

Примитивы синхронизации

Юрий Литвинов
yurii.litvinov@gmail.com

08.09.2020г

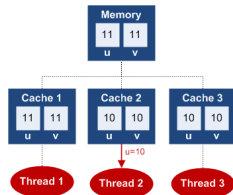
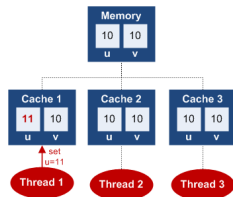
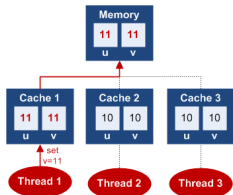
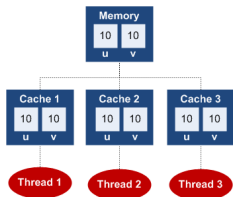
Примитивы синхронизации

- ▶ Лучше необходимости синхронизации вообще избегать
- ▶ Бывают:
 - ▶ User-mode — атомарные операции, реализующиеся на процессоре и не требующие участия планировщика
 - ▶ Kernel-mode — примитивы, управляющие тем, как поток обрабатывается планировщиком
 - ▶ Более тяжеловесные и медленные (до 1000 раз по сравнению с “без синхронизации вообще”)
 - ▶ Позволяют синхронизировать даже разные процессы

Атомарные операции

- ▶ Чтения и записи следующих типов всегда атомарны: Boolean, Char, (S)Byte, (U)Int16, (U)Int32, (U)IntPtr, Single, ссылочные типы
- ▶ Для других типов (например, Int64) операции чтения и записи могут быть прерваны посередине!

Volatile и модель памяти



© <https://igoro.com/archive/volatile-keyword-in-c-memory-model-explained/>

Volatile в .NET

▶ Volatile

- ▶ Volatile.Write
- ▶ Volatile.Read
- ▶ Связано с понятием Memory Fence, требует синхронизации ядер
- ▶ Есть ключевое слово volatile: **private** volatile **int** flag = 0;
- ▶ Volatile.Write должен быть последней операцией записи, Volatile.Read — первой операцией чтения

Пример

```
private int flag = 0;  
private int value = 0;
```

```
public void Thread1() {  
    value = 5;  
    Volatile.Write(ref flag, 1);  
}
```

```
public void Thread2() {  
    if (Volatile.Read(ref flag) == 1)  
        Console.WriteLine(value);  
}
```

Ещё один способ прострелить себе ногу

```
bool stop = false;
```

```
var thread = new Thread(() => {  
    while (!stop)  
    {  
        Console.WriteLine("Goodbye, cruel world!");  
        Thread.Sleep(500);  
    }  
});
```

```
thread.Start();  
Thread.Sleep(2000);  
stop = true;  
thread.Join();  
Console.WriteLine("Done.");
```

Interlocked

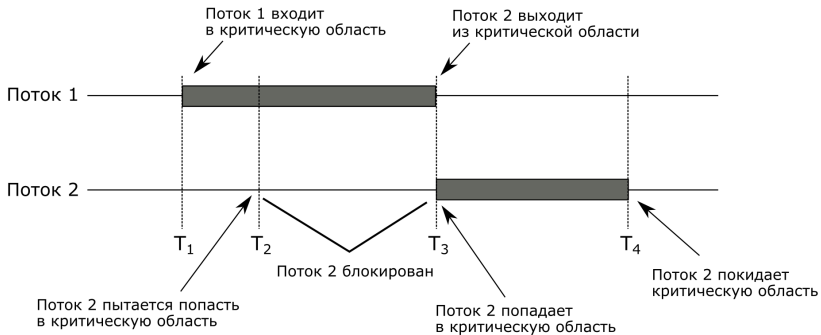
```
public int result;  
  
public void ThreadA()  
{  
    for (int i = 1; i <= 1000; i++)  
    {  
        result++;  
    }  
}
```

```
public void ThreadB()  
{  
    for (int i = 1; i <= 1000; i++)  
    {  
        result++;  
    }  
}
```

```
public int result;  
  
public void ThreadA()  
{  
    for (int i = 1; i <= 1000; i++)  
    {  
        Interlocked.Increment(ref result);  
    }  
}
```

```
public void ThreadB()  
{  
    for (int i = 1; i <= 1000; i++)  
    {  
        Interlocked.Increment(ref result);  
    }  
}
```


