Multithreading

# Понятие нити.

Нить – форма эмуляции CPU, когда код выполняется «параллельно» от другого кода. Нити существуют в самой операционной системе Windows, а нить CLR является оберткой для нити ОС.

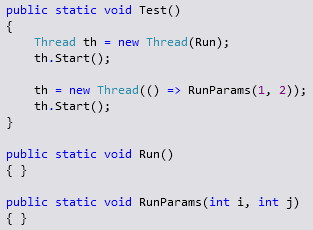
Нити не выполняются «параллельно». ОС может выполнять одновременно нитей не больше количества процессоров или ядер в нем. Для того, чтобы эмулировать параллельную работу нитей, ОС постоянно производит «переключение контекста», когда в каждый момент работает одна нить, а остальные находятся в режиме ожидания. По истечению определенного промежутка времени активная нить сменяется другой.

Переключение контекста - очень ресурсоемкая операция, а для поддержания «параллельности» эту операцию необходимо производить очень часто. Поэтому большое количество одновременно работающих нитей сильно сказываются на производительности всей системы в целом.

С каждой нитью, кроме того, связано очень большое количество других ресурсов, прежде всего памяти. Для каждой нити выделяется память под стек. В Windows размер стека изначально небольшой, он увеличивается по мере необходимости до максимума 2 Мб. В CLR стек сразу создается с максимальным размером в 2 Мб. Кроме этого, выделяется память и под другие данные, в частности, под контекст нити.

Таким образом, нити очень ресурсоемки. Не стоит создавать их в излишнем количестве.

# Синтаксис.



В данном примере создается объект типа *Thread*, которому в качестве параметра передается функция, которую нужно запустить в отдельной нити. Метод *Start* запускает новую нить, и функция *Run* выполнится уже в отдельной нити.

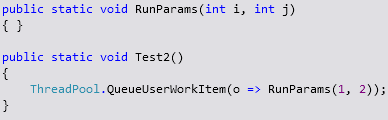
Во 2м примере используется «лямбда-синтаксис» для передачи функции в конструктор нити. С помощью такого синтаксиса можно передать параметры функции, которая должна выполниться в отдельной нити.

При работе с разными нитями нужно иметь в виду, что у каждой нити есть свой стек, то есть все локальные переменные в функциях принадлежат только одной нити, но управляемая куча доступна всем нитям одновременно.

# ThreadPool.

Его основная цель – не создавать нить в каждом случае, когда нужно выполнить операцию в отдельной нити, а использовать пул существующих нитей. При таком подходе не тратится время на инициализацию нитей и их уничтожение.

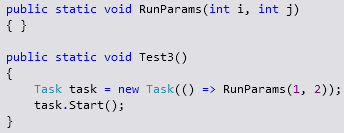
Второй плюс заключается в том, что оптимизируется работа с одновременным запуском большого числа нитей. Вместо того, чтобы сразу запускать много нитей под разные параллельные задачи, эти задачи ставятся в очередь на выполнение, а количество одновременно работающих нитей зависит от количества процессоров и/или ядер, а также некоторых других факторов. Например, долговременная операция, запущенная в выделенной из пула нити не будет блокировать остальные операции. Будет создана дополнительная нить, чтобы задачи не простаивали в очереди слишком долго.



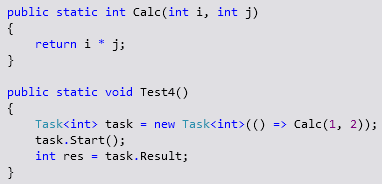
В данном примере вызов функции *RunParams* ставится в очередь в объекте *ThreadPool*. Этот объект сам решает, когда лучше всего выполнить эту задачу в отдельной нити.

# Задачи (Tasks).

Ключевым недостатком такого подхода является то, что невозможно управлять такой задачей. Например, нельзя выяснить, что задача выполнилась. Именно для этого в .NET 4.0 вводятся *Task*. Этот класс управляет задачей, которая ставится в очередь в *ThreadPool*.



