ОБРАЗОВАТЕЛЬНЫЙ ЦЕНТР
ПРОГРАММИРОВАНИЯ И ВЫСОКИХ ТЕХНОЛОГИЙ

2020г.

Основы многопоточного программирования

AGENDA

- ✓ Многопоточность
- ✓ Синхронизация потоков
- ✓ Понятие и виды блокировок
- ✓ TPL
- ✓ Parallel
- ✓ Специальные типы потоко-безопасных коллекций
- ✓ Отладка многопоточного кода

Многопоточность

Большинству приложений приходится иметь дело с более чем одной активностью, происходящей одновременно (параллелизм).

Общий механизм, с помощью которого приложение может выполнять код одновременно, называется многопоточностью. Многопоточность поддерживается как средой **CLR**, так и операционной системой (ОС), и в рамках параллелизма является фундаментальной концепцией.

Многопоточность – свойство платформы (в том числе и операционной системы) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «**параллельно**».

Такие потоки называют также потоками выполнения (от англ. **thread of execution**); иногда называют «**нитями**» (буквальный перевод англ. **thread**).

Процесс по отношению к потокам является контейнером, предоставляет им закрытое адресное пространство и следит за соблюдением политики операционной системы.

Многопоточность

Каждый поток запускается внутри процесса, который предоставляет изолированную среду для выполнения приложения. В однопоточном приложении внутри изолированной среды процесса функционирует только один поток, поэтому он получает монопольный доступ к среде. В многопоточном приложении внутри единственного процесса запускается множество потоков, разделяя одну и ту же среду выполнения (к примеру, память). Такие данные называются разделенныйм состоянием.

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен **System.Threading**. В нем определен класс, представляющий отдельный поток – класс **Thread**.

По умолчанию потоки, создаваемые явно, являются потоками переднего плана. Потоки переднего плана удерживают приложение в активном состоянии до тех пор, пока хотя бы один из них выполняется, в то время как фоновые потоки этого не делают. Как только все потоки переднего плана завершают свою работу, завершается и приложение, а любые все еще выполняющиеся фоновые потоки будут принудительно завершены.

Многопоточность

Имя элемента	Описания
Thread()	Конструктор с перегрузками для создания потока.
Abort()	Уничтожение потока.
Suspend()	Приостановление работы потока на неопределенное время.
Sleep()	Приостановление работы потока на заданное время.
Join()	Ожидание завершения работы потока.
CurrentThread	Свойство для извлечения текущего работающего потока.
CurrentThread.Name	Имя текущего потока.
IsBackground	Устанавливает работу потока в фоновом режиме.
IsAlive	Возвращает значение позволяющее определить, исполняется ли в данный момент поток.
ThreadState	Возвращает состояние текущего потока.
Interrupt()	Прерывает исполнение потока.
Resume()	Возобновляет работу ранее приостановленного потока.
Start()	Запускает поток на исполнение.

Синхронизация потоков

При построении многопоточного приложения необходимо учитывать, что любая часть разделяемых данных должна быть защищена от возможности изменения их значений множеством потоков.

Для решения подобных проблем в C# используется **синхронизация**. В её основе лежит понятие блокировки, посредством которой организуется управление доступом к кодовому блоку в объекте.

Простые методы блокировки потока:

- Sleep;
- Join.

Объекты синхронизации – служат для упорядочивания доступа потоков к ресурсу:

- Lock;
- Monitor;
- EventWaitHandle;
- Mutex;
- Semaphore.

Виды блокировок

Оператор **lock** определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. Это позволяет синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком.

```
object obj = new object();
Thread myThread = new Thread(Method);
…

Method()
{
  lock (obj)
  {
   //Блок кода
  }
}
```

Виды блокировок

Метод **Monitor.Enter** принимает два параметра - объект блокировки и значение типа **bool**. Этот метод блокирует объект **locker** так же, как это делает оператор **lock**. С помощью метода **Monitor.Exit** происходит освобождение объекта **locker**, и он становится доступным для других потоков.

```
object obj = new object();
Thread myThread = new Thread(Method);
…

Method()
{
 bool acquiredLock = false;
 Monitor.Enter(locker, ref acquiredLock);
  // Блок кода
 if(acquiredLock) Monitor.Exit(locker);
}
```

