Содержание

Об авторе

Шакти Танвар является генеральным директором Techpro Compsoft Pvt Ltd, глобального поставщика консалтинговых услуг в области информационных технологий. Шакти – IT-евангелист и архитектор программного обеспечения с 15-летним опытом работы в области разработки программного обеспечения и корпоративного обучения. Он также является сертифицированным преподавателем Microsoft и проводит обучение в сотрудничестве с Microsoft на Ближнем Востоке.

Шакти Танвар специализируется в таких областях, как .NET, машинное обучение в Azure, искусственный интеллект, применение чистого функционального программирования для построения отказоустойчивых систем и параллельные вычисления.

Его любовь к преподаванию привела к тому, что он запустил специальную программу «Обучение профессоров» с целью улучшения работы колледжей в Индии.

Эта книга была бы невозможна без неоценимой помощи моей жены Кирти и моего сына Шашвата, разделивших со мной все взлеты и падения. Благодаря их поддержке и желанию действовать я продолжал двигаться вперед в тяжелое время.

Я бесконечно благодарен своим родителям, братьям и сестрам, которые всегда побуждали меня к достижению новых высот.

Огромное спасибо моим друзьям, наставникам и команде Packt, которые сопровождали меня на протяжении всего пути.

О переводе

Данная книга далась нам непросто и потребовала больше года на перевод и кропотливую редактуру, но оно того стоило. Тема разработки параллельных программ сейчас актуальна как никогда, хотя и является очень непростой для понимания. В этой книге автор изложил не только сложные технические аспекты написания параллельных программ, но и объяснил механизмы реализации многопоточности в .NET, а также особенности языка C#.

Над переводом этой книги работали специалисты компании Devs Universe:

- **Алина Воронина** – переводчик, специалист по обучению разработчиков английскому языку в компании Devs Universe;
- **Вячеслав Черников** – редактор перевода, к. т. н. в области разработки ПО, основатель компании Devs Universe, автор книги «Разработка мобильных приложений на C# для iOS и Android», в прошлом – один из Microsoft MVP, Nokia Champion, Qt Certified Specialist, Qt Ambassador, автор статей для «Хабрахабра», «Хакера», Microsoft Developer Blogs, говоритель для конференций;
- **Максим Веркошанский, Дмитрий Милкин, Марина Королькова** – помощь с редактурой, специалисты компании Devs Universe.

Мы надеемся, что наши переводы помогут вам глубже понять суть современных технологий и стать суперразработчиками.

О рецензентах

Элвин Эшкрафт – разработчик, живущий недалеко от Филадельфии. Он провел свою 23-летнюю карьеру, создавая программное обеспечение при помощи C#, Visual Studio, WPF, ASP.NET и т. д. Был удостоен награды Microsoft Most Valuable Professional девять раз. Вы можете увидеть его ежедневные подборки ссылок в блоге для .Net-разработчиков *Morning Dew*. Ранее Элвин работал в софтверных компаниях, включая Oracle, сейчас он является главным специалистом по разработке ПО в Allscripts, создавая программное обеспечение для здравоохранения. Для Packt Publishing им также были написаны и другие рецензии на такие книги, как *Mastering ASP.NET Core 2.0, Mastering Entity Framework Core 2.0* и *Learning ASP.NET Core 2.0*.

Я хотел бы поблагодарить свою замечательную жену Стелену и трех наших очаровательных дочерей за их поддержку и понимание. Многие вечера и выходные дни были посвящены чтению и пересмотру глав этой книги, для того чтобы она смогла выйти в свет – первоклассная и полезная книга для разработчиков .NET.

Видья Врат Агарвал – любитель книг, спикер, автор публикаций для Apress и технический редактор более чем дюжины книг Apress, Packt и O'Reilly. Он является прикладным разработчиком с 20-летним опытом в области проектирования, создания и разработки распределенных программных решений для крупных предприятий. Будучи главным архитектором в T-Mobile, Видья Врат Агарвал работал с проектами B2C и B2B. Сейчас он также продолжает сотрудничество с другими архитекторами с целью разработки решений и дорожных карт для различных проектов T-Mobile для миллионов клиентов компании. Он рассматривает разработку программного обеспечения как ремесло и является большим сторонником архитектуры программного обеспечения и практики чистого кода (clean code).

Предисловие

Прошел почти год с момента, когда издательская компания Packt впервые связалась со мной по поводу книги. Я и предположить не мог, что этот долгий путь будет таким сложным, однако за это время я смог многому научиться. Книга, которую вы сейчас держите в руках, – это результат, который стоил долгих дней трудов, и я горжусь тем, что наконец-то представляю ее вам.

Процесс создания этой книги очень много для меня значит, так как я всегда мечтал написать о языке, с которого начинал свою карьеру. Язык программирования C# развивался очень стремительно, а платформа .NET Core еще сильнее улучшила его репутацию в сообществе разработчиков.

Для того чтобы книга была полезна широкому кругу разработчиков, мы поговорим как о классическом подходе, построенном на потоках (threads), так и о разработке с использованием библиотеки TPL (Task Parallel Library). Вначале будут рассмотрены основные концепции операционных систем (ОС), которые позволяют писать многопоточный код. Затем мы проанализируем различия между классическим подходом и TPL.

В этой книге я постараюсь подойти к параллельному программированию со стороны современных реалий. Все примеры будут короткими и простыми, чтобы облегчить ваше понимание. Содержание глав построено так, чтобы информация легко усваивалась, даже если вы не обладаете специальными знаниями.

Надеюсь, вы получите такое же удовольствие от чтения этой книги, как и я от ее написания.

Целевая аудитория

Данная книга предназначена для программистов C#, которые хотят изучить концепции параллельного программирования и многопоточности, а также использовать полученные знания для своих приложений, построенных на базе .NET Core. Книга будет полезна студентам и специалистам, желающим познакомиться с принципами работы параллельного программирования на современном оборудовании.

Предполагается, что вы уже имеете представление о C# и базовые знания о том, как работают операционные системы.

Краткий обзор

Глава 1 «Введение в параллельное программирование», в которой представлены важные понятия многопоточности и параллельного программирования, включает в себя описание того, как развивались операционные системы

для поддержки современных подходов параллельного программирования.

Глава 2 «Параллелизм задач» демонстрирует возможности разделения программы на задачи (tasks) с целью эффективного использования ресурсов процессора и повышения производительности.

В главе 3 «Реализация параллелизма данных», в которой основное внимание уделяется реализации параллелизма данных с использованием параллельных циклов, также рассматриваются дополнительные методы (extension methods), помогающие в достижении параллелизма, а также разделении данных.

Глава 4 «Использование PLINQ» рассказывает о преимуществах использования PLINQ, включая отмену запросов. Также рассматриваются особенности применения PLINQ.

В главе 5 «Примитивы синхронизации» рассматриваются конструкции, доступные в C# для работы с разделяемыми (shared) ресурсами в многопоточном коде.

Глава 6 «Использование параллельных коллекций» описывает использование преимуществ параллельных коллекций, доступных в .NET Core.

