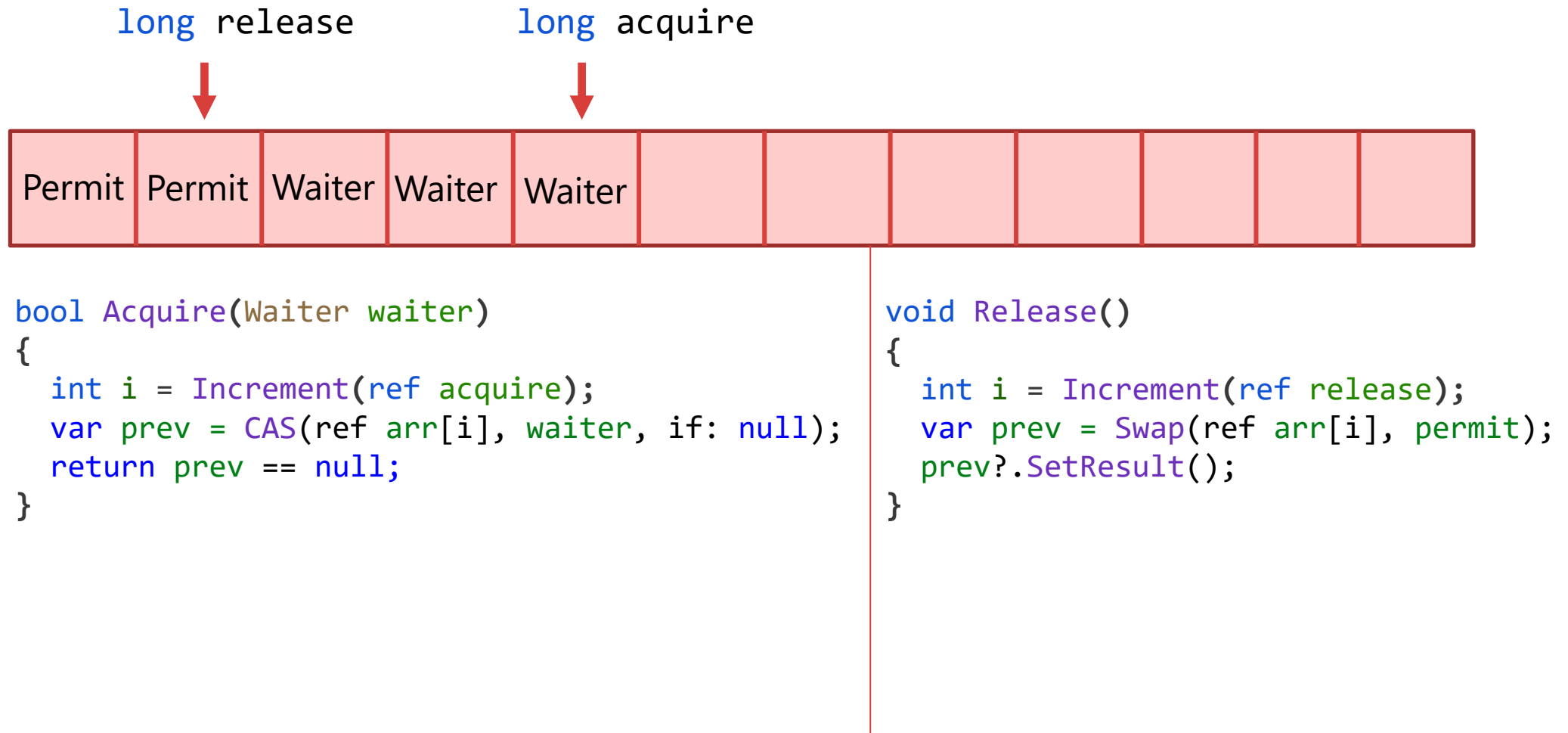
Lock-free: benchmark

Method	Permits	Threads	Mean	Contentions	Allocated
-----	-----	-----	-----:	-----:	-----:
NoSemaphore	16	16	44 ms	-	10 KB
SemaphoreSlim	16	16	134 ms	9452	10 KB
AnyLockFree	16	16	45 ms	0	10 KB
SemaphoreSlim	16	32	245 ms	10660	21645 KB
ConcurrentStack	16	32	88 ms	-	25250 KB
ConcurrentQueue	16	32	70 ms	0.001	22390 KB
VTConcurrentQueue	16	32	70 ms	0.001	24 KB

