

К.А. Туральчук

Параллельное программирование с помощью языка C#



ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Параллельное программирование с помощью языка C#

2-е издание, исправленное

Туральчук К.А.

Национальный Открытый Университет "ИНТУИТ"

2016

Параллельное программирование с помощью языка C#/ К.А. Туральчук - М.: Национальный Открытый Университет "ИНТУИТ", 2016

Курс предназначен для практического введения в параллельное программирование (ПП) и знакомства с основными возможностями ПП на платформе .NET 4.0

Курс включает общие теоретические сведения по дисциплине параллельное программирование в объеме необходимом для реализации эффективных параллельных приложений. Большая часть материала посвящена практическому знакомству с возможностями библиотеки TPL (Task Parallel Library). Все разделы иллюстрируются практическими примерами. Все конструкции и средства TPL закрепляются выполнением лабораторных работ.

(c) ООО "ИНТУИТ.РУ", 2013-2016

(c) Туральчук К.А., 2013-2016

Введение в параллельные вычисления

Архитектура ВС. Классификация вычислительных систем. Пути достижения параллелизма. Параллелизм на уровне команд, потоков, приложений. Анализ эффективности параллельных вычислений. Закон Амдала.

Мотивы параллелизма

- Параллельность повышает производительность системы из-за более эффективного расходования системных ресурсов. Например, во время ожидания появления данных по сети, вычислительная система может использоваться для решения локальных задач.
- Параллельность повышает отзывчивость приложения. Если один поток занят расчетом или выполнением каких-то запросов, то другой поток может реагировать на действия пользователя.
- Параллельность облегчает реализацию многих приложений. Множество приложений типа "клиент-сервер", "производитель-потребитель" обладают внутренним параллелизмом. Последовательная реализация таких приложений более трудоемка, чем описание функциональности каждого участника по отдельности.

Классификация вычислительных систем

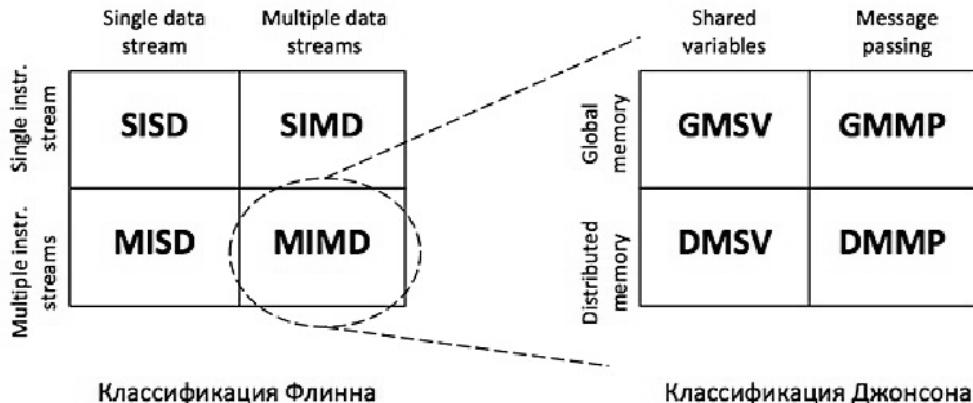
Одной из наиболее распространенных классификаций вычислительных систем является классификация Флинна. Четыре класса вычислительных систем выделяются в соответствие с двумя измерениями – характеристиками систем: поток команд, которые данная архитектура способна выполнить в единицу времени (одиночный или множественный) и поток данных, которые могут быть обработаны в единицу времени (одиночный или множественный).

- SISD (Single Instruction, Single Data) – системы, в которых существует одиночный поток команд и одиночный поток данных. В каждый момент времени процессор обрабатывает одиночный поток команд над одиночным потоком данных. К данному типу систем относятся последовательные персональные компьютеры с

одноядерными процессорами.

- SIMD (Single Instruction, Multiple Data) – системы с одиночным потоком команд и с множественным потоком данных; подобный класс составляют многопроцессорные системы, в которых в каждый момент времени может выполняться одна и та же команда для обработки нескольких информационных элементов. Такая архитектура позволяет выполнять одну арифметическую операцию над элементами вектора. Современные компьютеры реализуют некоторые команды типа SIMD (векторные команды), позволяющие обрабатывать несколько элементов данных за один такт.
- MISD (Multiple Instructions, Single Data) – системы, в которых существует множественный поток команд и одиночный поток данных; к данному классу относят систолические вычислительные системы и конвейерные системы;
- MIMD (Multiple Instructions, Multiple Data) – системы с множественным потоком команд и множественных потоком данных; к данному классу относится большинство параллельных вычислительных систем.

Классификация Флинна относит почти все параллельные вычислительные системы к одному классу – MIMD. Для выделения разных типов параллельных вычислительных систем применяется классификация Джонсона, в которой дальнейшее разделение многопроцессорных систем основывается на используемых способах организации оперативной памяти в этих системах. Данный подход позволяет различать два важных типа многопроцессорных систем: multiprocessors (мультипроцессорные или системы с общей разделяемой памятью) и multicompilers (мультикомпьютеры или системы с распределенной памятью).



Классификация Джонсоном основана на структуре памяти (global - глобальная или distributed - распределенная) и механизме коммуникаций и синхронизации (shared variables - разделяемые переменные или message passing - передача сообщений). Системы GMSV (global-memory-shared-variables) часто называются также мультипроцессорами с разделяемой памятью (shared-memory multiprocessors). Системы DMMP (distributed-memory-message-passing) также называемые мультикомпьютерами с распределенной памятью (distributed-memory multicompilers).

Архитектура однопроцессорной машины

Современная однопроцессорная машина состоит из нескольких компонентов: центрального процессорного устройства (ЦПУ), первичной памяти, одного или нескольких уровней кэш-памяти (кэш), вторичной (дисковой) памяти и набора периферийных устройств (дисплей, клавиатура, мышь, модем, CD, принтер и т.д.). Основными компонентами для выполнения программ являются ЦПУ, кэш и память.

Оперативная
память

Кэш уровня 2

Кэш уровня 1

Центральное
процессорное
устройство

Процессор выбирает инструкции из памяти, декодирует их и выполняет. Он содержит управляющее устройство (УУ), арифметико-логическое устройство (АЛУ) и регистры. УУ вырабатывает сигналы, управляющие действиями АЛУ, системой памяти и внешними устройствами. АЛУ выполняет арифметические и логические инструкции, определяемые набором инструкций процессора. В регистрах хранятся инструкции, данные и состояние машины (включая счетчик команд).

Мультикомпьютеры с распределенной памятью

В мультикомпьютерах с распределенной памятью существуют соединительная сеть, но каждый процессор имеет собственную память. Соединительная сеть поддерживает передачу сообщений. Мультикомпьютеры (многопроцессорные системы с распределенной памятью) не обеспечивают общий доступ ко всей имеющейся в системах памяти. Каждый процессор системы может использовать только свою локальную память, в то время как для доступа к данным, располагаемых на других процессорах, необходимо использовать

интерфейсы передачи сообщений (например, стандарт MPI). Данный подход используется при построении двух важных типов многопроцессорных вычислительных систем - массивно-параллельных систем (massively parallel processor or MPP) и кластеров (clusters).



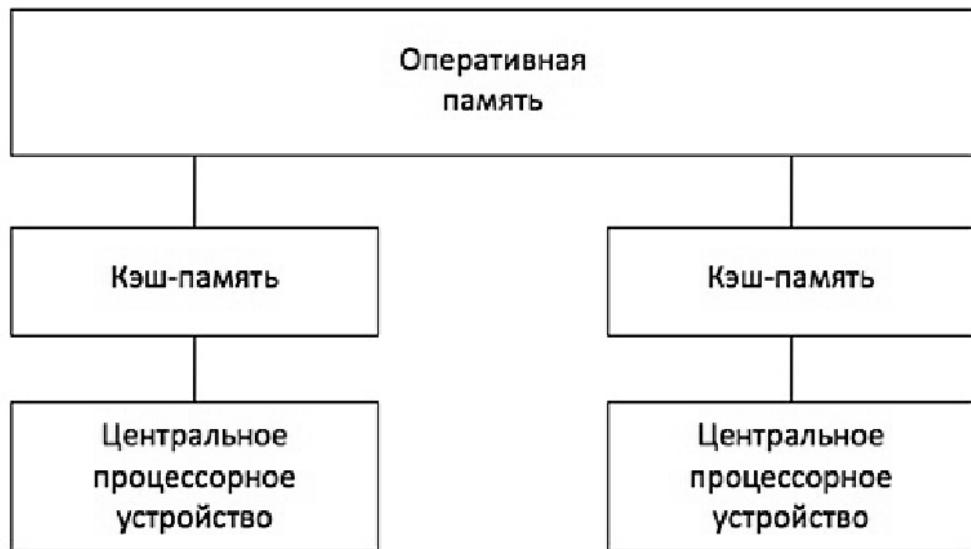
Мультикомпьютер (многомашинная система) – мультипроцессор с распределенной памятью, в котором процессоры и сеть расположены физически близко (в одном помещении). Также называют тесно связанный машинной. Она одновременно используется одним или небольшим числом приложений; каждое приложение задействует выделенный набор процессоров. Соединительная сеть с большой пропускной способностью предоставляет высокоскоростной путь связи между процессорами.

Сетевая система – это многомашинная система с распределенной памятью, связаны с помощью локальной сети или глобальной сети Internet (слабо связанные мультикомпьютеры). Здесь процессоры взаимодействуют также с помощью передачи сообщений, но время их доставки больше, чем в многомашинных системах, и в сети больше конфликтов. С другой стороны, сетевая система строится на основе обычных рабочих станций и сетей, тогда как в многомашинной системе часто есть специализированные компоненты, особенно у связующей

Под кластером обычно понимается множество отдельных компьютеров, объединенных в сеть, для которых при помощи специальных аппаратно-программных средств обеспечивается возможность унифицированного управления, надежного функционирования и эффективного использования. Кластеры могут быть образованы на базе уже существующих у потребителей отдельных компьютеров, либо же сконструированы из типовых компьютерных элементов, что обычно не требует значительных финансовых затрат. Применение кластеров может также в некоторой степени снизить проблемы, связанные с разработкой параллельных алгоритмов и программ, поскольку повышение вычислительной мощности отдельных процессоров позволяет строить кластеры из сравнительно небольшого количества (несколько десятков) отдельных компьютеров (*lowly parallel processing*). Это приводит к тому, что для параллельного выполнения в алгоритмах решения вычислительных задач достаточно выделять только крупные независимые части расчетов (*coarse granularity*), что, в свою очередь, снижает сложность построения параллельных методов вычислений и уменьшает потоки передаваемых данных между компьютерами кластера. Вместе с этим следует отметить, что организация взаимодействия вычислительных узлов кластера при помощи передачи сообщений обычно приводит к значительным времененным задержкам, что накладывает дополнительные ограничения на тип разрабатываемых параллельных алгоритмов и программ.

Мультипроцессор с разделяемой памятью

В мультипроцессоре и в многоядерной системе исполнительные устройства (процессоры и ядра процессоров) имеют доступ к разделяемой оперативной памяти. Процессоры совместно используют оперативную память.



У каждого процессора есть собственный кэш. Если два процессора ссылаются на разные области памяти, их содержимое можно безопасно поместить в кэш каждого из них. Проблема возникает, когда два процессора обращаются к одной области памяти. Если оба процессора только считывают данные, в кэш каждого из них можно поместить копию данных. Но если один из процессоров записывает в память, возникает проблема согласованности кэша: в кэш-памяти другого процессора теперь содержатся неверные данные. Необходимо либо обновить кэш другого процессора, либо признать содержимое кэша недействительным. Обеспечение однозначности кэшей реализуется на аппаратном уровне – для этого после изменения значения общей переменной все копии этой переменной в кэшах отмечаются как недействительные и последующий доступ к переменной потребует обязательного обращения к основной памяти. Необходимость обеспечения когерентности приводит к некоторому снижению скорости вычислений и затрудняет создание систем с достаточно большим количеством процессоров.

Наличие общих данных при выполнении параллельных вычислений приводит к необходимости синхронизации взаимодействия одновременно выполняемых потоков команд. Так, например, если изменение общих данных требует для своего выполнения некоторой последовательности действий, то необходимо обеспечить

взаимоисключение с тем, чтобы эти изменения в любой момент времени мог выполнять только один командный поток. Задачи взаимоисключения и синхронизации относятся к числу классических проблем, и их рассмотрение при разработке параллельных программ является одним из основных вопросов параллельного программирования.

Режимы выполнения независимых частей программы

При рассмотрении проблемы организации параллельных вычислений следует различать следующие возможные режимы выполнения независимых частей программы:

1) Режим разделения времени (многозадачный режим)

Режим разделения времени предполагает, что число подзадач (процессов или потоков одного процесса) больше, чем число исполнительных устройств. Данный режим является псевдопараллельным, когда активным (исполняемым) может быть одна единственная подзадача, а все остальные процессы (потоки) находятся в состоянии ожидания своей очереди на использование процессора; использование режима разделения времени может повысить эффективность организации вычислений (например, если один из процессов не может выполняться из-за ожидания вводимых данных, процессор может быть задействован для выполнения другого, готового к исполнению процесса), кроме того в данном режиме проявляются многие эффекты параллельных вычислений (необходимость взаимоисключения и синхронизации процессов и др.).

Многопоточность приложений в операционных системах с разделением времени применяется даже в однопроцессорных системах. Например, в Windows-приложениях многопоточность повышает отзывчивость приложения – если основной поток занят выполнением каких-то расчетов или запросов, другой поток позволяет реагировать на действия пользователя. Многопоточность упрощает разработку приложения. Каждый поток может планироваться и выполняться независимо. Например, когда пользователь нажимает кнопку мышки персонального компьютера, посыпается сигнал процессу, управляющему окном, в котором в данный момент находится курсор мыши. Этот

процесс (поток) может выполняться и отвечать на щелчок мыши. Приложения в других окнах могут продолжать при этом свое выполнение в фоновом режиме.

2) Распределенные вычисления

Компоненты выполняются на машинах, связанных локальной или глобальной сетью. По этой причине процессы взаимодействуют, обмениваясь сообщениями.

Такие системы пишутся для распределения обработки (как в файловых серверах), обеспечения доступа к удаленным данным (как в базах данных и в Web), интеграции и управления данными, распределенными по своей сути (как в промышленных системах), или повышения надежности (как в отказоустойчивых системах). Многие распределенные системы организованы как системы типа клиент-сервер. Например, файловый сервер предоставляет файлы данных для процессов, выполняемых на клиентских машинах. Компоненты распределенных систем часто сами являются многопоточными.

3) Синхронные параллельные вычисления.

Их цель – быстро решать данную задачу или за то же время решить большую задачу. Примеры синхронных вычислений:

- научные вычисления, которые моделируют и имитируют такие явления, как глобальный климат, эволюция солнечной системы или результат действия нового лекарства;
- графика и обработка изображений, включая создание спецэффектов в кино;
- крупные комбинаторные или оптимизационные задачи, например, планирование авиаперелетов или экономическое моделирование.

Программы решения таких задач требуют эффективного использования доступных вычислительных ресурсов системы. Число подзадач должно быть оптимизировано с учетом числа исполнительных устройств в системе (процессоров, ядер процессоров).

Уровни параллелизма в многоядерных архитектурах

Параллелизм на уровне команд (InstructionLevelParallelism, ILP) позволяет процессору выполнять несколько команд за один такт. Зависимости между командами ограничивают количество доступных для выполнения команд, снижая объем параллельных вычислений. Технология ILP позволяет процессору переупорядочить команды оптимальным образом с целью исключить остановки вычислительного конвейера и увеличить количество команд, выполняемых процессором за один такт. Современные процессоры поддерживают определенный набор команд, которые могут выполняться параллельно.

Параллелизм на уровне потоков процесса. Потоки позволяют выделить независимые потоки исполнения команд в рамках одного процесса. Потоки поддерживаются на уровне операционной системы. Операционная система распределяет потоки процессов по ядрам процессора с учетом приоритетов. С помощью потоков приложение может максимально задействовать свободные вычислительные ресурсы.

Параллелизм на уровне приложений. Одновременное выполнение нескольких программ осуществляется во всех операционных системах, поддерживающих режим разделения времени. Даже на однопроцессорной системе независимые программы выполняются одновременно. Параллельность достигается за счет выделение каждому приложению кванта процессорного времени.

Анализ эффективности параллельных вычислений

Анализ эффективности параллельных вычислений

$$S_p = \frac{T_1}{T_p}$$

Эффективность параллельного алгоритма определяется следующим образом:

$$E_p = \frac{T_1}{p \cdot T_p} = \frac{S_p}{p}$$

Эффективность показывает, насколько задействованы вычислительные ресурсы системы; идеальное теоретическое значение эффективности равно единице.

Пределы параллелизма

Достижению максимального ускорения может препятствовать существование в выполняемых вычислениях последовательных расчетов, которые не могут быть распараллелены. Джин Амдал (GeneAmdahl) показал, что верхняя граница для ускорения определяется долей последовательных вычислений алгоритма:

$$S_p \leq \frac{1}{f + (1-f)/p} \leq S^* = \frac{1}{f}$$

f – доля последовательных вычислений в применяемом алгоритме обработки данных, p – число процессоров.

Например, если доля последовательных вычислений составляет 25%, то максимально достижимое ускорение для параллельного алгоритма равно:

$$S^* = \frac{1}{f} = \frac{1}{0.25} = 4$$

В "законе Амдала" имеется несколько допущений, которые в реальных приложениях могут быть неверными. Одно из допущений заключается в том, что доля последовательных расчетов является постоянной величиной и не зависит от вычислительной сложности решаемой задачи. Однако, для большинства задач f является убывающей функцией от n , где n – параметр сложности задачи. В этом случае ускорение может быть увеличено при увеличении вычислительной сложности задачи. Нарушение "закона Амдала" также может быть связано с архитектурными особенностями параллельной вычислительной системы. Например, параллельный алгоритм уменьшает объем данных, используемых каждым процессором, и повышает эффективность использования кэш-памяти каждого процессора. Оптимальная работа с кэш-памятью может сильно увеличить быстродействие алгоритма.

Параллельный алгоритм называется масштабируемым, если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении постоянного уровня эффективности использования процессоров.

Проблемы разработки параллельных приложений

Основными этапами разработки параллельных приложений являются: декомпозиция, выявление информационных зависимостей между подзадачами, масштабирование подзадач и балансировка нагрузки для каждого процессора.

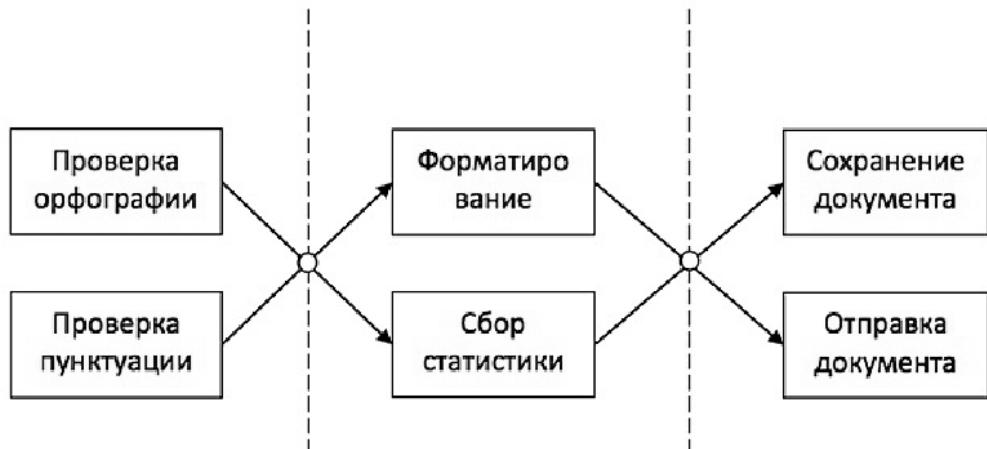
Декомпозиция

Под декомпозицией понимается разбиение задачи на относительно независимые части (подзадачи). Декомпозиция задачи может быть проведена несколькими способами: по заданиям, по данным, по информационным потокам.

Декомпозиция по заданиям (функциональная декомпозиция) предполагает присвоение разным потокам разных функций. Например, приложение выполняющее правку документа включает следующие функции: проверка орфографии CheckSpelling, проверка пунктуации CheckPuncto, форматирование текста в соответствие с выбранными стилями Format, подсчет статистики по документу CalcStat, сохранение изменений в файле SaveChanges и отправка документа по электронной почте SendDoc .



Функциональная декомпозиция разбивает работу приложения на подзадачи таким образом, чтобы каждая подзадача была связана с отдельной функцией. Но не все операции могут выполняться параллельно. Например, сохранение документа и отправка документа выполняются только после завершения всех предыдущих этапов. Форматирование и сбор статистики могут выполняться параллельно, но только после завершения проверки орфографии и пунктуации.



Декомпозиция по информационным потокам выделяет подзадачи, работающие с одним типом данных. В рассматриваемом примере могут быть выделены следующие подзадачи:

1. Работа с черновым документом (орфография и пунктуация);
2. Работа с исправленным документом (форматирование и сбор статистики);
3. Работа с готовым документом (сохранение и отправка);

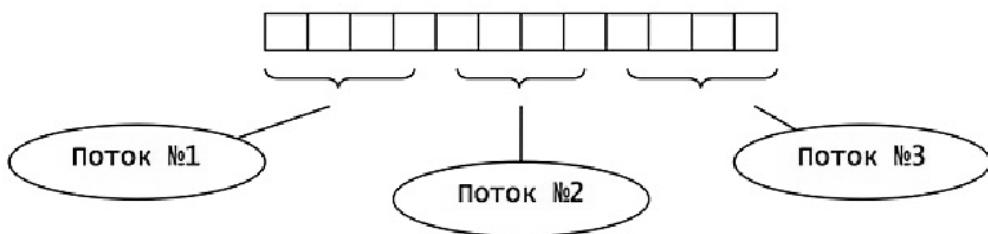
При декомпозиции по данным каждая подзадача работает со своим фрагментом данных, выполняя весь перечень действий. В рассматриваемом примере декомпозиция по данным может применяться к задачам, допускающим работу с фрагментом документа. Таким образом, функции CheckSpelling, CheckPuncto, CalcStat, Format объединяются в одну подзадачу, но создается несколько экземпляров этой подзадачи, которые параллельно работают с разными фрагментами документа. Функции SaveChanges и SendDoc составляют отдельные подзадачи, так как не могут работать с частью документа.

Декомпозиция по данным

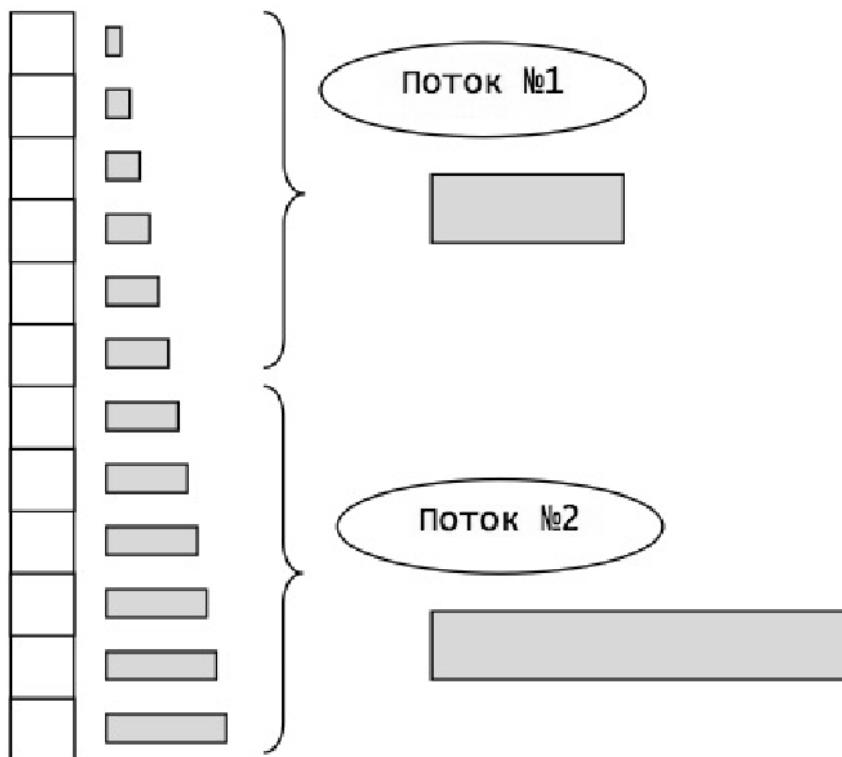
При декомпозиции по данным каждая подзадача выполняет одни и те же действия, но с разными данными. Во многих задачах, параллельных по данным, существует несколько способов разделения данных.

Например, задача матричного умножения допускает разделение по строчкам – каждый поток обрабатывает одну или несколько строчек матрицы, по столбцам – каждый поток обрабатывает один или несколько столбцов, а также по блокам заданного размера.

Два основных принципа разделения данных между подзадачами – статический и динамический. При статической декомпозиции фрагменты данных назначаются потокам до начала обработки и, как правило, содержат одинаковое число элементов для каждого потока. Например, разделение массива элементов может осуществляться по равным диапазонам индекса между потоками (*range partition*).

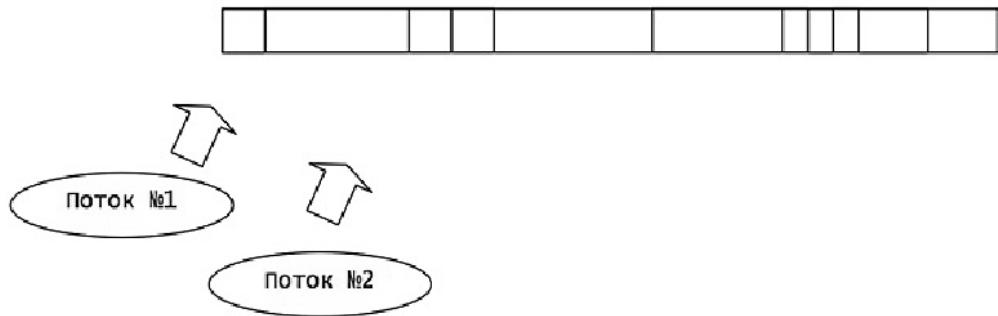


Основным достоинством статического разделения является независимость работы потоков (нет проблемы гонки данных) и, как следствие, нет необходимости в средствах синхронизации для доступа к элементам. Эффективность статической декомпозиции снижается при разной вычислительной сложности обработки элементов данных. Например, вычислительная нагрузка обработки i -элемента может зависеть от индекса элемента.



Разделение по диапазону приводит к несбалансированности нагрузки разных потоков. Несбалансированность нагрузки снижает эффективность распараллеливания. В некоторых случаях декомпозиция может быть улучшена и при статическом разбиении, когда заранее известно какие элементы обладают большей вычислительной трудоемкостью, а какие меньшей. Например, в случае зависимости вычислительной трудоемкости от индекса элемента, применение круговой декомпозиции выравнивает загруженность потоков. Первый поток обрабатывает все четные элементы, второй поток обрабатывает все нечетные.

В общем случае, когда вычислительная сложность обработки элементов заранее не известна, сбалансированность загрузки потоков обеспечивает динамическая декомпозиция.



/ При динамической декомпозиции каждый поток, участвующий в обработке, обращается за блоком данных (порцией). После обработки блока данных поток обращается за следующей порцией. Динамическая декомпозиция требует синхронизации доступа потоков к структуре данных. Размер блока определяет частоту обращений потоков к структуре. Некоторые алгоритмы динамической декомпозиции увеличивают размер блока в процессе обработки. Если поток быстро обрабатывает элементы, то размер блока для него увеличивается.

Масштабирование подзадач

Свойство масштабируемости заключается в эффективном использовании всех имеющихся вычислительных ресурсов. Параллельное приложение должно быть готово к тому, что завтра оно будет запускаться на системе с большими вычислительными возможностями. Свойство масштабируемости приложения тесно связано с выбранным алгоритмом решения задачи. Один алгоритм очень хорошо распараллеливается, но только на две подзадачи, другой алгоритм позволяет выделить произвольное число подзадач.

Обязательным условием масштабируемости приложения является возможность параметризации алгоритма в зависимости от числа процессоров в системе и в зависимости от текущей загруженности вычислительной системы. Такая параметризация позволяет изменять число выделяемых подзадач при конкретных условиях выполнения алгоритма. Современные платформы параллельного программирования предоставляют средства для автоматической балансировки нагрузки (пул потоков).

Проблема гонки данных

Потоки одного процесса разделяют единое адресное пространство, что упрощает взаимодействие подзадач (потоков), но требует обеспечения согласованности доступа к общим структурам данных.

Проблема гонки данных возникает при следующих условиях:

1. Несколько потоков работают с разделяемым ресурсом.
2. Конечный результат зависит от очередности выполнения командных последовательностей разных потоков.

Для иллюстрации проблемы гонки данных рассмотрим следующий фрагмент. Два потока выводят на экран значение общей переменной `Msg`.

```
// Код потока №1
(1) Msg = "I'm thread one";
(2) Console.WriteLine("Thread #1: " + Msg);

// Код потока №2
(3) Msg = "I'm thread two";
(4) Console.WriteLine("Thread #2: " + Msg);
```

Переменная `Msg` является общей – изменение переменной в одном потоке будут видны в другом потоке. При параллельной работе потоков вывод определяется конкретной последовательностью выполнения операторов.

Если операторы первого потока выполняются до операторов второго потока, т.е. при последовательности (1) – (2) – (3) – (4), то мы получаем:

```
Thread #1: I'm thread one
Thread #2: I'm thread two
```

Если же в выполнение операторов одного потока вмешаются операторы другого потока, например, при последовательности (1) – (3) – (2) – (4),

то получим следующее:

Thread #1: I'm thread two

Thread #2: I'm thread two

Проблема гонки данных возникает не только при выполнении нескольких операторов, но и при выполнении одного оператора. Рассмотрим следующий случай. Оба потока выполняют один оператор над общей переменной `x` типа `int`:

`x = x + 5`

Данный оператор предполагает выполнение следующих действий:

загрузить значение операндов в регистры процессора
осуществить суммирование
записать результат по адресу переменной `x`

При выполнении командных последовательностей одного потока на одном исполнительном устройстве в работу может вмешаться другой поток. Конечный результат зависит от очередности выполнения командных последовательностей. Если потоки осуществляют суммирование последовательно, то получаем конечный результат: 10. Если потоки осуществляют суммирование одновременно, то раннее изменение будет затерто поздним:

Действия	>Поток №1	Поток №2
Загрузка operandов	0	0
Вычисление	5	5
Запись результатов	5	5
Значение переменной	5	5

Еще одной "ловушкой" в многопоточной обработке является работа с динамическими структурами данных (списки, словари). Добавление и

удаление элементов в динамические структуры данных осуществляется с помощью одного метода:

```
list.Add("New element");
dic.RemoveKey("keyOne");
```

Реализация методов включает несколько действий. Например, при добавлении элемента в список необходимо записать элемент по текущему индексу (указателю) и сдвинуть индекс на следующую позицию. Для корректного осуществления добавления элемента одним потоком необходимо, чтобы другие потоки дождались завершения манипуляций со списком.

```
// Добавление элемента в массив по текущему индексу
data[current_index] = new_value;
current_index++;
```

Для решения проблемы гонки данных необходимо обеспечить взаимно исключительный доступ к тем командным последовательностям, в которых осуществляется работа с разделяемым ресурсом. Взаимная исключительность означает, что в каждый момент времени с ресурсом работает только один поток, другие потоки блокируются в ожидании завершения работы первого потока. Фрагмент кода, к которому должен быть обеспечен взаимно исключительный доступ, называется критической секцией.

Проблема гонки данных не всегда возникает при работе с общей переменной. Например, в следующем фрагменте два потока перед завершением осуществляют изменение разделяемой переменной, а третий поток читает это значение.

```
// Общая переменная
bool b = false;
//Поток №1
void f1()
{
    DoSomeWork1();
    b = true;
```

}

```
//Поток №2
void f2()
{
    DoSomeWork2();
    b = true;
}
```

```
//Поток №3
void f3()
{
    while(!b);
    DoSomeWork3();
}
```

В этом примере третий поток в цикле "ожидает" завершения хотя бы одного потока. Проблемы гонки данных не возникает, несмотря на работу трех потоков с общей переменной. Порядок выполнения потоков не влияет на конечный результат. Изменения, вносимые потоками, не противоречат друг другу.

Проблемы синхронизации

Решение проблемы гонки данных требует применения средств синхронизации, позволяющих обеспечить взаимно-исключительный доступ к критической секции. Применение синхронизации гарантирует получение корректных результатов, но снижает быстродействие приложения. Чем больше размер критических секций в приложении, тем большая доля последовательного выполнения и ниже эффективность от распараллеливания. Для повышения быстродействия размер критической секции должен быть предельно минимальным – только те операторы, последовательность которых не должна прерываться другим потоком.

В следующем фрагменте каждый поток сохраняет в разделяемом массиве `data` данные из файла. Для обеспечения согласованного

доступа к разделяемому ресурсу используются средства синхронизации.