Глава 7 «Повышение производительности с помощью отложенной инициализации» посвящается повышению производительности с помощью паттерна отложенной (lazy, «ленивой») инициализации.

В главе 8 «Введение в асинхронное программирование» рассматривается то, как нужно писать асинхронный код в более ранних версиях .NET.

В главе 9 «Основы асинхронного программирования с помощью async, await и задач» рассказывается о том, как использовать преимущества на новых конструкциях в .NET Core для реализации асинхронного кода.

Глава 10 «Отладка задач с Visual Studio» посвящена различным инструментам, доступным в Visual Studio 2019, которые облегчают отладку параллельных задач (tasks).

В главе 11 «Создание модульных тестов для параллельного и асинхронного кодов» рассматриваются различные способы написания тестов в Visual Studio и .NET Core.

Глава 12 «IIS и Kestrel в ASP.NET Core» расскажет, что такое IIS и Kestrel и как этим пользоваться. В этой главе также рассматривается поддержка асинхронных потоков.

Глава 13 «Шаблоны параллельного программирования» описывает различные паттерны (patterns), уже реализованные в языке C#. Она также включает в себя примеры реализации таких шаблонов.

В главе 14 «Управление распределенной памятью» рассматривается совместное использование памяти в распределенных программах.

Прежде чем начать

Вам необходимо установить Visual Studio 2019 вместе с .NET Core 3.1.

Также рекомендуется иметь базовые знания в языке C# и механизмах работы операционных систем.

Скачайте примеры исходных кодов

Вы можете скачать примеры исходных кодов для этой книги по адресу www.packtpub.com. Если вы приобрели эту книгу в другом месте, то можете перейти на сайт https://www.packtpub.com/support и получить примеры по электронной почте.

Шаги по загрузке файлов:

1. Войдите в систему или зарегистрируйтесь на сайте www.packtpub.com.
2. Выберите вкладку **Support**.
3. Нажмите на кнопку **Code Downloads**.
4. Введите название книги на английском языке «Hands-On Parallel Programming with C# 8 and .NET Core 3» в поле поиска и следуйте инструкциям.

Как только файл будет загружен, убедитесь, что вы его распаковываете, используя последнюю версию программ:

1) WinRAR/7-Zip для Windows;
2) Zipeg/iZip/UnRarX для Mac;
3) 7-Zip/PeaZip для Linux.

Примеры кода для этой книги также размещены на GitHub: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-NET-core-3.

При обновлении исходного кода он тоже меняется на GitHub.

В нашем обширном каталоге книг и видео, доступных по ссылке ниже, представлены и другие примеры: https://github.com/PacktPublishing/. Обязательно их посмотрите.

Изображения

Также предоставляем вам PDF-файл с использованными в книге цветными изображениями скриншотов/диаграмм, которые вы можете скачать по ссылке https://static.packt-cdn.com/downloads/9781789132410_ColorImages.pdf.

Условные обозначения

В данной книге используется целый ряд условных обозначений.

`CodeInText`: указывает кодовые слова, например названия таблиц в базах данных, имена папок, имена файлов и их расширения, имена путей, вводимую пользователем информацию или имена пользователей Twitter. Допустим: «Подключите загруженный образ диска `WebStorm-10*.dmg` в качестве нового диска вашей системы».

Блок кода обозначается следующим образом:

```
private static void PrintNumber10Times() {
  for (int i = 0; i < 10; i++) {
```

```
    Console.Write(1);
  }
  Console.WriteLine();
}
```

Некоторые строки или элементы кода могут быть выделены жирным шрифтом с целью привлечения внимания к ним:

```
private static void PrintNumber10Times() {
  for (int i = 0; i < 10; i++) {
    Console.Write(1);
  }
  onsole.WriteLine();
}
```

Жирным шрифтом обозначается новый термин, важное слово или слова, которые вы видите в диалоговых сообщениях. Пример: «Вместо того чтобы самим находить оптимальное количество потоков, мы можем предоставить это среде **Common Language Runtime**».

Предупреждение или важная информация.

Советы и рекомендации.

Часть I

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАБОТЫ С ПОТОКАМИ, МНОГОЗАДАЧНОСТИ И АСИНХРОННОСТИ

В данной части вы познакомитесь с понятиями потока, многозадачности и асинхронного программирования.

Содержание части I включает в себя следующие главы:

- глава 1 «Введение в параллельное программирование»;
- глава 2 «Параллелизм задач»;
- глава 3 «Реализация параллелизма данных»;
- глава 4 «Использование PLINQ».

Глава 1

Введение в параллельное программирование

Возможность использования параллельного программирования реализована в .NET с самого начала и после появления **Task Parellel Library** (TPL) в .NET Framework 4.0 получила широкое распространение.

Многопоточность (multithreading) является подмножеством параллельного программирования и выступает одной из самых сложных тем для начинающих разработчиков. Язык программирования C# заметно эволюционировал с момента своего появления и сейчас может быть использован не только для «классической» многопоточности, но и для «современного» асинхронного программирования. Многопоточность C# уходит своими корнями в версию 1.0. Язык C# является преимущественно синхронным, однако в версии 5.0 была добавлена поддержка асинхронности, которая сделала его отличным выбором для прикладных программистов. В то время как многопоточность имеет дело только с распараллеливанием внутри самих процессов, параллельное программирование также имеет дело с механизмами межпроцессного взаимодействия (inter-process communication, IPC).

До появления TPL использовались классы `Thread`, `BackgroundWorker` и `ThreadPool`, которые позволяли реализовать многопоточность. C# 1.0 опирался на потоки (threads) для отделения фоновой работы от **обработки событий пользовательского интерфейса** (user interface, или UI), что позволяло разрабатывать отзывчивые приложения. Эта модель теперь называется классической работой с потоками (classic threading). Со временем она уступила место другой модели программирования, называемой TPL, которая опирается на задачи (tasks) и скрывает от разработчика работу с потоками.

В этой главе мы познакомимся с различными концепциями, которые помогут вам научиться писать многопоточный код с нуля.

В главе 1 будут освещены следующие темы:

- основные понятия многоядерных вычислений, начиная с общих концепций и процессов **операционной системы** (ОС);
- потоки и разница между многопоточностью и многозадачностью;

- преимущества и недостатки написания параллельного кода и сценариев, в которых целесообразно использование параллельного программирования.

Технические требования

Продемонстрированные в данной книге примеры были созданы в Visual Studio 2019 при помощи C# 8. Со всеми исходниками вы можете ознакомиться на сайте: https://github.com/PacktPublishing/Hands-On-Parallel-Programming-with-C-8-and-.NET-Core-3/tree/master/Chapter01.

Подготовка к многоядерным вычислениям

В этом разделе вам будут представлены основные концепции ОС, начиная с процесса (process), внутри которого живут и исполняются потоки (threads). Далее мы рассмотрим развитие многозадачности с момента появления аппаратных возможностей, которые способствовали развитию параллельного программирования. После этого мы попытаемся разобраться в различных способах создания потоков в коде приложений.