Здесь вызов функции *RunParams* также ставится в очередь, однако мы можем управлять этой задачей. Можно узнать, выполнена ли она, подождать ее выполнения. Обратите внимание, что *task.Start* не запускает задачу, а только ставит ее в очередь.

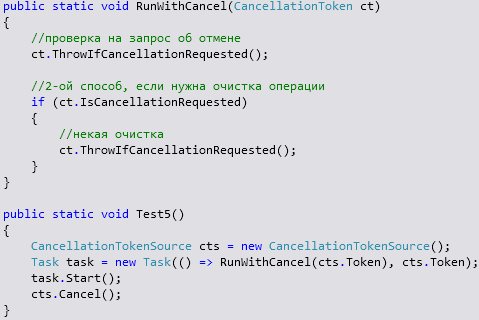


В данном примере использует класс-generic, который используется для функций, которые возвращают результат. Здесь мы в отдельной нити считаем результат функции *Calc*. Когда результат выполнения функции нам понадобится, мы запросим этот результат у объекта класса *Task*. При этом, если нить не закончила свою операцию, нить, вызвавшая задачу, будет ожидать результата.

# Отмена задачи.

Очень часто после старта некоторой задачи в отдельной нити, ее результат перестает быть нужен. В этом случае ее лучше отменить, чтобы эта задача не занимала лишние ресурсы.

В .NET 4.0 разработана схема для более упорядоченного механизма отмены задачи. Эта модель построена на простом объекте, называемом токеном отмены (*CancellationToken*). Объект, который вызывает отменяемую операцию, передает этот токен в операцию. Операция, в свою очередь, передает копии этого токена в другие операции. Некоторое время спустя объект, создавший токен, может использовать его для запроса остановки выполнения операции. Запрос на отмену может создавать только запрашивающий объект, и каждый прослушиватель должен обнаружить этот запрос и своевременно ответить на него.



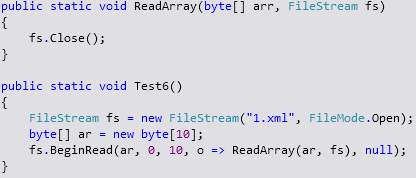
В данном примере создается объект класса *CancellationTokenSource*, который и может вызвать отмену операции. Он передает токен в запрашиваемую задачу. В самой задаче на каждом этапе необходимо проверять, что вызвана отмена операции.

# Синхронные IO операции.

При выполнении синхронные IO-операций управление передается устройству, которое должно выполнить эту операцию. Устройство – это мини-компьютер, который сам выполняет операцию. Соответственно, нить, вызвавшая операцию переводится в режим sleep. Она не тратит время процессора, однако она продолжает использовать память и прочее. Кроме того, поскольку нить остается используемой, она не может вернуться в пул нитей. Если будет запрошена еще одна асинхронная операция, то пул нитей будет вынужден создать еще одну нить. Если же первая нить оживет раньше завершения второй, то они обе будут активно работать, снижая производительность системы из-за частого переключения контекста.

# Модель асинхронных IO операций.

Такая ситуация – когда часто создаются нити – характерна для серверных приложений, поэтому рекомендуется использовать асинхронные вызовы для IO-операций.



В данном примере чтение из файла происходит асинхронно. Нить завершается, а выполнение задачи переносится на другую функцию – *ReadArray*. Такая модель очень громоздка, приходится все состояние операции переносить в параметрах функций и правильно завершать каждую нить.

# Синхронизация.

# Основы.

При работе нескольких нитей, необходимо следить за тем, какие объекты могут использоваться из разных нитей одновременно. Если такие объекты используются только для чтения, то здесь проблемы нет. Если же одна из нитей пишет в объекты, то необходима синхронизация работы нитей, чтобы объекты не использовались в своем изменяемом, а значит, некорректном состоянии.