// Поток №1

<Вход в критическую секцию>

```
StreamReader sr = File.OpenText("file" + ThreadNum);
newValue = GetValue(sr);
data[cur_index] = newValue;
cur_index++;
sr.Close();
```

<Выход из критической секции>

Размер критической секции в этом фрагменте не оправдано большой. Действия по подготовке данных для сохранения (открытие файла, уникального для каждого потока; поиск и чтение необходимой информации) могут быть вынесены за критическую секцию:

// Поток №1

```
StreamReader sr = File.OpenText("file" + ThreadNum);
newValue = GetValue(sr);
```

<Вход в критическую секцию>

```
data[cur_index] = newValue;
cur_index++;
```

<Выход из критической секции>

```
sr.Close();
```

Современные платформы для параллельного программирования, в том числе и среда Framework .NET, предлагают широкий выбор средств синхронизации. В каждой задаче применение того или иного инструмента синхронизации будет более эффективным. Например, для многопоточного увеличения разделяемого счетчика могут использоваться средства организации взаимно-исключающего доступа (объекты Monitor, Mutex), сигнальные сообщения (AutoResetEvent, ManualResetEventSlim), двоичные семафоры (Semaphore). Но максимально эффективным будет использование неблокирующих атомарных операторов (Interlocked.Increment).

Для работы с динамическими списками можно использовать как обычные коллекции с теми или иными средствами синхронизации, так и конкурентные коллекции с встроенной неблокирующей синхронизацией.

Проблемы кэшируемой памяти

Наличие кэшируемой памяти увеличивает быстродействие обработки данных, но усложняет работу системы при многопоточной обработке. Неоптимальная работа с кэшируемой памятью может сильно снизить эффективность параллельной обработки.

Кэш-память каждого процессора (ядра процессора) наполняется данными, необходимыми для работы потока, выполняющегося на этом процессоре. Если потоки работают с общими данными, то на аппаратном уровне должна обеспечиваться согласованность содержимого кэш-памяти. Изменение общей переменной в одном потоке, сигнализирует о недействительности значения переменной, загруженной в кэш-память другого процессора. При этом необходимо сохранить значение переменной в оперативной памяти и обновить кэш-память других процессоров. Большая интенсивность изменений общих переменных, которые используются в нескольких потоках, приводит к большому числу ошибок кэш-памяти (кэш-промахи) и увеличению накладных расходов, связанных с обновлением кэш-памяти.

Распространенной проблемой кэш-памяти является так называемое ложное разделение данных (*false sharing*). Проблема связана с тем, что потоки работают с разными переменными, которые в оперативной памяти расположены физически близко. Дело в том, что в кэш-память загружается не конкретная переменная, а блок памяти (строка кэша), содержащая необходимую переменную. Размер строки кэша может составлять 64, 128, 512 байт. Если в одной строке кэша расположены несколько переменных, используемых в разных потоках, то в кэш-память каждого процессора будет загружена одна и та же строка. При изменении в одном потоке своей переменной, содержимое кэш-памяти других процессоров считается недействительным и требует обновления.

В качестве иллюстрации проблемы ложного разделения рассмотрим

следующий фрагмент. В программе объявлена структура, содержащая несколько полей.

```
struct data
{
    int x;
    int y;
}
```

Первый поток работает только с полем `x`, второй поток работает только с полем `y`. Таким образом, разделения данных и проблемы гонки данных между потоками нет. Но последовательное расположение в памяти структуры `data`, приводит к тому, что в кэш-память одного и другого процессора загружается строка размером 64 байт, содержащая значения и поля `x` (4 байта), и поля `y` (4 байта). При изменении поля в одном потоке происходит обновление строки кэша в другом потоке.

```
// Поток №1
for(int i=0; i<N; i++)
    data1.x++;
```

```
// Поток №2
for(int i=0; i<N; i++)
    data1.y++;
```

Чтобы избежать последовательного расположения полей `x` и `y` в памяти, можно использовать дополнительные промежуточные поля.

Другой подход заключается в явном выравнивании полей в памяти с помощью атрибута `FieldOffsetAttribute`, который определен в пространстве `System.Runtime.InteropServices`:

```
// Явное выравнивание в памяти
[StructLayout(LayoutKind.Explicit)]
struct data
```

```

{
    [FieldOffset(0)] public int x;
    [FieldOffset(64)] public int y;
}

```

При достаточно большом значении N, разница в быстродействии кода с разделением кэша и без разделения может достигать 1.5 – 2 раз.

Все же самым эффективным решением при независимой обработке полей структуры будет применение локальных переменных внутри каждого потока. Разница в быстродействии может достигать нескольких десятков.

Модели параллельных приложений

Существуют следующие распространенные модели параллельных приложений:

- модель делегирования ("управляющий-рабочий");
- сеть с равноправными узлами;
- конвейер;
- модель "производитель-потребитель"

Каждая модель характеризуется собственной декомпозицией работ, которая определяет, кто отвечает за порождение подзадач и при каких условиях они создаются, какие информационные зависимости между подзадачами существуют.

Модель	Описание
Модель делегирования	Центральный поток ("управляющий") создает "рабочие" потоки и назначает каждому из них задачу. Управляющий поток ожидает завершения работы потоков, собирает результаты.
Модель с равноправными узлами	Все потоки имеют одинаковый рабочий статус.

Конвейер	Конвейерный подход применяется для поэтапной обработки потока входных данных.
Модель "производитель-потребитель"	Частный случай конвейера с одним производителем и одним потребителем.

Работа с потоками

Создание потоков, ожидания завершения потоков. Пул потоков Thread Pool.

В системах с общей памятью, включая многоядерные архитектуры, параллельные вычисления могут выполняться как при многопроцессном выполнении, так и при многопоточном выполнении. Многопроцессное выполнение подразумевает оформление каждой подзадачи в виде отдельной программы (процесса). Недостатком такого подхода является сложность взаимодействия подзадач. Каждый процесс функционирует в своем виртуальном адресном пространстве, не пересекающемся с адресным пространством другого процесса. Для взаимодействия подзадач необходимо использовать специальные средства межпроцессной коммуникации (интерфейсы передачи сообщений, общие файлы, объекты ядра операционной системы).

Потоки позволяют выделить подзадачи в рамках одного процесса. Все потоки одного приложения работают в рамках одного адресного пространства. Для взаимодействия потоков не нужно применять какие-либо средства коммуникации. Потоки могут непосредственно обращаться к общим переменным, которые изменяют другие потоки. Работа с общими переменными приводит к необходимости использования средств синхронизации, регулирующими порядок работы потоков с данными.

Потоки являются более легковесной структурой по сравнению с процессами ("облегченные" процессы). Поэтому параллельная работа множества потоков, решающих общую задачу, более эффективна в плане временных затрат, чем параллельная работа множества процессов.

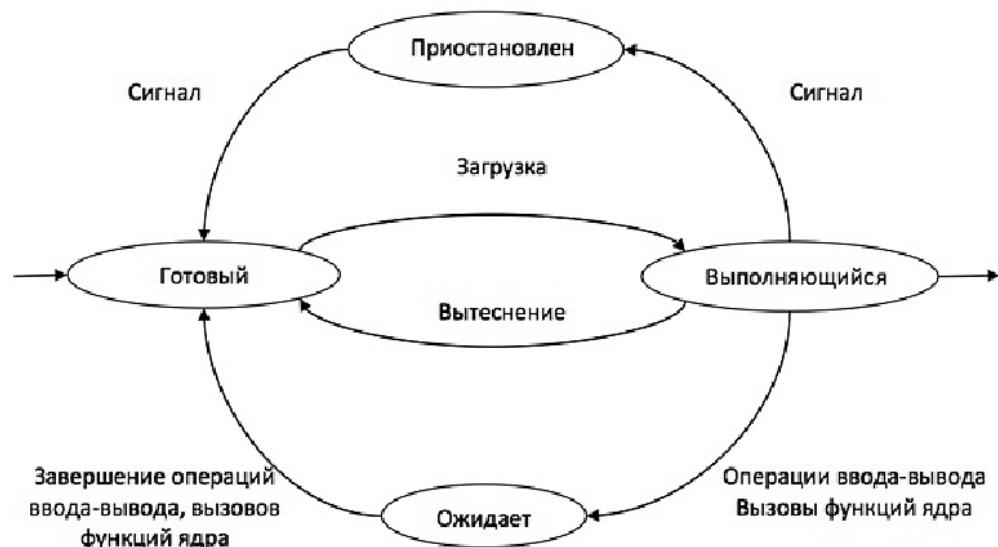
Структура потока

Поток состоит из нескольких структур. Ядро потока – содержит информацию о текущем состоянии потока: приоритет потока, программный и стековый указатели. Программный и стековые указатели образуют контекст потока и позволяют восстановить выполнение потока на процессоре. Блок окружения потока - содержит заголовок цепочки обработки исключений, локальное хранилище

данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL. Стек пользовательского режима – используется для передаваемых в методы локальных переменных и аргументов. Стек режима ядра – используется, когда код приложения передает аргументы в функцию операционной системы, находящуюся в режиме ядра. Ядро ОС вызывает собственные методы и использует стек режима ядра для передачи локальных аргументов, а также для сохранения локальных переменных.

Состояния потоков

Каждый поток может находиться в одном из нескольких состояний. Поток, готовый к выполнению и ожидающий предоставления доступа к центральному процессору, находится в состоянии "Готовый". Поток, который выполняется в текущий момент времени, имеет статус "Выполняющийся". При выполнении операций ввода-вывода или обращений к функциям ядра операционной системы, поток снимается с процессора и находится в состоянии "Ожидает". При завершении операций ввода-вывода или возврате из функций ядра поток помещается в очередь готовых потоков. При переключении контекста поток выгружается и помещается в очередь готовых потоков.



Переключение контекста

1. Значения регистров процессора для исполняющегося в данный момент потока сохраняются в структуре контекста, которая располагается в ядре потока.
2. Из набора имеющихся потоков выделяется тот, которому будет передано управление. Если выбранный поток принадлежит другому процессу, Windows переключает для процессора виртуальное адресное пространство.
3. Значения из выбранной структуры контекста потока загружаются в регистры процессора.

Работа с потоками в C#

Среда исполнения .NET CLR предоставляет возможность работы с управляемыми потоками через объект `Thread` пространства имен `System.Threading`. Среда исполнения стремится оптимизировать работу управляемых потоков и использовать для их выполнения потоки процесса, существующие на уровне операционной системы. Поэтому создание потоков типа `Thread` не всегда сопряжено с созданием потоков процесса.

Основные этапы работы

```
// Инициализация потока
Thread oneThread = new Thread(Run);
// Запуск потока
oneThread.Start();
// Ожидание завершения потока
oneThread.Join();
```

В качестве рабочего элемента можно использовать метод класса, делегат метода или лямбда-выражение. В следующем фрагменте создаются три потока. Первый поток в качестве рабочего элемента принимает статический метод `LocalWorkItem`. Второй поток инициализируется с помощью лямбда-выражения, третий поток связывается с методом общедоступного класса.

```
class Program
{
    static void LocalWorkItem()
    {
        Console.WriteLine("Hello from static method");
    }
    static void Main()
    {
        Thread thr1 = new Thread(LocalWorkItem);
        thr1.Start();
        Thread thr2 = new Thread(() =>
        {
            Console.WriteLine("Hello from
                lambda-expression");
        });
        thr2.Start();
        ThreadClass thrClass = new ThreadClass("Hello from thread-class");
        Thread thr3 = new Thread(thrClass.Run);
        thr3.Start();
    }
}
class ThreadClass
{
    private string greeting;
    public ThreadClass(string sGreeting)
    {
        greeting = sGreeting;
    }
    public void Run()
    {
        Console.WriteLine(greeting);
    }
}
```

Вызов метода `thr1.Join()` блокирует основной поток до завершения работы потока `thr1`.

Thread thr1 = new Thread(() =>

```
{  
    for(int i=0; i<5; i++)  
        Console.Write("A");  
};  
Thread thr2 = new Thread(() =>  
{  
    for(int i=0; i<5; i++)  
        Console.Write("B");  
});  
Thread thr3 = new Thread(() =>  
{  
    for(int i=0; i<5; i++)  
        Console.Write("C");  
});  
thr1.Start();  
thr2.Start();  
thr1.Join();  
thr2.Join();  
thr3.Start();
```

В общем случае порядок вывода первого и второго потоков не определен. Вывод третьего потока осуществляется только после завершения работы потоков `thr1` и `thr2`.

Можно получить такие результаты:

AAAAABBBBBCCCCC

или

BBBBBAAAAACCCCC

Маловероятны, но возможны варианты

AABAAABCCCCC

или

AAAABBBBBACCCCC

Передача параметров

Общение с потоком (передача параметров, возвращение результатов) можно реализовать с помощью глобальных переменных.

```
class Program
{
    static long Factorial(long n)
    {
        long res = 1;
        do
        {
            res = res * n;
        } while(--n > 0);
        return res;
    }
    static void Main()
    {
        long res1, res2;
        long n1 = 5000, n2 = 10000;
        Thread t1 = new Thread(() =>
        {
            res1 = Factorial(n1));
        });
        Thread t2 = new Thread(() => { res2=Factorial(n2); });
        // Запускаем потоки
        t1.Start(); t2.Start();
        // Ожидаем завершения потоков
        t1.Join(); t2.Join();
        Console.WriteLine("Factorial of {0} equals {1}",
            n1, res1);
        Console.WriteLine("Factorial of {0} equals {1}",
            n2, res2);
    }
}
```

}

Существует возможность передать параметры в рабочий метод потока с помощью перегрузки метода `Start`. Сигнатура рабочего метода строго фиксирована – либо без аргументов, либо только один аргумент типа `object`. Поэтому при необходимости передачи нескольких параметров в рабочем методе необходимо выполнить правильные преобразования.

```
class Program
{
    static double res;
    static void ThreadWork(object state)
    {
        string sTitle = ((object[])state)[0] as string;
        double d = (double)((object[])state)[1];
        Console.WriteLine(sTitle);
        res = SomeMathOperation(d);
    }
    static void Main()
    {
        Thread thr1 = new Thread(ThreadWork);
        thr1.Start(new object[] {"Thread #1", 3.14});
        thr1.Join();
        Console.WriteLine("Result: {0}", res);
    }
}
```

Работа в лямбда-выражениях и анонимных делегатах с общими переменными может приводить к непредсказуемым результатам.

```
for(int i=0; i<10; i++)
{
    Thread t = new Thread(() =>
        Console.Write("ABCDEFGHIJK "[i]))
    t.Start();
```

}

Ожидаем получить все буквы в случайному порядке, а получаем

BDDDEEJJKK

Если в строковой константе оставить только 10 букв, полагая, что индекс *i* может быть от 0 до 9, получаем ошибку "Индекс вышел за границы массива".

Проблема связана с тем, что при объявлении потока делегат метода или лямбда-выражение содержит только ссылку на индекс *i*. Когда созданный поток начинает свою работу фактическое значение индекса уже давно убежало вперед. Последнее значение индекса равно 10, что и приводит к возникновению исключения. Исправить данный фрагмент можно с помощью дополнительной переменной, которая на каждой итерации сохраняет текущее значение индекса.

```
for(int i=0; i<10; i++)
{
    int i_copy = i;
    Thread t = new Thread(new delegate(
        Console.WriteLine("ABCDEFGHIJK "[i_copy]))
    t.Start();
}
```

Теперь у каждого потока своя независимая переменная *i_copy* с уникальным значением. В результате получаем:

ABCDEFGHIJK

или в произвольном порядке, но все буквы по одному разу

BDACFGHEIJK

Приостановление потока

Метод `Sleep()` позволяет приостановить выполнение текущего потока на заданное число миллисекунд:

```
// Приостанавливаем поток на 100 мс
Thread.Sleep(100);
// Приостанавливаем поток на 5 мин
Thread.Sleep(TimeSpan.FromMinute(5));
```

Если в качестве аргумента указывается ноль `Thread.Sleep(0)`, то выполняющийся поток отдает выделенный квант времени и без ожидания включается в конкуренцию за процессорное время. Такой прием может быть полезен в отладочных целях для обеспечения параллельности выполнения определенных фрагментов кода.

Например, следующий фрагмент

```
static void ThreadFunc(object o)
{
    for(int i=0; i<20; i++)
        Console.Write(o);
}
static void Main()
{
    Thread[] t = new Thread[4];
    for(int i=0; i<4; i++)
        t[i] = new Thread(ThreadFunc);

    t[0].Start("A"); t[1].Start("B");
    t[2].Start("C"); t[3].Start("D");

    for(int i=0; i<4; i++)
        t[i].Join();
}
```

Выводит на консоль:

```
BBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAACCCC
```

Параллельность не наблюдается, так как каждый поток за выделенный квант процессорного времени успевает обработать все 20 итераций. Изменим тело цикла рабочей функции:

```
static void ThreadFunc(object o)
{
    for(int i=0; i<20; i++)
    {
        Console.Write(o);
        Thread.Sleep(0);
    }
}
```

Вывод стал более разнообразный:

```
AAAACACACACACACACACACACACACACACAACCBBDDDB
```

Существует аналог метода `Thread.Sleep(0)`, который позволяет вернуть выделенный квант – `Thread.Yield()`. При этом возврат осуществляется только в том случае, если для ядра, на котором выполняется данный поток, есть другой готовый к выполнению поток. Неосторожное применение методов `Thread.Sleep(0)` и `Thread.Yield()` может привести к ухудшению быстродействия из-за не оптимального использования кэш-памяти.

Свойства потока

Каждый поток имеет ряд свойств: `Name` - имя потока, `ManagedThreadId` – номер потока, `IsAlive` - признак существования потока, `IsBackground` – признак фонового потока, `ThreadState` – состояние потока. Эти свойства доступны и для внешнего вызова.

```
// Объявляем массив потоков
```

```
Thread[] arThr = new Thread[N];
for(int i=0; i<arThr.Length; i++)
{
    arThr[i] = new Thread(SomeFunc);
    arThr[i].Start();
}
for(int i=0; i<arThr.Length; i++)
{
    // Выводим информацию о потоках
    Console.WriteLine("Thread Id: {0}",
        name: {1}, IsAlive: {2}",
        arThr[i].ManagedThreadId,
        arThr[i].Name,
        arThr[i].IsAlive);
}
```

Свойства текущего потока можно получить с помощью объекта `Thread.CurrentThread`. Следующий фрагмент с помощью механизма рефлексии выводит все свойства текущего потока.

```
using System;
using System.Reflection;
using System.Threading;
class ThreadInfo
{
    static void Main()
    {
        Thread t = Thread.CurrentThread;
        t.Name = "MAIN THREAD";
        foreach(PropertyInfo p in t.GetType().GetProperties())
        {
            Console.WriteLine("{0}:{1}",
                p.Name,p.GetValue(t, null));
        }
    }
}
```

Вывод всех свойств текущего потока:

```
ManagedThreadId: 10
ExecutionContext: System.Threading.ExecutionContext
Priority: Normal
IsAlive: True
IsThreadPoolThread: False
CurrentThread: System.Threading.Thread
IsBackground: False
ThreadState: Running
ApartmentState: MTA
CurrentUICulture: ru-RU
CurrentCulture: ru-RU
CurrentContext: ContextID: 0
CurrentPrincipal: System.Security.Principal.GenericPrincipal
Name: MAIN THREAD
```

Приоритеты потоков

Приоритеты потоков определяют очередность выделения доступа к ЦП. Высокоприоритетные потоки имеют преимущество и чаще получают доступ к ЦП, чем низкоприоритетные. Приоритеты потоков задаются перечислением `ThreadPriority`, которое имеет пять значений: `Highest` - наивысший, `AboveNormal` – выше среднего, `Normal` - средний, `BelowNormal` – ниже среднего, `Lowest` - низший. По умолчанию поток имеет средний приоритет. Для изменения приоритета потока или чтения текущего используется свойство `Priority`. Влияние приоритетов сказывается только в случае конкуренции множества потоков за мощности ЦП.

В следующем фрагменте пять потоков с разными приоритетами конкурируют за доступ к ЦП с двумя ядрами. Каждый поток увеличивает свой счетчик.

```
class PriorityTesting
{
```

```
static long[] counts;
static bool finish;
static void ThreadFunc(object iThread)
{
    while(true)
    {
        if(finish)
            break;
        counts[(int)iThread]++;
    }
}
static void Main()
{
    counts = new long[5];
    Thread[] t = new Thread[5];
    for(int i=0; i<t.Length; i++)
    {
        t[i] = new Thread(ThreadFunc);
        t[i].Priority = (ThreadPriority)i;
    }
    // Запускаем потоки
    for(int i=0; i<t.Length; i++)
        t[i].Start(i);

    // Даём потокам возможность поработать 10 с
    Thread.Sleep(10000);

    // Сигнал о завершении
    finish = true;

    // Ожидаем завершения всех потоков
    for(int i=0; i<t.Length; i++)
        t[i].Join();
    // Вывод результатов
    for(int i=0; i<t.Length; i++)
        Console.WriteLine("Thread with priority {0, 15},",
                          Counts: {0}", (ThreadPriority)i, counts[i]);
}
}
```

Вывод программы

Thread with priority	Lowest, Counts: 7608195
Thread with priority	BelowNormal, Counts: 10457706
Thread with priority	Normal, Counts: 17852629
Thread with priority	AboveNormal, Counts: 297729812
Thread with priority	Highest, Counts: 302506232

Локальное хранилище потока

Типы, объявленные внутри рабочей функции потока, являются локальными – у каждого потока, выполняющего функцию, свои копии (приватные данные). Типы, объявленные вне рабочей функции потока, являются общими для всех потоков переменными. Изменения общих данных в одном потоке отражаются в другом потоке.

Существует возможность оперировать с локальными данными потока, объявленными вне рабочей функции, например, в общедоступном классе. Первый способ заключается в объявлении статического поля, локального для каждого потока.

В следующем фрагменте рабочая функция потока использует объект пользовательского типа `Data`. В этом классе

```
public class Data
{
    public static int sharedVar;
    [ThreadStatic] static int localVar;
}

class Program
{
    static void threadFunc(object i)
    {
        Console.WriteLine("Thread {0}: Before changing..
Shared: {1}, local: {2}",
i, Data.sharedVar, Data.localVar);
```

```

Data.sharedVar = (int)i;
Data.localVar = (int)i;
Console.WriteLine("Thread {0}: After changing..
    Shared: {1}, local: {2}",
    i, Data.sharedVar, Data.localVar);
}

static void Main()
{
    Thread t1 = new Thread(threadFunc);
    Thread t2 = new Thread(threadFunc);

    Data.sharedVar = 3; Data.localVar = 3;

    t1.Start(1); t2.Start(2);
    t1.Join(); t2.Join();
}

}

```

Вывод:

```

Thread 1: Before changing.. Shared: 3, local: 0
Thread 1: After changing.. Shared: 1, local: 1
Thread 2: Before changing.. Shared: 1, local: 0
Thread 2: After changing.. Shared: 2, local: 2

```

Ограничения этого способа связаны с тем, что атрибут используется только со статическими полями и инициализация поля осуществляется только одним потоком.

Второй способ объявления локальных данных заключается в использовании объекта `ThreadLocal<T>`:

```

static void Main()
{
    ThreadLocal<int> localSum = new ThreadLocal<int>(() => 0);
    t1 = new Thread(() => {
        for(int i=0; i<10; i++)

```

```
    localSum.Value++;
    Console.WriteLine(localSum.Value);
});

t2 = new Thread(() => {
    for(int i=0; i<10; i++)
        localSum.Value--;
    Console.WriteLine(localSum.Value);
});

t1.Start(); t2.Start();
t1.Join(); t2.Join();

Console.WriteLine(localSum.Value);
}
```

Получаем:

```
10
-10
0
```

Первый поток увеличивал свой счетчик, второй уменьшал, а третий (главный поток) ничего не делал со своим локальным счетчиком, поэтому получаем 0.

Третий способ заключается в использовании локальных слотов потока. Доступ к слотам обеспечивается с помощью методов `GetData`, `SetData`. Объект, идентифицирующий слот, можно получить с помощью строковой константы. Применение слотов является менее производительным по сравнению с локальными статическими полями. Но может быть полезным, если работа потока структурирована в нескольких методах. В каждом методе можно получить доступ к слоту потока по его имени.

```
public class ThreadWork
{
    private string sharedWord;
```

```
public void Run(string secretWord)
{
    sharedWord = secretWord;
    Save(secretWord);
    Thread.Sleep(500);
    Show();
}

private void Save(string s)
{
// Получаем идентификатор слота по имени
    LocalDataStoreSlot slot =
        Thread.GetNamedDataSlot("Secret");
// Сохраняем данные
    Thread.SetData(slot, s);
}

private void Show()
{
    LocalDataStoreSlot slot =
        Thread.GetNamedDataSlot("Secret");
    string secretWord = Thread.GetData(slot);
    Console.WriteLine("Thread {0}, secret word: {1},
                      shared word: {2}",
                      Thread.CurrentThread.ManagedThreadId,
                      secretWord, sharedWord);
}

class Program
{
    static void Main()
    {
        ThreadWork thr = new ThreadWork;
        new Thread(() => thr.Run("one")).Start();
        new Thread(() => thr.Run("two")).Start();
        new Thread(() => thr.Run("three")).Start();
        Thread.Sleep(1000);
    }

}
```

Вывод программы

```
Thread: 15, secret word: one, shared word: three
Thread: 16, secret word: two, shared word: three
Thread: 17, secret word: three, shared word: three
```

Переменная `sharedWord` является разделяемой, поэтому выводится последнее изменение, выполненное третьим потоком.

Пул потоков

Пул потоков предназначен для упрощения многопоточной обработки. Программист выделяет фрагменты кода (рабочие элементы), которые можно выполнять параллельно. Планировщик (среда выполнения) оптимальным образом распределяет рабочие элементы по рабочим потокам пула. Таким образом, вопросы эффективной загрузки оптимального числа потоков решаются не программистом, а планировщиком (исполняющей средой). Еще одним достоинством применения пула является уменьшение накладных расходов, связанных с ручным созданием и завершением потоков для каждого фрагмента кода, допускающего распараллеливание. Пул потоков используется для обработки задач типа `Task`. Задачи обладают рядом полезных встроенных механизмов (ожидания, отмены, продолжения и т.д.). Поэтому для распараллеливания рабочих элементов рекомендуется использовать именно задачи или шаблоны класса `Parallel`. Непосредственная работа с пулом без явного определения задач может быть полезна, когда нет необходимости в дополнительных возможностях объекта `Task`.

Для добавления рабочего элемента используется метод

```
// Добавление метода без параметров
ThreadPool.QueueUserWorkItem(SomeWork);
// Добавление метода с параметром
ThreadPool.QueueUserWorkItem(SomeWork, data);
```

В следующем фрагменте проиллюстрируем основные особенности пула

ПОТОКОВ.

```
for(int i=0; i<10; i++)
{
    ThreadPool.QueueUserWorkItem((object o)=> {
        Console.WriteLine("i: {0}, ThreadId: {1},
                          IsPoolThread: {2}",
                          i, Thread.CurrentThread.ManagedThreadId,
                          Thread.CurrentThread.IsThreadPoolThread);
    });
    Thread.Sleep(100);
}
```

Добавляем в пул потоков 10 экземпляров безымянного делегата, объявленного в виде лямбда-выражения. В рабочем элементе осуществляется вывод значения индекса *i*, номер потока и признак того, что поток принадлежит пулу.

```
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
```

Убеждаем, что действительно все рабочие элементы выполнялись потоками пула (признак *IsPoolThread* равен *true*). Всего в обработке участвовало только два потока. Для приведенного кода возможна ситуация, когда все рабочие элементы будут обрабатываться в одном потоке пула. Во всех рабочих элементах осуществляется вывод одного и того же значения индекса *i*, равного 10. Это связано с асинхронностью запуска рабочих элементов – основной поток продолжает обрабатывать цикл и увеличивать значение индекса, а

рабочие элементы фактически еще не выполняются. Для предотвращения такой ситуации необходимо использовать индивидуальные копии для каждого элемента.

Заменим работу с пулом на ручную работу с потоками:

```
for(int i=0; i<10; i++)
{
    new Thread((object o)=> {
        Console.WriteLine("i: {0}, ThreadId: {1},
                           IsPoolThread: {2}",
                           i, Thread.CurrentThread.ManagedThreadId,
                           Thread.CurrentThread.IsThreadPoolThread);
    });
    Start();
}
```

Вывод кода будет следующим:

```
i: 1, ThreadId: 5, IsPoolThread: False
i: 1, ThreadId: 6, IsPoolThread: False
i: 3, ThreadId: 7, IsPoolThread: False
i: 4, ThreadId: 8, IsPoolThread: False
i: 5, ThreadId: 9, IsPoolThread: False
i: 6, ThreadId: 10, IsPoolThread: False
i: 7, ThreadId: 11, IsPoolThread: False
i: 8, ThreadId: 12, IsPoolThread: False
i: 9, ThreadId: 13, IsPoolThread: False
i: 10, ThreadId: 14, IsPoolThread: False
```

Каждый рабочий элемент обрабатывался в своем потоке, не входящем в состав пула. С индексом *i* опять есть проблемы, но возникают они гораздо реже и не так явно проявляются – какое-то разнообразие в выводе все-таки есть. Это объясняется тем, что операция добавления делегата в очередь выполняется гораздо быстрее, чем инициализация запуска нового потока. Пока на второй итерации осуществляется запуск потока, первый поток уже приступил к работе и прочитал текущее значение индекса. Для гарантированной работы каждого потока с

уникальным индексом необходимо использовать копии индексов, создаваемые на каждой итерации:

```
for(int i=0; i<10; i++)
{
    int y = i;
    ThreadPool.QueueUserWorkItem((object o) => Console.WriteLine(y));
    new Thread(() => Console.WriteLine(y)).Start();
}
```

Основным неудобством работы с пулом, является отсутствие механизма ожидания завершения рабочих элементов. Необходимо использовать либо какие-то средства синхронизации (например, сигнальные сообщения `ManualResetEvent`, шаблон синхронизации `CountdownEvent`), либо общие переменные.

```
static void Func1(object o)
{
    var ev = (ManualResetEventSlim)o;
    // Работа
    Console.WriteLine("Func1: Working..");
    ev.Set();
}
static void Main()
{
    ManualResetEvent ev = new ManualResetEvent();
    ThreadPool.QueueWorkItem(Fun1, ev);
    ev.Wait()
}
```

В рабочую функцию передаем сигнальное сообщение. Основной поток блокируется в ожидании сигнала. Рабочий поток после завершения работы генерирует сигнал. В случае работы нескольких потоков можно использовать массив сообщений и метод `WaitHandle.WaitAll` для ожидания сигналов от всех потоков.

```
class Program
```

```
{  
    static void Main()  
    {  
        var events = new ManualResetEvent[10];  
        for(int i=0; i<10; i++)  
        {  
            int y = i;  
            events[i] = new ManualResetEvent(false);  
            ThreadPool.QueueWorkItem(() => {  
                SomeWork();  
                events[y].Set();  
            });  
        }  
        WaitHandle.WaitAll(events);  
    }  
}
```

Каждый рабочий элемент оперирует со своим уникальным сигнальным объектом. После завершения работы, в делегате устанавливается сигнальный объект. Основной поток дожидается завершения всех рабочих элементов.

Вопросы

1. Какие достоинства у непосредственной работы с потоками?
2. В чем заключаются основные проблемы организации многопоточной обработки?
3. В каких случаях работа с пулом потоков является более эффективной, чем непосредственная работа с потоками?

Упражнения

Реализуйте пользовательский пул потоков, который повторяет функциональность объекта `ThreadPool`. Добавьте новые методы пула, которые упрощают работу и делают общение с пулем более информативным. Например, методы ожидания запущенных задач; получение сводной информации о выполнении задач (число потоков,

участвовавших в обработке; время обработки).

Знакомство с многопоточной обработкой

Работа с потоками.

Задания

1. Реализуйте последовательную обработку элементов вектора, например, умножение элементов вектора на число. Число элементов вектора задается параметром N.
2. Реализуйте многопоточную обработку элементов вектора, используя разделение вектора на равное число элементов. Число потоков задается параметром M.
3. Выполните анализ эффективности многопоточной обработки при разных параметрах N (10, 100, 1000, 100000) и M (2, 3, 4, 5, 10) Результаты представьте в табличной форме.
4. Выполните анализ эффективности при усложнении обработки каждого элемента вектора.
5. Исследуйте эффективность разделения по диапазону при неравномерной вычислительной сложности обработки элементов вектора.
6. Исследуйте эффективность параллелизма при круговом разделении элементов вектора. Сравните с эффективностью разделения по диапазону.

Методические указания

В работе исследуется эффективность распараллеливания независимой обработки элементов вектора. В первом задании в качестве обработки можно выбрать то или иное математическое преобразование элементов вектора:

```
for(int i=0; i<a.Length; i++)
    b[i] = Math.Pow(a[i], 1.789);
```

Многопоточная обработка реализуется с помощью объектов Thread.

На многоядерной системе многопоточная обработка приводит к параллельности выполнения. Классы для работы с потоками расположены в пространстве имен `System.Threading`.