Виды блокировок

Класс **EventWaitHandle** также служит целям синхронизации потоков. Этот класс является оберткой над объектом **ОС Windows «событие»** и позволяет переключить данный объект-событие из сигнального в несигнальное состояние.

```
EventWaitHandle waitHandler = new AutoResetEvent(true);
Thread myThread = new Thread(Method);
…

Method()
{
 waitHandler.WaitOne();
  // Блок кода
 waitHandler.Set();
}
```

Виды блокировок

Mutex – класс-оболочка над соответствующим объектом **ОС Windows «мьютекс».**

Метод **mutexObj.WaitOne()** – приостанавливает выполнение потока.

Метод **mutexObj.ReleaseMutex()** – освобождает мьютекс, после выполнения всех действий.

Когда выполнение кода дойдет до вызова **mutexObj.WaitOne(),** поток будет ожидать, пока не освободится мьютекс.

```
static Mutex mutexObj = new Mutex();
Thread myThread = new Thread(Method);
…
Method()
{
 mutexObj.WaitOne();
 // Блок кода
 mutexObj.ReleaseMutex();
}
```

Виды блокировок

Семафор подобен мьютексу, однако он предоставляет одновременный доступ к общему ресурсу не одному, а нескольким потокам. Конструктор семафора принимает два параметра: изначальное число объектов семафора, второй – максимальное число.

Метод **sem.WaitOne()** – приостанавливает выполнение потока в ожидании места в семафоре.

Метод **sem.Release()** – освобождает место в семафоре, после выполнения всех действий.

```
static Semaphore sem = new Semaphore(3, 3);
Thread myThread = new thread(Method);
…
Method()
{
     sem.WaitOne();
     // Блок кода
     sem.Release();
}
```

Пулл потоков

Класс **System.Threading.ThreadPool** обеспечивает приложение пулом рабочих потоков, управляемых системой, позволяя пользователю сосредоточиться на выполнении задач приложения, а не на управлении потоками. Если имеются небольшие задачи, которые требуют фоновой обработки, пул управляемых потоков — это самый простой способ воспользоваться преимуществами нескольких потоков.

Чтобы запросить поток из пула для обработки вызова метода, можно использовать метод **QueueUserWorkItem().**

- Пул потоков управляет потоками эффективно, уменьшая количество создаваемых, запускаемых и останавливаемых потоков.

- Используя пул потоков, можно сосредоточиться на решении задачи, а не на инфраструктуре потоков приложения.

Параллельное программирование

TPL (Task Parallel Library) – библиотека параллельных задач основной функционал которой располагается в пространстве имен **System.Threading.Tasks.** Данная библиотека позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер. Кроме того, упрощается сама работа по созданию новых потоков. Поэтому начиная с **.NET 4.0.** рекомендуется использовать именно **TPL** и ее классы для создания многопоточных приложений.

Библиотека **TPL** позволяет автоматически распределять нагрузку приложений между доступными процессорами в динамическом режиме, используя пул потоков **CLR**. Библиотека **TPL** занимается распределением работы, планированием потоков, управлением состоянием и прочими низкоуровневыми деталями. В результате появляется возможность максимизировать производительность приложений **.NET** не имея дела со сложностями прямой работы с потоками.

Параллельное программирование

В основе библиотеки **TPL** лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов **.NET** задача представлена специальным классом – классом **Task**, который находится в пространстве имен **System.Threading.Tasks**.

Элементарная единица исполнения инкапсулируется в **TPL** средствами класса **Task**, а не **Thread**. По сравнению с потоком **Task** является абстракцией более высокого уровня – она представляет собой параллельную операцию, которая может быть или не быть подкреплена потоком. А в классе **Thread** инкапсулируется поток исполнения. Разумеется, на системном уровне поток по-прежнему остается элементарной единицей исполнения, которую можно планировать средствами операционной системы. Но соответствие экземпляра объекта класса **Task** и потока исполнения не обязательно оказывается взаимно-однозначным.