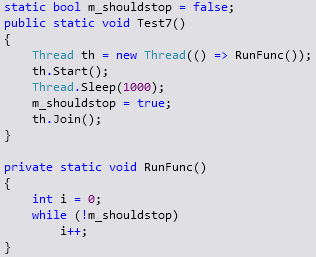
Для синхронизации используется блокировка нити с помощью какого-нибудь блокировщика. В начале критической секции необходимо поставить лок, а в ее конце – освобождение. Начало критической секции может пройти только одна нить. Все остальные будут останавливаться в ее начале, пока использующая критическую секцию нить не покинет ее. Блокировщики бывают разные, но даже самые быстрые нередко замедляют работу секции в разы. Кроме того, сама блокировка означает, что нить перестает работает и переходит в спящий режим. Что может привести к созданию дополнительных нитей, если понадобится выполнить асинхронную задачу. Поэтому необходимо на архитектурном уровне стараться создавать такое приложение, где использование общих объектов сведено к минимуму.

Блокировщики делятся на 2 главных типа – User-mod и Kernel-mod. Блокировки первого типа осуществляются на уровне приложения. В этом случае нить не прекращает свое выполнение, а переходит в активное ожидание, используя CPU. Блокировки второго типа осуществляются самой операционной системой. В этом случае заблокированная нить переходит в спящий режим.

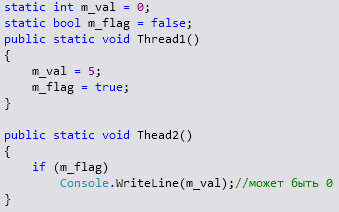
# Volatile блокировки.

Это блокировка User-mod типа.

Приведу 2 примера, где такие блокировки следует использовать.



Здесь приведен типичный способ сказать нити, что пора завершиться – путем изменения статичной переменной. Однако у такого механизма есть довольно неожиданный противник – JIT-оптимизатор. Дело в том, что он, видя, что переменная *m\_shouldstop* не меняется внутри цикла, в оптимизированном коде проверяет ее значение только в начале цикла. Таким образом, в оптимизированном варианте такая программа будет выполняться без остановки.



Здесь 2 функции, выполняющиеся в разных нитях. Кажется очевидным, что если *m\_flag* изменился, то *m\_val* тоже уже изменился. Однако, это не всегда может быть так. Причем, проблема с очередностью возможна в обеих функциях.

Прежде всего, оптимизатор снова мог оптимизировать работу функции так, что изменился порядок выполнения операторов. Действительно, если бы функция работала в одной нити, то нет разницы, какой из этих операторов выполнится раньше. И, если для оптимизации выгоднее раньше выполнить изменение *m\_flag*, то оптимизатор так и сделает.

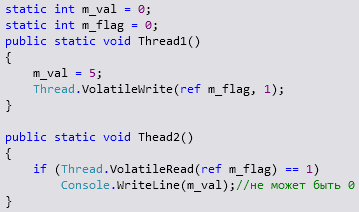
С другой стороны, даже если эти 2 поля изменились в нужном порядке, все равно есть возможность, что в консоль попадет 0. Дело в том, что эти 2 поля – всего лишь обертка. При работе внутри CPU используются регистры и прочее. И может так произойти, что на консоль может уйти неправильное значение регистра.

Для решения таких проблем используется Volatile блокировка, которая в C# реализована 3 статичными функциями класса *Thread*:

* *VolatileWrite(ref byte address, byte value)*;
* *byte VolatileRead(ref byte address);*
* *void MemoryBarrier();*

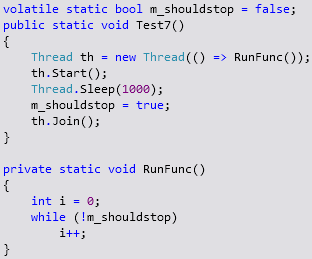
Первая функция пишет в *address* значение, при этом все операции записи, которые в коде находятся раньше, выполнялись. Вторая функция считывает из *address* значение и все операции чтения, которые в коде находятся позже, выполнятся после этого. Третья функция создает «барьер» в записи, комбинируя 2 первые функции. Но используется реже.

Используя блокировку, можно исправить 2 пример, чтобы он всегда работал корректно.



С помощью *VolatileWrite* мы гарантируем, что в первой функции запись полей будет произведена в правильном порядке. С помощью *VolatileRead* мы гарантируем, что во второй функции *m\_flag* будет прочитан раньше *m\_val*.