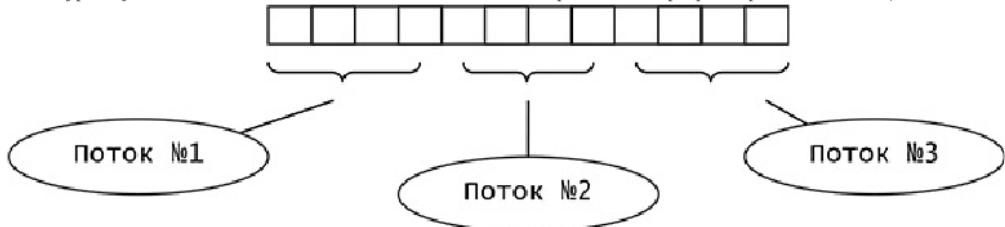
Для создания потока необходимо указать имя рабочего метода потока, который может быть реализован в отдельном классе, в главном классе приложения как статический метод или в виде лябда-выражения. Метод потока либо не принимает никаких аргументов, либо принимает аргумент типа `object`. Запуск потока осуществляется вызовом метода `Start`.

```
class Program
{
    static void Run(object some_data)
    {
        int m = (int) some_data;
        ..
    }
    static void Main()
    {
        ..
        Thread thr = new Thread(Run);
        thr.Start(some_data);
    }
}
```

Дождаться завершения работы потоков можно с помощью метода `Join`:

```
thr1.Join(); thr2.Join();
```

В функции потока необходимо предусмотреть возможность разбиения диапазона $0..(N-1)$ на число потоков `nThr`. При запуске потока в качестве аргумента передается либо "индекс потока", определяющий область массива, который обрабатывается в данном потоке, либо начальный и конечный индексы массива.



Многопоточное выполнение будет параллельным при наличии в вычислительной системе нескольких процессоров (ядер процессора). Число процессоров можно узнать с помощью свойства:

```
System.Environment.ProcessorCount;
```

Параллельное выполнение вычислений также можно реализовать с помощью классов библиотеки TPL (Task Parallel Library). Классы библиотеки располагаются в пространстве имен `System.Threading.Tasks`. Параллельное вычисление операций над элементами цикла выполняется с помощью метода `Parallel.For`:

```
Parallel.For(0, a.Length, i =>
{ b[i] = Math.Pow(a[i], 1.789); });
```

Для анализа производительности последовательного и параллельного выполнения можно использовать переменные типа `DateTime`. Например,

```
DateTime dt1, dt2;
dt1 = DateTime.Now;
// Вызов_вычислительной_процедуры;
dt2 = DateTime.Now;
TimeSpan ts = dt2 - dt1;
Console.WriteLine("Total time: {0}", ts.TotalMilliseconds);
```

Также можно использовать объект `Stopwatch` пространства `System.Diagnostics`:

```
Stopwatch sw = new Stopwatch();
```

```
sw.Start();
// Вызов_вычислительной_процедуры;
sw.Stop();
TimeSpan ts = sw.Elapsed;
Console.WriteLine("Total time: {0}", ts.TotalMilliseconds);
```

При оценке производительности необходимо учесть, что время выполнения алгоритма зависит от множества параметров. Поэтому желательно оценивать среднее время выполнения при нескольких прогонах алгоритма, исключая первый разогревающий прогон.

Эффективность параллельного алгоритма существенно зависит от элементов массива, числа потоков, сложности математической функции и т.д. Следует учитывать, что при малом объеме элементов массива, накладные расходы, связанные с организацией многопоточной обработки, превышают выигрыш от параллельности обработки. При последовательном выполнении примитивной циклической обработки быстродействие достигается за счет оптимального использования кэш-памяти.

Выполняя анализ зависимости быстродействия от числа потоков, следует учитывать число ядер процессора. Увеличение числа потоков сверх возможностей вычислительной системы приводит к конкуренции потоков и ухудшению быстродействия.

Усложнение обработки элементов массива предлагается реализовать с помощью внутреннего цикла. Например,

```
for(int i=0; i<a.Length; i++)
{
    // Обработка i-элемента
    for(int j=0; j < K; j++)
        b[i] += Math.Pow(a[i], 1.789);
}
```

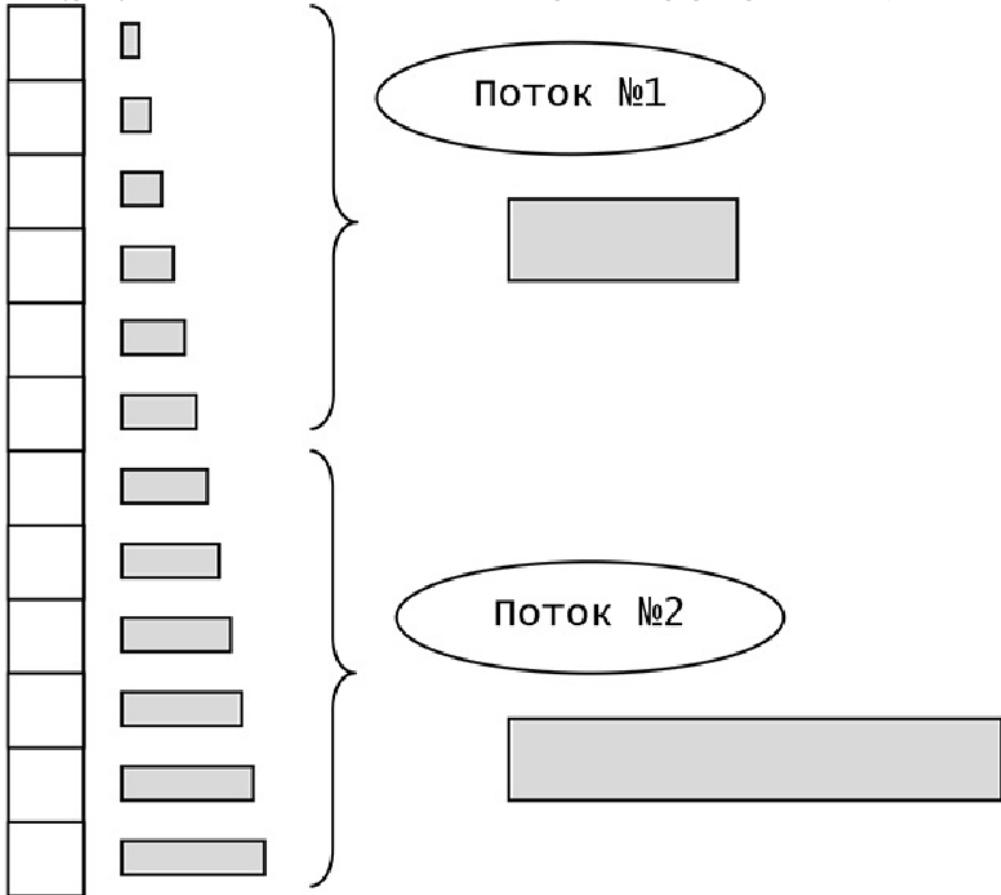
K – параметр "сложности". Увеличивая параметр K , наблюдаем повышение эффективности параллельной обработки при меньшем объеме массива чисел.

В рассмотренных вариантах обработки вычислительная нагрузка на

каждой итерации относительно одинакова. В ситуациях, когда вычислительная нагрузка зависит от индекса элемента, разделение массива по равным диапазонам может быть не эффективно. Рассмотрим следующий вариант обработки:

```
for(int i=0; i<a.Length; i++)
{
    // Обработка i-элемента
    for(int j=0; j <i; j++)
        b[i] += Math.Pow(a[i], 1.789);
}
```

Вычислительная нагрузка при обработке i -элемента зависит от индекса i . Обработка начальных элементов массива занимает меньшее время по сравнению с обработкой последних элементов. Разделение данных по диапазону приводит к несбалансированной загрузке потоков и снижению эффективности распараллеливания.



Одним из подходов к выравниванию загрузки потоков является применение круговой декомпозиции. В случае двух потоков получаем такую схему: первый поток обрабатывает все четные элементы, второй поток обрабатывает все нечетные элементы. Реализуйте круговую декомпозицию для нескольких потоков (больше двух).

Вопросы:

1. Почему эффект от распараллеливания наблюдается только при большем числе элементов?
2. Почему увеличение сложности обработки повышает эффективность многопоточной обработки?
3. Какое число потоков является оптимальным для конкретной вычислительной системы?

4. Почему неравномерность загрузки потоков приводит к снижению эффективности многопоточной обработки?
5. Какие другие варианты декомпозиции позволяют увеличить равномерность загрузки потоков?
6. В какой ситуации круговая декомпозиция не обеспечивает равномерную загрузку потоков?

Средства синхронизации

Критическая секция. Конструкция Lock. Атомарные операторы. Класс Interlocked. Семафоры. Semaphore и SemaphoreSlim. Классы Monitor и Mutex. Сообщения ManualResetEvent, AutoResetEvent. Классы SpinLock и SpinWait.

Средства синхронизации

Синхронизация необходима для координации выполнения потоков. Такая координация необходима для согласования порядка выполнения потоков или для согласования доступа потоков к разделяемому ресурсу.

Среда Framework .NET предоставляет широкий набор средств синхронизации.

Блокировка	Join, Sleep, SpinWait
Взаимно-исключительный доступ	Lock, Monitor, Mutex, SpinLock
Сигнальные сообщения	AutoResetEvent, ManualResetEvent, ManualResetEventSlim
Семафоры	Semaphore, SemaphoreSlim
>Атомарные операторы	Interlocked
Конкурентные коллекции	ConcurrentBag, ConcurrentQueue, ConcurrentDictionary, ConcurrentStack, BlockedCollection
Блокировки чтения-записи	ReaderWriterLock, ReaderWriterLockSlim
Шаблоны синхронизации	Barrier, CountdownEvent

В основе синхронизации лежит понятие блокировки – один поток блокируется в ожидании определенного события от других потоков, например, завершения работы определенного потока или освобождения

разделяемого ресурса.

Ожидание может быть активным или пассивным. При активном "ожидании" поток циклически проверяет статус ожидаемого события.

```
Thread thr = new Thread(SomeWork);
thr.Start();
while(thr.IsAlive) ;
```

Такая блокировка называется активной, так как фактически поток не прекращает своей работы и не освобождает процессорное время для других потоков. Активное ожидание эффективно только при незначительном времени ожидания.

Пассивное ожидание реализуется с помощью операционной системы, которая сохраняет контекст потока и выгружает его, предоставляя возможность выполнятся другим потокам. При наступлении ожидаемого события операционная система "будит" поток – загружает контекст потока и выделяет ему процессорное время. Пассивное ожидание требует времени на сохранение контекста потока при блокировке и загрузку контекста при возобновлении работы потока, но позволяет использовать вычислительные ресурсы во время ожидания для выполнения других задач.

В следующем фрагменте используются два типа ожидания. В первом случае применяется циклическая проверка статуса. Во втором случае используется метод `Join`.

```
class Program
{
    static bool b;
    static double res;
    static void SomeWork()
    {
        for (int i=0; i<100000; i++)
            for(int j=0; j<20; j++)
                res += Math.Pow(i, 1.33);
        b = true;
    }
}
```

```

static void Main()
{
    Thread thr1 = new Thread(SomeWork);
    thr1.Start();
    // Активное ожидание в цикле
    while(!b) ;
    Console.WriteLine("Result = " + res);

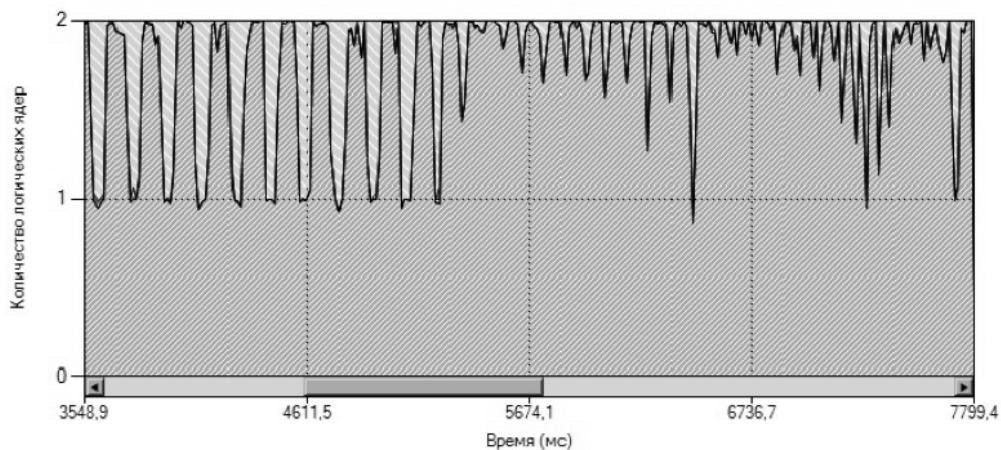
    res = 0;
    Thread thr2 = new Thread(SomeWork);
    thr2.Start();
    // Ожидание с выгрузкой контекста
    thr2.Join();

}
}

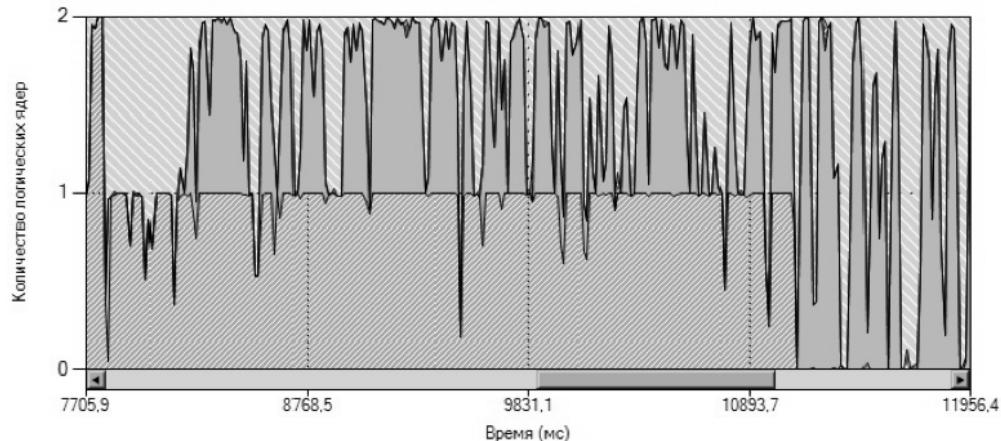
```

Анализ выполнения программы с помощью инструмента Visual Studio 12 "Визуализатор параллелизма" позволяет зафиксировать особенности загрузки вычислительной системы.

Загрузка процессора при активном ожидании в среднем равна 92%



При пассивном ожидании, основной поток выгружается и занятость ЦП в среднем равна 50%.



Существуют гибридные средства синхронизации, сочетающие в себе достоинства активного и пассивного ожидания. Гибридную блокировку используют объекты синхронизации, введенные в .NET 4.0: SpinWait, SpinLock, SemaphoreSlim, ManualResetEventSlim, ReaderWriterLockSlim и др. Потоки, блокируемые с помощью гибридных средств синхронизации, в начале фазы ожидания находятся в активном состоянии – не выгружаются, циклически проверяют статус ожидаемого события. Если ожидание затягивается, то активная блокировка становится не эффективной и операционная система выгружает ожидающий поток.

Средства для взаимного исключения

Одно из основных назначений средств синхронизации заключается в организации взаимно исключающего доступа к разделяемому ресурсу. Изменения общих данных одним потоком не должны прерываться другими потоками. Фрагмент кода, в котором осуществляется работа с разделяемым ресурсом и который должен выполняться только в одном потоке одновременно, называется критической секцией.

```
public string data;
void DoSomeWork1()
{
    Thread.Name = "First";
    data = "AAAA";
    Console.WriteLine("Thread: {0}, Data: {1}", Thread.Name, data);
```

```
        }
```

```
void DoSomeWork2()
{
    Thread.Name = "Second";
    data = "BBBB";
    Console.WriteLine("Thread: {0}, Data: {1}",
        Thread.Name, data);

}
```

В этом фрагменте предполагается, что два потока изменяют значение общей переменной `data`. После внесения изменений поток выводит на экран новое значение `data` и имя потока. Отсутствие средств синхронизации могло бы привести к некорректному результату:

```
Thread: First, Data: BBBBB
Thread: Second, Data: BBBBB
```

Поток `First` внес свои изменения и собирался вывести сообщение, но второй поток успел вклиниваться и изменить данные.

Выделение критической секции с помощью конструкции `lock` позволяет избежать такой ситуации:

```
lock(sync_obj)
{
    data = "AAAA";
    Console.WriteLine("Thread #1 has changed
        data to: {0}", data);
}
```

Когда один поток входит в критическую секцию (захватывает объект синхронизации), другой поток ожидает завершения всего блока (освобождения объекта синхронизации). Если заблокировано было несколько потоков, то при освобождении критической секции только один поток разблокируется и входит критическую секцию.

Для выделения критической секции с помощью конструкции `lock` необходимо указать объект синхронизации, который выступает в качестве идентификатора блокировки.

Monitor

Конструкция `lock` введена для удобства как аналог применения объекта синхронизации `Monitor`.

Итак, конструкция

```
lock(sync_obj)
{
    // Critical section
}
```

аналогична применению объекта `Monitor`:

```
try
{
    Monitor.Enter(sync_obj);
    // Critical section
}
finally
{
    Monitor.Exit(sync_obj);
}
```

Блоки `try-finally` формируются для того, чтобы гарантировать освобождение блокировки (критической секции) в случае возникновения какого-либо исключения внутри критической секции.

Кроме "обычного" входа в критическую секцию класс `Monitor` предоставляет "условные" входы:

```
b = Monitor.TryEnter(sync_obj);
if(!b)
```

```
{  
    // Выполняем полезную работу  
    DoWork();  
    // Снова пробуем войти в критическую секцию  
    Monitor.Enter(sync_obj);  
}  
// Критическая секция  
ChangeData();  
// Выходим  
Monitor.Exit(sync_obj);
```

Если критическая секция уже выполняется кем-то другим, то поток не блокируется, а выполняет полезную работу. После завершения всех полезных работ поток пытается войти в критическую секцию с блокировкой.

Если доступ к критической секции достаточно интенсивен со стороны множества потоков, то полезным может быть метод `TryEnter` с указанием интервала в миллисекундах, в течение которого поток пытается захватить блокировку.

```
while(! Monitor.TryEnter(sync_obj, 100))  
{  
    // Полезная работа  
    DoWork();  
}  
// Критическая секция  
ChangeData();  
// Выходим  
Monitor.Exit(sync_obj);
```

Объект `Monitor` также предоставляет методы для обмена сигналами с ожидающими потоками `Pulse` и `Wait`, которые могут быть полезны для предотвращения взаимоблокировки в случае работы с несколькими разделяемыми ресурсами.

В следующем фрагменте два потока пытаются захватить ресурсы `P` и `Q`.

Первый поток захватывает сначала P, затем пытается захватить Q. Второй поток сначала захватывает ресурс Q, а затем пытается захватить P. Применение обычной конструкции lock привело бы в некоторых случаях к взаимоблокировке потоков – потоки успели захватить по одному ресурсу и пытаются получить доступ к недостающему ресурсу. Следующий фрагмент решает проблему с помощью объекта Monitor:

```
void ThreadOne()
{
    // Получаем доступ к ресурсу P
    Monitor.Enter(P);
    // Пытаемся захватить ресурс Q
    if(!Monitor.TryEnter(Q))
    {
        // Если Q занят другим потоком,
        // освобождаем P и
        // ожидаем завершения работы потока
        Monitor.Wait(P);
        // Освободился ресурс P, смело захватываем и Q
        Monitor.Enter(Q);
    }
    // Теперь у потока есть и P, и Q, выполняем работу
    ..
    // Освобождаем ресурсы в обратной последовательности
    Monitor.Exit(Q);
    Monitor.Exit(P);
}

void ThreadTwo()
{
    Monitor.Enter(Q);
    Monitor.Enter(P);
    // Выполняем необходимую работу
    ..
    // Обязательный сигнал для потока, который
    // заблокировался при вызове Monitor.Wait(P)
    Monitor.Pulse(P);
    Monitor.Exit(P);
    Monitor.Exit(Q);
```

Первый поток после захвата ресурса P пытается захватить Q. Если Q уже занят, то первый поток, зная, что второму нужен еще и P, освобождает его и ждет завершения работы второго потока с обеими ресурсами. Вызов Wait блокирует первый поток и позволяет другому потоку (одному из ожидающих) войти в критическую секцию для работы с ресурсом P. Работа заблокированного потока может быть продолжена после того как выполняющийся поток вызовет метод Pulse и освободит критическую секцию. Таким образом, первый поток возобновляет работу не после вызова Pulse, а после вызова Exit(P).

Mutex

Объект Mutex используется, как и Monitor, для обеспечения взаимно-исключительного доступа к фрагменту кода. В основе объекта Mutex лежит вызов функции ядра операционной системы, и поэтому блокировка с помощью Mutex является менее эффективной по сравнению с классом Monitor и конструкцией lock.

Отличие от Monitor заключается в возможности использования глобальных именованных блокировок, доступных в рамках нескольких приложений. Таким образом, с помощью объекта Mutex можно организовать синхронизацию нескольких приложений. Ядро операционной системы контролирует взаимную исключительность выполнения критических секций. Одним из примеров использования межпроцессной синхронизации является контроль количества запущенных копий приложения.

```
class MyApplication
{
    static void Main()
    {
        var mutex = new Mutex(false, "MyApp ver 2.0");

        if(!mutex.WaitOne(TimeSpan.FromSeconds(5), false))
```

```
{  
    Console.WriteLine("Приложение уже запущено");  
    return;  
}  
Run();  
mutex.Dispose();  
}  
  
static void Run ()  
{  
    Console.WriteLine("Welcome to MyApp ver 2.0");  
    ..  
}  
}
```

В этом фрагменте создается "именованный" мьютекс. Вызов метода `WaitOne` позволяет одному потоку или процессу войти в критическую секцию. Вызов метода `WaitOne` для дополнительных копий приложения возвращает признак занятости критической секции. По этому признаку приложение распознает факт, что копия уже запущена. Временной интервал в 5с используется для случаев, если запущенное приложение завершает выполнение. Освобождение мьютекса не выполняется, так как нет необходимости передавать управление другому процессу. Второй аргумент в вызове `WaitOne` указывает, что используется ожидание глобального мьютекса.

Сигнальные сообщения

Сигнальные сообщения позволяют реализовать разные схемы синхронизации, как взаимное исключение, так и условную синхронизацию. При условной синхронизации поток блокируется в ожидании события, которое генерируется в другом потоке. Платформа .NET предоставляет три типа сигнальных сообщений: `AutoResetEvent`, `ManualResetEvent` и `ManualResetEventSlim`, а также шаблоны синхронизации, построенные на сигнальных сообщениях (`CountdownEvent`, `Barrier`). Первые два типа построены на объекте ядра

операционной системы. Третий тип `ManualResetEventSlim` является облегченной версией объекта `ManualResetEvent`, является более производительным.

В следующем фрагменте два потока используют один и тот же объект типа `ManualResetEvent`. Первый поток выводит сообщение от второго потока. Сообщение записывается в разделяемую переменную. Вызов метода `WaitOne` блокирует первый поток в ожидании сигнала от второго потока. Сигнал генерируется при вызове метода `Set`.

```
void OneThread(object o)
{
    ManualResetEvent mre = (ManualResetEvent)o;
    mre.WaitOne();
    Console.WriteLine("Data from thread #2: " + data);
}

void SecondThread(object o)
{
    ManualResetEvent mre = (ManualResetEvent)o;
    Console.WriteLine("Writing data");
    data = "BBBBBB";
    mre.Set();
}
```

Вывод:

```
Writing data..
Data from thread#2: BBBB
```

Отличия инструментов `AutoResetEvent` и `ManualResetEvent` заключаются в режиме сброса статуса сигнального события: автоматическое (`auto reset`) или ручное (`manual reset`). Сигнал с автоматическим сбросом снимается сразу же после освобождения потока, блокированного вызовом `WaitOne`. Сигнал с ручным сбросом не снимается до тех пор, пока какой-либо поток не вызовет метод `Reset`.

В следующем фрагменте рассматриваются отличия сигнальных сообщений. Управляющий поток Manager запускает пять рабочих потоков и каждому передает один и тот же сигнальный объект. Рабочие потоки ожидают сигнала от управляющего потока.

```
void Worker(object initWorker)
{
    string name = ((object[])initWorker)[0] as string;
    ManualResetEvent mre =
        (object[])initWorker[1] as ManualResetEvent;
    // Waiting to start work
    mre.WaitOne();
    Console.WriteLine("Worker {0} starts ..", name);
    // useful work
}
void Manager()
{
    int nWorkers = 5;
    Thread[] worker = new Thread[nWorkers];
    ManualResetEvent mre = new ManualResetEvent(false);
    for(int i=0; i<nWorkers; i++)
    {
        worker[i] = new Thread(Worker);
        worker[i].Start(new object[]{"#" + i, mre});
    }
    // preparing data in shared variables for workers
    // let start work
    mre.Set();
}
```

При установлении события `mre` работу начнут все ожидающие рабочие потоки. При замене объекта на `AutoResetEvent` событие будет сбрасываться автоматически и "поймает" его только какой-то один поток. Таким образом, объект `AutoResetEvent` можно использовать для реализации взаимно исключительного доступа.

```
static void ThreadFunc(object o)
{
```

```
var lockEvent = o as AutoResetEvent;
ParallelWork();

lockEvent.WaitOne();
CriticalWork();
lockEvent.Set();

}

static void Main()
{
    Thread[] workers = new Thread[5];
    for(int i=0; i<5; i++)
        workers[i] = new Thread(ThreadFunc);
    var lockEvent = new AutoResetEvent(true);
    for(int i=0; i<5; i++)
        workers[i].Start(lockEvent);

}
```

В этом примере пять рабочих потоков часть работы могут выполнять параллельно, но какой-то фрагмент должны выполнять последовательно (критическая секция). Сообщение типа `AutoResetEvent` используется для организации взаимоисключительного доступа к критической секции. Объект инициализируется с установленным сигналом для того, чтобы вначале работы один из потоков вошел в критическую секцию. При завершении выполнения критической секции поток дает сигнал одному из ожидающих потоков. Порядок вхождения потоков в критическую секцию, также как и при использовании объекта `Monitor`, не определен.

Объект `ManualResetEventSlim` функционально соответствует сигнальному событию с ручным сбросом. Применение гибридной блокировки повышает производительность в сценариях с малым временем ожидания. Вызов метода `Wait` в течение ограниченного промежутка времени сохраняет поток в активном состоянии (циклическая проверка статуса сигнала), если сигнал не поступил, то осуществляется вызов дескриптора ожидания ядра операционной

Объекты `AutoResetEvent`, `ManualResetEvent`, а также объекты `Semaphore`, `Mutex` происходят от объекта, инкапсулирующего дескриптор ожидания ядра `WaitHandle`. Тип `WaitHandle` содержит полезные статические методы ожидания нескольких объектов синхронизации ядра:

```
var ev1 = new ManualResetEvent(false);
var ev2 = new ManualResetEvent(false);
new Thread(SomeFunc).Start(ev1);
new Thread(SomeFunc).Start(ev2);
// Ожидаем все сигналы
WaitHandle.WaitAll(new ManualResetEvent[] {ev1, ev2});
ev1.Reset(); ev2.Reset();
// Ожидаем хотя бы один сигнал
int iFirst = WaitHandle.WaitAny(new ManualResetEvent[]
{ev1, ev2});
```

Семафоры

Объект синхронизации `Semaphore` отличается от сигнальных событий наличием внутреннего счетчика с устанавливаемым максимальным значением. Объект `AutoResetEvent` можно интерпретировать как семафор с максимальным счетчиком равным 1 (двоичный семафор).

В следующем фрагменте рассматривается применение семафоров. В коде используется объект `SemaphoreSlim`. Вместо него можно использовать объект `Semaphore`.

```
// Применение семафоров
class SemaphoreSlimTesting
{
    private static SemaphoreSlim sem;
    private static void Worker(object num)
    {
        // Ждем сигнала от управляющего
```

```
sem.Wait();
// Начинаем работу
Console.WriteLine("Worker {0} starting", num);
}

private static void Main()
{
    // Максимальная емкость семафора: 5
    // Начальное состояние: 0 (все блокируются)
    sem = new SemaphoreSlim(0, 5);
    Thread[] workers = new Thread[10];
    for(int i=0; i<workers.Length; i++)
    {
        workers[i] = new Thread(Worker);
        workers[i].Start(i);
    }
    Thread.Sleep(300);
    Console.WriteLine("Разрешаем работу трем рабочим");
    sem.Release(3);
    Thread.Sleep(200);
    Console.WriteLine("Разрешаем работу еще двум рабочим");
    sem.Release(2);

}
}
```

В методе `Main` инициализируется семафор `SemaphoreSlim`. Начальное значение внутреннего счетчика равно 0, максимальное значение – 5. Рабочие потоки блокируются, так как счетчик семафора равен нулю. Главный поток увеличивает счетчик на три единицы, тем самым освобождая три потока. После небольшой паузы главный поток освобождает еще два потока.

Вывод программы:

```
Разрешаем работу трем рабочим
Worker 9 starting
Worker 6 starting
Worker 0 starting
```

Разрешаем работу еще двум рабочим

Worker 8 starting

Worker 1 starting

Атомарные операторы

Библиотека .NET 4.0 предоставляет высокоэффективные атомарные операторы, которые реализованы как статические методы класса `System.Threading.Interlocked`. Атомарные операторы предназначены для потокобезопасного неблокирующего выполнения операций над данными, преимущественно целочисленного типа.

Оператор	Метод	Типы данных
Увеличение счетчика на единицу	Increment	Int32, Int64
>Уменьшение счетчика на единицу	Decrement	Int32, Int64
Добавление	Add	Int32, Int64
Обмен значениями	Exchange	Int32, Int64, double, single, object
Условный обмен	CompareExchange	Int32, Int64, double, single, object
Чтение 64-разрядного целого	Read	Int64

Атомарность означает, что при выполнении оператора никто не вмешается в работу потока. Функционально, атомарные операторы равносильны критической секции, выделенной с помощью `lock`, `Monitor` или других средств синхронизации.

```
lock (sync_obj)
{
    counter++;
}
```

```
    }  
    // можно выполнить с помощью атомарного оператора  
    Interlocked.Increment(ref counter);
```

Атомарные операторы являются неблокирующими - поток не выгружается и не ожидает, поэтому обеспечивают высокую эффективность. Выполнение оператора `Interlocked` занимает вдвое меньшее время, чем выполнение критической секции с `lock`-блокировкой без конкуренции.

Оператор `Interlocked.CompareExchange` позволяет атомарно выполнить конструкцию "проверить-присвоить":

```
lock(LockObj)  
{  
    if(x == curVal)  
        x = newVal;  
}
```

```
oldVal = Interlocked.CompareExchange(ref x, newVal, curVal);
```

Если значение переменной `x` равно значению, задаваемому третьим аргументом `curVal`, то переменной присваивается значение второго аргумента `newVal`. Возвращаемое значение позволяет установить, осуществилась ли замена значения.

Атомарный оператор `Read` предназначен для потокобезопасного чтения 64-разрядных целых чисел (`Int64`). Чтение значений типа `long` (`Int64`) на 32-разрядной вычислительной системе не является атомарной операцией на аппаратном уровне. Поэтому многопоточная работа с 64-разрядными переменными может приводить к некорректным результатам. В следующем фрагменте проиллюстрируем проблему чтения переменных типа `Int64`:

```
Int64 bigInt = Int64.MinValue;  
Thread t = new Thread(() => {  
    while(true) {
```

```
if(bigInt == Int64.MinValue)
    bigInt = Int64.MaxValue;
else
    bigInt = Int64.MinValue;
});
}
t.Start();
List<Int64> lstBig = new List<Int64>();
for(int i=0; i < 1000; i++)
{
    Thread.Sleep(100);
    lstBig.Add(bigInt);
}
t.Abort();

Console.WriteLine("Distinct values: "
+ lstBig.Distinct().Count());
lstBig.Distinct().AsParallel().ForAll(Console.WriteLine);
```

В этом примере значение переменной `bigInt` изменяется только в одном потоке. Основной поток периодически читает текущие значения `bigInt`. Поток `t` циклически меняет значение переменной `bigInt` с `MinValue` на `.MaxValue` и с `.MaxValue` на `MinValue`. Тем не менее, вывод показывает, что основной поток прочитал и другие значения. Эти "промежуточные" значения появились из-за не атомарности действий над 64-разрядными переменными – пока основной поток прочитал первые 32 бита числа, дочерний поток изменил следующие 32 бита. Предпоследняя строчка выводит число различных значений переменной `bigInt`, прочитанных в основном потоке. Последняя строчка выводит на консоль все различные значения.

```
Distinct values: 4
-9223372036854775808
-9223372032559808513
9223372036854775807
9223372032559808512
```

Для устранения проблемы необходимо сделать атомарным запись и чтение переменной `bigInt`:

```
Int64 bigInt = Int64.MinValue;
Thread t = new Thread(() => {
    Int64 oldValue = Interlocked.CompareExchange(ref bigInt,
        Int64.MinValue, Int64.MaxValue);
    Interlocked.CompareExchange(ref bigInt,
        Int64.MaxValue, oldValue);
});
t.Start();
List<Int64> lstBig = new List<64>();
for(int i=0; i < 1000; i++)
{
    Thread.Sleep(100);
    lstBig.Add(Interlocked.Read(ref bigInt));
}
t.Abort();

Console.WriteLine("Distinct values: "
    + lstBig.Distinct().Count());
lstBig.Distinct().AsParallel().ForAll(Console.WriteLine);
```

Изменение `bigInt` реализовано с помощью двух операторов `CompareExchange`. Первый оператор пытается присвоить `MinValue`, если текущее значение равно `MaxValue`. Оператор возвращает старое значение. Сравнивая текущее со старым значением, определяем, произошло ли изменение. Если изменения не было, то присваиваем максимальное значение `MaxValue`. Атомарное чтение реализовано с помощью оператора `Interlocked.Read`. Вывод результатов свидетельствует о решении проблемы:

```
Distinct value: 2
-9223372036854775808
9223372036854775807
```

Операции над 64 разрядными целыми на 64-разрядной системе являются атомарными на аппаратном уровне, поэтому не требуют средств синхронизации при параллельной записи и чтении. Но при параллельной записи в нескольких потоках, возникает проблема гонки данных.