Процессы

Говоря простым языком, слово «процесс» (process) относится к программе, которая запущена на компьютере. Однако с точки зрения ОС процесс – это адресное пространство в оперативной памяти. Каждому приложению нужны процессы для запуска, вне зависимости от того, написано ли это приложение для смартфона, Windows или веб-браузера. Процессы обеспечивают защиту одних программ от других, работающих в той же системе: выделенные одной программой данные не могут случайным образом быть доступными для другой. Кроме этого, процессы также обеспечивают изоляцию, при которой программы могут запускаться и останавливаться независимо друг от друга и от самой ОС.

Дополнительно об ОС

Производительность приложений во многом зависит от конфигурации аппаратного обеспечения, которая включает в себя следующее:

- скорость центрального процессора;
- объем оперативной памяти;
- скорость жесткого диска (HDD);
- тип диска, то есть HDD или SSD.

За последние несколько десятилетий мы стали свидетелями существенного прогресса в области аппаратных технологий. Например, микропроцес-

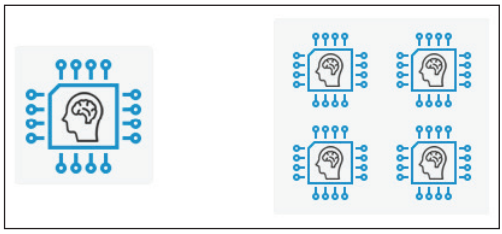
соры раньше имели одно ядро, которое представляло собой чип с одним **центральным процессором** (central processing unit, CPU). На рубеже веков мы увидели появление многоядерных процессоров, которые представляют собой чипы с двумя или более процессорами, где каждый обладает своим собственным кешем.

Многозадачность

Многозадачность – это способность компьютерной системы одновременно запускать несколько процессов или приложений. Количество процессов, которые могут выполняться системой, прямо пропорционально количеству ядер центрального процессора (ЦП). Таким образом, одноядерный процессор может выполнять только одну задачу за раз, двухъядерный процессор может одновременно выполнять две задачи, а четырехъядерный – четыре задачи. Если мы добавим к этому понятие планирования (scheduling) ЦП, то увидим, что ЦП одновременно запускает больше приложений, планируя или переключая их на основе алгоритмов планирования ЦП.

Hyper-threading

Hyper-threading (HT) – это запатентованная технология, разработанная компанией Intel, которая улучшает распараллеливание вычислений, выполняемых на процессорах x86. Впервые технология HT была представлена на серверных процессорах Xeon в 2002 году. Однопроцессорные чипы с поддержкой HT работают с двумя виртуальными (логическими) ядрами и способны выполнять две задачи одновременно. На следующей диаграмме показана разница между одноядерными и многоядерными чипами:

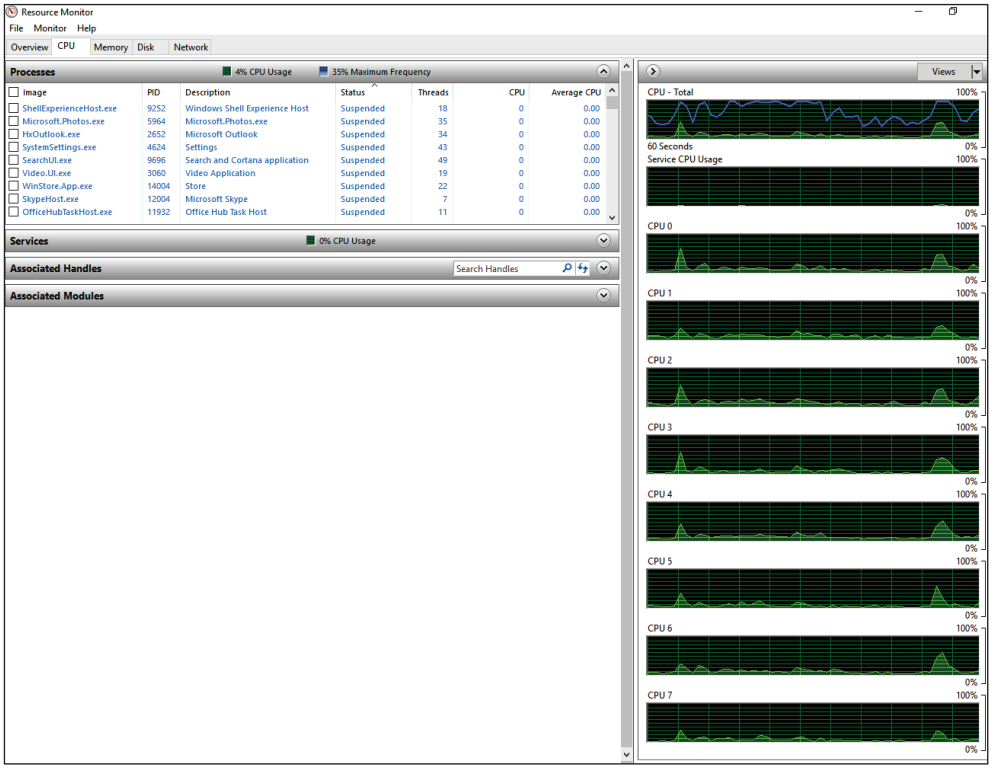


Ниже представлены примеры конфигураций процессоров и количество задач, которые они могут выполнять:

- **процессор с одноядерным чипом:** 1 задача за раз;
- **процессор с одноядерным чипом с поддержкой HT:** 2 задачи одновременно;
- **процессор с двухъядерным чипом:** 2 задачи одновременно;
- **процессор с двухъядерным чипом с поддержкой HT:** 4 задачи одновременно;

- **процессор с четырехъядерным чипом:** 4 задачи одновременно;
- **процессор с четырехъядерным чипом с поддержкой HT:** 8 задач одновременно.

На правой стороне в приложении **Resource Monitor** (Монитор ресурсов) (скриншот ниже) для четырехъядерной процессорной системы с поддержкой HT отображено восемь доступных процессоров.



Должно быть, вам интересно, насколько улучшится производительность компьютера при простом переходе от одноядерного процессора к многоядерному. На момент подготовки этой книги архитектура большинства самых быстрых суперкомпьютеров была основана на подходе «**множество команд, множество данных**» (Multiple Instruction, Multiple Data, MIMD). Данный вид архитектуры является одной из классификаций компьютерной архитектуры, предложенной Майклом Дж. Флинном в 1966 году.

Попробуем разобраться в этой классификации.

