Синхронизация потоков — это краеугольный камень многопоточного программирования, и понимание её инструментов критически важно для создания стабильных и эффективных приложений.

Введение: Зачем нужна синхронизация?

Когда несколько потоков имеют одновременный доступ к общим данным или ресурсам, возникает хаос. Планировщик операционной системы в любой момент может переключить контекст с одного потока на другой, что приводит к непредсказуемому порядку выполнения операций.

Основные проблемы, которые решаем:

- **Race Condition (Состояние гонки):** Результат выполнения программы зависит от того, в каком порядке и когда выполняются потоки. Это приводит к недетерминированному и часто ошибочному поведению.
- **Data Corruption (Порча данных):** Когда один поток читает данные, которые другой поток находится в процессе изменения и ещё не закончил, это приводит к чтению "битых" или неконсистентных данных. Классический пример — инкремент разделяемой переменной (неатомарная операция counter++).
- **Deadlock (Взаимная блокировка):** Два или более потока бесконечно ожидают освобождения ресурсов, захваченных друг другом. Представьте, что Поток 1 ждёт ресурс A, удерживаемый Потоком 2, а Поток 2 в это же время ждёт ресурс B, удерживаемый Потоком 1. Оба повисли навечно.

Примитивы синхронизации в .NET

Давайте рассмотрим каждый из указанных примитивов, их особенности и применение.

1. lock (ключевое слово) / Monitor (класс) – критическая секция

Подход: Эксклюзивная блокировка (mutual exclusion). Особенности:

- Самый простой и часто используемый примитив.
- Ключевое слово lock является "синтаксическим сахаром" для пары методов Monitor.Enter и Monitor.Exit в блоке try-finally.

- Гарантирует, что только один поток может выполнить блок кода, защищённый lock, в любой момент времени.
- **ВАЖНО:** Объект, используемый для блокировки (syncRoot), должен быть **private, readonly и ссылочного типа (обычно object)**. Никогда не используйте this, строки или Type-объекты.

Способ работы: Поток пытается войти в критическую секцию. Если блокировка свободна — он захватывает её и выполняет код. Если занята — поток переходит в состояние ожидания до её освобождения.

Пример:

```
public class Counter
{
   private int _count = 0;
   private readonly object _padlock = new object(); // Объект-заглушка для блокировки

   public void Increment()
   {
      // Критическая секция. В один момент времени здесь может быть только один поток.
      lock (_padlock)
      {
         _count++; // Эта операция теперь потокобезопасна
      }
   }

   public int GetCount()
   {
      lock (_padlock)
      {
         return _count;
      }
   }
}
```

Эквивалент с использованием Monitor напрямую:

```
public void IncrementWithMonitor()
{
   bool lockTaken = false;
   try
```

```
    {
        Monitor.Enter(_syncRoot, ref lockTaken);
        _count++;
    }
    finally
    {
        if (lockTaken)
            Monitor.Exit(_syncRoot);
    }
}
```

2. Mutex

Подход: Межпроцессная эксклюзивная блокировка. Особенности:

- Как и lock, обеспечивает эксклюзивный доступ, но на уровне **операционной системы**.
- Может использоваться для синхронизации между разными процессами (приложениями), если мьютекс имеет глобальное имя.
- Более тяжеловесный, чем Monitor, поэтому для синхронизации внутри одного приложения предпочтительнее lock.

Способ работы: Работает аналогично lock, но требует явного вызова ReleaseMutex.

Пример (внутрипроцессный):

```
private static Mutex _mutex = new Mutex();

public void DoWork()
{
    _mutex.WaitOne(); // Захват мьютекса. Ожидаем, если он занят.
    try
    {
        // Критическая секция
        Console.WriteLine($"Поток
{Thread.CurrentThread.ManagedThreadId} в критической секции.");
        Thread.Sleep(1000);
    }
    finally
    {
```

```
        _mutex.ReleaseMutex(); // Освобождение мьютекса ОБЯЗАТЕЛЬНО
    }
}
```

Пример (межпроцессный):

```
// Создаём именованный мьютекс. Если он уже существует, получим ссылку на него.
using var mutex = new Mutex(false, "Global\\MyAppUniqueMutexName");

if (!mutex.WaitOne(TimeSpan.FromSeconds(5)))
{
    // Не удалось захватить мьютекс за 5 секунд (возможно, уже запущена другая копия приложения)
    Console.WriteLine("Другая копия приложения уже запущена!");
    return;
}

try
{
    // Основной код приложения. Только один экземпляр будет выполнять его.
    Console.WriteLine("Приложение работает. Нажмите Enter для выхода...");
    Console.ReadLine();
}
finally
{
    mutex.ReleaseMutex();
}
```

3. SemaphoreSlim

Подход: Ограничение количества одновременных доступов. Особенности:

- Позволяет ограничить не до одного, а до N потоков, которые могут одновременно работать с ресурсом.
- SemaphoreSlim — легковесная версия для использования внутри одного процесса. Имеет асинхронную поддержку (WaitAsync).

- Классический Semaphore тяжелее и может работать между процессами.

Способ работы: Счётчик. При создании указывается начальное и максимальное количество "разрешений". Поток вызывает Wait() (или WaitAsync()), чтобы получить одно разрешение (счётчик уменьшается). По окончании работы он вызывает Release(), чтобы вернуть разрешение (счётчик увеличивается). Если счётчик = 0, новые потоки ждут.