Ограничения подхода

- ConcurrentQueue и Stack поддерживают удаление только с концов
- Не получится честной поддержки CancellationToken
- Что делать?
- Искать решение за пределами .NET стека

Segment queue synchronizer



Hydra 2020. Nikita Koval Synchronization primitives can be faster with SegmentQueueSynchronizer

<https://www.youtube.com/watch?v=2uxsNJ0TdIM>

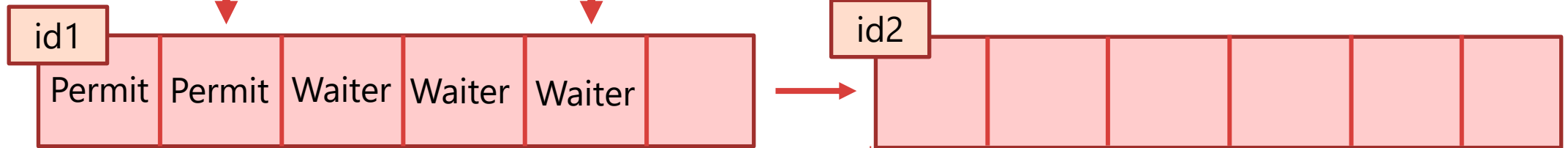
Segment queue synchronizer

Segment releaseSeg;

long release

Segment acquireSeg;

long acquire



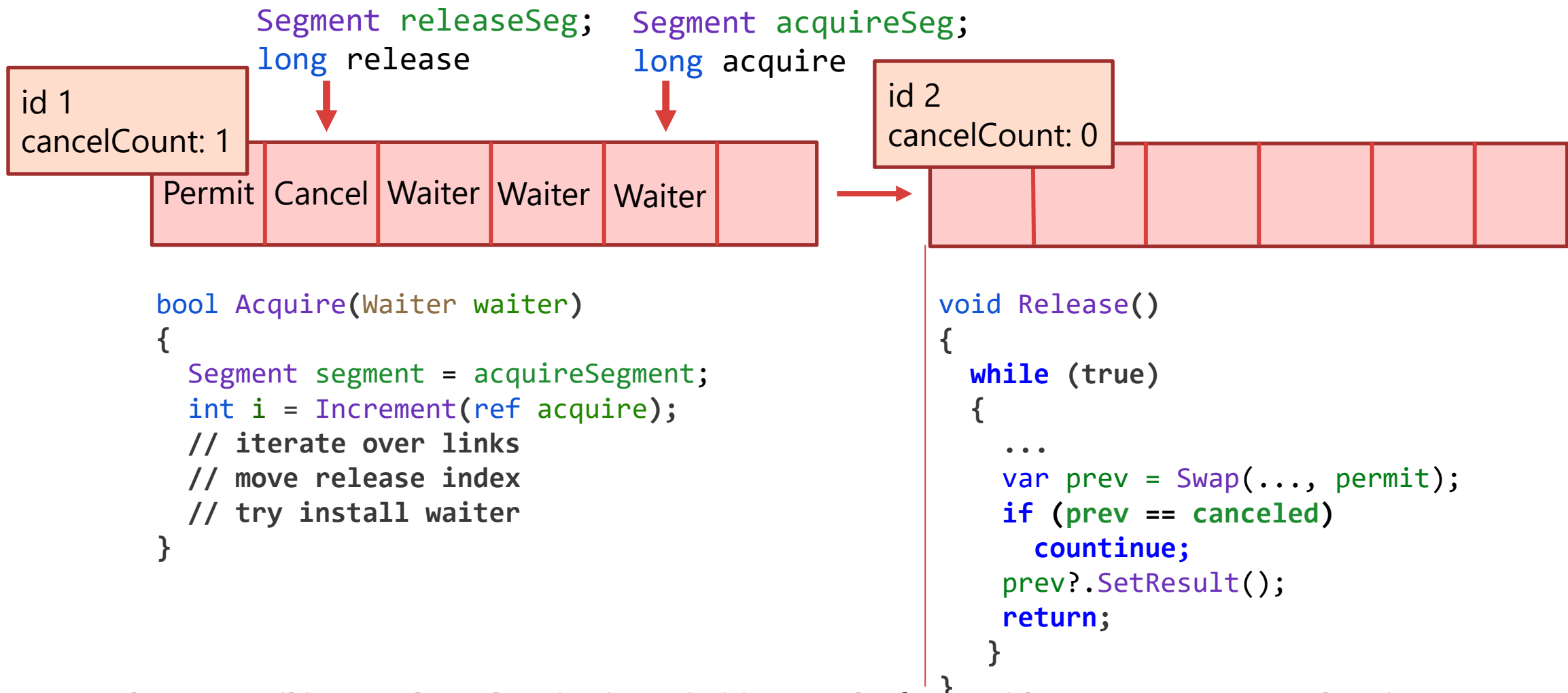
```
bool Acquire(Waiter waiter)
{
    Segment segment = acquireSegment;
    int i = Increment(ref acquire);
    // iterate over links
    // move release index
    // try install waiter
}
```