Вопросы

1. Можно ли организовать работу нескольких потоков без средств синхронизации?
2. Может ли многопоточное приложение, использующее только конструкции `lock`, войти в состояние взаимоблокировки?
3. Какие средства синхронизации позволяют реализовать функциональность критической секции? В каких случаях следует отдавать предпочтение тому или иному объекту?

Упражнения

1. Исследуйте эффективность легковесных средств синхронизации по сравнению с аналогичными объектами ядра операционной системы: `SemaphoreSlim` – `Semaphore`, `ManualResetEvent` – `ManualResetEventSlim`.
2. Для анализа можно использовать задачу обращения к разделяемому счетчику:

```
void ThreadFunc() {
    // Вход в критическую секцию с помощью
    // того или иного средства синхронизации
    totalCount++;
    // Выход из критической секции
}
```

3. Исследуйте эффективность потокобезопасных коллекций по сравнению с синхронизированным доступом к обычным коллекциям.
4. Исследуйте эффективность атомарных операторов по сравнению со средствами организации критической секции (`lock`, `Monitor`, `Mutex`).

5. Самостоятельно освойте работу с объектами, реализующими типовые схемы синхронизации, CountdownEvent и Barrier. Реализуйте функциональность этих объектов с помощью средств синхронизации, рассмотренных в лекции. Исследуйте эффективность и удобство работы объектов CountdownEvent и Barrier.

Поиск простых чисел

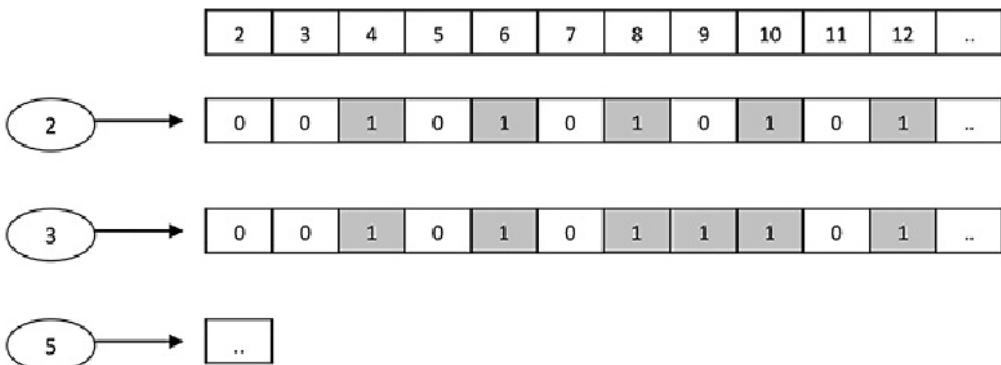
Применение средств синхронизации для организации взаимодействия потоков.

Задачи:

реализовать последовательный и параллельные алгоритмы поиска простых чисел; выполнить анализ быстродействия алгоритмов при разном объеме данных, разном числе потоков; рассчитать ускорение и эффективность выполнения алгоритмов; сделать выводы о целесообразности применения параллельных алгоритмов и необходимости использования синхронизации.

Последовательный алгоритм "Решето Эратосфена".

Алгоритм заключается в последовательном переборе уже известных простых чисел, начиная с двойки, и проверке разложимости всех чисел диапазона $(m, n]$ на найденное простое число m . На первом шаге выбирается число $m = 2$, проверяется разложимость чисел диапазона $(2, n]$ на 2-ку. Числа, которые делятся на двойку, помечаются как составные и не участвуют в дальнейшем анализе. Следующим непомеченным (простым) числом будет $m = 3$, и так далее.



При этом достаточно проверить разложимость чисел на простые числа в интервале $(2, \sqrt{n}]$. Например, в интервале от 2 до 20 проверяем все

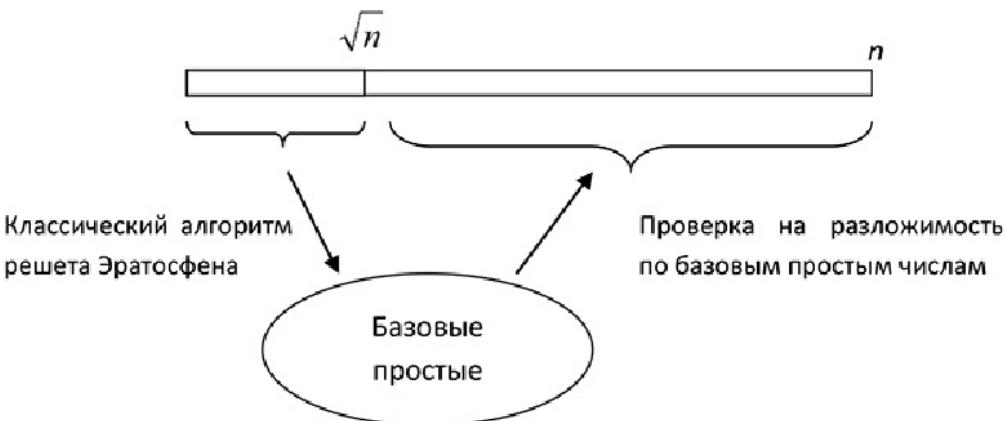
числа на разложимость 2, 3. Составных чисел, которые делятся только на пятерку, в этом диапазоне нет.

Модифицированный последовательный алгоритм поиска

В последовательном алгоритме "базовые" простые числа определяются поочередно. После тройки следует пятерка, так как четверка исключается при обработке двойки. Последовательность нахождения простых чисел затрудняет распараллеливание алгоритма. В модифицированном алгоритме выделяются два этапа:

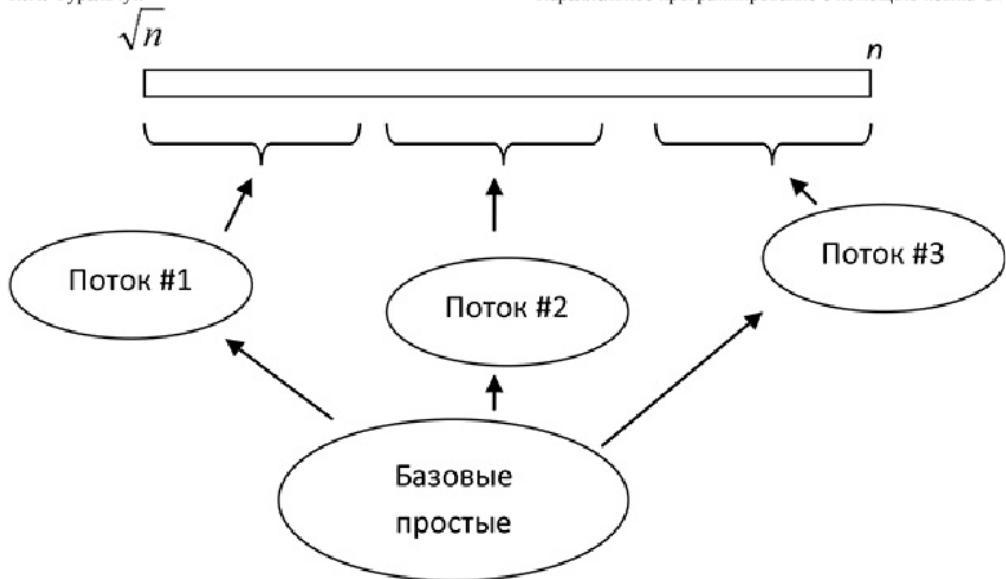
1-ый этап: поиск простых чисел в интервале от $2 \dots \sqrt{n}$ с помощью классического метода решета Эратосфена (базовые простые числа).

2-ой этап: поиск простых чисел в интервале от $\sqrt{n} \dots n$, в проверке участвуют базовые простые числа, выявленные на первом этапе.



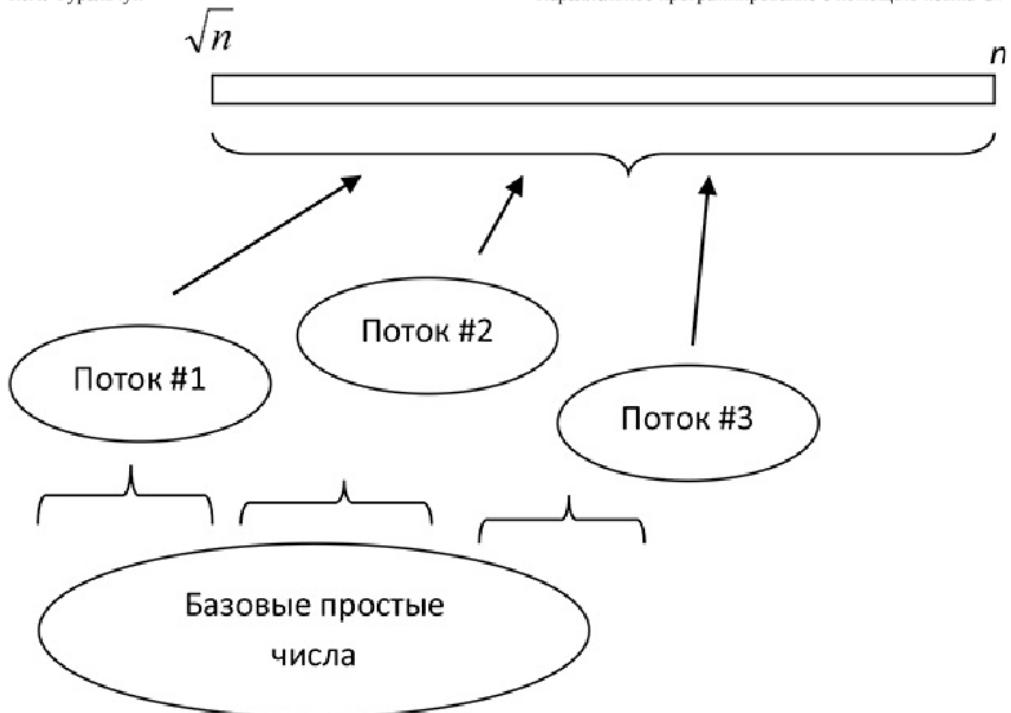
На первом этапе алгоритма выполняется сравнительно небольшой объем работы, поэтому нецелесообразно распараллеливать этот этап. На втором этапе проверяются уже найденные базовые простые числа. Параллельные алгоритмы разрабатываются для второго этапа.

Параллельный алгоритм №1: декомпозиция по данным



Идея распараллеливания заключается в разбиении диапазона $\sqrt{n} \dots n$ на равные части. Каждый поток обрабатывает свою часть чисел, проверяя на разложимость по каждому базовому простому числу.

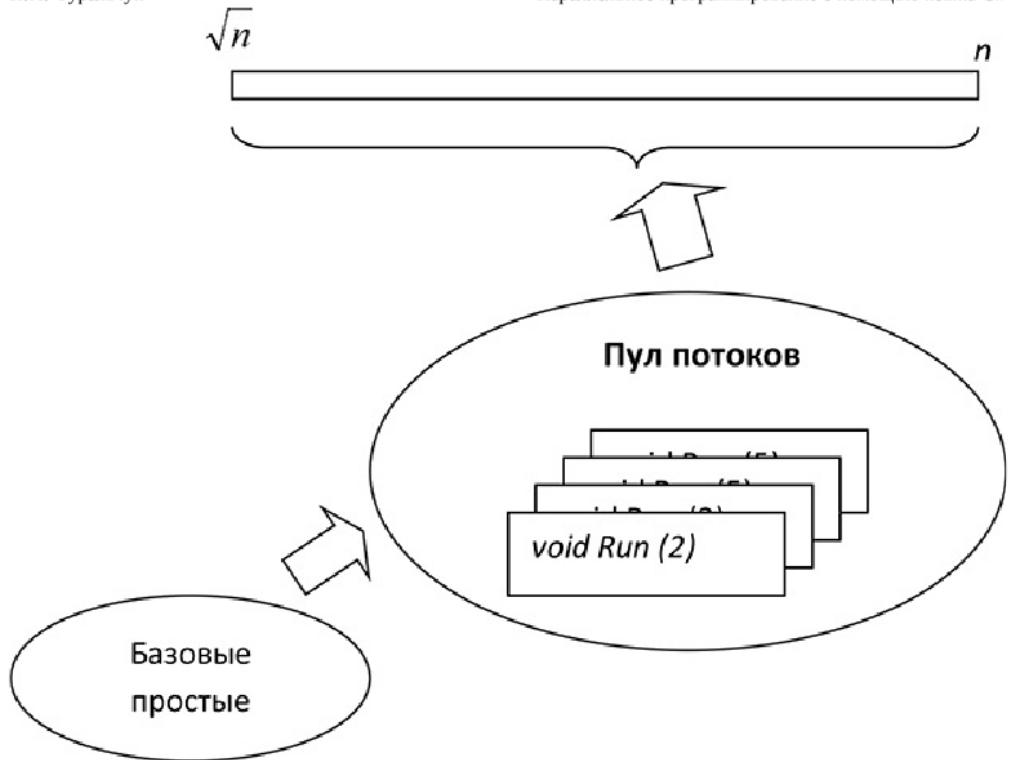
Параллельный алгоритм №2: декомпозиция набора простых чисел



В этом алгоритме разделяются базовые простые числа. Каждый поток работает с ограниченным набором простых чисел и проверяет весь диапазон $\sqrt{n} \dots n$.

Параллельный алгоритм №3: применение пула потоков

Применение пула потоков позволяет автоматизировать обработку независимых рабочих элементов. В качестве рабочих элементов предлагается использовать проверку всех чисел диапазона от $\sqrt{n} \dots n$ на разложимость по одному базовому простому числу.



Для применения пула потоков необходимо загрузить рабочие элементы вместе с необходимыми параметрами в очередь пула потоков:

```
for(int i=0; i<basePrime.Length; i++)
{
    ThreadPool.QueueUserWorkItem(Run, basePrime[i]);
}
```

`Run` – метод обработки всех чисел диапазона $\sqrt{n} \dots n$ на разложимость простому числу `basePrime[i]`.

Выполнение рабочих элементов осуществляется автоматически после добавления в пул потоков. Не существует встроенного механизма ожидания завершения рабочих элементов, добавленных в пул потоков. Поэтому вызывающий поток (метод `Main`) должен контролировать завершение либо с помощью средств синхронизации (например, сигнальных сообщений), либо с помощью общих переменных и цикла

ожидания в методе Main.

Применение сигнальных сообщений может быть реализовано следующим образом:

```
static void Main()
{
    // Поиск базовых простых
    ..
    int[] basePrime = ..;

    // Объявляем массив сигнальных сообщений
    ManualResetEvent [] events =
        new ManualResetEvent [basePrime.Length];

    // Добавляем в пул рабочие элементы с параметрами
    for(int i=0; i<basePrime.Length; i++)
    {
        events[i] = new ManualResetEvent(false);
        ThreadPool.QueueUserWorkItem(Run,
            new object[] {basePrime[i], events[i]});
    }
    // Дожидаемся завершения
    WaitHandle.WaitAll(events);

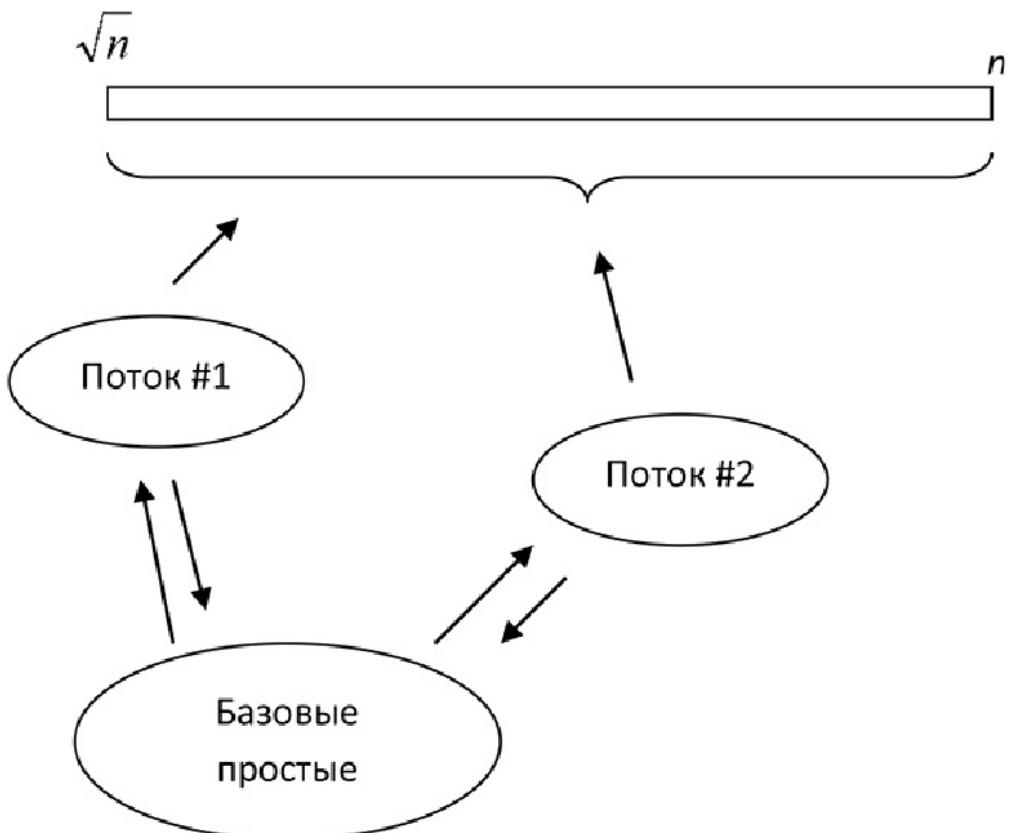
    // Выводим результаты
    ..
}

static void F(object o)
{
    int prime = (int)((object[])o)[0];
    ManualResetEvent ev = ((object[])o)[1] as ManualResetEvent;
    // Обработка чисел на разложимость простому числу prime
    ..
    ev.Set();
}
```

Параллельный алгоритм №4: последовательный

перебор простых чисел

Идея алгоритма заключается в последовательном переборе базовых простых чисел разными потоками. Каждый поток осуществляет проверку всего диапазона на разложимость по определенному простому числу. После обработки первого простого числа поток не завершает работу, а обращается за следующим необработанным простым числом.



Для получения текущего простого числа поток выполняет несколько операторов:

```
while(true)
{
    if (current_index >= basePrime.Length)
        break;
    current_prime = basePrime[current_index];
```

```
current_index++;
// Обработка текущего простого числа
..
}
```

В этой реализации существует разделяемый ресурс – массив простых чисел. При одновременном доступе к ресурсу возникает проблема гонки данных. Следствием этой проблемы являются: лишняя обработка, если несколько потоков одновременно получают одно и то же число; пропущенная задача - потоки, получив одно число, последовательно увеличивают текущий индекс; исключение "Выход за пределы массива", когда один поток успешно прошел проверку текущего индекса, но перед обращением к элементу массива, другой поток увеличивает текущий индекс.

Для устранения проблем с совместным доступом необходимо использовать средства синхронизации (критические секции, атомарные операторы, потокобезопасные коллекции).

Критическая секция позволяет ограничить доступ к блоку кода, если один поток уже начал выполнять операторы секции:

```
lock (sync_obj)
{
    критическая_секция
}
```

где `sync_obj` – объект синхронизации, идентифицирующий критическую секцию (например, строковая константа).

Вопросы и упражнения

1. Какими достоинствами и недостатками обладает каждый вариант распараллеливания?
2. Какие средства синхронизации можно использовать вместо конструкции `lock`? Какой вариант будет более эффективным?
3. Какой вариант ожидания завершения работ, запущенных пулом

потоков, более эффективный и почему?

4. Реализуйте один или несколько вариантов распараллеливания с помощью объектов Task и с помощью метода Parallel.For. Выполните эффективность алгоритмов.
5. Реализуйте алгоритм поиска простых чисел как LINQ-запрос к массиву чисел.

Конкурентные коллекции

Потокобезопасные коллекции пространства имен
System.Collections.Concurrent

Динамические структуры данных пространства System.Collections и System.Collections.Generic не являются потокобезопасными. Обращение к коллекции нескольких потоков может приводить к проблемам гонки данных: потеря элементов, выход индекса за пределы и аварийное завершение работы приложения.

Рассмотрим многопоточную работу с объектом Dictionary< TKey, TValue>. При одновременном инкрементировании элемента с ключом "one" возможно наложение изменений, которое приводит к затиранию изменения одного потока изменением другого потока. При инкрементировании затирание приводит к меньшим значениям счетчика.

```
// Создаем обычный словарь
var dic = new Dictionary<string, int>();
// Параллельно обновляем значение элемента с ключом "one"
Parallel.For(0, 100000, i=>
{
    if(dic.ContainsKey("one"))
        dic["one"]++;
    else
        dic.Add("one", 1);
});
Console.WriteLine("Element \"one\": {0}", dic["one"]);
```

Выполняя этот фрагмент, часто получаем правильный ответ: 100000. Тем не менее, встречаются и 86675, и 56670, и 92030. Возможен вариант возникновения исключения с ошибкой выхода индекса за границы.

Для обеспечения потокобезопасного доступа можно использовать средства синхронизации, рассмотренные выше.

```

var lst = new List<int>();
Parallel.For(0, 100000, i =>
{
    lock("sync")
{
    lst.Add(i);
}
});

```

Более эффективным является применение конкурентных коллекций, обеспечивающие потокобезопасность операций добавления и удаления элементов. Конкурентные коллекции реализованы с применением легковесных средств синхронизации и по возможности избегают блокировок там, где они не нужны.

ConcurrentQueue	FIFO-очередь
ConcurrentStack	LIFO-стэк
ConcurrentBag	Неупорядоченная коллекция
ConcurrentDictionary	Словарь
BlockingCollection	Ограниченнная коллекция

В следующем фрагменте осуществляем многопоточную работу с разделяемым списком типа ConcurrentBag:

```

var bag = new ConcurrentBag<int>();
Parallel.For(0, 100000, i =>
{
    bag.Add(i);
});

```

Применение конкурентных коллекций не требует использования дополнительных средств синхронизации.

Обычные коллекции не позволяют изменять объект, который используется в foreach-перечислении. Конкурентные коллекции, обеспечивая многопоточный доступ, позволяют добавлять

элементы внутри цикла `foreach` при переборе элементов. При этом изменения, вносимые внутри перечисления, не отражаются на текущем перечислителе:

```
var bag = new ConcurrentBag<int>();  
for(int i=0; i<10; i++)  
    bag.Add(i);  
  
foreach(int k in bag)  
{  
    bag.Add(k);  
    Console.WriteLine(bag.Count + " ");  
}
```

В этом фрагменте в `foreach`-цикле осуществляем добавление новых элементов и вывод размера коллекции. Получаем вывод 10 итераций, но размер коллекции при этом увеличивается до 20:

```
11 12 13 14 15 16 17 18 19 20
```

Замена конкурентной коллекции `ConcurrentBag` на список `List` привела бы к возникновению необработанного исключения.

Эффективность конкурентных коллекций

Конкурентные коллекции спроектированы для применения в многопоточных сценариях и не являются в полной мере эквивалентом обычных коллекций с блокировками. В случае однопроцессорной системы или при использовании конкурентных коллекций в одном потоке их эффективность может быть ниже, чем использование обычных коллекций. Поэтому рекомендуется использовать конкурентные коллекции только в случае многопоточной работы.

Интерфейс `IProducerConsumerCollection<T>`

Все конкурентные коллекции реализуют интерфейс "производитель-

"потребитель". Основные операции интерфейса TryAdd и TryTake проверяют возможность операций записи/извлечения и в случае наличия возможности осуществляют эти действия. Операции осуществляются атомарно, то есть потокобезопасно. Если какой-то поток убедился в возможности извлечения элемента, то другой поток не сможет вмешаться до завершения первым потоком операции извлечения элемента.

Метод TryTake возвращает `false` в случае, если коллекция пуста. Метод TryAdd для конкурентных коллекций всегда возвращает `true` и успешно завершает добавление элемента.

Методы TryTake для конкурентных стеков и очередей возвращают элементы в определенном порядке – для `ConcurrentStack` получаем последний добавленный элемент, для `ConcurrentQueue` получаем первый добавленный элемент.

ConcurrentBag<T>

Коллекция `ConcurrentBag<T>` предназначена для хранения неупорядоченной коллекции объектов (повторы разрешены). Отсутствие определенного порядка извлечения элементов повышает производительность операции чтения и добавления для `ConcurrentBag`. Добавление элементов в коллекции `ConcurrentStack` или `ConcurrentQueue` в нескольких потоках приводит к дополнительным накладным расходам (неблокирующая синхронизация). Объект `ConcurrentBag` одинаково эффективен при однопоточном добавлении и при многопоточном.

Внутри объекта `ConcurrentBag` содержатся связанные списки для каждого потока. Элементы добавляются в тот список, который ассоциирован с текущим потоком. При извлечении элементов сначала опустошается локальная очередь данного потока. Если в локальной очереди содержатся элементы, то извлечение является максимально эффективным. Если локальная очередь пуста, то поток заимствует элементы из локальных очередей других потоков. Таким образом, извлечение элементов из `ConcurrentBag` осуществляется по принципу LIFO с учетом локальности.

BlockingCollection<T>

Объект `BlockingCollection` позволяет сформировать модифицированную конкурентную коллекцию на базе `ConcurrentStack`, `ConcurrentQueue` или `ConcurrentBag`. Модификации конкурентных коллекций применяются для реализации шаблона "производитель-потребитель". Метод `Take` для `BlockingCollection` вызывается потоком-потребителем и приводит к блокировке потока в случае отсутствия элементов. Добавление элементов потоком-производителем приводит к разблокировке (освобождению) потребителя.

При создании коллекции существует возможность установления максимальной емкости. Если коллекция полностью заполнена, то операция добавления элемента приводит к блокировке текущего потока до тех пор, пока поток-потребитель не извлечет, хотя бы один элемент.

Метод `CompleteAdding` позволяет завершить добавление элементов в коллекцию. Обращения к операции `Add` будут приводить к исключениям. Свойство `IsAddingCompleted` позволяет проверить статус завершения. Извлечение элементов из "завершенной" коллекции разрешено, пока коллекция не пуста. Операция `Take` для пустой завершенной коллекции приводит к генерации исключения. Свойство `IsCompleted` позволяет установить, является ли коллекция пустой и завершенной одновременно.

Тип коллекции, которая используется для формирования `BlockingCollection`, определяет порядок извлекаемых элементов. Если объект `BlockingCollection` создается без указания конкурентной коллекции, то в качестве базовой коллекции используется очередь типа `ConcurrentQueue`.

ConcurrentDictionary

Конкурентный словарь кроме потокобезопасности добавления и удаления элементов предоставляет расширенные функциональные возможности: методы условного добавления и методы обновления значений.

```
// Создаем конкурентный словарь
var cd = new ConcurrentDictionary<string, int>();

// Хотим получить элемент с ключом "b", если нет - создаем
int value = cd.GetOrAdd("b", (key) => 555);
// Проверяем: value = 555;
value = cd.GetOrAdd("b", -333);
// Параллельно пытаемся обновить элемент с ключом "a"
Parallel.For(0, 100000, i =>
{
    // Если ключа нет – добавляем
    // Если есть – обновляем значение
    cd.AddOrUpdate("a", 1, (key, oldValue) => oldValue + 1);
});
Console.WriteLine("Element \"a\": {0}", cd["a"]);
```

Метод `GetOrAdd` реализует возможность получения значения по ключу или в случае отсутствия элемента добавления.

```
if(dic.ContainsKey(sKey))
    val = dic[sKey];
else
    dic.Add(sKey, sNewValue);
```

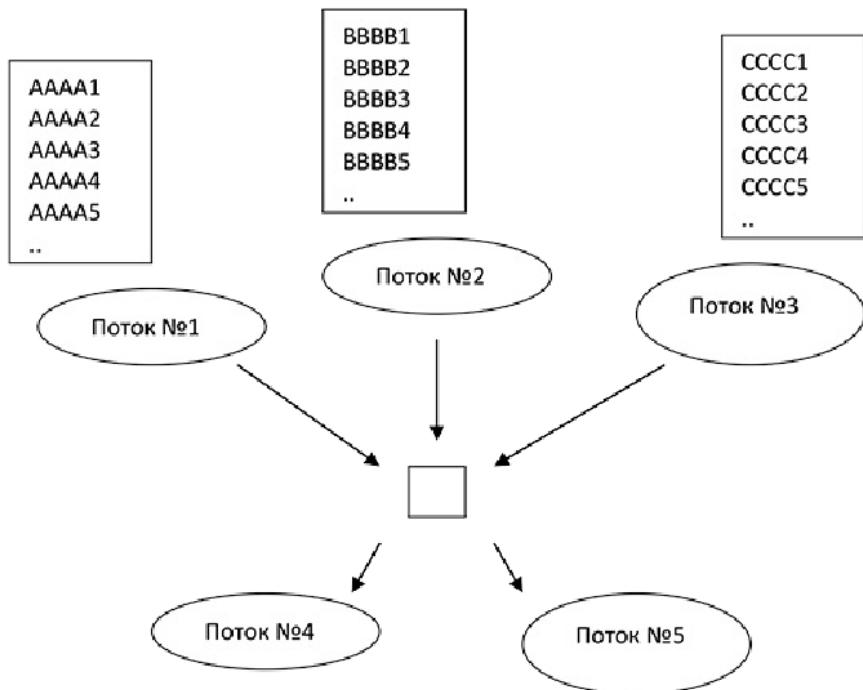
Но метод `GetOrAdd` выполняется атомарно в отличие от приведенного фрагмента, то есть несколько потоков не могут одновременно проверить наличие элемента и осуществить добавление.

Синхронизация доступа к одноэлементному буферу

Применение потокобезопасных структур данных в приложении

Задача

Несколько потоков работают с общим одноэлементным буфером. Потоки делятся на "писателей", осуществляющих запись сообщений в буфер, и "читателей", осуществляющих извлечение сообщений из буфера. Только один поток может осуществлять работу с буфером. Если буфер свободен, то только один писатель может осуществлять запись в буфер. Если буфер занят, то только один читатель может осуществлять чтение из буфера. После чтения буфер освобождается и доступен для записи. В качестве буфера используется глобальная переменная, например, типа `string`. Работа приложения заканчивается после того, как все сообщения писателей через общий буфер будут обработаны читателями.



Задание

1. Реализуйте взаимодействие потоков-читателей и потоков-писателей с общим буфером без каких-либо средств синхронизации. Проиллюстрируйте проблему совместного доступа. Почему возникает проблема доступа?
2. Реализуйте доступ "читателей" и "писателей" к буферу с применением следующих средств синхронизации:
 3.
 - блокировки (`lock`);
 - сигнальные сообщения (`ManualResetEvent`, `AutoResetEvent`, `ManualResetEventSlim`);
 - семафоры (`Semaphore`, `SemaphoreSlim`).
 - атомарные операторы (`Interlocked`)
4. Исследуйте производительность средств синхронизации при разном числе сообщений, разном объеме сообщений, разном числе потоков.
5. Сделайте выводы об эффективности применения средств синхронизации.

Методические указания

В случае одноэлементного буфера достаточно использовать флаг типа `bool` для контроля состояния буфера. Читатели обращаются к буферу, только если он свободен:

```
// Работа читателя
while (!finish)
{
    if (!bEmpty)
    {
        MyMessages.Add(buffer);
        bEmpty = true;
    }
}
```

Писатели обращаются к буферу, только если он пуст:

```
// Работа писателя
while(i < n)
{
    if (bEmpty)
    {
        buffer = MyMessages[i++];
        bEmpty = false;
    }
}
```

Писатели работают, пока не запишут все свои сообщения. По окончании работы писателей основной поток может изменить статус переменной `finish`, который является признаком окончания работы читателей.

```
static void Main()
{
    // Запускаем читателей и писателей
    ..
    // Ожидаем завершения работы писателей
    for(int i=0; i < writers.Length; i++)
        writers[i].Join();
    // Сигнал о завершении работы для читателей
    finish = true;

    // Ожидаем завершения работы читателей
    for(int i=0; i < readers.Length; i++)
        readers[i].Join();
}
```

Отсутствие средств синхронизации при обращении к буферу приводит к появлению гонки данных – несколько читателей могут прочитать одно и то же сообщение, прежде чем успеют обновить статус буфера; несколько

писателей могут одновременно осуществить запись в буфер. В данной задаче следствием гонки данных является потеря одних сообщений и дублирование других. Для фиксации проблемы предлагается выводить на экран число повторяющихся и потерянных сообщений.