Классификация Флинна

В зависимости от количества параллельных потоков команд (или управления) и потоков данных Флинн классифицировал компьютерные архитектуры на четыре категории:

- **одиночный поток команд, одиночный поток данных** (Single Instruction, Single Data или SISD): в этой модели имеется один блок управления и один поток команд. Эти системы могут выполнять только одну команду за раз (без какой-либо параллельной обработки). Все одноядерные процессоры основаны на архитектуре SISD;
- **одиночный поток команд, множество потоков данных** (Single Instruction, Multiple Data или SIMD): в этой модели имеется только один поток команд и несколько потоков данных. Одна и та же программа параллельно применяется к нескольким наборам данных. Такая архитектура обеспечивает разделение данных на части и их параллельную обработку одним и тем же алгоритмом;
- **множество потоков команд, одиночный поток данных** (Multiple Instructions, Single Data или MISD): в этой модели множество потоков команд работают с одним потоком данных. Таким образом, несколько операций могут параллельно применяться к одним и тем же данным. Как правило, эта модель используется для обеспечения отказоустойчивости, например на ЭВМ, которые управляют полетом космических кораблей;
- **множество потоков команд, множество потоков данных** (Multiple Instructions, Multiple Data, MIMD): как видно из названия, эта модель предполагает наличие нескольких потоков команд и нескольких потоков данных. Благодаря такому подходу можно достичь истинного параллелизма, при котором каждый процессор способен выполнять несколько программ с разными наборами данных. В настоящее время большинство компьютерных систем используют этот тип архитектуры.

Теперь, когда мы разобрались с основами, давайте перейдем к обсуждению потоков.

Потоки

Поток (thread) – это единица выполнения, исполняемая внутри процесса (process). В любой момент времени программа может состоять из одного или нескольких потоков для лучшей производительности. Приложения Windows на основе графического интерфейса, такие как устаревшие **Windows Forms** (WinForms) или **Windows Presentation Foundation** (WPF), имеют выделенный поток для управления пользовательским интерфейсом и обработки действий пользователя. Этот поток также называется **потоком пользовательского интерфейса** (UI thread), или **потоком переднего плана** (foreground thread). Он владеет всеми элементами управления, которые создаются как часть пользовательского интерфейса.

Типы потоков

Существует два типа управляемых потоков: поток переднего плана и фоновый поток (background thread). Разница между ними заключается в следующем:

- **потоки переднего плана** оказывают непосредственное влияние на время жизни приложения. Приложение продолжает работать до тех пор, пока выполняется поток переднего плана;
- **фоновые потоки** не влияют на время жизни приложения. Таким образом, при закрытии приложения все фоновые потоки уничтожаются.

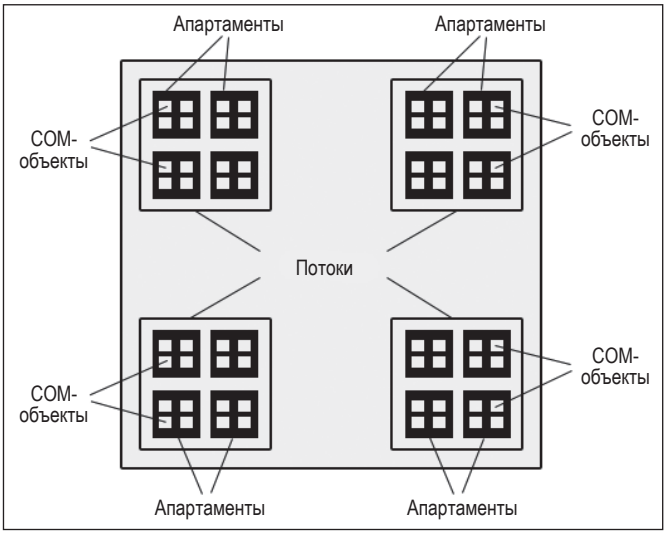
Приложение может содержать любое количество фоновых потоков и потоков переднего плана. Поток переднего плана поддерживает работу приложения, когда оно активно. Приложение полностью прекращает работу, когда последний поток переднего плана останавливается или прерывается. При выходе из приложения система останавливает все фоновые потоки.

Апартаментное состояние в модели COM

Другой немаловажной особенностью потоков выступает апартаментное состояние (apartment state). Это область внутри потока, где находятся объекты так называемой **модели компонентных объектов** (Component Object Model, COM).

COM является объектно-ориентированной системой для создания двоичного программного обеспечения, с которым может взаимодействовать пользователь. Также COM – это распределенная и кросс-платформенная система, на которой базируются технологии Microsoft OLE и ActiveX.

Как вы, возможно, знаете, все элементы управления Windows Forms содержат COM-объекты. Всякий раз, когда вы создаете приложение .NET WinForms, вы фактически размещаете на сервере провайдера эти COM-компоненты. Состояние подразделений – это отдельная область внутри прикладного процесса, где создаются COM-объекты. На следующей схеме показана связь между апартаментом потока и COM-объектами.



Из вышеуказанной схемы следует, что каждый поток имеет апартаменты, где расположены COM-объекты.

Поток может принадлежать одному из двух апартаментных состояний:

- **однопоточный апартамент** (Single-Threaded Apartment, STA): исходный COM-объект может быть доступен только через один поток;
- **многопоточный апартамент** (Multi-Threaded Apartment, MTA): исходный COM-объект может быть доступен одновременно через несколько потоков.

Ниже приведен список, содержащий важные моменты, касающиеся апартаментных состояний потоков:

- процессы могут иметь несколько потоков, среди которых – как потоки переднего плана, так и фоновые потоки;
- каждый поток может иметь один апартамент (STA либо MTA);
- каждый апартамент имеет модель параллелизма – либо однопоточную, либо многопоточную. Мы также можем программно изменять состояние потока;
- прикладной процесс может иметь более одного STA, но не MTA;
- примером приложения STA является Windows-приложение, а примером MTA – веб-приложение;
- COM-объекты создаются в апартаментах. Один COM-объект может находиться только в одном апартаменте потока, при этом апартаменты не могут быть общими.

Приложение можно принудительно запустить в режиме STA, используя атрибут STAThread в качестве основного метода. Пример метода `Main` для WinForms старого образца:

```
static class Program {
  ///<summary>
  ///Главная точка входа для приложения
  ///</summary>
  [STAThread]
  static void Main() {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
  }
}
```

Атрибут `STAThread` также присутствует в WPF, однако скрыт от пользователей. Ниже приведен код скомпилированного класса `App.g.cs`, который находится в хранилище `obj/Debug` проекта WPF после компиляции:

```
///<summary>
///Приложение
///</summary>
public partial class App: System.Windows.Application {
  ///<summary>
  ///Инициализация компонента
  ///</summary>
  [System.Diagnostics.DebuggerNonUserCodeAttribute()]
```

```
  [System.CodeDom.Compiler.GeneratedCodeAttribute(
    "PresentationBuildTasks", "4.0.0.0")]

  public void InitializeComponent() {
    #line 5 "..\..\App.xaml"
    this.StartupUri = new System.Uri("MainWindow.xaml",
      System.UriKind.Relative);
    #line
    default
    #line hidden
  }
  ///<summary>
  ///Точка входа в приложение
  ///</summary>
  [System.STAThreadAttribute()]
  [System.Diagnostics.DebuggerNonUserCodeAttribute()]
  [System.CodeDom.Compiler.GeneratedCodeAttribute(
    "PresentationBuildTasks", "4.0.0.0")]

  public static void Main() {
    WpfApp1.App app = new WpfApp1.App();
    app.InitializeComponent();
    app.Run();
  }
}
```

Как вы могли заметить, метод `Main` снабжен атрибутом `STAThread`.