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

Hydra 2020. Nikita Koval Synchronization primitives can be faster with SegmentQueueSynchronizer

<https://www.youtube.com/watch?v=2uxsNJ0TdIM>

Segment queue synchronizer



Hydra 2020. Nikita Koval Synchronization primitives can be faster with SegmentQueueSynchronizer

<https://www.youtube.com/watch?v=2uxsNJ0TdIM>

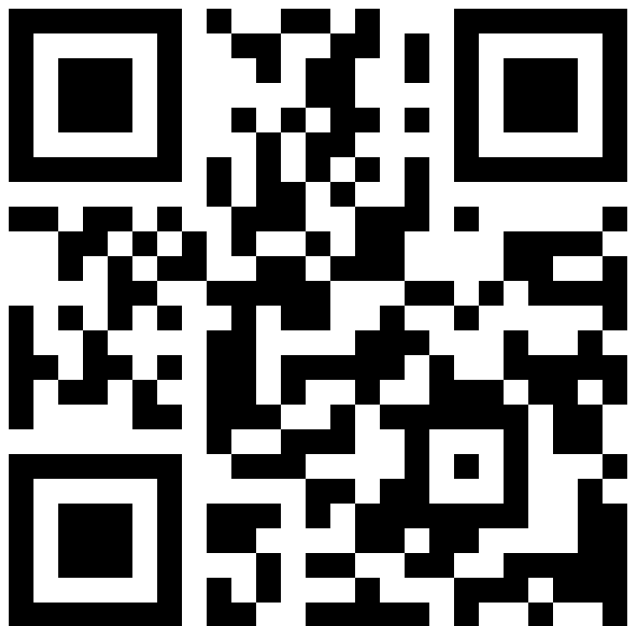
Lock-free: benchmark

Method	Permits	Threads	Mean	Contentions	Allocated
-----	-----	-----	-----:	-----:	-----:
NoSemaphore	16	16	44 ms	-	10 KB
SemaphoreSlim	16	16	134 ms	9452	10 KB
AnyLockFree	16	16	45 ms	0	10 KB
SemaphoreSlim	16	32	245 ms	10660	21645 KB
ConcurrentStack	16	32	88 ms	-	25250 KB
ConcurrentQueue	16	32	70 ms	0.001	22390 KB
VTConcurrentQueue	16	32	70 ms	0.001	24 KB
KSemaphore	16	32	74 ms	-	4740 KB
SimpleKSemaphore	16	32	65 ms	-	3920 KB

Выводы

- Не все новое лучше старого
- За асинхронным API может скрываться блокирующий код
- Иногда полезно заглядывать за пределы .NET-стека

Links



- <https://github.com/epeshk/dotnext-2022-semaphores/>
 - <https://github.com/epeshk/blog/>
- **Hydra 2020. Nikita Koval. SegmentQueueSynchronizer**
 - <https://www.youtube.com/watch?v=2uxsNJ0TdIM>
- **DotNext 2021 Autumn. Станислав Сидристый ThreadPool для сервиса, адаптирующегося под внешнюю нагрузку**
 - https://www.youtube.com/watch?v=EGDD-nB_EtI
- **[DotNext 2022 Spring. Станислав Сидристый — Тонкие настройки стандартного ThreadPool**
 - <https://www.youtube.com/watch?v=zeWhoFWGWKo>