Пояснительная записка

# Дополнительные ограничения

* Я принципиально не пользовался стандартным ThreadPool (и никакими другими объектами, косвенно его использующими: ни Task ни Timer).
* Уточнение к задаче: количество задач в пуле – это количество задач, которые находятся в очереди и ожидают своего выполнения. Выполняющиеся на текущий момент задачи в это число не входят.
* Уточнение к задаче: «ждет завершения обработки всех ранее добавленных задач» означает, что если задача была добавлена, принята в обработку рабочим потоком, этот поток попросили остановиться (балансировка), начали высвобождать пул, то мы должны дождаться завершения той задачи.
* Я не делал захвата ExecutionContext, поэтому моя реализация получилась аналогом ThreadPool.UnsafeQueueUserWorkItem. Причины и последствия – описаны ниже.
* Абсолютное большинство классов и internal. Некоторые из них – по причине незащищённости (например, ThreadCollection), многие – ради идейной чистоты.
* Принятие решений по удалению (и добавлению) рабочих потоков я постарался сделать отдельно стоящим классом (PerformanceBalancer). Прошу не оценивать это решение критически, потому как есть 3 «но». Но 1) само это решение сделано максимально дёшево. Но 2) класс ThreadPool держит ссылку на балансировщик, что бы высвободить его при своём высвобождении – тоже исключительно ради дешевизны. Но 3) сбор метрик, на основании которых балансер принимает решение, так же сделан очень дёшево.

# Пояснения к реализации

## Инстанцирование пула

Предусмотрена фабрика SimpleThreadPoolCreator, и структуры с настройками PerformanceBalanceSettings и SimpleThreadPoolSettings. Настроек получилось очень много – я старался обойтись без констант (не везде удалось).

## Идея решения

Я строил решение вокруг BlockingCollection – его функционал на 90% удовлетворяет нашим задачам. Он даёт возможности кооперативной отмены ожидающих потоков, блокировка читающего процесса и много другое.

Стандартный ThreadPool сильно интегрирован с VM: набором рабочих потоков управляет сама VM, а ThreadPool даёт ей команды. Рабочий поток просыпается по сигналу от ThreadPool, который даётся при добавлении задачи в очередь. Добавление потоков происходит по вызову ThreadPool.RequestWorkerThread. Когда совершать этот вызов - определяется эвристикой, зашитой в самом ThreadPool (если в крадце - количество элементов в очереди больше количества процессоров). Удаление скучающих потоков делает сама VM – не известно какими способами.

У меня не было задачи гоняться со стандартным решением, поэтому использования BlockingCollection будет вполне достаточно.

Удаление и добавление потоков (хотя второе не требовалось) я старался сделать отдельным объектом – об этом написано выше. Конечно это дополнительная точка отказа, но это позволяет делать решение чуть более гибким.

## Архитектура пула

Использовать Task нельзя т.к. он будет работать на стандартном ThreadPool, что противоречит задаче.

Стандартный ThreadPool устроен так. Там каждый поток обращается в ThreadPool.Dispatch и выбирает себе работу. Поток активируется из VM, а при добавлении задачи мы сообщаем в VM что нужно пнуть какой-нибудь поток. Про изменение количества потоков – см. выше.

В методе Dispatch каждый из потоков выбирает задачи сначала из своей очереди, потом пытается "своровать" из общей очереди, потом пытается "своровать" из очереди соседа.

**Варианты алгоритма 1**. По аналогии со стандартным ThreadPool, при попадании элемента в WorkingQueue мы уведомляем наблюдателя, что появилась работа (эту работу берёт на себя BlockingCollection). Один из потоков просыпается и направляется в метод ThreadPool.Dispatch. После завершения текущей задачи опять повисает в ожидании события.

Поддерживаем кооперативную и одиночную отмену.

При этом каждый поток не знает об устройстве очереди, но знает о своём хозяине - ThreadPool-е. Ответственность выбора следующего элемента ложится на THreadPool.Dispatch.

**Вариант алгоритма 2**. Пользуемся возможностью BlockingCollection.GetConsumingEnumerator. В каждый Thread передаём IEnumerable, и поток крутится в вечном foreach. Если работа в колекции заканчивается - поток просто подвисает внутри MoveNext().

Поддерживаем кооперативную и одиночную отмену (это позволяет BlockingCollection).

В этом случае поток не знает о хозяине, явно не использует объектов синхронизации, но получает косвенную ответственность - выбирать следующий элемент для обработки.

**Я выбрал вариант 1**, хотя второй тоже имеет право на существование.

## Красота кода

У меня здесь нет решарпера и нет stylecop. Я очень старался писать чисто, добавлять комментарии и т.п. Надеюсь, читаемость будет приемлемая.

## SimpleThreadPool

Является фасадом для колекции задач, колекции потоков, диспатчера задач и балансера производительности.

Использует особенности BlockingCollection для реализации конкурентного потребления из очереди - это кастомизировать нельзя.

PerformanceBalancer умеет автоматически изменять количество рабочих потоков.

Использует ThreadCollection для управления потоками.

В методе Dispose вызывает ThreadCollection.Dispose, который дожидается выполнения всех когда-либо добавленных задач, что может вызвать performance hit. (Кстати идея – не очень)

## Рабочие потоки

Классы WorkingThread и PerformanceBalancerThread я старался сделать максимально автономными. Они наследуются от ThreadBase , который содержит деструктор (если даже мы потеряем ссылку на класс обёртку, то финализатор прикончит Thread). Экстренную отмену потока я сделал через Abort(). Я знаю, что это опасная штука.