Многопоточность

Параллельное исполнение кода в .NET реализуется благодаря многопоточности. В зависимости от аппаратных возможностей процесс (или приложение) может использовать любое количество потоков. Любое приложение, включая консоль, существующие WinForms, WPF и даже веб-приложения, по умолчанию запускается одним потоком. Можно с легкостью достигнуть многопоточности, программно создавая больше потоков по мере необходимости.

Многопоточность обычно реализуется с помощью **планировщика потоков** (thread scheduler), отслеживающего момент, при котором активные потоки исчерпают отведенное им на выполнение время внутри процесса. Каждому созданному потоку присваивается приоритет (свойство `System.Threading.ThreadPriority`), который может иметь лишь одно из нижеуказанных допустимых значений. Приоритетом по умолчанию является значение «нормальный». Все возможные значения приоритета:

- Самый высокий (Highest);
- Выше нормы (AboveNormal);
- Нормальный (Normal);
- Ниже нормы (BelowNormal);
- Самый низкий (Lowest).

Для каждого потока внутри процесса ОС назначает временной отрезок на основе алгоритма планирования и приоритета потока. Каждая ОС имеет свой алгоритм планирования для потоков, поэтому в разных операционных системах порядок исполнения может отличаться. Это затрудняет поиск и исправление ошибок в работе потоков. Наиболее распространенный алгоритм планирования выглядит следующим образом.

1. Найти потоки с наивысшим приоритетом и запланировать их запуск.
2. Если имеется несколько потоков, обладающих наивысшим приоритетом, то каждому из них присваивается фиксированный временной диапазон (квант), в рамках которого они могут выполняться.
3. Как только потоки с наивысшим приоритетом заканчивают работу, то запускаются потоки с более низким приоритетом, для которых также выделяется определенное время.
4. Если создается новый поток с наивысшим приоритетом, то потоки с низким приоритетом снова сдвигаются назад.

Размеры временных квантов также зависят от длительности переключения между активными потоками. Кванты могут варьироваться в зависимости от конфигурации оборудования. Одноядерный процессор может одновременно запускать только один поток, поэтому планировщик проводит разделение времени между потоками. Квантование времени во многом зависит от тактовой частоты процессора, которая определяет производительность систем, однако по сравнению с многопоточностью – не в таком объеме. Более того, во время переключения контекста (загруженные в ОЗУ и кеш данные, необходимые для работы потока) появляются дополнительные издержки. Если потоку необходимо несколько временных промежутков для выполнения работы, то в таком случае поток должен быть выгружен из памяти, а затем снова загружен.

Понятие **параллелизма** в основном используется в контексте многоядерных процессоров. В основе многоядерного процессора лежит большее количество доступных процессорных модулей (ядер), как было упомянуто ранее, и поэтому различные потоки могут одновременно выполняться на разных ядрах. Чем больше будет процессоров, тем выше степень параллелизма.

Существует несколько способов создания потоков в программах:

- класс `Thread`;
- класс `ThreadPool`;
- класс `BackgroundWorker`;
- асинхронные делегаты;
- TPL.

В этой книге будут даны объяснения первым трем способам, а также подробно рассмотрены асинхронные делегаты и TPL.

Класс Thread

Самый простой и легкий способ создания потоков – через класс `Thread`, который определен в пространстве имен `System.Threading`. Этот подход использовался на платформе .NET с момента появления версии 1.0 и сейчас также применяется на .NET Core. Для создания потока необходимо предоставить

метод, который поток будет выполнять. Метод может либо содержать, либо не содержать параметры. Чтобы «обернуть» эти методы, библиотека классов предоставляет два делегата:

- `System.Threading.ThreadStart`;
- `System.Threading.ParameterizedThreadStart`.

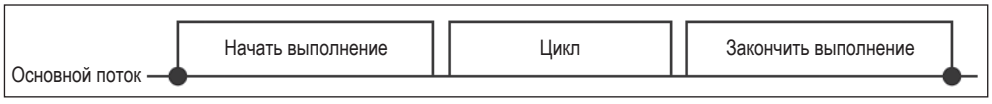
Каждый из них будет позже продемонстрирован на примерах. Для начала мы рассмотрим, как создается поток, затем я постараюсь объяснить вам работу синхронной программы. После мы обратимся к понятию многопоточности, которое позволит понять асинхронный подход. Пример *создания потока*:

```
using System;
namespace Ch01 {
  class _1Synchronous {
    static void Main(string[] args) {
      Console.WriteLine("Start Execution!!!");
      PrintNumber10Times();
      Console.WriteLine("Finish Execution");
      Console.ReadLine();
    }
    private static void PrintNumber10Times() {
      for (int i = 0; i < 10; i++)
    }
    Console.Write(1);
  }
  Console.WriteLine();
}
```

В предыдущем примере все действия выполняются в основном потоке. Мы запросили метод `PrintNumber10Times` из Main-метода. Поскольку Main-метод вызывается основным потоком графического интерфейса, код выполняется синхронно. Если код будет долго запускаться из-за перегрузки основного потока, то это может привести к зависанию.

Так выглядит вывод в консоль:

Ниже приведена схема, отображающая последовательность выполнения кода в **основном потоке** (Main Thread):



На предыдущей диаграмме схематично изображено последовательное выполнение кода в основном потоке.

В данной ситуации мы можем сделать программу многопоточной, создав поток для вывода сообщения. Основной поток печатает инструкции, написанные в Main-методе:

```
using System;
namespace Ch01 {
  class _2ThreadStart {
    static void Main(string[] args) {
      Console.WriteLine("Start Execution!!!");
      //Использование потока без параметра
      CreateThreadUsingThreadClassWithoutParameter();
      Console.WriteLine("Finish Execution");
      Console.ReadLine();
    }

    private static void
    CreateThreadUsingThreadClassWithoutParameter() {
      System.Threading.Thread thread;
      thread = new System.Threading.Thread
      (new System.Threading.ThreadStart(PrintNumber10Times));
      thread.Start();
    }

    private static void PrintNumber10Times() {
      for (int i = 0; i < 10; i++) {
        Console.Write(1);
      }
      Console.WriteLine();
    }
  }
}
```

В этом коде мы делегировали выполнение `PrintNumber10Times()` новому потоку, созданному с помощью класса `Thread`. Команды `Console.WriteLine` в Main-методе по-прежнему выполняются через основной поток, однако `PrintNumber10Times` не вызывается через дочерний поток.

Ниже представлен вывод в консоль:

Далее схематично изображен представленный выше процесс. Можно заметить, что инструкция `Console.WriteLine` выполняется в **основном потоке**, а реализация самого цикла происходит в **дочернем потоке** (child thread).

Приведенная выше схема является примером многопоточной реализации.