Самый простой вариант решения проблемы заключается в использовании критической секции (`lock` или `Monitor`).

```
// Работа читателя
while (!finish)
{
    lock ("read")
    {
        if (!bEmpty)
        {
            MyMessage[i++] = buffer;
            bEmpty = true;
        }
    }
}
```

Для писателей существует своя критическая секция:

```
// Работа писателя
while(i < n)
{
    lock("write")
    {
        if (bEmpty)
        {
            buffer = MyMessage[i++];
            bEmpty = false;
        }
    }
}
```

Данная реализация не является оптимальной. Каждый из читателей

поочередно входит в критическую секцию и проверяет состояние буфера, в это время другие читатели блокируются, ожидая освобождения секции. Если буфер свободен, то синхронизация читателей избыточна. Более эффективным является вариант двойной проверки:

```
// Работа читателя
while (!finish)
{ if (!bEmpty)
  {
    lock ("read")
    {
      if (!bEmpty)
      {
        bEmpty = true;
        MyMessage[i++] = buffer;
      }
    }
  }
}
```

Если буфер свободен, то читатели "крутятся" в цикле, проверяя состояние буфера. При этом читатели не блокируются. Как только буфер заполняется, несколько читателей, но не все, успевают войти в первый `if`-блок, прежде чем самый быстрый читатель успеет изменить статус буфера `bEmpty = true`.

Применение сигнальных сообщений позволяет упростить логику синхронизации доступа. Читатели ожидают сигнала о поступлении сообщения, писатели – сигнала об опустошении буфера. Читатель, освобождающий буфер, сигнализирует об опустошении. Писатель, заполняющий буфер, сигнализирует о наполнении буфера. Сообщения с автоматическим сбросом `AutoResetEvent` обладают полезным свойством – при блокировке нескольких потоков на одном и том же объекте `AutoResetEvent` появление сигнала освобождает только один поток, другие потоки остаются заблокированными. Порядок освобождения потоков при поступлении сигнала не известен, но в данной задаче это не существенно.

```
// Работа читателя
void Reader(object state)
{
    var evFull = state[0] as AutoResetEvent;
    var evEmpty = state[1] as AutoResetEvent;
    while(!finish)
    {
        evFull.WaitOne();
        MyMessage.Add(buffer);
        evEmpty.Set();
    }
}
```

```
// Работа писателя
void Writer(object state)
{
    var evFull = state[0] as AutoResetEvent;
    var evEmpty = state[1] as AutoResetEvent;
    while(i < n)
    {
        evEmpty.WaitOne();
        buffer = MyMessage[i++];
        evFull.Set();
    }
}
```

Данный фрагмент приводит к зависанию работы читателей. Писатели закончили работу, а читатели ждут сигнала о наполненности буфера `evFull`. Для разблокировки читателей необходимо сформировать сигналы `evFull.Set()` от писателей при завершении работы или от главного потока. Чтобы отличить ситуацию завершения можно осуществлять проверку статуса `finish` непосредственно после разблокировки.

```
// Рабочий цикл читателей
while(true)
{
    evFull.Wait();
```

```
// Сигнал о завершении работы
if(finish) break;
MyMessage.Add(buffer);
evEmpty.Set();
}
```

Применение семафоров (`Semaphore`, `SemaphoreSlim`) в данной задаче аналогично использованию сигнальных сообщений `AutoResetEvent`. Кроме предложенного варианта обмена сигналами между читателями и писателями, семафоры и сигнальные сообщения могут использоваться в качестве критической секции читателей и писателей.

```
void Reader(object state)
{
    var semReader = state as SemaphoreSlim;
    while(!finish)
    {
        if(!bEmpty)
        {
            semReader.Wait();
            if(!bEmpty)
            {
                bEmpty = true;
                myMessages.Add(buffer);
            }
            semReader.Release();
        }
    }
}
```

```
void Writer(object state)
{
    var semWriter = state as SemaphoreSlim;
    while(i < myMessages.Length)
    {
        if(bEmpty)
        {
```

```
    semWriter.Wait();
    if(bEmpty)
    {
        bEmpty = false;
        buffer = myMessages[i];
    }
    semWriter.Release();
}
}
```

Вопросы и упражнения

1. Почему проблема гонки данных проявляется не при каждом прогоне?
2. Какие факторы увеличивают вероятность проявления проблемы гонки данных?
3. Возможно ли в данной задаче при отсутствии средств синхронизации возникновение исключения и аварийное завершение программы?
4. Можно ли в данной задаче использовать атомарные операторы для обеспечения согласованности доступа? Необходимы ли при этом дополнительные средства синхронизации?
5. Можно ли в данной задаче использовать потокобезопасные коллекции для обеспечения согласованного доступа?
6. Какие средства синхронизации обеспечивают наилучшее быстродействие в данной задаче? Объясните с чем это связано.

Работа с задачами

Основные возможности TPL: асинхронные задачи, императивный параллелизм, декларативный параллелизм. Работа с задачами. Вложенные задачи. Задачи-продолжения. Обработка исключений в задачах. Механизм отмены задач

Задачи (`tasks`) являются основным строительным блоком библиотеки `Task Parallel Library`. Задачи представляют собой рабочие элементы, которые могут выполняться параллельно. В качестве рабочих элементов могут использоваться: методы, делегаты, лямбда-выражения.

```
static void HelloWorld()
{
    Console.WriteLine("Hello, world!");
}

static void Main()
{
    // Используем обычный метод
    Task t1 = new Task(HelloWorld);

    // Используем делегат Action
    Task t2 = new Task(new Action(HelloWorld));

    // Используем безымянный делегат
    Task t3 = new Task(delegate
    {
        HelloWorld();
    });

    // Используем лямбда-выражение
    Task t4 = new Task(() => HelloWorld());

    // Используем лямбда-выражение
    Task t5 = new Task(() =>
    {
        HelloWorld();
    });

    Task t6 = new Task(() =>
    {
        Console.WriteLine("Hello, world!");
    });
}
```

```
// Запускаем задачи  
t1.Start(); t2.Start(); t3.Start();  
t4.Start(); t5.Start(); t6.Start();  
// Дожидаемся завершения задач  
Task.WaitAll(t1, t2, t3, t4, t5, t6);  
}
```

Работа с задачами, как правило, включает три основные операции: объявление задачи, добавление задачи в очередь готовых задач, ожидание завершения выполнения задачи.

Работа с потоками:

```
Thread threadOne = new Thread(SomeWork);  
threadOne.Start();  
threadOne.Join();
```

Работа с задачами:

```
Task taskOne = new Task(SomeWork);  
taskOne.Start();  
taskOne.Wait();
```

Важное отличие заключается в том, что вызов метода `Start` для задачи не создает новый поток, а помещает задачу в очередь готовых задач – пул потоков. Планировщик (`TaskScheduler`) в соответствии со своими правилами распределяет готовые задачи по рабочим потокам. Действия планировщика можно корректировать с помощью параметров задач. Момент фактического запуска задачи в общем случае не определен и зависит от загруженности пула потоков.

Для более лаконичного "запуска" задачи существует шаблон, замещающий этапы объявления и добавления задачи в пул потоков:

```
Task t = Task.Factory.StartNew(SomeWork);
```

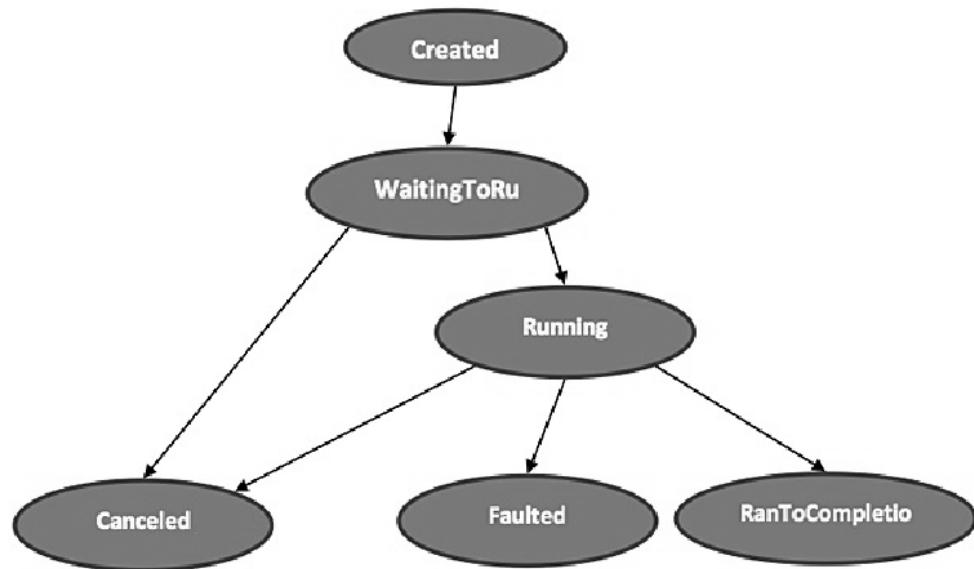
Ожидание завершения конкретной задачи осуществляется с помощью метода `Wait`. Для ожидания завершения нескольких задач существуют статические методы класса `Task`, принимающие в качестве аргумента массив задач:

```
Task t1 = Task.Factory.StartNew(DoWork1);
Task t2 = Task.Factory.StartNew(DoWork2);
Task t3 = Task.Factory.StartNew(DoWork3);
// Дожидаемся завершения хотя бы одной задачи
int firstTask = Task.WaitAny(t1, t2, t3);
// Дожидаемся завершения всех задач
Task.WaitAll(t1, t2, t3);
```

Методы ожидания `WaitAll` и `WaitAny` могут принимать в качестве аргументов, как массив задач (один параметр), так и сами задачи (произвольное число параметров). Вызов `WaitAll` блокирует текущий поток до завершения всех указанных задач. Вызов `WaitAny` блокирует текущий поток до завершения хотя бы одной из указанных задач и возвращает номер первой завершенной задачи.

Статусы задачи

Задача может находиться в нескольких состояниях: `Created`, `Running`, `WaitingToRun`, `Faulted`, `Canceled`, `RanToCompletion`, `WaitingForActivation`, `WaitingForChildrenToComplete`. Переходы между основными состояниями изображены на рисунке.



При объявлении задача получает статус `Created`. Запуск задачи с помощью метода `Start` или `StartNew` помещает задачу в пул потоков со статусом `WaitingToRun`. Задача, выполняющаяся в данный момент, имеет статус `Running`. Статус `Canceled` соответствует задаче, отмененной с помощью объекта `CancellationTokenSource`. Статус `Faulted` – при выполнении задачи произошла какая-то ошибка, необработанная в коде задачи. При успешном завершении задачи – статус `RanToCompletion`. Задачи-продолжения (см. ниже) начинают со статуса `WaitingForActivation`. При наличии дочерних вложенных задач родительская задача после завершения своей работы (`Running`) ожидает завершения вложенных задач со статусом `WaitingChildrenToComplete`.

Работа с данными в задаче

В задаче можно оперировать всеми переменными, находящимися в области видимости. Если рабочий элемент это метод класса, то работать можно с переменными этого класса. Если рабочим элементом является лямбда-выражение, то работать можно со всеми локальными переменными метода, порождающего задачу.

```

class Program
{
    static string programName;
    static void ShowTaskInfo(object taskName)
    {
        Console.WriteLine("Task name: {0}, Task ID: {1},
                           thread id: {2}, Program name: {3}",
                           taskName, Task.CurrentId,
                           Thread.ThreadId, programName);
    }
    static void Main()
    {
        programName = "Working with data";
        Task t1 = Task.Factory.StartNew(new Action<object>(ShowTaskInfo),
                                         "First worker");
        Task t2 = Task.Factory.StartNew(o => ShowTaskInfo(o),
                                         "Second worker");
        string t3Name = "Third worker";
        Task t3 = Task.Factory.StartNew(() =>
                                         ShowTaskInfo(t3Name));
    }
}

```

Для получения результата работы задачи существует специальный тип `Task<T>`. Свойство `Result` содержит результат задачи. Обращение к свойству блокирует поток до завершения задачи.

```

static void Main()
{
    long lNumber = 123456789;
    Task<double> SqrtTask = Task.Factory.StartNew((obj) =>
    {
        return Math.Sqrt((long)obj);
    }, lNumber);
    // Дожидаемся завершения вычислений без явной блокировки
    double sqrtNumber = SqrtTask.Result;
    Console.WriteLine("Result: {0}", sqrtNumber);
}

```

При объявлении задачи или при использовании шаблона `StartNew` можно указать ряд важных параметров:

- токен отмены (`CancellationToken`), с помощью которого можно реализовать "согласованную" отмену задачи;
- опции задачи (`TaskCreationOptions`), позволяющие скорректировать действия планировщика при обработке задачи;
- параметр задачи - аргумент метода, ассоциированного с задачей;
- планировщик, который будет использоваться для обработки задачи.

Вложенные задачи

В коде задачи можно запускать вложенные задачи, которые могут быть дочерними и недочерними. Недочерние задачи обладают независимостью от родительской задачи: родитель не дожидается завершения вложенной задачи, статусы задач не взаимосвязаны. Дочерняя задача действительно является вложенной – родитель дожидается завершения дочерней задачи, статусы задач при исключениях взаимосвязаны.

```
Task tParent = Task.Factory.StartNew( () =>
{
    Console.WriteLine("Parent task starts");
    Task t1 = Task.Factory.StartNew( () =>
        Console.WriteLine("Inner task"));
    Task t2 = Task.Factory.StartNew( () =>
        Console.WriteLine("Child task"),
        TaskCreationOptions.AttachedToParent);
    Console.WriteLine("Parent task ends");
});
tParent.Wait();
Console.WriteLine("Parent task really ends");
```

Вывод приведенного фрагмента определяется загруженностью системы и пулом потоков. Тем не менее, можно быть уверенным, что вызов

`tParent.Wait()` завершится только после вывода вложенной дочерней задачи. Вывод вложенной недочерней задачи может быть и после завершения родительской задачи.

Parent task starts

Parent task ends

Child task

Parent task really ends

Inner task

Механизм отмены задач

Встроенный механизм согласованной отмены задач позволяет унифицированным образом реализовывать корректное досрочное завершение выполнения задач.

Для реализации механизма отмены необходимо выполнить следующие шаги:

1. Создать объект `CancellationTokenSource` в области видимости метода, который порождает и запускает задачу.
2. Получить объект `CancellationToken`, через который осуществляется взаимодействие с задачей.
3. Передать объект `CancellationToken` при запуске задачи.
4. Реализовать в задаче процедуру отмены
5. При необходимости отмены вызвать метод `Cancel()`.

Обработчик отмены задачи можно реализовать либо в самой задаче, либо назначить отдельный делегат, который будет вызываться при возникновении сигнала об отмене задачи.

```
static void Main()
{
    var cts = new CancellationTokenSource();
    var token = cts.Token;
    Task t1 = new Task () =>
    {
        while(true)
```

```

{
    DoSomeWork();
    if(token.IsCancellationRequested)
    {
        // Код обработки отмены
        SaveSomeData();
        break;
    }
}
}, token);

t1.Start();
Thread.Sleep(1000);
cts.Cancel();
t1.Wait();

var cts2 = new CancellationTokenSource();
var token2 = cts2.Token;
// Регистрируем обработчик отмены
token2.Register(() => {
    Console.WriteLine("Task #2 was cancelled");
    throw new TaskCanceledException();
});
Task t2 = new Task(() => {
    while(true)
        DoSomeWork();
}, token2);

t2.Start();
Thread.Sleep(1000);
cts2.Cancel();
t2.Wait();
}

```

В первой задаче сигнал отмены контролируется в рабочем цикле с помощью свойства `IsCancellationRequested`. Во второй задаче отмена обрабатывается с помощью делегата, который регистрируется методом `Register`.

Исключения в задачах

Исключения, которые могут возникнуть при выполнении задач, обрабатываются в коде метода, который объявляет, запускает и ожидает завершения задач.

```
void SomeMethod()
{
    Task t1 = Task.Factory.StartNew(WorkOne);
    Task t2 = Task.Factory.StartNew(WorkTwo);
    try
    {
        Task.WaitAll(t1, t2);
    }
    catch(AggregateException ae)
    {
        // Обработка ошибок
    }
}
```

Исключения могут возникнуть в нескольких задачах. Поэтому для обработки исключений параллельного кода используется объект типа `AggregateException`, который агрегирует все возникнувшие исключения. Список единичных исключений можно получить с помощью свойства `InnerExceptions`.

```
catch(AggregateException ae)
{
    foreach(Exception exc in ae.InnerExceptions)
        Console.WriteLine(exc.Message);
}
```

Если у задачи есть вложенные дочерние задачи, то объект `exc` представляет собой опять тип `AggregateException` и для обработки исключений дочерних задач необходимо во вложенном цикле обрабатывать элементы `exc.InnerExceptions`.

Метод `Flatten()` объекта `AggregateException` возвращает все исключения, возникнувшие в задачах и вложенных задачах в одном списке, делая более удобной обработку.

```
Task[] tasks = new Task[N];
// объявляем и запускаем задачи
..
// Обработка исключений
try {
    Task.WaitAll(tasks);
}
catch(AggregateException ae)
{
    foreach(Exception e in ae.Flatten().InnerExceptions)
        Console.WriteLine("Message:{0}", e.Message);
}
```

Метод `Handle` позволяет назначить делегаты-обработчики для конкретных исключений:

```
Task t1 = new Task(() => {
    throw new OutOfMemoryException(); });
Task t2 = new Task(() => {
    throw new DivideByZeroException(); });
t1.Start(); t2.Start();
try {
    Task.WaitAll(t1, t2);
} catch (AggregateException ae) {
    ae.Handle((inner) => {
        if (inner is OperationCancelledException) {
            // обработчик исключения
            return true;
        } else if(inner is DivideByZeroException) {
            // обработчик исключения
            return true;
        } else
            return false;
    });
}
```

Делегат, определяемый в методе Handle, возвращает true, если исключение обработано, и false, если исключение не обработано.

Задачи-продолжения

Задачи-продолжения предназначены для планирования запуска задач после завершения предшествующих задач с тем или иным статусом завершения: OnlyOnRanToCompletion, OnlyOnCanceled, OnlyOnFaulted, NotOnCancelled, NotOnRanToCompletion.

```
// Основная задача, выполняющая расчёт
Task t1 = Task<int>.Factory.StartNew(() => FindDecision());
// Вывод результатов в отдельной задаче
Task t2 = t1.ContinueWith((prev) =>
    Console.WriteLine("Result: {0}", prev.Result),
    TaskContinuationOptions.OnlyOnRanToCompletion);
// Обработчик ошибок
Task t3 = t1.ContinueWith((prev) =>
    Console.WriteLine("Error: {0}",
        prev.Exception.InnerException.Message),
    TaskContinuationOptions.OnlyOnFaulted);
// Задача была отменена
Task t4 = t1.ContinueWith((prev) =>
    Console.WriteLine("Task was cancelled"),
    TaskContinuationOptions.OnlyOnCancelled);
```

Первая задача осуществляет основной расчет. Следующие задачи выполняются в зависимости от статуса завершения первой задачи. Вторая задача выполняется при успешном завершении. Третья задача выполняется при возникновении необработанного исключения. Четвертая задача выполняется только в случае отмены первой.

Задачи-продолжения позволяют без дополнительных средств синхронизации реализовать критическую секцию и конструкцию барьера:

```
// Объявляем задачи, которые могут выполняться параллельно
Task[] tasks = new Task[3];
tasks[0] = new Task(Work1);
tasks[1] = new Task(Work2);
tasks[2] = new Task(Work3);
// Планируем выполнение критической секции
Task tCr = Task.Factory.ContinueWhenAll(tasks, (tt) => {
    // Критическая секция
});
// Параллельные задачи
Task t5 = tCr.ContinueWith(Work5);
Task t6 = tCr.ContinueWith(Work6);
// Запускаем задачи
tasks[0].Start(); tasks[1].Start(); tasks[2].Start();
// Ожидаем завершения последней задачи
t6.Wait();
```

Вопросы

1. В чем основные достоинства работы с задачами по сравнению с непосредственной работой с потоками?
2. В каком случае задача завершается со статусом `RanToCompletion` при возникновении исключения?
3. Каким образом можно реализовать критическую секцию, используя задачи и не используя средств синхронизации, рассмотренных в предыдущей главе?
4. Для каких целей можно использовать вложенные недочерние задачи?

Упражнения

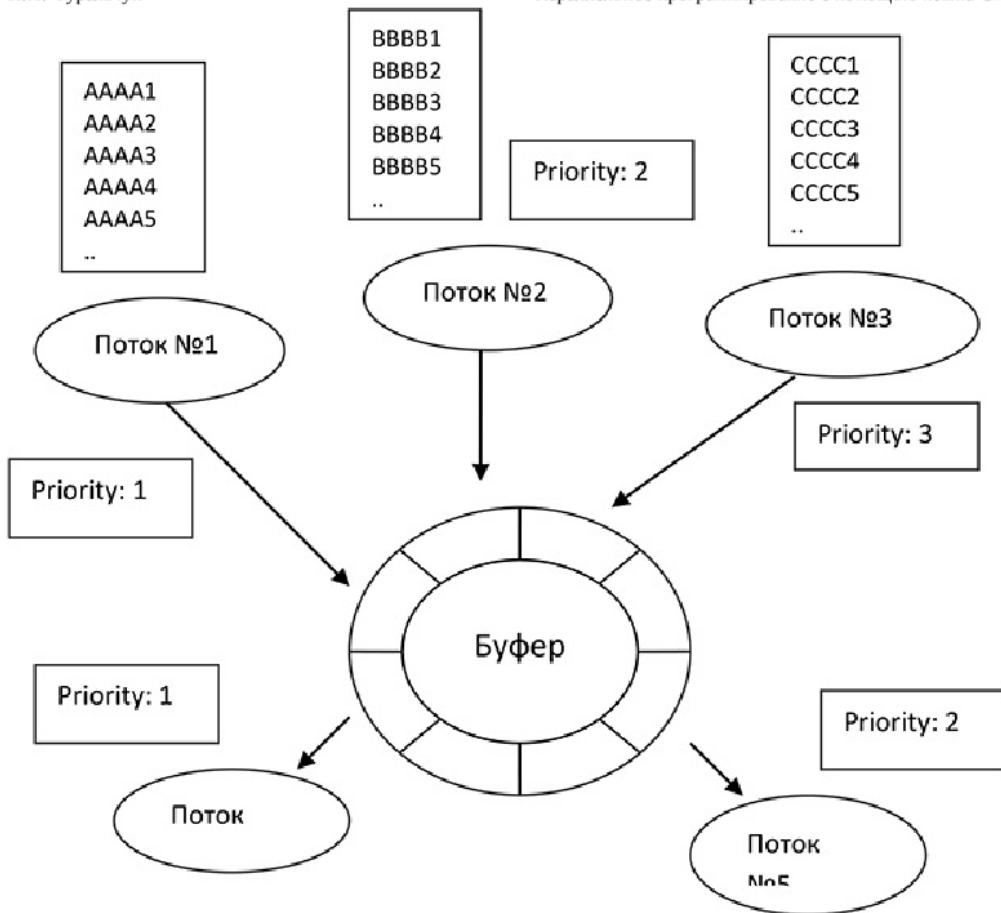
1. Исследуйте эффективность работы большего числа задач по сравнению с работой большого числа потоков.

Синхронизация приоритетного доступа к многоэлементному буферу

Реализация параллельных вычислений с помощью задач.

Задача

Несколько потоков работают с общим многоэлементным буфером. Потоки делятся на "читателей" и "писателей", каждый поток обладает приоритетом. Писатели осуществляют запись в буфер, если есть свободные ячейки. Читатели извлекают содержимое буфера, если есть заполненные ячейки. Работа приложения заканчивается после того, как все сообщения писателей будут обработаны читателями через общий буфер. В качестве буфера используется "кольцевой массив".



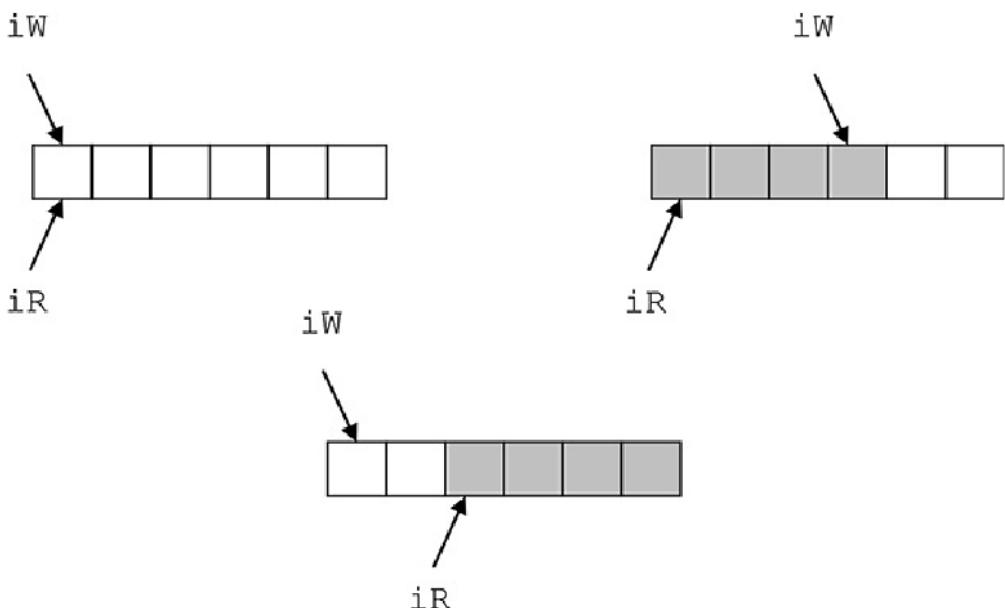
Задания

1. Реализуйте синхронизированное взаимодействие читателей и писателей с учетом приоритета. Аргументируйте выбор средств синхронизации.
2. Вывод программы включает: время работы каждого писателя и читателя; число сообщений, обработанных каждым писателем и читателем.
3. Выполните прогон программы при разных параметрах: разном числе писателей и читателей, разном объеме сообщений, разных приоритетах потоков. Результаты прогонов представьте в табличной форме.

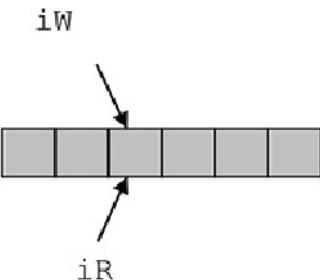
Методические указания

В качестве многоэлементного буфера используется кольцевой массив. Он представляет собой обычный массив размера n . Буфер называется кольцевым, так как при смещении текущего индекса после крайнего элемента следует первый. Для доступа к буферу используются два индекса: один для чтения и один для записи. Такая организация обеспечивает независимость операций чтения и записи – если в массиве есть свободные элементы и есть заполненные элементы, то операции чтения и записи могут производиться одновременно без каких-либо средств синхронизации.

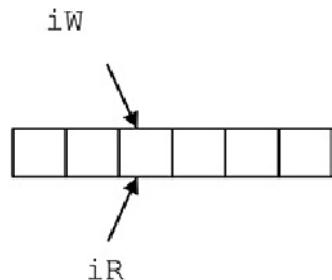
В начале работы буфер является пустым – оба индекса указывают на первый элемент. При осуществлении операций чтения или записи соответствующие индексы смещаются.



Операция чтения блокируется, если буфер пуст, запись при этом разрешена. Операция записи блокируется, если буфер полностью заполнен, чтение при этом разрешено. Равенство индексов чтения и записи является признаком и занятости буфера, и пустоты. Чтобы различать эти ситуации необходимо контролировать, какая операция привела к равенству индексов. Если операция записи, то буфер



`bFull = true`



`bFull = false`

Операции чтения и записи могут быть реализованы следующим образом:

```
bool Write(string Msg)
{
    if(bFull)
        return false;
    buffer[iW] = Msg;
    iW = (iW + 1) % n;
    // Если индексы совпали после записи,
    // буфер заполнен
    if(iW == iR)
        bFull = true;
    return true;
}
bool Read(ref string Msg)
{
    // Если индексы совпадают, но не после операции записи
    // буфер пуст
    if(iW == iR && !bFull)
        return false;
    Msg = buffer[iR];
    iR = (iR + 1) % n;
    // Если буфер был заполнен, то снимаем отметку
    if(bFull)
        bFull = false;
    return true;
}
```

Главный поток контролирует статус завершения операций чтения и записи. Если операция чтения не выполнена, то поток читателя блокируется.

Ситуация усложняется, если доступ к буферу осуществляют несколько читателей и несколько писателей. Один из вариантов решения проблемы – добавить конструкции критической секции в функции чтения и записи.

Другой подход заключается в реализации схемы "управляющий-рабочие", где управляющий контролирует все операции, требующие синхронизации. Рабочие потоки (читатели и писатели) обращаются к управляющему (основной поток) с сигналом о готовности осуществлять операцию чтения или записи. Управляющий поток фиксирует обращения читателей и писателей, вычисляет текущие индексы для чтения и записи, контролирует состояние буфера (полностью заполнен или полностью пуст), выбирает читателя и писателя, которым разрешает доступ. Операции чтения и записи по корректным индексам, полученным от управляющего потока, осуществляются читателями и писателями уже без контроля.

Взаимодействие рабочих и управляющего удобно организовать с помощью сигнальных сообщений типа `ManualResetEventSlim`.

Сигналы о готовности `evReadyToRead`, `evReadyToWrite` генерируют читатели и писатели, готовые осуществлять операции с буфером. Управляющий контролирует состояние сигналов у каждого рабочего.

Сигналы о возможности операций чтения и записи `evStartReading`, `evStartWriting` генерируются управляющим потоком конкретным читателям и писателям. Перед генерацией сигналов управляющий вычисляет индекс чтения или записи и сохраняет его в индивидуальной ячейке конкретного рабочего.

Такая организация взаимодействия позволяет достаточно легко изменять правила доступа: вводить приоритеты читателей и писателей, учитывать время обращения к управляющему потоку и обеспечивать

"справедливость" доступа в плане очередности.

```
void ReaderThread(int iReader,
    ManualResetEventSlim evReadyToRead,
    ManualResetEventSlim evStartReading)
{
    // Инициализация внутреннего буфера
    var Messages = new List<string>();
    // Рабочий цикл чтения
    while(true)
    {
        // Сигнализирует о готовности
        evReadyToRead.Set();
        // Ждем сигнала от менеджера
        evStartReading.Wait();
        // Разрешено чтение по текущему индексу
        int k = ReadIndexCopy[iReader];
        Messages.Add(buffer[k]);
        // Сбрасываем сигнал о чтении
        evStartReading.Reset();
        // Проверяем статус завершения работы
        if (finish) break;
    }
}

// Код писателя практически идентичен коду читателя
void WriterThread(int iWriter,
    ManualResetEventSlim evReadyToWrite,
    ManualResetEventSlim evStartWriting)
{
    // Инициализация массива сообщений писателя
    Messages = ..;
    // Рабочий цикл записи
    while(true)
    {
        // Сигнализируем о готовности менеджеру
        evReadyToWrite.Set();
        // Ждем сигнала от менеджера
        evStartWriting.Wait();
        // Разрешена запись по текущему индексу
```

```
k = WriteIndexCopy[iWriter];
buffer[k] = Messages[j];
// Проверяем статус завершения работы
if (finish || j >= Messages.Length)
break;
j++
}
}
// Код менеджера
void Manager(int nReaders, int nWriters)
{
// Запуск читателей
for(int i=0; i<nReaders; i++)
{
evReadyToRead[i] =
new ManualResetEventSlim(false);
evStartReading[i] =
new ManualResetEventSlim(false);
tReaders[i] = new Task( () =>
Reader(i, evReadyToRead[i], evStartReading[i]));
tReaders[i].Start();
}
// Запуск писателей
for(int i=0; i < nWriters; i++)
{
var evReadyToWrite[i] =
new ManualResetEventSlim(false);
var evStartWriting[i] =
new ManualResetEventSlim(false);
tWriters[i] = new Task( () =>
Writer(i, evReadyToWrite[i], evStartWriting[i]));
tWriters[i].Start();
}
// Рабочий цикл
while(true)
{
// Если в буфере есть свободные ячейки
// пытаемся обработать готовых писателей
if(!bFull)
```

```
{  
    // Получаем текущий индекс записи  
    iW = GetBufferWriteIndex();  
    if(iW != -1)  
    {  
        // Устанавливаем писателя,  
        // которому разрешаем работать  
        iWriter = GetWriter();  
        if (iWriter != -1)  
        {  
            // Сбрасываем сигнал готовности  
            // выбранного писателя  
            evReadyToWrite[iWriter].Reset();  
            // Сохраняем копию индекса для записи  
            ReadIndexCopy[iWriter] = iW;  
            // Разрешаем писателю начать работу  
            evStartWriting[iWriter].Set();  
        }  
    }  
    else  
        bFull = true;  
}  
// Если буфер не пуст, пытаемся  
// обработать готовых писателей  
if(!bEmpty)  
{  
    // Получаем текущий индекс для чтения  
    iR = GetBufferReadIndex();  
    if(iR != -1)  
    {  
        //Устанавливаем готового читателя  
        iReader = GetWriter();  
        if (iReader != -1)  
        {  
            evReadyToRead[iReader].Reset();  
            WriteIndexCopy[iReader] = iR;  
            evStartReading[iReader].Set();  
        }  
    }  
}
```

```
else
    bEmpty = false;
}
}
}

// Код функции получения номера готового писателя
// с учетом приоритетов
int GetWriter()
{
    // Устанавливаем готовых писателей
    var ready = new List<int>();
    for(int i=0; i<nWriter; i++)
        if(evReadyToWrite[i].IsSet())
            ready.Add(i);
    if(ready.Count == 0)
        return -1;
    return ready.OrderBy(i => WriterPriority[i]).First();
}
```

Вопросы и упражнения

1. Можно ли вместо объектов `ManualResetEventSlim` использовать другие типы сигнальных сообщений: `AutoResetEvent` или `ManualResetEvent`?
2. Какие особенности задачи не позволяют использовать объект `ReaderWriterSlim`?
3. Почему структура кольцевого буфера не требует синхронизации при работе одного читателя и одного писателя?
4. Почему в предложенной реализации не используются критические секции?
5. Реализуйте учет времени обращения рабочих потоков к буферу.
6. Реализуйте решение задачи с использованием конкурентных коллекций в качестве буфера.