Пример (ограничение подключений к БД):

```
// Разрешаем не более 3 одновременных подключений
private static SemaphoreSlim _semaphore = new SemaphoreSlim(3);

public async Task AccessDatabaseAsync()
{
   await _semaphore.WaitAsync(); // Ждём, если уже 3 потока внутри
   try
   {
      // Имитация работы с БД
      Console.WriteLine($"Поток
{Thread.CurrentThread.ManagedThreadId} начал работу с БД.");
      await Task.Delay(2000);
      Console.WriteLine($"Поток
{Thread.CurrentThread.ManagedThreadId} закончил работу с БД.");
   }
   finally
   {
      _semaphore.Release(); // Освобождаем слот
   }
}
```

4. ManualResetEventSlim и AutoResetEvent

Подход: Уведомления между потоками (сигналы).

Особенности:

- **ManualResetEventSlim:** Как шлагбаум. Когда установлен в сигнальное состояние (Set()), он пропускает **все** ожидающие потоки. Чтобы остановить потоки, его нужно вручную сбросить (Reset()).

- **AutoResetEvent:** Как турникет. При каждом Set() пропускает **ровно один** ожидающий поток и автоматически сбрасывается. Не имеет "Slim"-версии, но она и не нужна, т.к. ManualResetEventSlim часто быстрее.

Способ работы: Потоки вызывают Wait() для ожидания сигнала. Другой поток устанавливает сигнал с помощью Set().

Пример (старт нескольких worker'ов по сигналу):

```
// Используем легковесный ManualResetEventSlim
private static ManualResetEventSlim _startEvent = new ManualResetEventSlim(false);

public static void Main()
{
   for (int i = 0; i < 5; i++)
   {
      new Thread(Worker).Start();
   }

   Console.WriteLine("Нажмите Enter, чтобы разрешить воркерам начать работу...");
   Console.ReadLine();

   _startEvent.Set(); // Отправляем сигнал всем воркерам одновременно

   Console.ReadLine();
}

private static void Worker()
{
   Console.WriteLine($"Воркер {Thread.CurrentThread.ManagedThreadId} ждёт разрешения...");
   _startEvent.Wait(); // Ожидаем сигнала

   // Как только Main() вызовет Set(), все ждущие воркеры продолжат работу.
   Console.WriteLine($"Воркер {Thread.CurrentThread.ManagedThreadId} начал работу!");
}
```

5. ReaderWriterLockSlim

Подход: Множественное чтение, эксклюзивная запись. Особенности:

- Оптимизация для сценариев, где чтение происходит часто, а запись — редко.
- Позволяет нескольким потокам **читать** данные одновременно.
- Но когда поток хочет **записать**, он получает эксклюзивный доступ: все читатели и другие писатели блокируются.

Способ работы: Потоки-читатели используют EnterReadLock()/ExitReadLock(), а потоки-писатели — EnterWriteLock()/ExitWriteLock().

Пример (потокобезопасный кэш):

```
public class ThreadSafeCache<TKey, TValue>
{
   private readonly Dictionary<TKey, TValue> _cache = new
Dictionary<TKey, TValue>();
   private readonly ReaderWriterLockSlim _lock = new
ReaderWriterLockSlim();

   public bool TryGetValue(TKey key, out TValue value)
   {
      _lock.EnterReadLock(); // Захватываем блокировку для чтения
      try
      {
         return _cache.TryGetValue(key, out value);
      }
      finally
      {
         _lock.ExitReadLock(); // Освобождаем блокировку для чтения
      }
   }

   public void AddOrUpdate(TKey key, TValue value)
   {
      _lock.EnterWriteLock(); // Захватываем ЭКСКЛЮЗИВНУЮ
блокировку для записи
      try
```

```
        {
            _cache[key] = value;
        }
        finally
        {
            _lock.ExitWriteLock(); // Освобождаем блокировку для записи
        }
    }
}
```

Важные правила и лучшие практики (ещё раз, потому что это ОЧЕНЬ важно)

1. **Минимизируйте объём кода в критических секциях.** Чем дольше вы держите блокировку, тем больше потоков ждут, тем ниже производительность. Выносите всё, что можно, за пределы lock.
2. **Используйте неизменяемые (immutable) структуры данных.** Если данные не могут быть изменены после создания, то не нужно и синхронизировать к ним доступ. Это самый эффективный способ избежать проблем.
3. **Никогда не блокируйте поток в async методе (используйте await).** Вместо lock или Wait() внутри асинхронных методов используйте асинхронные примитивы, такие как SemaphoreSlim.WaitAsync(). Блокировка потока в асинхронном контексте может привести к взаимоблокировкам (deadlock) и уничтожению производительности.
4. **Всегда освобождайте блокировки в finally блоке.** Это гарантирует, что блокировка будет снята даже в случае исключения.
5. **Избегайте вложенных блокировок.** Захват нескольких блокировок в непредсказуемом порядке — прямая дорога к Deadlock. Если это необходимо, строго контролируйте порядок их захвата.

Надеюсь, этот подробный разбор поможет вам уверенно использовать инструменты синхронизации в ваших .NET приложениях. Это та база, без которой многопоточное программирование превращается в охоту на призрачные баги