Сравнив выходные данные, можно увидеть, что программа завершает все инструкции в основном потоке, а затем начинает печатать цифры 10 раз. В представленном примере используются простые команды, поэтому они и выполняются всегда одним и тем же образом. Однако если в основном потоке трудоемкие операторы (требующие больше времени) появятся раньше, чем будет отображено **Finish Execution**, то результаты могут отличаться. Чтобы получить более полное представление об этом, позже мы рассмотрим, как работает многопоточность и как она связана со скоростью процессора.

Вот другой пример, демонстрирующий передачу данных в поток с помощью делегата `System.Threading.ParameterizedThreadStart`:

```
using System;
namespace Ch01 {
  class _3ParameterizedThreadStart {
    static void Main(string[] args) {
      Console.WriteLine("Start Execution!!!");
      //Использование потока без параметра
      CreateThreadUsingThreadClassWithParameter();
      Console.WriteLine("Finish Execution");
      Console.ReadLine();
    }

    private static void CreateThreadUsingThreadClassWithParameter() {
      System.Threading.Thread thread;
      thread = new System.Threading.Thread
      new System.Threading.ParameterizedThreadStart
      PrintNumberNTimes));
      thread.Start(10);
    }

    private static void PrintNumberNTimes(object times) {
      int n = Convert.ToInt32(times);
      for (int i = 0; i < n; i++) {
        Console.Write(1);
      }
      Console.WriteLine();
    }
  }
}
```

Вывод этого кода выглядит так:

Использование класса `Thread` имеет свои преимущества и недостатки. Давайте попробуем в них разобраться.

Преимущества и недостатки потоков

Преимущества класса `Thread`:

- дочерние потоки могут быть использованы для освобождения основного потока;
- потоки можно использовать для разбиения задач на мелкие подзадачи, которые могут выполняться параллельно.

Недостатки класса `Thread`:

- при большом количестве потоков трудно отлаживать и поддерживать код;
- создание потоков нагружает систему относительно ресурсов памяти и процессора;
- необходимо выполнять обработку исключений внутри фоновых методов, так как любые необработанные исключения могут привести к сбою программы.

Класс ThreadPool

Создание потоков требует больших затрат с точки зрения как памяти, так и ресурсов ЦП. В среднем каждый поток потребляет около 1 МБ памяти и несколько сотен микросекунд процессорного времени. Производительность приложений – понятие относительное, поэтому она не всегда улучшается при большом количестве потоков. Напротив, создание большого количества потоков иногда может резко снизить производительность приложения. В приоритете – создание оптимального количества потоков в зависимости от загрузки процессора целевой системы, то есть запущенных в системе программ. Это происходит потому, что каждая программа получает свой временной квант, который затем распределяется между потоками внутри приложения. Если вы создадите слишком много потоков, они могут не успеть справиться с выполнением какого-либо конструктивного задания, прежде чем будут выгружены из памяти, чтобы другие потоки с аналогичным приоритетом могли выполниться.

Поиск оптимального количества потоков может вызвать сложности, поскольку зависит от конкретной системы, обладающей своей уникальной конфигурацией и определенным количеством одновременно работающих

приложений. То, что может быть оптимальным количеством для одной системы, может негативно сказаться для другой. Вместо самостоятельного поиска оптимального количества потоков мы можем воспользоваться средой **Common Language Runtime** (CLR). CLR обладает алгоритмом определения оптимального числа потоков на основе загрузки процессора в любой момент времени. Эта среда поддерживает пул потоков, известный как `ThreadPool`. `ThreadPool` привязан к процессу, при этом все приложения имеют свой собственный пул потоков. Преимущество пула потоков заключается в том, что он поддерживает оптимальное количество потоков и динамически связывает их с конкретными задачами (tasks). По завершении работы потоки возвращаются в пул, где могут быть привязаны к новой задаче, таким образом убираются затраты на создание и уничтожение потоков.

Оптимальное количество потоков, которые могут быть созданы в различных фреймворках внутри `ThreadPool`:

- 25 на ядро в .NET Framework 2.0;
- 250 на ядро в .NET Framework 3.5;
- 1023 в .NET Framework 4.0 в 32-битной среде;
- 32 768 в .NET Framework 4.0 и в .NET Core в 64-битной среде.

Работая с инвестиционным банком, мы столкнулись со сценарием, при котором торговый процесс занимал почти 1800 секунд для синхронной брони 1000 сделок. Попробовав различные оптимальные числа, мы наконец переключились на `ThreadPool` и сделали процесс многопоточным. С помощью .NET Framework версии 2.0 приложение завершилось почти за 72 секунды. С версией 3.5 то же самое приложение завершилось всего за несколько секунд. Вместо того чтобы изобретать велосипед, достаточно прибегнуть к использованию фреймворка. Вы можете добиться необходимого прироста производительности, просто обновив фреймворк.

Создание потока через `ThreadPool` при помощи команды `ThreadPool.QueueUserWorkItem` показано в следующем примере.

Параллельный вызов нужного метода:

```
private static void PrintNumber10Times(object state) {
  for (int i = 0; i < 10; i++) {
    Console.Write(1);
  }
  Console.WriteLine();
}
```

Пример создания потока с помощью `ThreadPool.QueueUserWorkItem` и делегата `WaitCallback`:

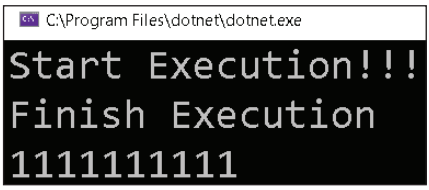
```
private static void CreateThreadUsingThreadPool() {
  ThreadPool.QueueUserWor(newWaitCallback(PrintNumber10Times));
}
```

Вызов из Main-метода:

```
using System;
using System.Threading;
namespace Ch01 {
```

```
  class _4ThreadPool {
    static void Main(string[] args) {
      Console.WriteLine("Start Execution!!!");
      CreateThreadUsingThreadPool();
      Console.WriteLine("Finish Execution");
      Console.ReadLine();
    }
  }
}
```

Вывод кода отражен на скриншоте:



Каждый пул потоков поддерживает минимальное и максимальное количества потоков. Эти значения можно изменить, вызвав следующие статические методы:

- `ThreadPool.SetMinThreads`;
- `ThreadPool.SetMaxThreads`.

Поток создается через `System.Threading`. Класс `Thread` не принадлежит к `ThreadPool`.

Далее мы рассмотрим преимущества и недостатки, связанные с использованием класса `ThreadPool`, и разберем случаи, когда следует избегать его использования.

Преимущества и недостатки ThreadPool. И когда лучше не обращаться к этому классу

Преимущества `ThreadPool`:

- дочерние потоки могут быть использованы для освобождения основного потока;
- оптимально потоки создаются и поддерживаются с помощью CLR.

Недостатки `ThreadPool`:

- при большом количестве потоков код становится трудно отлаживать и поддерживать;
- нам нужно выполнить обработку исключений внутри рабочего метода, так как любое необработанное исключение может привести к сбою программы;
- отчеты о ходе выполнения, отмена команд и логика завершения должны быть написаны с нуля.