Шаблоны параллелизма Parallel

Параллельные циклы `Parallel.For`, `Parallel.ForEach` Дополнительные возможности циклов: досрочный выход из цикла, пакетная обработка диапазонов, реализация агрегированных вычислений. Методы `Parallel.Invoke` для реализации алгоритмов Эразделяй-и-властвуй".

Класс `Parallel` предоставляет два наиболее распространенных шаблона параллельной обработки: параллельные циклы (`Parallel.For`, `Parallel.ForEach`) и параллельный запуск нескольких независимых задач (`Parallel.Invoke`). Реализация шаблонов построена на задачах (`tasks`), поэтому при использовании методов `Parallel` поддерживаются механизмы отмены с помощью объекта `CancellationToken` и обработка исключений типа `AggregateException`. Важной особенностью шаблонов является "императивность". Оператор, следующий за вызовом метода класса `Parallel`, будет выполняться только после завершения всех задач, неявно созданных в методе.

Parallel.Invoke

Вызов метода позволяет "запустить" (добавить в очередь готовых к выполнению задач) несколько рабочих элементов и дождаться завершения их работы.

Таким образом, вызов метода `Invoke`:

```
Parallel.Invoke(FuncOne, FuncTwo)
```

с помощью задач можно переписать следующим образом:

```
Task taskTwo = Task.Factory.StartNew(() => FuncTwo());  
Task taskOne = Task.Factory.StartNew(() => FuncOne());  
Task.WaitAll(taskOne, taskTwo);
```

Методы `Parallel` создают задачи не для каждого рабочего элемента, а для набора задач, которые будут выполняться в одном потоке. Поток, в

котором осуществляется вызов метода `Invoke`, также используется для обработки элементов.

Использовать метод `Invoke` вместо непосредственного манипулирования с задачами полезно, когда рабочие элементы можно запустить одновременно (нет какой-либо подготовительной работы перед каждой задачей) и вызывающий поток должен дождаться завершения работы всех рабочих элементов. Такая ситуация возникает в алгоритмах типа "разделяй и властвуй" (быстрая сортировка, обработка графов).

В качестве аргументов метода `Parallel.Invoke` можно указывать методы, лямбда-выражения, а также массив делегатов типа `Action`:

```
Action[] actions = new Action[4];
actions[0] = new Action(() => Console.WriteLine("one"));
actions[1] = new Action(() => Console.WriteLine("two"));
actions[2] = new Action(() => Console.WriteLine("three"));
actions[3] = new Action(() => Console.WriteLine("four"));
Parallel.Invoke(actions);
```

В качестве параметров выполнения в методе `Invoke` можно указать токен отмены, максимальную степень параллелизма и используемый планировщик:

```
ParallelOptions pOptions = new ParallelOptions()
{
    maxDegreeOfParallelism = 4,
    cancellationToken = cToken,
    TaskScheduler = tScheduler
};
Parallel.Invoke(pOptions, actions);
```

Параллельные циклы `Parallel.For` и `Parallel.ForEach`

Методы `Parallel.For` и `Parallel.ForEach` позволяют распараллелить обработку итераций или обработку элементов какой-

либо структуры данных перечислимого типа (массив, список). Методы имеют несколько перегрузок, позволяющие настраивать параллелизм обработки.

В случае независимости обработки элементов перечислимого типа используется самый простой вариант вызова:

```
int[] data = new int[500];
int[] results = new int[500];
// Последовательный цикл
for(int i=0; i<data.Length; i++)
    data[i] = ..;
// Параллельный цикл
Parallel.For(0, data.Length, i =>
{
    results[i] = SomeWork(data[i]);
});
Console.WriteLine("Total sum: {0}", results.Sum());
```

Аргументами являются границы индекса *i*, который используется в делегате обработки элементов массива. Как и в последовательном цикле, конечный индекс не участвует в обработке. Делегат *Action<int>* вызывается на каждой итерации и в качестве аргумента принимает значения индекса целочисленного типа.

Метод *Parallel.ForEach* позволяет параллельно обрабатывать элементы перечислимого типа.

```
List<string> words = new List<string> {"first",
"second", "third", "four", "five" };
Parallel.ForEach(words, s => Console.WriteLine(s));
```

Метод *Parallel.ForEach* позволяет создавать произвольные итераторы в отличие от *Parallel.For*.

```
// Используем произвольный итератор: от 0 до 500 с шагом 10
Parallel.ForEach(SteppedIterator(0, 500, 10), index =>
    Console.WriteLine("Index: {0}", index));
```

```
double[] dblData = new double[500];
// Инициализируем данные типа double
Parallel.ForEach(dblData, InitData);
// Вычисляем квадратный корень для каждого элемента массива
Parallel.ForEach(dblData, Math.Sqrt);
```

Параметры цикла

Параллельные циклы поддерживают возможность указания токена отмены и максимальную степень параллелизма:

```
Parallel.For(0, 1000,
    new ParallelOptions()
    { maxDegreeOfParallelism = 4,
        cancellationToken = ctoken},
    i => SomeFunc(i));
```

Досрочный выход из цикла

В последовательных циклах часто используется досрочный выход из цикла с помощью оператора `break`. Досрочный выход используется в двух типовых случаях:

- поиск единственного решения, которое возникает на заранее неизвестной итерации, при этом результаты выполнения предыдущих итераций не представляют интереса;
- обработка всех итераций до итерации, на которой выполняется определенное условие; интерес представляют и все предшествующие итерации.

При последовательном выполнении оба случая можно реализовать с помощью одного оператора `break`. При параллельном выполнении используются два разных метода `Stop` и `Break` объекта `ParallelLoopState`, который является аргументом метода обработчика.

```
var bag1 = new ConcurrentBag<int>();
Parallel.For(0, 1000, (int i, ParallelLoopState pState) =>
{
    if(i == 50)
        pState.Break();
    else
    {
        Thread.Sleep(10);
        bag1.Add(i);
    }
});

var bag2 = new ConcurrentBag<int>();
Parallel.For(0, 10000, (i, state) =>
{
    if (i == 50)
        state.Stop();
    else
    {
        Thread.Sleep(10);
        bag2.Add(i);
    }
});

Console.WriteLine("Break 50, Smaller: {0}, bigger: {1}",
    bag1.Where(i => i < 50).Count(),
    bag1.Where(i => i > 50).Count());

Console.WriteLine("Stop 50, Smaller: {0}, bigger: {1}",
    bag2.Where(i => i < 50).Count(),
    bag2.Where(i => i > 50).Count());
```

Выход:

Break 50, Smaller: 50, Bigger: 20
Stop 50, Smaller: 38, Bigger: 24

Метод `Break` вызван на 50-итерации. Все итерации с меньшими номерами гарантировано будут выполнены, даже если они еще не начались выполняться. Итерации с большими номерами отменяются, если они еще не начались выполняться. Метод `Stop` отменяет все итерации, которые еще не начались выполняться.

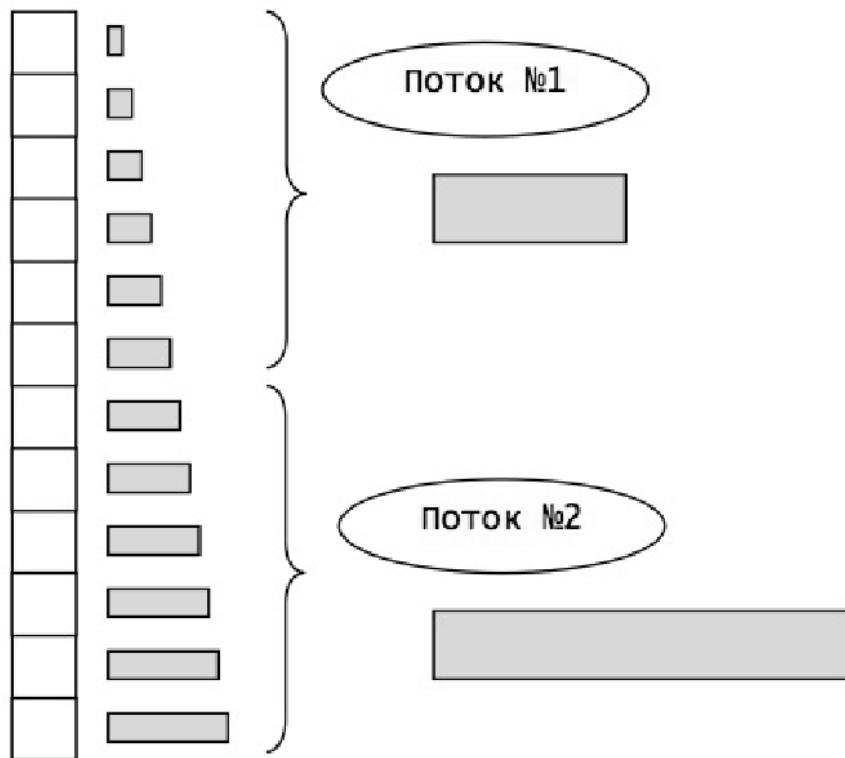
Разделение данных

Распределение итераций по рабочим потокам осуществляется либо по равным диапазонам индекса (`range-partitioning`), либо по блокам (`chunk-partitioning`). По умолчанию осуществляется разделение по диапазону (статическая декомпозиция). Планировщик до начала выполнения цикла разделяет итерации по свободным рабочим потокам. В процессе обработки цикла нет необходимости в синхронизации доступа.

```
Parallel.For(0, 100, i =>
    Console.WriteLine("Iteration: {0}, task: {1}, thread: {2}", i, Task.TaskId,
        Thread.ManagedThreadId);
```

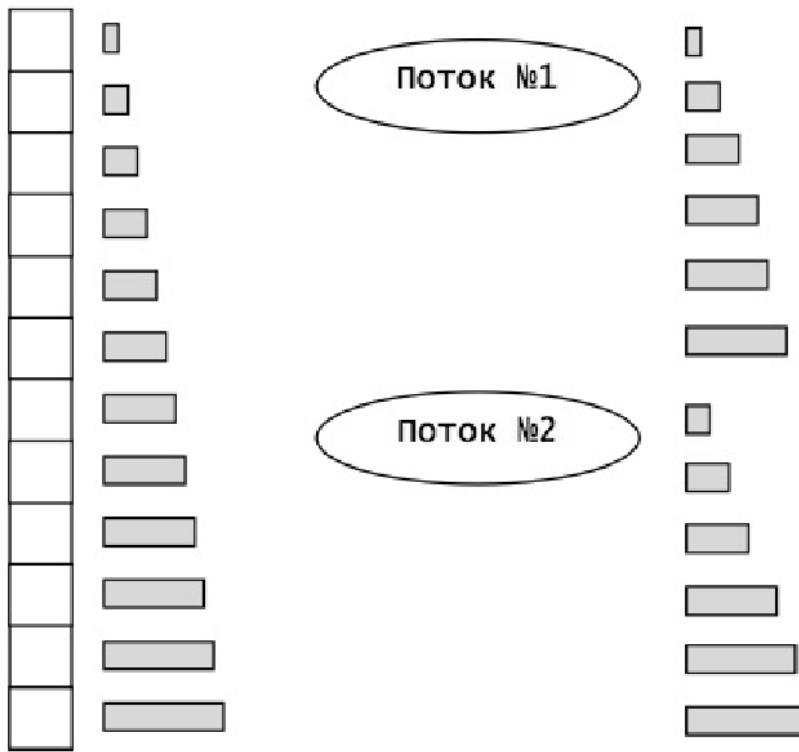
В результате получаем, что первые 50 итераций выполнялись одним рабочим потоком, следующие 50 итераций выполнялись вторым рабочим потоком. Статическая декомпозиция является эффективной при относительно одинаковой вычислительной нагрузке на каждой итерации.

Часто обработка i -элемента зависит от номера индекса. Разная вычислительная "нагрузка" при обработке элементов приводит к несбалансированности параллельной обработки – некоторые потоки быстро выполняют свою работу, а какие-то потоки будут продолжать работать. При статической декомпозиции освободившиеся потоки не помогают загруженным, но могут использоваться планировщиком для обработки других задач.



При блочной (динамической) декомпозиции распределение элементов (итераций цикла) по рабочим потокам осуществляется динамически на протяжении всей обработки цикла. Блочная декомпозиция приводит к более сбалансированному разбиению, но требует затрат на синхронизацию доступа к элементам структуры. Для выполнения блочной декомпозиции необходимо использовать объект `Partitioner` пространства `System.Collections.Concurrent`:

```
// Список элементов
List<string> list = ..
Parallel.ForEach(Partitioner.Create(list, true), s => {
    // Обработка элементов списка
});
```



/

При обработке итераций цикла можно обращаться к разделяемым переменным, например, при вычислении агрегированных величин: суммы, среднего значения и т.п., но необходимо использовать средства синхронизации (блокировки, атомарные операторы).

```
Parallel.For(0, N, i =>
{
    double d = Compute(ar[i]);
    Interlocked.Add(ref totalSum, d);
});
```

Если несколько потоков, участвующих в обработке цикла, будут осуществлять обновление общей переменной, то возникает проблема гонки данных. Для предотвращения искажений, используется

атомарный оператор суммирования `Interlocked.Add`. Тем не менее, синхронизация всех итераций является избыточной. Те итерации, которые обрабатываются одним рабочим потоком, будут выполняться строго последовательно, и поэтому между ними нет конфликта доступа к общей переменной.

Класс `Parallel` предоставляет перегрузку `Parallel.For`, которая позволяет использовать частные переменные – общие для всех итераций, выполняющихся в одном рабочем потоке.

```
// Вычисление арифметического среднего
int [] ar = new int[] {3, 2, 7, 1};
Parallel.For(
    0, N, // индексы массива
    () => 0.0, // инициализация частных переменных
    (i, state, partial) =>
    {// нет необходимости в блокировке оператора
        partial += ar[i];
    },
    // конечная редукция частных переменных
    partial => Interlocked.Sum(ref sum, partial))
);
avgValue = sum / N;
```

В этом примере перегрузка `Parallel.For` содержит следующие параметры. Первые два аргумента задают диапазон итераций – от 0 до `N`. Третий параметр определяет делегат инициализации. Этот делегат вызывается один раз в каждом потоке, участвующем в обработке итераций. Четвертый аргумент задает делегат обработки, который вызывается на каждой итерации. Делегат обработки принимает три аргумента: индекс элемента, объект `ParallelLoopState` и локальная переменная, общая для всех итераций, выполняющихся в одном потоке. Последний параметр задает делегат финальной обработки. Таким образом, вычисление суммы распараллеливается с помощью частных сумм в каждом потоке и агрегированием частных сумм в конце обработки. Синхронизация необходима только в финальном делегате, так как здесь возможна параллельная работа с общей переменной `sum`.

Пакетная обработка итераций

Метод `Parallel.ForEach` обладает еще одной полезной перегрузкой, позволяющей вызывать обработчик не на каждой итерации, а только по одному разу для каждого рабочего потока. В случае большего числа итераций, во много раз превосходящее число рабочих потоков, "пакетная" обработка цикла может увеличить быстродействие. Ниже приведен вызов метода `Parallel.ForEach` с пакетной обработкой.

```
Parallel.ForEach(  
    //  
    Partitioner.Create(0, N),  
    // Начальная инициализация  
    () => 0.0,  
    // Обработчик цикла  
    (range, state, partial) =>  
    {  
        for(int i=range.Item1; i< range.Item2; i++)  
            partial += ar[i];  
        return partial;  
    },  
    // финальный этап  
    partial => Interlocked.Sum(ref sum, partial)  
);
```

В качестве первого параметра выступает разделитель, который создается для обрабатываемой структуры данных. В примере используется перегрузка разделителя, создающая ряд целых чисел от 0 до $N-1$. Второй параметр определяет делегат, который вызывается при инициализации обработки на каждом потоке. В примере делегат обнуляет значение локальной переменной потока. Третий параметр задает делегат, который вызывается в каждом потоке один раз для обработки всех элементов данного потока. Делегат обработки содержит три аргумента: первый аргумент `range` – предоставляет доступ к граничным элементам, обрабатываемым в данном потоке; второй аргумент `state` типа `ParallelLoopState` предоставляет

возможности по досрочному выходу из цикла; третий аргумент `partial` представляет собой локальную переменную потока. Делегат обработки возвращает значение типа локальной переменной. Возвращаемые значения делегатов обработки используются как входные аргументы финального делегата. В рассматриваемом примере каждый поток вычисляет свои локальные суммы. Финальный делегат вычисляет итоговую сумму с помощью атомарного оператора `Interlocked.Sum`. Применение средств синхронизации в финальном обработчике связано с возможной параллельностью вызовов. Таким образом, в каждом потоке, участвующем в обработке цикла, последовательно вызывается инициализирующий делегат, делегат обработки и финальный делегат. Каждый делегат обработки работает со своей локальной переменной, поэтому синхронизация при обработке элементов не требуется.

Вопросы

1. Почему для досрочного выхода из параллельного цикла используются два метода `Break` и `Stop`?
2. В чем преимущество пакетной обработки?

Упражнения

1. Исследуйте эффективность цикла `Parallel.For` в задаче матричного умножения.
2. Исследуйте эффективность цикла `Parallel.For` с пакетной обработкой и без пакетной обработки.

Клеточная модель "Игра Жизнь" Дж.Конвея

Применение шаблонов Parallel.For, Parallel.Invoke.

Задания

1. Реализуйте Windows-приложение, которое последовательно отображает состояния клеточной модели "Игра Жизнь".
2. Реализуйте последовательный алгоритм расчета состояний модели.
3. Реализуйте параллельные алгоритмы расчета состояний модели. Для распараллеливания используйте задачи (tasks) или метод Parallel.For.
4. Реализуйте возможность отмены расчета с помощью объекта CancellationToken.
5. Выполните анализ эффективности разработанных алгоритмов.

Методические указания

Клеточная модель представляет собой множество клеток (ячеек таблицы), принимающих одно из нескольких состояний. Состояние каждой клетки определяется состоянием её ближайших соседей. Одной из известных моделей является "Игра Жизнь" математика Дж. Конвея.

В модели Конвея каждая клетка может находиться в двух состояниях: живая или неживая. Состояние клетки на следующем шаге определяется потенциалом клетки (числом живых соседних клеток):

- если потенциал клетки равен двум, то клетка сохраняет свое состояние;
- если потенциал равен трем, то клетка оживает;
- если потенциал меньше двух или больше трех, то клетка погибает.

Правила изменения состояния клетки можно описать следующим лямбда-выражением:

```
var lifeRules = new Func<int, bool, bool>((p, state) =>
{
    if (p == 2 && state)
        return true;
    if (p == 3 && !state)
        return true;
    return false;
})
```

```

if(p == 3)
    return true;
else if (p == 2)
    return state;
else
    return false;
});

```

Последовательный алгоритм расчета представляет собой расчет состояния каждой клетки

```

LifeTable tableNew = new LifeTable;
for(int i=0; i < Height; i++)
    for(int j=0; j < Width; j++)
    {
        p = CalcPotential(table[i,j]);
        tableNew[i, j] = lifeRules(p, table[i, j]);
    }
table = tableNew;

```

Потенциал клетки вычисляется по восьми ближайшим соседям клетки.

```

int CalcPotential(int i, int j)
{
    int p=0;
    for(int x = i-1; x <= i + 1; x++)
        for(int y = j-1; y <= j + 1;y++)
        {
            if(x < 0 || y < 0 || x >= Height || y >= Width
                || (x == i && y == j))
                continue;

            if(table[x,y]) p++;
        }
    return p;
}

```

{

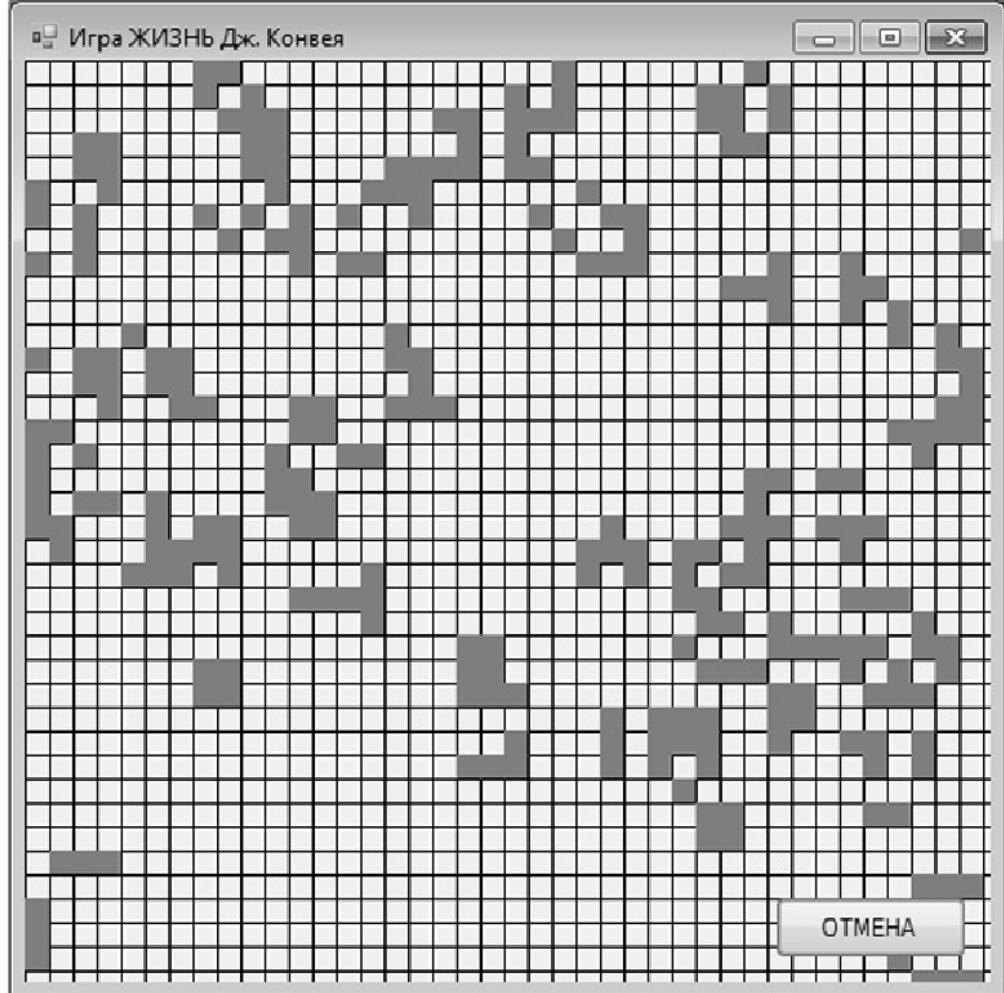
Вычисления новых состояний для каждой клетки являются независимыми и могут выполняться параллельно.

```
Parallel.For(0, Height, (i) => {
    for(int j=0; j < Width; j++)
    {
        p = CalcPotential(table[i,j]);
        tableNew[i, j] = lifeRules(p, table[i, j]);
    }
});
```

Также можно изменить порядок расчета и распараллелить внешний цикл по столбцам:

```
Parallel.For(0, Width, (j) => {
    for(int i=0; i < Height; i++)
    {
        p = CalcPotential(table[i,j]);
        tableNew[i, j] = lifeRules(p, table[i, j]);
    }
});
```

Вывод таблицы состояний клеточной модели может быть следующим:



Вопросы и упражнения

1. Имеет ли смысл распараллеливание внутреннего цикла расчета?
Почему?

```
for(int i=0; i < Height; i++)  
{  
    Parallel.For(0, Width, j =>  
    {  
        p = CalcPotential(table[i,j]);  
        tableNew[i, j] = lifeRules(p, table[i, j]);  
    }  
}
```

```
});  
}
```

2. Как вариант расчета – по строкам или по столбцам – более эффективен и с чем это связано?
3. Продумайте вариант блочной декомпозиции, где блок выступает матрицей размера $K \times K$. В чем достоинства и недостатки блочной декомпозиции для этой задачи? Какое значение параметра K следует выбирать?

Технология PLINQ

Параллелизм PLINQ-запросов. Анализ оптимальности. Вынужденный параллелизм. Упорядочивание элементов в PLINQ-запросах. Разделение данных при выполнении PLINQ-запросов: разделение по диапазону, блочное разделение, хеш-секционирование. Буферизация выполнения PLINQ-запросов. Агрегированные вычисления с помощью PLINQ-запросов.

Технология PLINQ (Parallel LINQ) позволяет автоматически распараллеливать LINQ-запросы для обработки локальных структур данных.

```
IEnumerable<int> numbers = Enumerable.Range(1, 10000);
// Последовательный запрос
var seqQ = from n in numbers
           where n % 2 == 0
           select Math.Pow(n, 2);

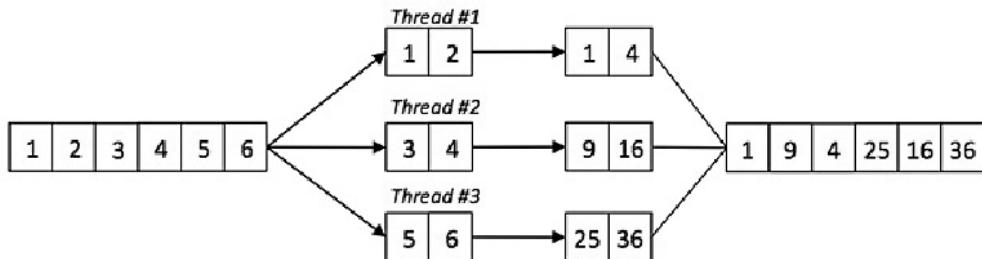
// Объявляем запрос, который выполняется параллельно
var parQ = from n in numbers.AsParallel()
           where n % 2 == 0
           select Math.Pow(n, 2);
```

Метод `AsParallel` преобразуют исходную последовательность типа `IEnumerable<TSource>` в последовательность типа `ParallelQuery<TSource>`. Этот тип содержит методы-расширения с теми же именами, как и тип `IEnumerable<T>`, но предполагающие возможное параллельное исполнение на многопроцессорной системе. Другой способ выполнения параллельного запроса связан с использованием объекта `ParallelEnumerable`, который позволяет сформировать диапазон аналогично объекту `Enumerable`:

```
// Альтернативный вызов параллельных запросов
var parQ2 = from n in ParallelEnumerable.Range(1, 1000)
             where n % 2 == 0
             select Math.Pow(n, 2);
```

Основные этапы выполнения PLINQ-запроса: разделение данных по рабочим потокам, параллельное исполнение запросов на каждом потоке, агрегирование результатов в конечную последовательность.

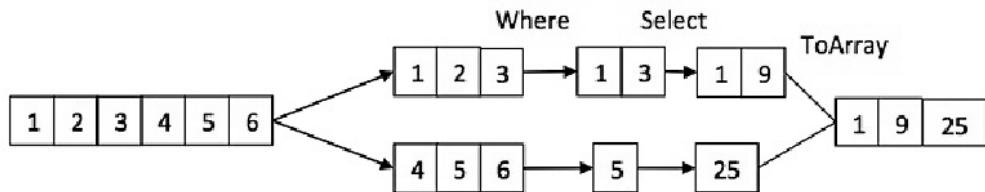
```
numbers.AsParallel().Select(n => Math.Pow(n, 2)).ToArray()
```



Удобство PLINQ заключается в том, что выполняя несколько запросов к последовательности, разделение элементов по потокам осуществляется по возможности вначале обработки. Элементы, попавшие в один поток при выполнении первого оператора, продолжают обрабатываться в этом потоке.

На рисунке изображено выполнение запроса

```
ParallelEnumerable.Range(1, 6).Where(n => n % 2 != 0).Select(n => n * n).To
```



Как и в случае шаблонов класса Parallel, для обработки элементов используются не только рабочие потоки пула, но и в первую очередь пользовательский поток, в котором осуществляется вызов запроса.

```
var threads =
from n in ParallelEnumerable.Range(1, 1000)
select Thread.CurrentThread.ManagedThreadId;
```

В этом запросе сохраняем номер потока, в котором происходит обработка элементов. Получаем, что половина элементов обрабатывается в основном потоке, другая половина в рабочем потоке пула.

Эффективность распараллеливания

Распараллеливание выполнения запроса связано с накладными расходами на разделение данных, агрегирование результатов. Многие закономерности при обработке PLINQ-запросов сохраняются: чем больше элементов, тем выше эффективность распараллеливания; чем больше вычислительная сложность обработки элемента, те выше эффективность распараллеливания.

Некоторые запросы обладают низкой эффективностью – последовательные аналоги работают быстрее, чем параллельные. Например, запросы, связанные со сравнением элементов (`GroupBy`, `Distinct`, `Join` и др.) разделяют элементы с помощью хэш-секционирования: перед распараллеливанием для каждого элементы вычисляет хэш-код, элементы с одинаковыми хэш-кодами объединяются для обработки в одном потоке. Сложная процедура разделения значительно снижает эффект распараллеливания дальнейшей обработки. Рассмотрим следующий фрагмент:

```
var threads =  
from n in ParallelEnumerable.Range(1, 1000)  
select Thread.CurrentThread.ManagedThreadId;  
var diff = threads.Distinct().ToArray();  
int nThr = diff.Length;
```

Запрос `threads` сохраняет номер потока, в котором осуществляется обработка элемента. Для получения массива уникальных номеров `diff` используем оператор выявления различий `Distinct` и оператор преобразования в массив `ToArray`. Если изменить

последовательность операторов `Distinct` и `ToArray`, то получим более производительный код:

```
var diff = threads.ToArray().Distinct();
```

Быстродействие выполнения второго варианта в 2 – 2.5 раза превышает параллельную версию. Разный порядок операторов приводит к тому, что метод `Distinct` применяется для массива, то есть типа `IEnumerable<T>`, а не для типа `ParallelQuery<T>`. Поэтому вызывается последовательная версия поиска различных элементов.

Запросы `Take`, `TakeWhile`, `Skip`, `SkipWhile` работают только с исходным порядком элементов. Например, запрос `Take(5)` отбирает первые пять элементов. Анализатор PLINQ не распараллеливает такие запросы. Также не распараллеваются индексные перегрузки методов `Select`, `Where`, если предыдущие операторы привели к изменению индексов элементов в структуре.

```
var threads = ParallelEnumerable.Range(1, 1000)
    .Where(n => n % 2 == 0)
    .Select((int n, int i) =>
        Thread.CurrentThread.ManagedThreadId);
int nThr = threads.ToArray().Distinct().Count();
```

В этом фрагменте получаем число потоков, участвовавших в обработке элементов. Оператор `Where` выполняет отбор элементов и изменяет индексы элементов в структуре - двойка была вторым элементом, становится первым. Изменения индексов препятствует распараллеливанию индексной версии `Select`. Поэтому запрос не распараллеливается и число потоков `nThr` равно одному.

Для обязательного распараллеливания запроса вне зависимости от эффективности применяется модификатор `WithExecutionMode` с параметром `ForceParallelism`:

```
var threads = "abcdef".AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
```

```
.Where(c => true)
.Take(3)
.Select(c=> Thread.CurrentThread.ManagedThreadId);

int nThr = threads.ToArray().Distinct().Count();
```

Несмотря на то, что оператор `Where` фактически пропускает все элементы, анализатор PLINQ-запросов по умолчанию не распараллеливал бы выполнение запроса. Но режим `ForceParallelism` позволяет включить распараллеливание, например, в отладочных целях.

Если какая-то часть запроса обязательно должна выполняться последовательно, необходимо выполнить обратное преобразование `ParallelEnumerable` в `IEnumerable` с помощью метода `AsSequential`. Такое преобразование может быть полезным, если в запросе используются потоконебезопасные методы, параллельное выполнение которых может привести к ошибкам.

```
var q = data.AsParallel()
    .Select(item => DoSome(item)).Where(item => IsValid(item))
    .AsSequential().Select(item => DoSomeSeq(item))
    .ToArray();
```

Первые два оператора `Select` и `Where` выполняются параллельно. Перед вызовом последнего оператора элементы собираются в одну последовательность.

Буферизация

При выполнении LINQ-запросов элементы обрабатываются последовательно в процессе перебора, отсутствует какая-либо буферизация результатов. Технология PLINQ предоставляет три режима буферизации.

По умолчанию используется авто-буферизация (`AutoBuffered`) -

объем буфера для вычисления результатов определяется исполняющей средой.

```
var numbers = ParallelEnumerable.Range(1, 10);
var query = numbers.Select(n =>
{
    Thread.Sleep(500);
    Console.WriteLine("Prepare item {0}, thread {1}", n, Thread.CurrentThread);
    return n;
});
foreach(int i in query)
    Console.WriteLine("Got item {0}", i);
```

При обращении к первому элементу запроса осуществляется параллельная обработка части элементов.

```
Prepare item 1, thread 10
Prepare item 6, thread 11
Prepare item 2, thread 10
Prepare item 7, thread 11
Prepare item 3, thread 10
Prepare item 8, thread 11
Prepare item 4, thread 10
Prepare item 9, thread 11
Prepare item 5, thread 10
Got item 1
Got item 2
Got item 3
Got item 4
Got item 5
Prepare item 10, thread 11
Got item 6
Got item 7
Got item 8
Got item 9
Got item 10
```

Так как число элементов достаточно небольшое, то практически все элементы были обработаны при первом обращении к запросу.