Кроме того, исполнением задач управляет планировщик задач, который работает с пулом потоков. Это, например, означает, что несколько задач могут разделять один и тот же поток.

Класс Parallel

Класс **Parallel** также является частью **TPL** и предназначен для упрощения параллельного выполнения кода. **Parallel** имеет ряд методов, которые позволяют распараллелить задачи.

Метод **Invoke()** – позволяет параллельное выполнение задач, в качестве параметра принимает массив объектов **Action**. И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах.

Метод **Parallel.For()** – позволяет выполнять итерации цикла параллельно. Принимает следующие парамтры: первый параметр задает начальный индекс элемента в цикле, а второй параметр – конечный индекс. Третий параметр – делегат **Action** – указывает на метод, который будет выполняться в каждой итерации.

Метод **Parallel.ForEach()** – осуществляет итерацию по коллекции, реализующей интерфейс **IEnumerable**, подобно циклу **foreach**, только осуществляет параллельное выполнение перебора. Принимает следующие парамтры: где первый параметр представляет перебираемую коллекцию, а второй параметр – делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции.

Потокобезопасные коллекции

Параллельные(потокобезопасные) коллекции – поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен **System.Collections.Concurrent.** Данные коллекции могут безопасно использоваться в многопоточном приложении, где возможен одновременный доступ к коллекции со стороны двух или больше параллельно исполняемых потоков.

Использование коллекций:

- **Однопоточные сценарии** – только классические коллекции с лучшей производительностью.
- **Запись из множества пото**ков – только **concurrent** коллекции, защищающие внутреннее состояние и имеющие подходящее для конкурентной записи **API**.
- **Чтение из множества потоков** – однозначных рекомендаций нет. **Concurrent** коллекции, могут создать проблемы с производительностью при интенсивных запросах состояния всей коллекции. Однако для классических коллекций **Microsoft** не гарантирует работоспособность даже для операций чтения.
- **И чтение и запись из нескольких потоков** – однозначно **concurrent** коллекции, как реализующие защиту состояния и безопасное **API**.

Потокобезопасные коллекции

ConcurrentDictionary – потокобезопасная коллекция общего назначения.

ConcurrentBag, ConcurrentStack, ConcurrentQueue – коллекции специального назначения.

- Отсутствие доступа к произвольному элементу.
- **Stack** и **Queue** имеют заданный порядок добавления и извлечения элементов.
- **ConcurrentBag** для каждого потока поддерживает собственную коллекцию для добавления элементов. При извлечении он «крадет» элементы из соседнего потока, если у текущего потока коллекция пуста.

BlockingCollection – используется в сценариях, когда одни потоки заполняют коллекцию, а другие извлекают из нее элементы. Вызвав метод **CompleteAdding()** мы можем указать, что коллекция больше не будет пополняться, тогда при чтении не будет выполнятся ожидание нового элемента. Проверить состояние коллекции можно с помощью свойств **IsAddingCompleted** (**true** если данные больше не будут добавляться) и **IsCompleted** (**true** если данные больше не будут добавляться и коллекция пуста).

Отладка многопоточного кода

В Visual Studio доступны различные средства для отладки многопоточных приложений.

- Основные средства для отладки потоков – это окно **Threads** , маркеры потоков в окнах исходного кода, окно **Parallel Stacks**, окно **Parallel Watch** и панель инструментов **Debug Location**.
- Для кода, использующего библиотеку параллельных задач **(TPL)** или среду выполнения с параллелизмом, основными средствами отладки являются окно **Parallel Stacks**, окно **Parallel Watch** и окно **Tasks.**
- Основным средством отладки потоков в GPU является окно Потоки GPU.
- Основные средства для работы с процессами – это диалоговое окно Attach to Process, окно **Processes** и панель инструментов **Debug Location**.

Visual Studio также предоставляет эффективные средства для работы с точками останова и трассировки, что может быть полезно при отладке многопоточных приложений. Используйте условия и фильтры точек останова для установки точек останова в отдельных потоках. Точки трассировки дают возможность отслеживать выполнение программы без прерывания ее выполнения. Это может быть полезно для изучения неполадок, таких как взаимоблокировки.