Причины, по которым не следует обращаться к `ThreadPool`:

- когда нам нужен поток переднего плана (foreground thread);

- при необходимости установить явный приоритет потоку;
- когда у нас есть длительные или блокирующие задачи. Наличие большого количества заблокированных потоков в пуле предотвратит запуск новых задач из-за ограниченного количества потоков, доступных для `ThreadPool`;
- если нам нужны потоки STA, так как потоки в `ThreadPool` по умолчанию являются MTA;
- при необходимости привязать отдельный идентификационный номер к потоку нельзя присвоить название потоку `ThreadPool`.

BackgroundWorker

`BackgroundWorker` – это компонент, предоставляемый .NET для создания управляемых потоков в `ThreadPool`. Как вы ранее видели в примере приложения с графическим интерфейсом, Main-метод был снабжен атрибутом `STAThread`. Этот атрибут гарантирует безопасность управления, поскольку элементы пользовательского интерфейса создаются в апартаментах, принадлежащих основному потоку, и не могут быть совместно использованы другими потоками. В приложениях Windows есть основной поток, владеющий пользовательским интерфейсом и элементами управления, которые создаются при запуске приложения. Он отвечает за прием информации от пользователя и визуальное оформление (перекраску) интерфейса на основе действий пользователя. Для получения отзывчивых приложений мы должны постараться максимально освободить пользовательский интерфейс от длительных операций и делегировать все трудоемкие задачи фоновым потокам. Некоторые общие задачи, которые обычно выполняются рабочими потоками:

- загрузка изображений с сервера;
- взаимодействие с базой данных;
- взаимодействие с файловой системой;
- взаимодействие с веб-сервисами;
- сложные локальные вычисления.

Очевидно, что большинство из этих задач являются операциями **ввода-вывода** (I/O), которые выполняются центральным процессором. В момент вызова фрагмента кода, запускающего операцию ввода-вывода, выполнение передается от потока к процессору, который и выполняет задачу. Когда задача будет завершена, результат операции вернется обратно к вызывающему потоку. Период времени с момента передачи полномочий на выполнение задачи до получения результатов является периодом бездействия для потока, поскольку он просто должен ждать завершения операции процессором. Если это происходит в основном потоке, то приложение перестает отвечать на действия пользователя. Существуют и другие проблемы, которые необходимо решить для создания отзывчивых приложений. Давайте рассмотрим пример.

Сценарий:

Нам нужно получить данные от сервиса, который передает их в потоковом режиме. Затем необходимо проинформировать пользователя о проценте вы-

полнения работы. Как только работа будет завершена, нужно будет показать пользователю загруженные данные.

Проблемы:

Обращение к внешнему сервису занимает много времени, поэтому нам нужно выполнить его в фоновом потоке, чтобы избежать «заморозки» пользовательского интерфейса.

Решение:

`BackgroundWorker` – это класс, предоставленный `System.ComponentModel`. Как было упомянуто ранее, он может быть использован для создания рабочего потока на базе `ThreadPool`. `BackgroundWorker` также умеет сообщать свой статус и поддерживает отмену.

Требуемый сценарий может быть реализован при помощи следующего кода:

```
using System;
using System.ComponentModel;
using System.Text;
using System.Threading;
namespace Ch01 {
  class _5BackgroundWorker {
    static void Main(string[] args) {
      var backgroundWorker = new BackgroundWorker();
      backgroundWorker.WorkerReportsProgress = true;
      backgroundWorker.WorkerSupportsCancellation = true;
      backgroundWorker.DoWork += SimulateServiceCall;
      backgroundWorker.ProgressChanged += ProgressChanged;
      backgroundWorker.RunWorkerCompleted += RunWorkerCompleted;
      backgroundWorker.RunWorkerAsync();
      Console.WriteLine("To Cancel Worker Thread Press C.");
      while (backgroundWorker.IsBusy) {
        if (Console.ReadKey(true).KeyChar == 'C') {
          backgroundWorker.CancelAsync();
        }
      }
    }
    //Этот метод выполняется, когда завершается фоновый поток
    private static void RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e) {
      if (e.Error != null) {
        Console.WriteLine(e.Error.Message);
      } else
        Console.WriteLine($"Result from service call is {e.Result}");
    }
    //Этот метод выполняется, когда фоновый поток
    //сообщает о прогрессе
    private static void ProgressChanged(object sender, ProgressChangedEventArgs e) {
      Console.WriteLine($"{e.ProgressPercentage}% completed");
    }
    //Обращение к сервису, который мы хотим симулировать
    private static void SimulateServiceCall(object sender, DoWorkEventArgs e) {
```

```
      var worker = sender as BackgroundWorker;
      StringBuilder data = new StringBuilder();
      //Симулировать обращение к потоковому сервису
      for (int i = 1; i <= 100; i++) {
        //worker.CancellationPending будет true, если пользователь
        //нажмет C
        if (!worker.CancellationPending) {
          data.Append(i);
          worker.ReportProgress(i);
          Thread.Sleep(100);
          //Можете раскомментировать строку ниже и посмотреть на
          //исключительную ситуацию
          //throw new Exception("Some Error has occurred");
        } else {
          //Отменяет работу фонового потока
          worker.CancelAsync();
        }
      }
      e.Result = data;
    }
  }
}
```

Класс `BackgroundWorker` является абстракцией над низкоуровневыми потоками, предоставляя разработчику больше контроля и возможностей. Также `BackgroundWorker` отлично подходит для реализации асинхронной модели, основанной на событиях (**Event-Based Asynchronous Pattern**, или **EAP**), и способен более эффективно взаимодействовать с кодом, чем низкоуровневые потоки. Чтобы получать отчеты о ходе выполнения и отмене событий, необходимо установить следующие свойства в значение `true`:

```
backgroundWorker.WorkerReportsProgress = true;
backgroundWorker.WorkerSupportsCancellation = true;
```

Также вам нужно будет установить обработчик на `ProgressChanged` для получения прогресса выполнения работы; на `DoWork` – для передачи метода, выполняющего работу; `RunWorkerCompleted` – для получения результатов работы или сообщения об ошибках:

```
backgroundWorker.DoWork += SimulateServiceCall;
backgroundWorker.ProgressChanged += ProgressChanged;
backgroundWorker.RunWorkerCompleted += RunWorkerCompleted;
```

После данной настройки вы можете запустить исполнение с помощью команды

```
backgroundWorker.RunWorkerAsync();
```

У вас есть возможность в любой момент отменить выполнение потока, вызвав метод `backgroundWorker.CancelAsync()`, который устанавливает свойство `CancellationPending` в рабочем потоке. Также необходимо написать код, который будет постоянно проверять статус отмены и корректно завершать работу.

При отсутствии исключений результат выполнения потока может быть направлен обратно к вызывающему объекту со следующей установкой:

```
e.Result = data;
```

Если в программе есть какие-либо необработанные исключения, они корректно возвращаются вызывающему объекту. Это можно осуществить путем его «обертывания» в `RunWorkerCompletedEventArgs` для передачи в качестве параметра обработчику события `RunWorkerCompleted`.

В следующем разделе мы рассмотрим преимущества и недостатки использования `BackgroundWorker`.