Полная буферизация (*fully-buffered*) позволяет выполнить запрос полностью до предоставления результатов вне зависимости от числа элементов.

```
var numbers = ParallelEnumerable.Range(1, 10);
var q = numbers
    .WithMergeOptions(ParallelMergeOptions.FullyBuffered)
{
    Thread.Sleep(500);
    Console.WriteLine("Prepare item {0}, thread {1}", n, Thread.CurrentThread);
    return n;
});
foreach(int i in query)
    Console.WriteLine("Got item {0}", i);
```

Обращение к первому элементу инициирует параллельную обработку всех элементов:

```
Prepare item 6, thread 10
Prepare item 1, thread 11
Prepare item 7, thread 10
Prepare item 2, thread 11
Prepare item 8, thread 10
Prepare item 3, thread 11
Prepare item 9, thread 10
Prepare item 4, thread 11
Prepare item 10, thread 10
Prepare item 5, thread 11
Got item 1
Got item 2
Got item 3
Got item 4
Got item 5
Got item 6
Got item 7
```

```
Got item 8  
Got item 9  
Got item 10
```

Режим с полной буферизацией обладает худшим временем получения первого элемента, но получение всей результирующей последовательности может занимать меньше времени, чем другие режимы.

Третий режим не использует буферизацию.

```
var numbers = ParallelEnumerable.Range(1, 10);  
var q = numbers  
.WithMergeOptions(ParallelMergeOptions.NotBuffered)  
{  
    Thread.Sleep(500);  
    Console.WriteLine("Prepare item {0}, thread {1}", n, Thread.CurrentThread.M  
    return n;  
}  
foreach(int i in query)  
    Console.WriteLine("Got item {0}", i);
```

Элементы вычисляются по мере обращения:

```
Prepare item 1, thread 6  
Got item 1  
Prepare item 6, thread 11  
Got item 6  
Prepare item 2, thread 6  
Got item 2  
Prepare item 7, thread 11  
Got item 7  
Prepare item 3, thread 6  
Got item 3  
Prepare item 8, thread 11  
Got item 8  
Prepare item 4, thread 6  
Got item 4
```

Prepare item 9, thread 11

Got item 9

Prepare item 5, thread 6

Got item 5

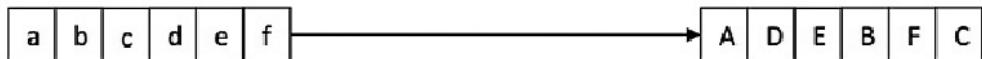
Prepare item 10, thread 11

Got item 10

Порядок элементов

При выполнении параллельных запросов не гарантируется сохранение порядка элементов в результирующей последовательности.

```
select Char.ToUpper(c)
```



Для гарантированного сохранения порядка необходимо использовать метод `AsOrdered`, который после агрегирования результатов выполняет восстановление порядка.

```
var q = "abcdefgh".AsParallel()
    .Select(c=>Char.ToUpper(c)).AsOrdered().ToArray();
```

Обязательное упорядочивание результатов приводит к ухудшению эффективности распараллеливания. Модификатор `AsUnordered` позволяет снять требование сохранения порядка и повысить эффективность распараллеливания.

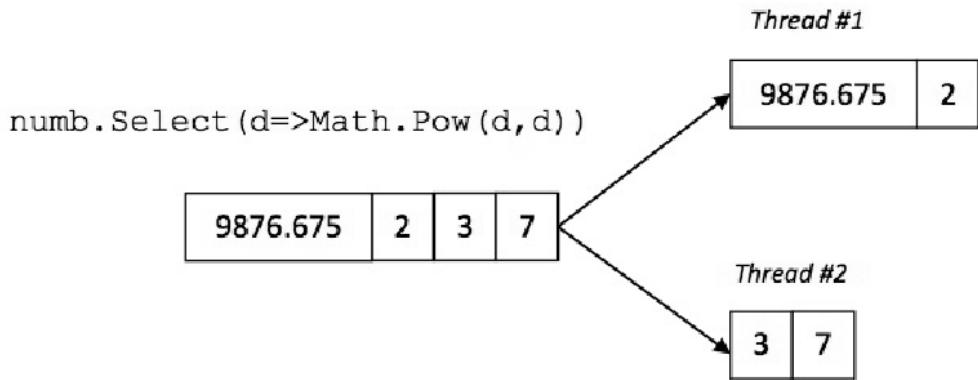
Разделение данных

Разбиение элементов последовательности по потокам осуществляется в соответствии с одной из трех стратегий: блочное разделение (`chunk partitioning`), разделение по диапазону (`range partitioning`), хэш-секционирование (`hash sectioning`).

Хэш-секционирование требует расчета хэш-значений для всех

элементов последовательности; элементы с одинаковыми хэш-значениями обрабатываются одним и тем же потоком. Хэш-секционирование выполняется для операторов, сравнивающих элементы: `GroupBy`, `Join`, `GroupJoin`, `Intersect`, `Except`, `Union`, `Distinct`.

При разделении по диапазону последовательность разбивается на равное число элементов, каждая порция обрабатывается в одном рабочем потоке. Такое разделение является достаточно эффективным, так как приводит к полной независимости обработки элементов на разных потоках и не требует какой-либо синхронизации. Недостатком такой декомпозиции является несбалансированность загруженности потоков в случае разных вычислительных затрат при обработке элементов последовательности.

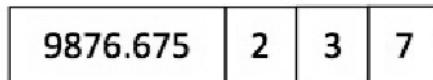
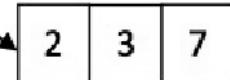


Разная вычислительная нагрузка при обработке элементов приводит к несбалансированности загрузки потоков.

Динамическое разделение данных позволяет получить более равномерную загрузку потоков, но требуют синхронизации потоков для доступа к исходной последовательности.

Thread #1

```
numb.Select(d => d * d)
```

**9876.675****Thread #2**

При динамическом (блочном) разделении каждый поток, участвующий в обработке, получает по фиксированной порции элементов (`chunk`). В качестве порции может быть и один элемент. После обработки своей порции поток обращается за следующей порцией.

По умолчанию при выполнении PLINQ-запросов выполняется разделение по диапазону, кроме запросов, требующих хэш-секционирования. Для выполнения запросов с динамическим разделением необходимо использовать объект `Partitioner`.

```
var parNumbers = ParallelEnumerable.Range(1, 1000);
// Range-partition
var q1 = (from n in parNumbers
          where n % 3 == 0
          select n * n).ToArray();
// Range-partition
double[] ard =
    new double[] {3.4, 56565.634, 7.8, 9.9, 2.4};
var q2 = ard.AsParallel().Select(d =>
    Math.Sqrt(d)).ToArray();
// Block-partition
var q3 = Partitioner.Create(ard, true).AsParallel()
    .Select(d=>Math.Sqrt(d)).ToArray();
```

Первый и второй запросы используют разделение по диапазону. Третий запрос использует динамическую декомпозицию. Второй аргумент метода `Partitioner.Create` задает режим декомпозиции: `false` – разделение по диапазону, `true` – динамическая (блочная) декомпозиция.

Обработка исключений

Исключения, которые могут произойти при выполнении PLINQ-запросов, обрабатываются также как и в случае задач с помощью объекта `AggregateException` в блоке `catch`. Блок `try` располагается там, где осуществляется фактический вызов обработки элементов.

```
var q = numbers.Select(n => SomeWork(n)).Where(n =>
{
    if(n > 0)
        return true;
    else
        throw new Exception();
});
try
{
    foreach(int n in q)
        Console.WriteLine(n);
}
catch(AggregateException ae)
{
    Console.WriteLine("Some error was happened!");
    return true;
}
```

Обработка элементов начинается при переборе элементов в `foreach`-цикле. Исключение обрабатывается в `catch`-блоке. Объект `AggregateException` содержит список ошибок, возникнувших при обработке элементов, в списке `InnerExceptions`. Для обработки исключений можно применять методы `Flatten` и `Handle`.

Отмена запроса

Для отмены выполнения PLINQ-запросов используется объект `CancellationToken`, который передается с помощью метода `WithCancellation`.

```
CancellationTokenSource cts = new CancellationTokenSource();
var q = someData.AsParallel().WithCancellation(cts.Token)
    .Select(d => d * d);

// Задача, которая отменит запрос
Task t = Task.Factory.StartNew(() =>
{
    Thread.Sleep(100);
    cts.Cancel();
});

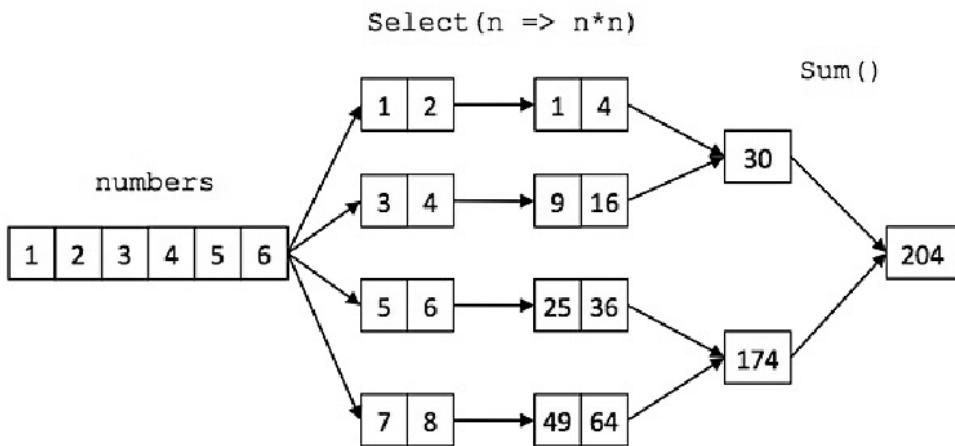
try
{
    var results = data.ToList();
}
catch(OperationCanceledException e)
{
    Console.WriteLine("The query was cancelled!");
}
```

В этом фрагменте отмена запроса осуществляется с помощью отдельной задачи. При отмене генерируется исключение `OperationCanceledException`, которое при необходимости можно обработать.

Агрегирование вычислений

Под агрегированием (редукцией) понимается вычисление какого-либо итогового значения для исходного набора элементов. Примером редукции являются: вычисление минимального значения, максимального значения, суммы, произведения и т.д.

Для получения агрегированных результатов можно использовать методы: Sum, Min, Max для вычисления суммы, минимального и максимального значения.



Реализация произвольных агрегированных функций осуществляется с помощью метода Aggregate. В следующем фрагменте реализован метод вычисления среднеквадратичного отклонения

```

double CalcStdDevParallel(double[] data)
{
    double mean = data.AsParallel().Average();
    double stdDev = data.AsParallel().Aggregate(
        // Инициализация локальной переменной
        0.0,
        // Вычисления в каждом потоке
        (subtotal, item) =>
        subtotal + Math.Pow(item - mean, 2),
        // Агрегирование локальных значений
        (total, subtotal) =>
        total + subtotal,
        // Итоговое преобразование
        (total) =>
        Math.Sqrt(total/(data.Length - 1))
    );
    return stdDev;
}
    
```

Первый аргумент – делегат инициализации локальных переменных потоков, который вызывается один раз для каждого потока перед началом вычислений.

$() \Rightarrow 0.0;$

Второй аргумент – делегат обработки каждого элемента, принимающий в качестве параметров текущее значение локальной переменной и значение обрабатываемого элемента; возвращается значение локальной переменной. Делегат обработки вызывается для каждого элемента.

$(local, item) \Rightarrow local + item;$

Третий аргумент метода агрегирования принимает делегат редукции, определяющий, как частные переменные будут сворачиваться (редуцироваться) в одно итоговое значение. Делегат редукции принимает текущее значение итоговой переменной, локальное значение переменной и возвращает обновленное значение итоговой переменной.

$(total, local) \Rightarrow total + local;$

Делегат редукции вызывается столько раз, сколько потоков участвует в обработке метода агрегирования.

Четвертый аргумент – делегат финальной обработки итоговой переменной. Делегат вызывается только один раз после завершения редукции локальных переменных.

$(total) \Rightarrow Math.Sqrt(total) / (data.Length - 1)$

Вопросы

1. Почему LINQ-запросы не распараллеливаются автоматически?
2. С какой целью используется оператор возвращения к

последовательному выполнению запроса AsSequential?

3. Почему при выполнении параллельного запроса порядок элементов может сохраняться?

Упражнения

1. Составьте запросы, которые демонстрируют неэффективность статической декомпозиции.
2. Исследуйте эффективность выполнения PLINQ-запросов и шаблона Parallel.For.
3. Составьте запрос, с помощью которого можно убедиться в параллельности или последовательности его выполнения.
4. Исследуйте эффективность выполнения запроса с разными режимами буферизации.

Знакомство с "Визуализатором параллелизма" в Visual Studio 12

Реализация PLINQ-запросов.

Задачи

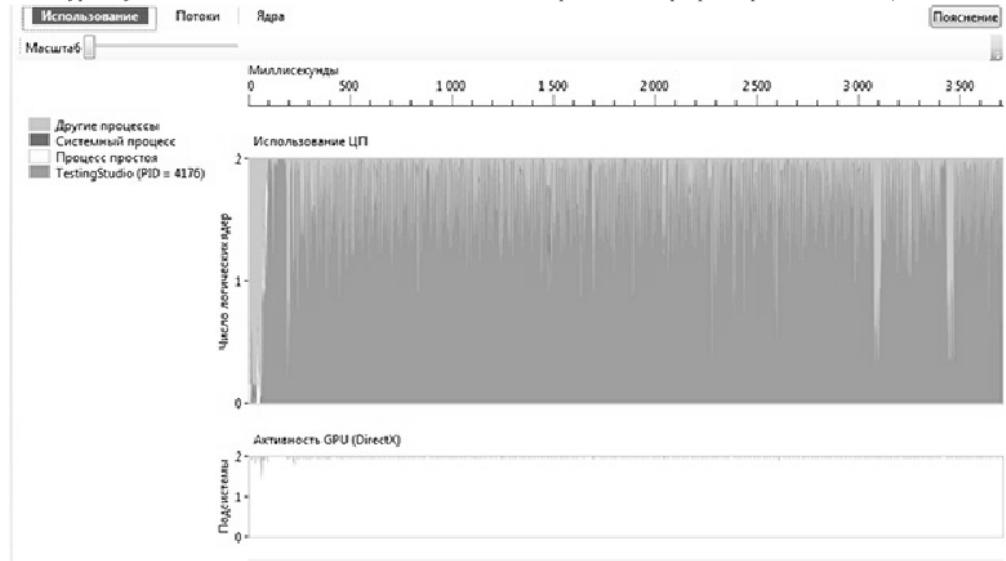
1. Выполните анализ загруженности вычислительной системы при разном режиме ожидания потока.
2. Выполните анализ автоматического распараллеливания циклической обработки шаблоном `Parallel.For`.
3. Выполните анализ распараллеливания PLINQ-запросов.

Методические указания

Среда разработки Visual Studio 12 содержит полезный инструмент для анализа эффективности выполнения параллельной программы – "Визуализатор параллелизма". Для запуска инструмента на вкладке "Анализ" запускаем "Визуализатор параллелизма" -> "Выполнить анализ с текущим процессом". Инструмент запускает программу и собирает информацию о её фактическом исполнении на данной вычислительной системе. Основные вкладки результатов: "Использование", "Потоки", "Ядра".

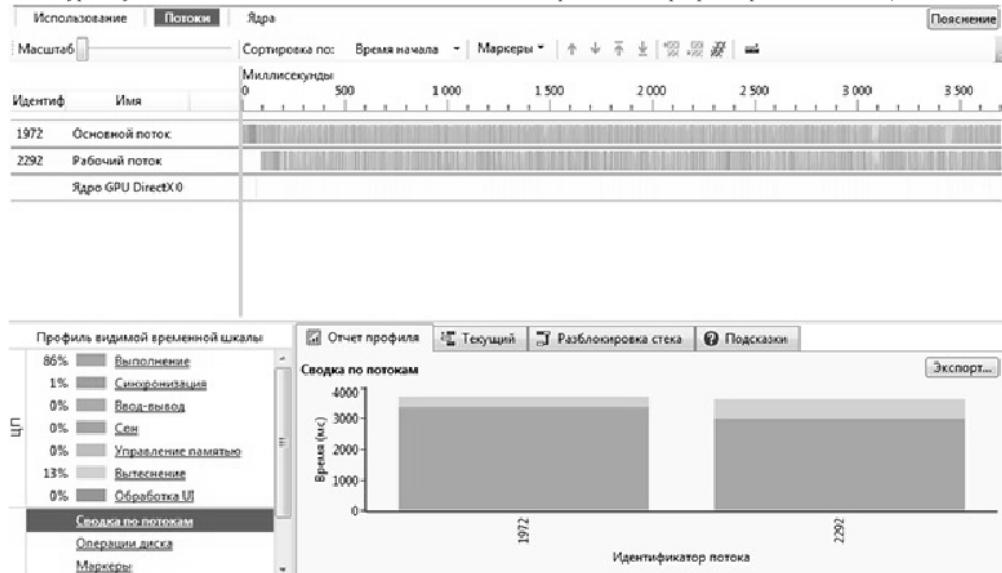
Использование ЦП

Вкладка содержит информацию об общей загрузке ЦП приложением в течение всего интервала выполнения.



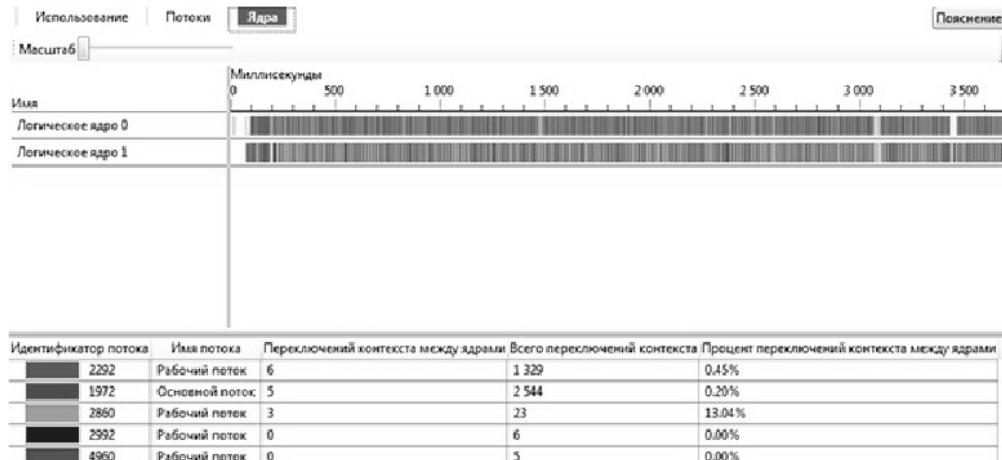
Потоки

Вкладка "Потоки" выводит информацию о выполнении потоков, как пользовательских, так и рабочих потоков пула. Если в программе не используются объекты `ThreadPool`, `Parallel`, `Task`, то рабочие потоки пристаивают. Для удобства восприятия информацию об отдельных потоках, например, пристаивающих, можно скрыть. График "Сводка по потокам" суммирует время нахождения потоков в том или ином состоянии.



Ядра

Вкладка "Ядра" позволяет проанализировать перемещение потоков по фактическим исполнителям - ядрам процессора.



Анализ блокировки потоков

Применение "Визуализатора" позволяет исследовать особенности

разных средств синхронизации.

Следующие фрагменты осуществляют блокировку основного потока с помощью цикла ожидания (активная блокировка) и с помощью встроенного механизма Join с выгружением контекста потока.

// Фрагмент 1

```
Thread t = new Thread(() => {  
    for (int i=0; i<10000; i++)  
        for(int j =0; j < 100; j++)  
            a[i] += Math.Sin(j) + i;  
});
```

t.Start();

// Цикл ожидания

```
while(t.IsAlive) ;
```

// Фрагмент 2

```
Thread t = new Thread(() => {  
    for (int i=0; i<10000; i++)  
        for(int j =0; j < 100; j++)  
            a[i] += Math.Sin(j) + i;
```

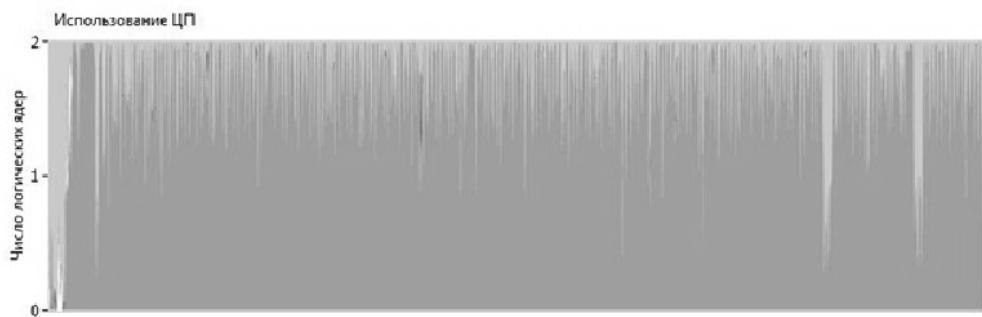
});

t.Start();

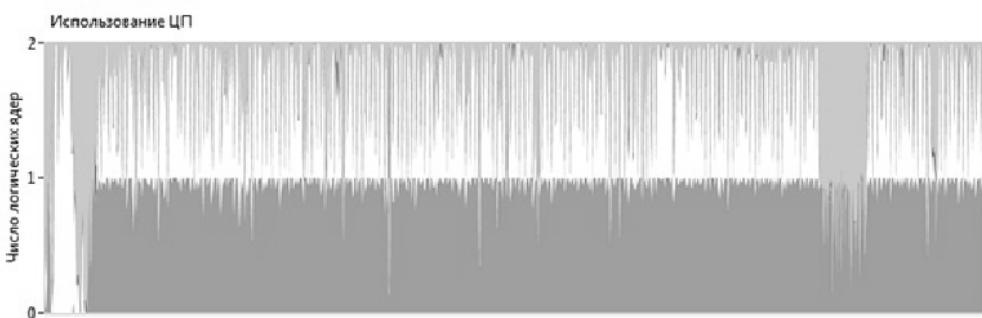
t.Join();

Окно "Использование ЦП" иллюстрирует основной недостаток циклической блокировки – полезную работу осуществляет один поток, а вычислительные ресурсы заняты двумя потоками

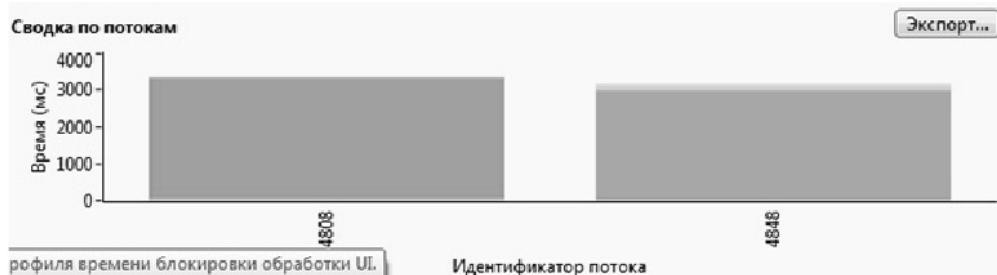
Циклическая блокировка



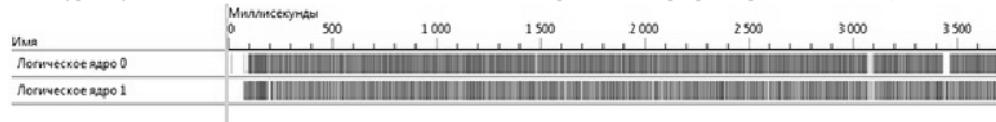
Блокировка Join



При `Join`-блокировке основной поток №4808 почти все время находится в состоянии "Синхронизация", ожидая завершения потока №4848.



Вкладка "ядра" позволяет зафиксировать интересную особенность блокировок. При циклическом ожидании работают два потока (рабочий и основной), на двухъядерной системе потоки практически не перемещаются между ядрами.



При блокировке основного потока методом `Join`, вычислительная система с двумя ядрами полностью предоставлена одному рабочему потоку. Операционная система активно перемещает поток с одного ядра на другое.



Идентификатор потока	Имя потока	Переключений контекста между ядрами	Всего переключений контекста	Процент переключений контекста между ядрами
4848	Рабочий поток	929	1929	48.16%
4808	Основной поток	10	84	11.90%
4948	Рабочий поток	2	30	6.67%
4724	Рабочий поток	2	7	28.57%
4276	Рабочий поток	1	8	12.50%

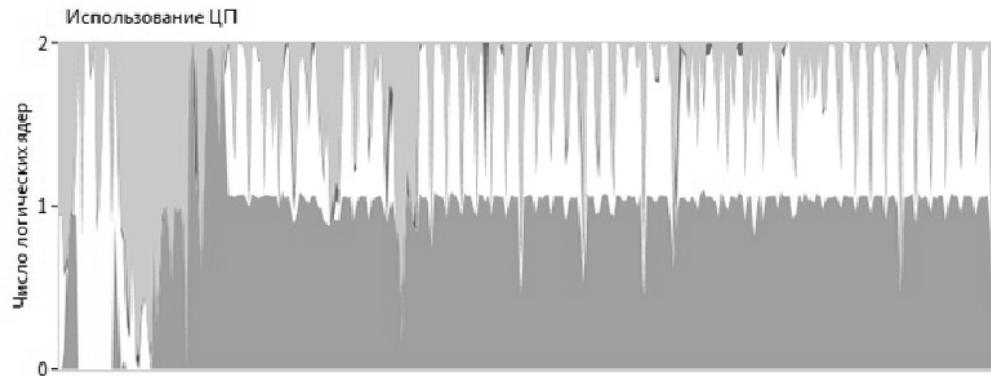
При таком выполнении существуют дополнительные накладные расходы, связанные с переключением контекста между ядрами (процент переключений между ядрами 48.16% от общего числа переключений).

При выполнении полезной обработки двумя и большим числом потоков загруженность системы увеличится, и доля переключений между ядрами будет минимальной.

Блокировка `SpinWait` комбинирует два типа ожидания.

```
// Фрагмент 1
Thread t = new Thread(() => {
    for (int i=0; i<10000; i++)
        for(int j =0; j < 100; j++)
            a[i] += Math.Sin(j) + i;
});

t.Start();
// Цикл ожидания
SpinWait.SpinUntil(() => !t.IsAlive);
```

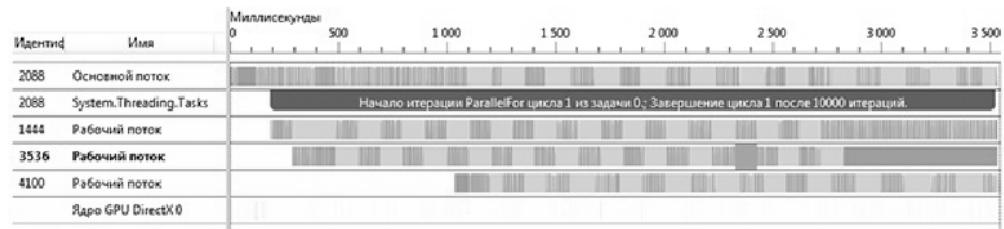


В начале цикла ожидания выполняется несколько прокруток, после чего ожидающий поток выгружается, освобождая вычислительные ресурсы.

Анализ выполнения объектов библиотеки TPL

Применение объектов TPL для распараллеливания скрывает от программиста работу по созданию и синхронизации потоков. "Визуализатор параллелизма" позволяет получить основную информацию о действиях среды выполнения по распараллеливанию задач.

Метод `Parallel.For` автоматически распределяет итерации цикла по рабочим потокам. Вкладка "Потоки" раскрывает особенности выполнения параллельной циклической обработки.



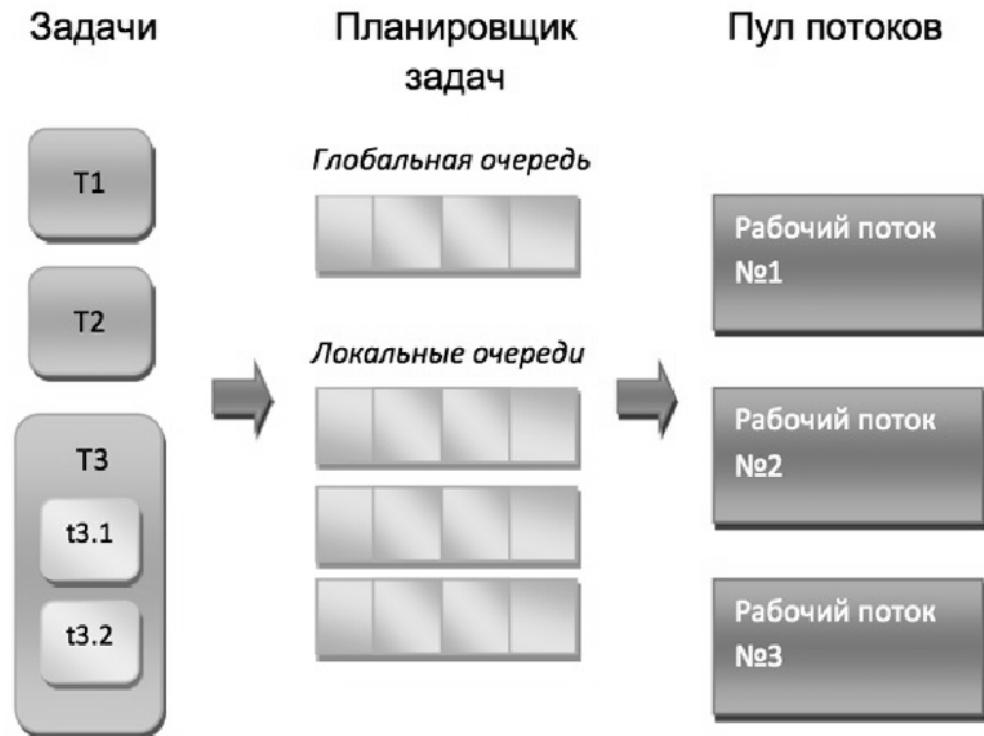
Специальная строка `System.Threading.Tasks` указывает на вызов и работу объекта TPL. В окне "Сводка" отображается итоговая информация о выполнении цикла. Рассматривая работу потоков,

видим, что в каждый момент времени работают только два потока, другие потоки вытесняются. Основной поток также участвует в обработке цикла. Сначала для обработки используются два дополнительных рабочих потока, спустя какое-то время в работу включается еще один поток.

Планировщик задач

Организация планировщика: глобальная очередь задач, локальные очереди рабочих потоков. Стратегии планировщика: WorkStealing, Inlined threading, Inject threads Опции задач: LongRunning, PreferFairness.

Планировщик задач играет связующую роль между задачами (рабочими элементами) и потоками. Множество задач приложения выполняется одними и теми же рабочими потоками, число которых динамически оптимизируется планировщиком с учетом возможностей вычислительной системы, фактической загруженности системы и прогрессом выполнения задач. Планировщик задач включает в себя очереди задач (одна глобальная и множество локальных очередей), стратегии распределения задач и рабочие потоки, которые фактически выполняют задачи.



Организация параллельного приложения с помощью пула потоков позволяет избежать накладных расходов, связанных с запуском и

завершением потоков для каждого фрагмента в программе, допускающего распараллеливание.

Таким образом, приложение состоит из основного потока, в котором могут запускаться новые пользовательские потоки, и набора рабочих потоков. Рабочие потоки существуют на протяжении всей жизни приложения; изначально находятся в "спящем" состоянии, но могут быть задействованы для обработки пользовательских задач. Рабочие потоки включаются в обработку, если в коде программы есть работа с объектами библиотеки Task Parallel Library (Task, Parallel, PLINQ) или непосредственная работа с пулем потоков ThreadPool.

В следующем фрагменте проверяем, действительно ли рабочие потоки осуществляют обработку пользовательских задач:

```
static void Main()
{
    Action IsPool = () => {
        Console.WriteLine(Thread.CurrentThread.IsThreadPoolThread);
    }

    Console.WriteLine("Parallel.Invoke");
    Parallel.Invoke(IsPool, IsPool, IsPool, IsPool, IsPool);

    Console.WriteLine("Parallel.For");
    Parallel.For(0, 5, i => IsPool());

    Console.WriteLine("Task");
    Task.Factory.StartNew(IsPool).Wait();

    Console.WriteLine("Thread");
    new Thread(() => IsPool()).Start();
}
```

Итак, получаем, что задача Task выполняется в пуле, методы Parallel.Invoke, Parallel.For не явно создают задачи, которые тоже выполняются в пуле. Метод Main и пользовательский поток не являются рабочими потоками пула. Часть работы,

определенной в методах `Parallel.Invoke`, `Parallel.For`, может выполнять в основном потоке.