Критические области



Активное ожидание

```
private int turn = 0;
```

```
void Task1()
```

```
{
```

```
    while (true)
```

```
    {
```

```
        while (turn != 0) ;
```

```
        CriticalSection();
```

```
        turn = 1;
```

```
        NonCriticalSection();
```

```
    }
```

```
}
```

```
void Task2()
```

```
{
```

```
    while (true)
```

```
    {
```

```
        while (turn != 1) ;
```

```
        CriticalSection();
```

```
        turn = 0;
```

```
        NonCriticalSection();
```

```
    }
```

```
}
```

Активное ожидание, обсуждение

- ▶ Не требует поддержки ОС
 - ▶ Поэтому переключение может быть очень быстрым
- ▶ Ждущий поток полностью занимает ядро
 - ▶ Греет процессор и очень быстро сажает аккумулятор
- ▶ Потоки работают строго по очереди
 - ▶ Это можно побороть, есть алгоритм Петерсона

Проблема производителя и потребителя

Producer-consumer problem

```
private Queue<int> buffer = new Queue<int>();
```

```
private void Producer() {  
    while (true) {  
        var item = ProduceItem();  
        if (buffer.Count == 100)  
            Sleep();  
        buffer.Enqueue(item);  
        if (buffer.Count == 1)  
            WakeUp(Consumer);  
    }  
}
```

```
private void Consumer() {  
    while (true) {  
        if (buffer.Count == 0)  
            Sleep();  
        var item = buffer.Dequeue();  
        if (buffer.Count == 100 - 1)  
            WakeUp(Producer);  
        ConsumeItem(item);  
    }  
}
```

Семафоры

Дейкстры (того самого), 1965 год

- ▶ Целочисленный счётчик, который можно поднять и опустить (**up()** и **down()**)
- ▶ **down()** уменьшает счётчик на 1, если он больше нуля или блокирует вызывающего, если он 0
- ▶ **up()** увеличивает счётчик на один и, если он был нулём, будит одного из ожидающих потоков (случайного!)
- ▶ **down()** обычно делается при входе в критическую секцию, **up()** — при выходе
- ▶ Позволяет быть в критической секции не более чем заданному количеству потоков
 - ▶ Например, Google Drive не позволяет качать более чем с 10 подключениями одновременно, семафор решает проблему

Производитель-потребитель на семафорах

```
private Queue<int> buffer = new Queue<int>();  
private Semaphore mutex = new Semaphore(0, 1);  
private Semaphore empty = new Semaphore(100, 100);  
private Semaphore full = new Semaphore(0, 100);
```

```
private void Producer()
```

```
{  
    while (true)  
    {  
        var item = ProduceItem();  
        empty.WaitOne();  
        mutex.WaitOne();  
        buffer.Enqueue(item);  
        mutex.Release();  
        full.Release();  
    }  
}
```

```
private void Consumer()
```

```
{  
    while (true)  
    {  
        full.WaitOne();  
        mutex.WaitOne();  
        var item = buffer.Dequeue();  
        mutex.Release();  
        empty.Release();  
        ConsumeItem(item);  
    }  
}
```

Мьютекс

- ▶ Мьютекс — бинарный семафор
 - ▶ Пускает ровно один поток в критическую секцию
- ▶ Существенно проще в реализации и использовании, чем семафор
- ▶ Тоже требует поддержки операционной системы
 - ▶ Может использоваться для синхронизации даже процессов