C# предлагает особую конструкцию, позволяющую любое чтение/запись какого-либо поля Volatile.

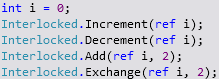


Здесь мы защищаем поле *m\_shouldstop*, и оно будет считываться правильно.

Суммируем правило использования данной блокировки. Последнее изменение в статичных полях, которые могут использоваться в разных нитях, необходимо блокировать с помощью *VolatileWrite*. Первое чтение в таких полях необходимо блокировать с помощью *VolatileRead*.

# Interlocked блокировки.

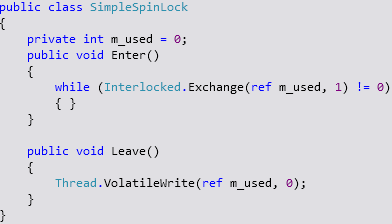
Данная блокировка используется для атомарного одновременно чтения и записи в разделяемый объект. Для этого используется класс Interlocked.



Здесь показаны примеры статичных функций, которые используют данную блокировку. В каждой из этих функций сначала считывается значение переменной и потом записывается в нее новое значение. Эта пара операций (чтений и запись) атомарная. Главным недостатком является то, что большинство таких методов работают только с *Int32*.

# «Активная» блокировка.

Используя предыдущую блокировку, можно реализовать простую блокировку критической секции:



В данном примере нить, входящая в критическую секцию, изменит *m\_used* на 1. Если до изменения значение этого поля 0, то выйдет за пределы цикла и войдет в критическую секцию. Если до того как 1-ая нить покинет критическую секцию, 2-ая нить тоже попробует туда зайти, то она попадет в цикл, постоянно меняя 1 на 1. Когда 1-ая нить выйдет из критической секции, то она поменяет поле на 0, и 2-ая нить закончит цикл. Если в таком цикле находилось несколько нитей, то выйдет из него только одна, поскольку операция атомарная, и только одна нить увидит 0 и тут же изменит его на 1.

Главной особенностью такой блокировки является то, что заблокированная нить продолжает работать, активно используя CPU. Такой блокировкой можно пользоваться, если критическая секция выполняется очень быстро. В .NET есть класс *SpinLock*, который работает примерно так же.

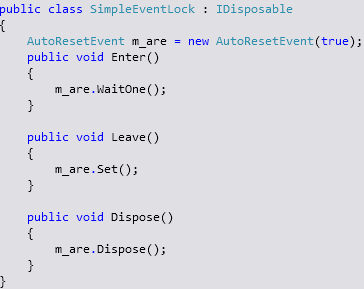
# Kernel-mode блокировки.

Данные блокировки используют ядро операционной системы. Все такие блокировки значительно медленнее user-mode блокировок, особенно если сами блокировки не происходят. В то же время позволяют переводить заблокированные нити в спящий режим, а также осуществлять блокировки, недоступные для user-mode, например, блокировки между управляемым и неуправляемым кодом.

Все Kernel-mode блокировки наследуются от класса *WaitHandle*, который внутри содержит handle на системный объект, отвечающий за блокировки.

# Event блокировки.

Данная блокировка основана на работе с булевой переменной. Блокирует нить, когда значение переменной *false*, и разблокирует нить, если *true*.

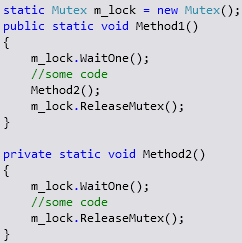


Данный класс аналогичен по внешнему функционалу классу *SimpleSpinLock*, но работает значительно медленнее. Для случаев, когда блокировка не понадобится данный код работает в 80 раз медленнее.

# Semaphore блокировки.

Данная блокировка основана на внутреннем *int* поле. Если значение 0, то нить разблокируется, иначе блокируется. Главным отличием является очередность нитей. То есть раньше разблокируется та нить, которая раньше была заблокирована.

# Mutex блокировки.



Mutex – готовый класс блокировок. Он позволяет рекурсивно ставить критические секции без опасности быть самозаблокированным.