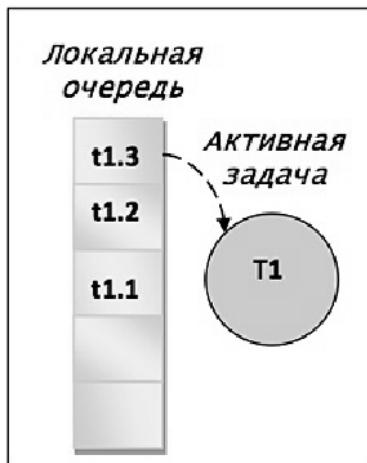
Пул потоков состоит из глобальной очереди, организованной по принципу FIFO, и множества локальных очередей, организованных по принципу LIFO. Дисциплина очереди FIFO (first-in, first-out) предполагает, что первый добавленный элемент первым поступит на обработку. Рабочие потоки обращаются к глобальной очереди и получают задачу на обработку. Доступ нескольких потоков к одному ресурсу – очереди задач – требует применения синхронизации для избегания проблемы гонки данных. Синхронизированный доступ потоков снижает быстродействие обработки. Чем меньше "вычислительная нагрузка" задач, тем чаще потоки соревнуются за доступ к очереди.

Для уменьшения накладных расходов, связанных с необходимостью синхронизации доступа потоков к глобальной очереди, в структуру пула потоков введены локальные очереди. Каждая локальная очередь соответствует одному рабочему потоку.

Локальные очереди предназначены для вложенных задач. Задачи, которые объявляются и запускаются из пользовательского потока, являются задачами верхнего уровня (top-most level tasks). Такие задачи помещаются в глобальную очередь и распределяются по рабочим потокам. При выполнении задачи верхнего уровня рабочий поток добавляет в свою локальную очередь все вложенные задачи.



Рабочий поток №1



Рабочий поток №2



Вложенные задачи помещаются в локальную очередь рабочего потока, в котором выполняется родительская задача. Локальные очереди организованы по принципу LIFO (last-in, first-out). Первой вложенной задачей, которая будет выполняться рабочим потоком, будет та задача, которая добавлена в очередь последней. Такой порядок объясняется возможной локальностью данных. Последняя добавленная вложенная задача использует общие данные в ближайшем окружении.

```
Task t = Task.Factory.StartNew(() =>
{
    // готовим данные для подзадачи 1
    data1 = f1(data);
    // добавляем в очередь подзадачу 1
    Task t1 = Task.Factory.StartNew(() => { work(data1); });

    // готовим данные для подзадачи 2
    data2 = f2(data);
    Task t2 = Task.Factory.StartNew(() => { work(data2); });
});
```

```
// готовим данные для подзадачи 3
data3 = f3(data);
Task t3 = Task.Factory.StartNew(() => {work(data3); });

// ожидаем завершения подзадач
Task.WaitAll(t1, t2, t3);
});
```

Благодаря стратегии планировщика `inlined execution` (см. ниже), не стоит опасаться блокировки потока, если последняя вложенная задача ссылается на предыдущую подзадачу (ожидает завершения или события из предыдущей задачи).

Опция PreferFairness

Если обратный порядок выполнения задач не желателен, то можно использовать опцию `PreferFairness` при создании задач. Такие задачи помещаются в глобальную очередь. Организация глобальной очереди по принципу `FIFO` позволяет надеяться, что вложенные задачи, созданные с опцией `PreferFairness`, будут выполняться в порядке добавления.

Рассмотрим следующий фрагмент:

```
static void Main()
{
    Task tMain = Task.Factory.StartNew(() => {
        t1 = Task.Factory.StartNew(() => ..);
        t2 = Task.Factory.StartNew(() => ..);
        t3 = Task.Factory.StartNew(() => ...,
            TaskCreationOptions.PreferFairness);
        t4 = Task.Factory.StartNew(() => ...,
            TaskCreationOptions.PreferFairness);
    });
}
```

В этом фрагменте – одна задача верхнего уровня, которая помещается в

глобальную очередь. При обработке задачи `tMain` в одном из рабочих потоков вложенные задачи `t1` и `t2` помещаются в локальную очередь данного рабочего потока. Задачи `t3` и `t4` созданы с опцией `PreferFairness`, поэтому добавляются в глобальную очередь. Порядок обработки вложенных задач локальной очереди – обратный порядку добавления. Сначала будет обрабатываться задача `t2`, затем `t1`. Задачи `t3` и `t4` обрабатываются в порядке добавления, т.е. сначала `t3`, затем `t4`. В каком рабочем потоке фактически будет выполняться задачи `t1`, `t2`, `t3` и `t4` зависит от загруженности других рабочих потоков (см. механизм `WorkStealing`). Но можно сказать, что для задач `t1` и `t2` созданы предпосылки последовательного выполнения в том же рабочем потоке, что и родительская задача `tMain`. Для задач `t3` и `t4` созданы предпосылки параллельного выполнения в других рабочих потоках путь. Следует не забывать, что выполнение вложенных задач в других рабочих потоках может быть сопряжено с накладными расходами при использовании данных родительской задачи.

Таким образом, опция `PreferFairness` применяется не для изменения порядка выполнения задач (для этого проще изменить порядок добавления вложенных задач в локальную очередь), а для обеспечения условий параллельного выполнения вложенных задач в разных рабочих потоках.

Оптимизация выполнения задач выполняется с помощью трех стратегий планировщика: стратегия заимствования задач (`work-stealing`), стратегия динамического изменения числа рабочих потоков (`thread-injection`), стратегия `inlined execution`.

Стратегия Work Stealing

Стратегия `work-stealing` заключается в том, что свободный рабочий поток при условии отсутствия ожидающих задач в глобальной очереди заимствует задачу у одного из загруженных рабочих потоков. Для решения проблемы синхронизации доступ к каждой локальной очереди осуществляется с помощью двух ключей: `private key` – указатель на последний добавленный элемент (задачу), который используется только одним рабочим потоком - держателем очереди; `public key` – указатель на первый добавленный элемент, который

используется сторонними рабочими потоками для заимствования (work-stealing).



Синхронизация реализована только для открытого ключа, так как существует возможность обращения нескольких свободных рабочих потоков к локальной очереди занятого потока. Механизм заимствования задач вносит корректиды в обработку вложенных задач. Если заимствование задач не происходит, то вложенные задачи выполняются последовательно в одном рабочем потоке в порядке обратном добавлению. При заимствовании вложенные задачи могут выполняться параллельно в разных рабочих потоках.

Стратегия Inlined Execution

Стратегия *inlined execution* применяется в случае блокировки выполняющейся задачи. Если выполняющаяся задача блокируется вызовом ожидания завершения задачи, которая находится в локальной очереди этого же рабочего потока, то планировщик выполняет задачу из очереди. Такая стратегия позволяет избежать взаимоблокировки рабочего потока (*dead-lock*), которая может возникнуть при определенном порядке вложенных задач в локальной очереди. Стратегия *inlined execution* применяется только для задач в локальной очереди блокированного рабочего потока.

Рассмотрим следующий фрагмент:

```
static void Main()
{
    Task tParent = Task.Factory.StartNew(() =>
```

```
{  
    tChild1 = Task.Factory.StartNew(() => ..);  
    tChild2 = Task.Factory.StartNew(() => ..);  
  
    tChild1.Wait();  
});  
}
```

В задаче `tParent` поочередно добавляются в локальную очередь `tChild1`, `tChild2`. Предполагается, что сначала должна завершить свою работу родительская задача, затем управление будет передано задаче `tChild2`, и только после этого задаче `tChild1`. Но оператор ожидания `Wait` блокирует родительскую задачу. Без стратегии планировщика `Inlined thread` текущий рабочий поток был бы заблокированным. Но в действительности планировщик (среда выполнения) фиксирует вызов `Wait` и передает управление ожидаемой задаче `tChild1`.

Стратегия `Inlined execution` работает при вызовах `Wait` для конкретной задачи в локальной очереди и при вызове `WaitAll` для ожидания нескольких задач локальной очереди. Вызов `WaitAny` не приводит к передаче управления вложенным задачам. Это связано с тем, что в случае метода `WaitAny` возникает ситуация неопределенности – кому передать управление? Вложенная задача, которая будет выбрана первой для выполнения и будет победительницей. Метод ожидания `WaitAny` изначально предполагает соревновательность выполнения задач с целью выявления победителя – задачи, первой завершившей выполнение.

Стратегия Thread-Injection

Стратегия `thread-injection` применяется для оптимизации числа рабочих потоков. Применяются два механизма динамического изменения числа рабочих потоков. Первый механизм применяется с целью избежать блокировки рабочих потоков: планировщик добавляет еще один рабочий поток, если не наблюдает прогресса при выполнении задачи. При этом планировщик не различает, находится ли поток

действительно в заблокированном состоянии, ожидая какого-либо события, или поток загружен полезной длительной работой (`long-running task`). В случае выполнения "длинных" задач такая стратегия планировщика не улучшает, а даже ухудшает производительность. При росте числа потоков возникает конкуренция за физические ядра вычислительной системы, появляются накладные расходы на переключение контекстов. Чтобы избежать неправильных действий планировщика рекомендуется делать более "короткие" задачи, а длинные вычислительно-ёмкие задачи запускать с опцией `LongRunning`:

```
Task t = Task.Factory.StartNew(VeryLongComputing,  
    TaskCreationOptions.LongRunning);
```

Эта опция вынуждает планировщик выделить для задачи собственный поток и не контролировать прогресс выполнения данной задачи. Поток "длинной" задачи не входит в группу рабочих потоков пула, поэтому планировщик не применяет стратегии оптимизации работы этого потока и не использует поток для выполнения других задач, ожидающих в пуле.

Второй механизм стратегии `thread-injection` добавляет или удаляет рабочие потоки в зависимости от результатов выполнения предыдущих задач. Если увеличение числа потоков привело к росту производительности, то планировщик увеличивает число потоков. Если производительность не растет, то число потоков сокращается. Эвристический алгоритм стремится найти оптимальное число потоков при текущей загрузке вычислительной системы. Уменьшение объема задач также помогает планировщику найти оптимальный вариант. Второй механизм дополняет первый и позволяет избежать неограниченного добавления рабочих потоков без улучшения производительности.

Вопросы

1. В каком потоке будут выполняться вложенные задачи, если родительская задача запущена с опцией `LongRunning`?
2. Возможна ли ситуация блокировки потока при ожидании

завершения задачи, находящейся в глобальной очереди?

3. В каких случаях возможно параллельное выполнение вложенных задач?
4. В каких случаях нарушается порядок выполнения вложенных задач?

Упражнения

1. Напишите код, который позволяет оценить достоинства и/или недостатки организации локальной очереди потока по принципу LIFO.
2. Напишите код с вычислительно-емкими задачами и оцените эффективность выполнения задач в пуле потоков и в сторонних потоках.

Типовые модели параллельных приложений

Шаблон MapReduce. Реализация с помощью PLINQ-запросов. Шаблон Scan/Fold. Реализация с помощью Parallel.For. Распараллеливание рекурсивных алгоритмов.

Существуют следующие распространенные модели параллельных приложений:

- модель делегирования ("управляющий-рабочий");
- сеть с равноправными узлами;
- конвейер ("производители-потребители");

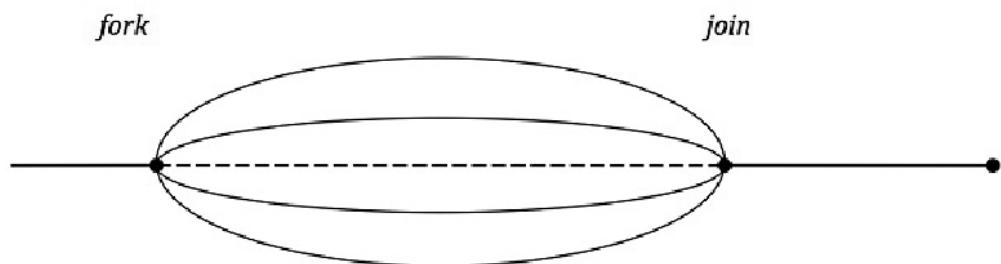
Каждая модель характеризуется собственной декомпозицией работ, которая определяет, кто отвечает за порождение подзадач и при каких условиях они создаются, какие информационные зависимости между подзадачами существуют.

Модель	Описание
Модель делегирования	Центральный поток ("управляющий") создает "рабочие" потоки и назначает каждому из них задачу. Управляющий поток ожидает завершения работы потоков, собирает результаты.
Модель с равноправными узлами	Нет центрального узла, все потоки участвуют в обработке.
Конвейер	Конвейерный подход применяется для поэтапной обработки потока входных данных. Входные данные обрабатываются строго последовательно.

Модель делегирования

В модели делегирования выделяется один центральный поток (менеджер, управляющий, мастер) и несколько рабочих потоков. Управляющий поток запускает рабочие потоки, передает им все необходимые данные, контролирует работу и обрабатывает результаты после их завершения.

К модели делегирования относится так называемая схема "fork-join". Этап "fork" (разветвление) – делегирование полномочий рабочим потокам: создание, запуск, передача параметров. Этап "join" (присоединение) – ожидание завершения работы потоков.



Самым простым способом реализации схемы fork-join является применение шаблона `Parallel.Invoke`:

```
Parallel.Invoke(worker1, worker2, worker3);
```

Шаблон параллельно запускает рабочие элементы и блокирует основной поток до завершения работы.

Если в главном потоке выполняется какая-либо работа параллельно с рабочими потоками, то можно использовать объекты `Task` и встроенные механизмы ожидания.

```
Task t1 = Task.Factory.StartNew(..);
Task t2 = Task.Factory.StartNew(..);
Task t3 = Task.Factory.StartNew(..);
// Параллельная работа центрального узла
fManager();
// Ожидаем завершения работы
Task.WaitAll(t1, t2, t3);
// Обрабатываем результаты
fResults();
```

Объект синхронизации `CountdownEvent` позволяет выполнять координацию работы управляемого и рабочих потоков. Объект

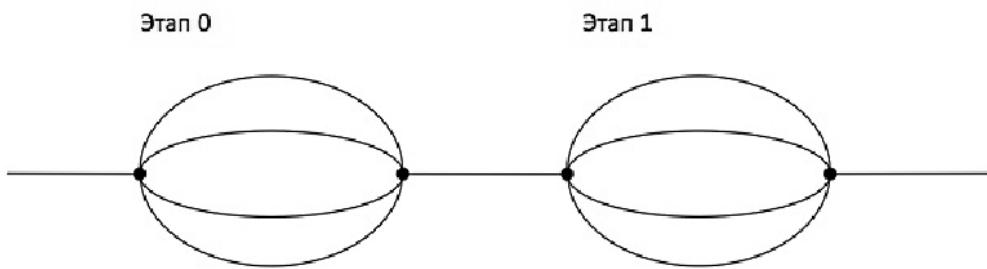
обладает внутренним счетчиком, который устанавливается при инициализации объекта. При завершении работы или достижении какого-либо этапа рабочий поток вызывает метод `Signal`, уменьшающий внутренний счетчик. Когда внутренний счетчик становится равным нулю, объект сигнализирует потоку, ожидающему с помощью метода `Wait`.

```
static void Main()
{
    int N = 5;
    CountdownEvent ev = new CountdownEvent(N);
    Thread[] workers = new Thread[N];
    for(int i=0; i<N; i++)
    {
        int y = i;
        workers[i] = new Thread(() => {
            DoSomeWork1(y);
            ev.Signal();
            DoSomeWork2(y);
        });
        workers[i].Start();
    }
    while(!ev.IsSet)
    {
        ev.Wait(TimeSpan.FromSeconds(5));
        Console.WriteLine("{0}-рабочих закончили",
            ev.CurrentCount);
        SaveCurrentResults();
    }
}
```

При инициализации объекта указывается начальное значение внутреннего счетчика объекта. Каждый рабочий поток при завершении работы уменьшает значение счетчика с помощью метода `Signal`. Уменьшение счетчика осуществляется потокобезопасно. Когда значение счетчика уменьшится до нуля, объект синхронизации сигнализирует ожидающему главному потоку, при этом главный поток разблокируется.

Кроме блокирующего ожидания с помощью метода `Wait`, также можно использовать перегрузку, принимающую в качестве аргумента интервал в миллисекундах, в течение которого поток блокируется. Текущее значение внутреннего счетчика можно узнать с помощью свойства `CurrentCount`, свойство `IsSet` позволяет определить установлен ли сигнал или нет.

Объект `Barrier` инкапсулирует барьерную синхронизацию с возможностью нескольких этапов. При барьерной синхронизации несколько участников могут параллельно выполнять свою работу. При завершении работы участники дожидаются остальных. После завершения работы всех участников выполняется финальный обработчик на данном этапе. После завершения работы финального обработчика начинается следующий этап, и работа участников возобновляется.



При инициализации объекта указывается число участников барьерной синхронизации и делегат, вызываемый в конце каждого этапа.

```
Barrier bar = new Barrier(3, (bar) =>
{
    Console.WriteLine("Phase: {0}",
        bar.CurrentPhaseNumber);
});
```

```
Action worker = () => {
    Work1();
    bar.SignalAndWait();
    Work2();
    bar.SignalAndWait();
    Work3();
```

```
};

var w1 = worker; var w2 = worker; var w3 = worker;
Parallel.Invoke(w1, w2, w3);
```

Метод `SignalAndWait` сигнализирует о завершении работы данным участником и блокирует поток до завершения работы всех участников. Объект `Barrier` позволяет изменять число участников в процессе работы.

Рекурсивные алгоритмы относятся к моделям делегирования. В качестве примера рассмотрим алгоритм быстрой сортировки. Алгоритм рекурсивно разбивает диапазон чисел на два диапазона в соответствии с выбранным ведущим элементом – левый диапазон содержит только числа, меньшие или равные ведущему элементу; правый диапазон содержит числа, большие ведущего элемента. Распараллеливание алгоритма сводится к одновременному выполнению обработки левого и правого диапазона.

```
static void ParallelSort(T[] data, int startIndex, int endIndex,
    IComparer<T> comparer,
    int minBlockSize=10000) {

    if (startIndex < endIndex) {
        // мало элементов – выполняем последовательную сортировку
        if (endIndex - startIndex < minBlockSize) {
            // Последовательная сортировка
            Array.Sort(data, startIndex,
                endIndex - startIndex + 1, comparer);
        } else {
            // Определяем ведущий элемент
            int pivotIndex = partitionBlock(data, startIndex,
                endIndex, comparer);
            // обрабатываем левую и правую часть
            Action leftTask = () =>
            {
                ParallelSort(data, startIndex,
                    pivotIndex - 1, comparer,
                    comparer);
                ParallelSort(data, pivotIndex + 1, endIndex,
                    comparer);
            };
        }
    }
}
```

```
    depth + 1, maxDepth, minBlockSize);
}
Action rightTask = () =>
{
    ParallelSort(data, pivotIndex + 1,
endIndex, comparer,
depth + 1, maxDepth, minBlockSize);
}
// wait for the tasks to complete
Parallel.Invoke(leftTask, rightTask);
}
}
```

// Осуществляем перераспределение элементов

```
static int partitionBlock(T[] data, int startIndex,
int endIndex, IComparer<T> comparer) {
```

// Ведущий элемент

```
T pivot = data[startIndex];
```

```
// Перемещаем ведущий элемент в конец массива
swapValues(data, startIndex, endIndex);
```

// индекс ведущего элемента

```
int storeIndex = startIndex;
```

// цикл по всем элементам массива

```
for (int i = startIndex; i < endIndex; i++) {
```

// ищем элементы меньшие или равные ведущему

```
if (comparer.Compare(data[i], pivot) <= 0) {
```

// перемещаем элемент и увеличиваем индекс

```
swapValues(data, i, storeIndex);
```

```
    storeIndex++;
}
```

```
}
```

```
swapValues(data, storeIndex, endIndex);
```

```
return storeIndex;
```

```
}
```

// Обмен элементов

```
static void swapValues(T[] data,
```

```
int firstIndex, int secondIndex) {
```

```
T holder = data[firstIndex];
    data[firstIndex] = data[secondIndex];
    data[secondIndex] = holder;
}

static void Main(string[] args) {
    // generate some random source data
    Random rnd = new Random();
    int[] sourceData = new int[5000000];
    for (int i = 0; i < sourceData.Length; i++) {
        sourceData[i] = rnd.Next(1, 100);
    }

    QuickSort(sourceData, new IntComparer());
}
```

Основная проблема рекурсивных алгоритмов заключается в снижении эффективности при большой глубине рекурсии. Для ограничения рекурсивного разбиения множества данных применяется пороговая величина `MinBlock`. Если число элементов в блоке незначительно, то выполняется нерекурсивная сортировка (пузырьковая или сортировка со вставками). Распараллеливание быстрой сортировки приводит к еще одному источнику накладных расходов – рекурсивное порождение задач, конкурирующих за рабочие потоки пула. При использовании пользовательских потоков (работа с объектами `Thread`) конкуренция будет фатальной – рекурсивное порождение потоков приводит к значительным накладным расходам. Для контроля степени параллелизма применяют несколько подходов. Самый простой способ заключается в контроле глубины рекурсии – если глубина рекурсии превышает некий порог, то выполняется последовательная быстрая сортировка.

```
static void ParallelSort(T[] data, int startIndex, int endIndex,
    IComparer<T> comparer,
    int minBlockSize=10000,
    int depth = 0,
    int MaxDepth)
```

```
{  
    // Последовательная сортировка  
    if (endIndex - startIndex < minBlockSize)  
        InsertionSort(data, startIndex, endIndex, comparer);  
    else  
    {  
        // Определяем ведущий элемент  
        int pivotIndex = partitionBlock(data, startIndex,  
                                         endIndex, comparer);  
        // обработчик левой части  
        Action leftTask = () =>  
        {  
            ParallelSort(data, startIndex,  
                         pivotIndex - 1, comparer,  
                         depth + 1, maxDepth, minBlockSize);  
        };  
        // обработчик правой части  
        Action rightTask = () =>  
        {  
            ParallelSort(data, pivotIndex + 1,  
                         endIndex, comparer,  
                         depth + 1, maxDepth, minBlockSize);  
        };  
  
        if(depth >= MaxDepth)  
        {  
            leftTask();  
            rightTask();  
        }  
        else  
        {  
            Parallel.Invoke(leftTask, rightTask);  
        }  
    }  
}
```

Глубина рекурсии является простым критерием, но не оптимальным. При плохом выборе ведущего элемента, блоки будут неравномерными, и глубина рекурсии для обработки каждого блока будет различной. Если правая часть содержит мало элементов, то обработка правой части будет завершена достаточно быстро. Поэтому распараллеливание обработки левой части может осуществляться и при большей глубине, чем задано параметром `MaxDepth`. Реализовать "адаптивный" параллелизм можно с помощью разделяемого счетчика фактических выполняющихся параллельных вызовов. При параллельном запуске быстрой сортировки - счетчик увеличивается, при завершении параллельных вызовов – счетчик уменьшается. Изменения счетчика необходимо выполнять атомарно с помощью методов `Interlocked.Increment`, `Interlocked.Decrement`.

```
static int parallelCalls;
static void ParallelSort(T[] data, int startIndex, int endIndex,
    IComparer<T> comparer,
    int minBlockSize=10000)
{
    // Последовательная сортировка
    if (endIndex - startIndex < minBlockSize)
        InsertionSort(data, startIndex, endIndex, comparer);
    else
    {
        // Определяем ведущий элемент
        int pivotIndex = partitionBlock(data, startIndex,
            endIndex, comparer);
        // обработчик левой части
        Action leftTask = () =>
        {
            ParallelSort(data, startIndex,
                pivotIndex - 1, comparer,
                depth + 1, maxDepth, minBlockSize);
        });
        // обработчик правой части
        Action rightTask = () =>
        {
            ParallelSort(data, pivotIndex + 1,
                endIndex, comparer,
                depth + 1, maxDepth, minBlockSize);
        });
    }
}
```

```
    depth + 1, maxDepth, minBlockSize);  
});
```

```
if (parallelCalls > MaxParallelCalls)  
{  
    leftTask();  
    rightTask();  
}  
else  
{  
    Interlocked.Increment(ref parallelCalls);  
    Parallel.Invoke(leftTask, rightTask);  
    Interlocked.Decrement(ref parallelCalls);  
}  
  
}  
}
```

Максимальное число параллельных вызовов `MaxParallelCalls` можно выбирать в зависимости от числа ядер вычислительной системы:

```
MaxParallelCalls = System.Environment.ProcessorCount / 2;
```

Таким образом, для двуядерной системы разрешается один параллельный вызов двух методов.

Модель с равноправными узлами

В модели с равноправными узлами все потоки (или задачи) участвуют в обработке; центрального узла нет. Работа узлов может осуществляться параллельно. Задачи, в которых декомпозиция осуществляется по данным, являются примером модели с равноправными узлами. Для реализации алгоритмов с параллелизмом по данным можно использовать средства TPL, автоматически осуществляющие декомпозицию и агрегацию результатов с учетом возможностей

вычислительной системы и текущей загруженностью. В эту группу входят шаблоны `Parallel.For`, `Parallel.ForEach`, а также технология `PLINQ`.

В следующем примере рассматривается матричное умножение с декомпозицией по строкам первой матрицы. Каждый рабочий поток (подзадача) оперирует с одной или несколькими строками первой матрицы и всей второй матрицей. Таким образом, каждый поток вычисляет соответствующие строки результирующей матрицы. Для эффективного разделения строк матрицы можно использовать шаблон `Parallel.For`.

```
Parallel.For(0, N, i => {
    for(int j=0; j<N; j++)
        for(int k=0; k<N; k++)
            C[i,j] += A[i,k] * B[k, j];
});
```

В модели с равноправными узлами работа каждого участника может включать несколько последовательных этапов. Примером многоэтапной обработки является шаблон `Map/Reduce`. Обработка элементов включает следующие этапы:

- **Map.** Обработка элементов, формирование для каждого элементы пары "ключ-значение", группировка элементов по ключам.
- **Reduce.** Обработка сгруппированных элементов и выполнение для каждой группы заданной редукции.

Примером шаблона `Map/Reduce` является задача подсчёта встречаемости слов в тексте. Операция `Map` создает для каждого уникального слова пару "ключ-значение", ключ соответствует слову, значение равно единице. Все слова группируются. Операция `Reduce` вычисляет количество слов в каждой группе.

```
// Операция Map
// Генерируем пары ключ-значение (word, 1)
ILookup<string, int> map =
```

```
words.AsParallel().ToLookup(p => p, k => 1);
```

```
// Операция Reduce
// Вычисляем встречаемость слов
// Отбираем с частотой встречаемости больше 1
var reduce = from IGrouping<string, int> wordMap
    in map.AsParallel()
    where wordMap.Count() > 1
    select new { Word = wordMap.Key,
        Count = wordMap.Count() };
// Отображение результатов
foreach (var word in reduce)
    Console.WriteLine("Word: '{0}'; Count: {1}",
        word.Word, word.Count);
Console.ReadLine();
```

Для повышения эффективности алгоритм можно переписать в виде одного PLINQ-запроса:

```
var files =
    Directory.EnumerateFiles(dirPath, "*.txt").AsParallel();

var counts = files
    .SelectMany(f =>
        File.ReadLines(f).SelectMany(line =>
            line.Split(delimiters)))
    .GroupBy(w => w)
    .Select(g => new {Word = g.Key, Count = g.Count()});
```

В первой строке инициализируется перечислимый список файлов с расширением *.txt в директории dirPath. Список файлов представляет собой тип ParallelQuery<File>, поэтому все запросы выполняются параллельно. Первый запрос SelectMany формирует общий список слов из всех файлов. Для разделения строк файла на слова используется массив разделителей delimiters. Оператор GroupBy осуществляет группировку слов, последний оператор Select для каждой группы формирует безымянный тип с полями Word и Count.

Модель конвейера

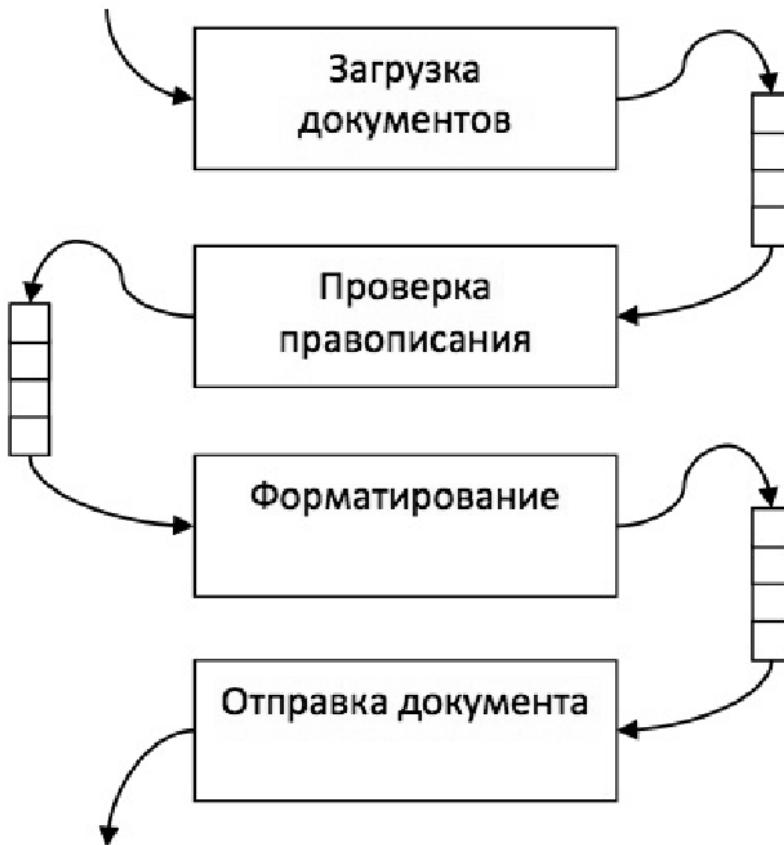
В модели конвейерной обработки (*pipelines*) поток обрабатываемых данных проходит через несколько этапов. Прохождение этапов осуществляется строго последовательно. Параллелизм достигается за счет одновременной обработки разных элементов на разных этапах.

Если последовательность элементов заранее определена и порядок обработки элементов не важен, то для распараллеливания эффективнее использовать модель с равноправными узлами (шаблон MapReduce, шаблон Parallel.For). Конвейерная обработка возникает при работе с последовательными потоками данных (потоки событий, потоки видеосигналов, потоки изображений), а также при многоэтапной обработке элементов последовательности в строго заданном порядке.

Рассмотрим конвейер, обрабатывающий документы. Обработка документов включает следующие этапы: загрузка документа (из файла, из почтового сервера и т.д.), проверка правописания, форматирование, отправка документа. Обработка документов выполняется строго последовательно, то есть первым отправляется тот документ, который загружен первым. Но при организации конвейера обработка разных документов на разных этапах может осуществляться параллельно.



Реализацию конвейерной обработки можно реализовать с помощью задач и конкурентных очередей. Каждая задача реализует этап конвейера, очереди выступают буферами, накапливающими элементы.



Последовательный алгоритм обработки изображений выглядит следующим образом:

```

while(bWorking)
{
    doc = LoadDoc(..);
    spelledDoc = CheckSpelling(doc);
    formattedDoc = FormatDoc (spelledDoc);
    Send(formattedDoc);
    nSlide++;
}
  
```

Для реализации конвейера необходимо ввести буферы, предназначенные для параллельной работы, и оформить этапы в виде асинхронных задач:

```
var downloads = new BlockingCollection<MyDoc>(limit);
var checked = new BlockingCollection<MyDoc>(limit);
var formatted = new BlockingCollection<MyDoc>(limit);

var factory =
    new TaskFactory(TaskCreationOptions.LongRunning,
        TaskContinuationOptions.None);

var loadTask = factory.StartNew(() =>
    LoadingDocs(downloads));

var checkTask = factory.StartNew(() =>
    CheckingDocs(downloads, checked));

var filterTask = factory.StartNew(() =>
    FilteringDocs(checked, filtered));

var sendTask = factory.StartNew(() =>
    SendingDocs(formatted.GetConsumingEnumerable()));

Task.WaitAll(loadTask, checkTask, filterTask, sendTask);
```

Задачи создаются с параметром `LongRunning`, чтобы с каждым этапом был связан собственный поток, и планировщик не оценивал производительность выполнения задач. Задачи читают элементы из одной очереди и пишут в другую. Если во входной очереди нет элементов, то поток блокируется для ожидания поступления элементов. Если выходная очередь переполнена, то поток, осуществляющий запись, блокируется в ожидании извлечения, хотя бы одного элемента.

Содержание

Титульная страница	2
Выходные данные	3
Лекция 1. Введение в параллельные вычисления	4
Лекция 2. Проблемы разработки параллельных приложений	16
Лекция 3. Работа с потоками	30
Лекция 4. Знакомство с многопоточной обработкой	53
Лекция 5. Средства синхронизации	60
Лекция 6. Поиск простых чисел	81
Лекция 7. Конкурентные коллекции	90
Лекция 8. Синхронизация доступа к одноэлементному буферу	96
Лекция 9. Работа с задачами	104
Лекция 10. Синхронизация приоритетного доступа к многоэлементному буферу	116
Лекция 11. Шаблоны параллелизма Parallel	125
Лекция 12. Клеточная модель "Игра Жизнь" Дж.Конвея	136
Лекция 13. Технология PLINQ	141
Лекция 14. Знакомство с "Визуализатором параллелизма" в Visual Studio 12	157
Лекция 15. Планировщик задач	165
Лекция 16. Типовые модели параллельных приложений	175