Производитель-потребитель на семафорах и мьютексе

```
private Queue<int> buffer = new Queue<int>();  
private Mutex mutex = new Mutex();  
private Semaphore empty = new Semaphore(100, 100);  
private Semaphore full = new Semaphore(0, 100);
```

```
private void Producer()  
{  
    while (true)  
    {  
        var item = ProduceItem();  
        empty.WaitOne();  
        mutex.WaitOne();  
        buffer.Enqueue(item);  
        mutex.ReleaseMutex();  
        full.Release();  
    }  
}
```

```
private void Consumer()  
{  
    while (true)  
    {  
        full.WaitOne();  
        mutex.WaitOne();  
        var item = buffer.Dequeue();  
        mutex.ReleaseMutex();  
        empty.Release();  
        ConsumeItem(item);  
    }  
}
```


Монитор

Хоара, 1974 год

- ▶ Пользоваться семафорами очень сложно — например, поменять `empty.WaitOne();` и `mutex.WaitOne();` в `Producer()` — хороший способ устроить дедлок
 - ▶ Представим, что буфер полон. `Producer()` захватывает мьютекс и встаёт на семафоре `empty`, потому что он 0, управление передаётся `Consumer()`. Он тут же встаёт на `mutex.WaitOne()`, потому что он захвачен `Producer()`-ом. Теперь оба потока ждут друг друга.
- ▶ Поэтому придумали мониторы
- ▶ Монитор — набор методов (или функций), внутри которых может находиться ровно один поток
- ▶ Реализуется через мьютексы, требует поддержки в языке программирования

Производитель-потребитель на мониторе

```
private class SynchronizedQueue {
    private Queue<int> buffer =
        new Queue<int>();

    public void Enqueue(int item) {
        lock (buffer) {
            while (buffer.Count == 100)
                Monitor.Wait(buffer);
            buffer.Enqueue(item);
            Monitor.Pulse(buffer);
        }
    }

    public int Dequeue() {
        lock (buffer) {
            while (buffer.Count == 0)
                Monitor.Wait(buffer);
            var result = buffer.Dequeue();
            Monitor.Pulse(buffer);
            return result;
        }
    }
}
```

```
private SynchronizedQueue buffer =
    new SynchronizedQueue();

private void Producer() {
    while (true) {
        var item = ProduceItem();
        buffer.Enqueue(item);
    }
}

private void Consumer() {
    while (true) {
        var item = buffer.Dequeue();
        ConsumeItem(item);
    }
}
```

lock в .NET

- ▶ У каждого объекта (сылочного типа) есть скрытое поле, указывающее на структуру синхронизации
- ▶ lock использует именно её
 - ▶ То есть lock в одной критической секции, но на разные объекты — это разные мониторы
 - ▶ Но lock в разных секциях на один объект — один монитор
 - ▶ lock умеет обрабатывать исключения и отпускать замок
 - ▶ Предыдущие примеры с семафорами и мьютексами были неправильными — не учитывались исключения
- ▶ Хорошая практика — создавать объект специально для синхронизации, **lock(this)** писать нельзя!

```
private Object lockObject = new Object();
```

```
private void SomeMethod() {  
    lock (lockObject) {  
        ...  
    }  
}
```

WaitHandle

- ▶ WaitHandle — всё, что можно ожидать
 - ▶ EventWaitHandle
 - ▶ AutoResetEvent — по сути, булевый флаг, поддерживаемый ОС
 - ▶ ManualResetEvent — тоже булевый флаг, но сбрасывается вручную
- ▶ Остальные примитивы синхронизации — наследники WaitHandle

Пример (самодельный замок на Event-ax)

```
internal class SimpleWaitLock : IDisposable {  
    private readonly AutoResetEvent available;  
    public SimpleWaitLock() {  
        available = new AutoResetEvent(true);  
    }  
  
    public void Enter() {  
        available.WaitOne();  
    }  
  
    public void Leave() {  
        available.Set();  
    }  
  
    public void Dispose() { available.Dispose(); }  
}
```

Литература

Эндрю Таненбаум, Х. Бос, Современные операционные системы, Питер, 2017. 1120 С.



Jeffrey Richter, CLR via C# (4th Edition), Microsoft Press, 2012. 894pp.