Так же код, вызываемый из финализатора не должен бросать исключений – я дополнительно это проверил.

## Защита от ThreadAbortedException

Один из участков кода очень критичен к прерыванию – речь идёт о процедуре удаления потока. Так вот кусок кода от удаления потока из основной коллекции и добавления во вспомогательную не должен быть прерван (ThreadCollection.TryRemoveThread). Для этого я применил приём с try finally – runtime гарантирует что при возникновении TAE все блоки finally доработают.

## Ожидание выполнения всех задач

Задача решается в два этапа. Класс SimpleThreadPool завершает добавление элементов в очередь и высвобождает ThreadCollection, а тот в свою очередь ожидает окончания всех потоков. Делает он это методом Join c таймаутом, а если таймаут вышел – жестко обрывает поток методом Abort. Это в задаче не просили, но моя паранойя не позволяет мне делать Join без параметра. Вместо Join можно было бы использовать WaitHandle.WaitAll, но это потребовало бы введения дополнительных объектов.

## Удаление рабочего потока и потеря ссылки на него

При решении этой задачи особо остро ощутил нехватку стандартного ThreadPool. Мы должны попросить поток остановиться, а после того как он остановится – потерять на него ссылку. Наоборот делать нельзя – при высвобождении мы этот поток должны будем ожидать.

Нельзя использовать ThreadPool, поэтому приходится выкручиваться событием (Complete). Иначе я бы использовал RegisterSingleWait или простенький континьюейшен.

## Удаление и добавление рабочих потоков

Эта часть задачи увеличила скоуп работы примерно в 2 раза.

Балансировщик – это отдельно-работающий наблюдатель, который при помощи стратегий (внутри единственной стратегии – простая эвристика) и оперируя накапливаемыми метриками, даёт команду либо добавить, либо удалить рабочий поток. Извинения относительно архитектурной не-идеальности см. выше.

Само удаление и добавление потоков реализовано в ThreadCollection – эти операции требуют валидации – количество потоков не может быть больше определённого или меньше определённого. И только из-за этого приходится использовать объект синхронизации SemaphoreSlim(1) – он работает быстрее, чем lock.

Существует патерн, решающий эту задачу (валидация + действие) при помощи цикла, SpinWait и Interlocked.CompareExchange, но его реализация даст не большой профит (большой конкуренции не предполагается) а времени у меня уже не хватает. Пример идеи здесь - <https://msdn.microsoft.com/ru-ru/library/801kt583%28v=vs.110%29.aspx>

## Тесты

Нагрузочных тестов я сделать не успел. Юниттестами покрывал в первую очередь требования и ответственности основных классов.

## Почему я не сделал захват ExecutionContext

Вот пример кода, который пытался бы захватить контекст и коментарии о том, почему это не получиться

/// <summary>

/// Внимание! контекст синхронизации так же захватывается, но пользовательский обработчик будет работать в рабочем потоке.

/// </summary>

/// <remarks>

/// Класс называется по аналогии с internal sealed class QueueUserWorkItemCallback из System.Threading

/// вовсе не потому что я позаимствовал его оттуда.

/// Текущая задача чуть проще общего случая. Мне не нужно думать про unsafe вызовы (хотя это делается просто),

/// нужно работать с Action а не WaitHandle (не нужно передавать stateObj), поэтому я слегка переделал класс.</remarks>

internal class QueueUserWorkItemCallbackWithExecutionContextFlow : IThreadPoolWorkItem

{

/// <summary>

/// Этот метод может срабатывать в любом SecurityContext

/// </summary>

[System.Security.SecuritySafeCritical]

static QueueUserWorkItemCallbackWithExecutionContextFlow() { }

private Action callback;

private ExecutionContext context;

/// <summary>

/// Внутри происходит захват контекста

/// </summary>

[SecurityCritical]

internal QueueUserWorkItemCallbackWithExecutionContextFlow(Action waitCallback)

{

callback = waitCallback;

if (!ExecutionContext.IsFlowSuppressed())

{

// захват контекста. Вообще то там внутри отметка стека замеряется как StackCrawlMark.LookForMyCaller,

// и захватывается synchronizationContext, а это не совсем то, что нужно в продакшен коде.

// Если на synchronizationContext можно наплевать (при определённой осторожности), то

// отметка стека замеряемая внутри полностью лишает смысла захват SecurityContext! Ведь в этом случае,

// захватится SecurityContext для вот этого вот метода, а должен был бы - для кода который пытался положить

// задачу в threadPool.

context = ExecutionContext.Capture();

// правильный метод - разумеется internal:

// context = ExecutionContext.Capture(ref stackMark, ExecutionContext.CaptureOptions.IgnoreSyncCtx | ExecutionContext.CaptureOptions.OptimizeDefaultCase);

// Обещали в следующих версиях .net решить эту проблему. Сейчас можно было бы вызвать его через рефлексию.

}

}

[SecurityCritical]

void IThreadPoolWorkItem.ExecuteWorkItem()

{

if (context == null)

{

Action cb = callback;

callback = null;

cb();

}

else

{

ExecutionContext.Run(context, ccb, this);

}

}

[System.Security.SecurityCritical]

static internal ContextCallback ccb = new ContextCallback(WaitCallback\_Context);

[System.Security.SecurityCritical]

static private void WaitCallback\_Context(Object state)

{

QueueUserWorkItemCallbackWithExecutionContextFlow obj = (QueueUserWorkItemCallbackWithExecutionContextFlow)state;

Action wc = obj.callback as Action;

wc();

}

}