Преимущества и недостатки использования BackgroundWorker

Преимущества использования `BackgroundWorker`:

- потоки могут быть использованы для освобождения основного потока;
- потоки создаются и должным образом поддерживаются при помощи `ThreadPool`;
- способствует корректной и автоматической обработке исключений;
- предоставляет отчетность о ходе выполнения работ, поддерживает отмену команд и логику завершения с использованием событий.

Недостатком использования `BackgroundWorker` является то, что с большим количеством потоков код становится трудно отлаживать и поддерживать.

Многопоточность и многозадачность

Мы рассмотрели работу многопоточности и многозадачности, и нами было обнаружено, что каждый из этих подходов имеет свои преимущества и недостатки. Ниже представлены сценарии, для которых лучше подходит использование многопоточности.

- **Если необходима система, которую легко настроить и остановить**: многопоточность может оказаться полезным свойством в том случае, если имеется процесс с большим ресурсопотреблением. При использовании потоков можно параллельно выполнять необходимые методы, передав в них нужные данные. Если же реализовывать параллельную обработку на основе процессов (запуск новых экземпляров одной и той же программы), то это потребует гораздо больше ресурсов.
- **Если необходимо быстро переключаться между задачами**: кеш-память процессора и программный контекст могут легко сохраняться между потоками в процессе. Если же происходит переключение процессов, то это требует перезагрузки кеша и контекста.
- **Если необходимо обмениваться данными с другими потоками**: внутри процесса все потоки совместно используют общую память, что облегчает обмен данными. Если же осуществлять обмен данными между процессами, то нужны специальные операции ввода-вывода и транспортные протоколы, что требует гораздо больших ресурсов.

В данном разделе мы обсудили основы многопоточности и многозадачности наряду с различными подходами, которые были использованы для создания потоков в более старых версиях .NET. В следующем разделе мы попытаемся разобрать некоторые сценарии, где возможно использование техник параллельного программирования.

Сценарии, при которых полезно параллельное программирование

Ниже представлены сценарии, в которых может быть использовано параллельное программирование.

- **Создание отзывчивого пользовательского интерфейса для приложений с графическим интерфейсом**: мы можем переносить ресурсозатратные задачи на рабочий поток, тем самым предоставляя возможность основному потоку обрабатывать пользовательские взаимодействия и задачи перерисовки интерфейса.
- **Обработка одновременных запросов**: при реализации серверных приложений нам необходимо обрабатывать большое количество параллельных подключений. Мы можем создавать отдельный поток для обработки каждого запроса. Например, можно использовать модель запроса ASP.NET, которая реализуется через ThreadPool (пул потоков) – под каждый поступающий на сервер запрос назначается поток из пула. Затем поток выполняет обработку запроса и отправляет ответ клиенту.
- **Эффективное использование ЦП**: при использовании многоядерных процессоров без многопоточности обычно применяется лишь одно ядро. Можно полноценно использовать ресурсы ЦП, создавая несколько потоков, каждый из которых будет работать на своем ядре. Таким образом, распределение нагрузки приводит к повышению производительности. Это важно для длительных и сложных вычислений, которые можно выполнить быстрее, используя стратегию «разделяй и властвуй».
- **Оценивающие задачи**: сценарии, включающие несколько алгоритмов, например для сортировки входного набора чисел в кратчайшие сроки. Единственный способ сделать это – передать входные данные всем алгоритмам и запустить их параллельно. Принимается тот алгоритм, который первым завершится, а остальные отменяются.

Преимущества и недостатки параллельного программирования

Многопоточность приводит к параллелизму со своими плюсами и минусами. Освоив основные понятия параллельного программирования, обратимся теперь к его преимуществам и недостаткам.

Преимущества параллельного программирования:

- **повышенная производительность**. Можно достигнуть роста производительности, поскольку задачи распределяются по работающим параллельно потокам;
- **улучшенная отзывчивость графического интерфейса**. Так как задачи производят неблокирующий ввод-вывод, поток графического интерфейса всегда может получить входные данные от пользователя. Это приводит к лучшей отзывчивости;
- **одновременный и параллельный запуск задач**. По причине того, что задачи выполняются параллельно, мы можем одновременно запускать несколько алгоритмов;
- **эффективное использование возможностей ЦП, а также кеш-памяти.** Задачи могут выполняться на разных ядрах, что приводит к увеличению пропускной способности.

Параллельное программирование также имеет следующие недостатки:

- **сложные процессы отладки и тестирования**. В связи с параллельной работой потоков их сложно отлаживать без специальных инструментов;
- **издержки на переключение контекста**. Каждый поток работает на выделенном ему отрезке времени. Как только время заканчивается, происходит переключение контекста, что также приводит к неэффективному использованию ресурсов;
- **высокая вероятность возникновения взаимной блокировки**. Если несколько потоков работают с общим ресурсом, то необходима блокировка для обеспечения потокобезопасности. Однако если несколько потоков одновременно блокируют и ждут один и тот же ресурс, это может привести к взаимоблокировкам;
- **трудности в программировании**. Написание параллельных программ может вызвать сложности (по сравнению с синхронными версиями);
- **непредсказуемые результаты**. Поскольку параллельное программирование опирается на процессорные ядра, мы можем получить разные результаты на разных конфигурациях машин.

Следует помнить, что параллельное программирование является относительным понятием. То, что подходит для одних задач, не всегда подойдет остальным, поэтому рекомендуем попробовать этот подход самостоятельно и проверить его на практике.

Резюме

В этой главе мы рассмотрели сценарии, преимущества и особенности параллельного программирования. За последние несколько десятилетий компьютерные системы эволюционировали от одноядерных к многоядерным. Аппаратное обеспечение в чипах также поддерживает Hyper Threading, что дополнительно повышает производительность современных систем.

Прежде чем перейти к параллельному программированию, полезно повторить основные понятия, связанные с ОС, такие как процессы, задачи и разница между многопоточностью и многозадачностью.

В следующей главе мы полностью сосредоточимся на TPL и связанных с ней реализациях. Однако в реальном мире существует много устаревшего кода, который все еще опирается на более старые конструкции, поэтому знание о нем не будет лишним.

Вопросы

1. Многопоточность является подмножеством параллельного программирования.
    1. Да
    2. Нет
2. Сколько ядер будет в однопроцессорной двухъядерной ЭВМ с включенным HyperThreading?
    1. 2
    2. 4
    3. 8
3. Когда приложение завершается, все потоки переднего плана также прекращают работу. Нет необходимости реализовывать логику закрытия потоков переднего плана при выходе из приложения.
    1. Да
    2. Нет
4. Какое исключение возникает, когда поток пытается получить доступ к элементам управления, которые ему не принадлежат (которых он не создавал)?
    1. `ObjectDisposedException`
    2. `InvalidOperationException`
    3. `CrossThreadException`
5. Какой из классов поддерживает отмену команд и предоставляет отчетность о ходе выполнения работ?
    1. `Thread`
    2. `BackgroundWorker`
    3. `ThreadPool`