ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Средства асинхронного программирования – мощный механизм, добавленный в версию 5.0 языка программирования C#. Это произошло как раз в тот момент, когда производительность и распараллеливание вызывают всё более пристальный интерес у разработчиков программного обеспечения. При правильном использовании новые средства позволяют создавать программы с такими характеристиками производительности и параллелизма, которых раньше можно было добиться только за счет написания объемного и громоздкого кода. Однако тема эта непроста, и у нее есть масса нюансов, не очевидных с первого взгляда.

Если не считать Visual Basic .NET, в который средства асинхронного программирования были добавлены одновременно с C#, то ни в одном из других широко распространенных языков программирования ничего эквивалентного нет[1]. Рекомендации по их применению в реальных программах найти трудно, опыт еще не наработан. В этой книге я хочу поделиться своей практикой работы, а также идеями, почерпнутыми у проектировщиков C# и из теоретической информатики. Но главное – я покажу, что такое механизм async, как он работает и почему его стоит использовать.

Предполагаемая аудитория

Эта книга рассчитана на программистов, уверенно владеющих языком C#. Быть может, вы хотите понять, что такое механизм async и стоит ли его использовать в своих программах. А, быть может, уже вовсю работаете с ним, но хотели бы познакомиться с передовыми приемами и узнать о подводных камнях.

Тем не менее, знакомство с другими продвинутыми средствами C# не предполагается, так что книга может быть полезна также начинающим программистам на C# и тем, кто пишет на других языках.

На C# разрабатываются самые разные приложения, и во всех них механизм async может найти применения – для различных целей. Поэтому в этой книге мы будем рассматривать как клиентские, так

1 Это чрезмерно категоричное утверждение оставим на совести автора. – *Прим. перев.*

и серверные программы и посвятим отдельные главы технологиям ASP.NET и WinRT.

Как читать эту книгу

Предполагается, что вы будете читать книгу последовательно, от начала до конца – для того чтобы познакомиться с механизмом async. Новые идеи излагаются по порядку и иллюстрируются на примерах. В особенности это относится к первым пяти главам.

Лучший способ чему-то научиться – практиковаться, поэтому я рекомендую выполнять все примеры. Для этого вам понадобится какая-нибудь среда разработки на C#, например Microsoft Visual Studio или MonoDevelop. Не упускайте возможность развить примеры и применить полученные знания в собственных программах – так вы сможете глубже усвоить материал.

Прочитав всю книгу, вы, возможно, захотите использовать главы, начиная с шестой, как справочник по продвинутым способам работы с async. Эти главы относительно независимы.

- В главах 6 и 7 рассматриваются приемы, применяемые при программировании с помощью async.
- Главы 8 и 9 посвящены нетривиальным аспектам механизма async.
- В главах с 10 по 13 обсуждаются ситуации, в которых механизм async может быть полезен.
- В главах 14 и 15 речь идет о внутреннем устройстве async.

Принятые соглашения

В этой книге используются следующие графические выделения:

Курсив
: Таким шрифтом набраны новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширинный шрифт`
: Применяется для набора листингов, а также внутри основного текста для обозначения элементов программы: имен переменных и функций, баз данных, типов данных, переменных окружения, языковых конструкций и ключевых слов.

`Моноширинный курсив`
: Так набран текст, вместо которого нужно подставить задаваемые пользователем или зависящие от контекста значения.

Таким значком обозначаются советы, предложения и замечания общего характера.

О примерах кода

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Async in C# 5.0 by Alex Davies O'Reilly). Copyright 2012 Alex Davies, 978-1-449-33716-2».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: *http://oreil.ly/Async_in_CSharp5*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *http://www.oreilly.com*.

Читайте нас на Facebook: *http://facebook.com/oreilly*.

Следите за нашей лентой в Twitter: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Я благодарен Стивену Тоубу (Stephen Toub) не только за техническое редактирование, но и за ценные сведения о некоторых аспектах параллельных вычислений. Именно из его блога я впервые узнал о многих идеях, которые объясняю на страницах этой книги. Спасибо Хэмишу за корректуру и Кэти за то, что она подавала мне чай во время работы.

Я также благодарен Рэчел Румелиотис (Rachel Roumeliotis), моему редактору, и всему коллективу издательства O'Reilly, оказывавшему мне неоценимую помощь в работе над книгой.

Я признателен своей семье, особенно маме, которая ухаживала за мной после операции, когда и была написана большая часть текста. И наконец, благодарю своих коллег по компании Red Gate, которые создали благоприятствующую экспериментам атмосферу, в которой у меня и зародилось желание исследовать механизм async на практике.

Об авторе

Алекс Дэвис – программист, блогер и энтузиаст параллельного программирования. Проживает в Англии. В настоящее время является разработчиком и владельцем продукта в компании Red Gate, занимающейся созданием инструментария для платформы .NET. Получил ученую степень по информатике в Кэмбриджском университете и до сих пор теоретическая информатика у него в крови. В свободное время разрабатывает каркас Actors с открытым исходным кодом для .NET, цель которого – упростить написание параллельных программ.

ГЛАВА 1. Введение

Начнем с общего введения в средства асинхронного программирования (или просто *async*) в C# 5.0.

Асинхронное программирование

Код называется асинхронным, если он запускает какую-то длительную операцию, но не дожидается ее завершения. Противоположностью является блокирующий код, который ничего не делает, пока операция не завершится.

К числу таких длительных операций можно отнести:

- сетевые запросы;
- доступ к диску;
- продолжительные задержки.

Основное различие заключается в том, в каком *потоке* выполняется код. Во всех популярных языках программирования код работает в контексте какого-то потока операционной системы. Если этот поток продолжает делать что-то еще, пока выполняется длительная операция, то код асинхронный. Если поток в это время ничего не делает, значит, он заблокирован и, следовательно, вы написали блокирующий код.

Разумеется, есть и третья стратегия ожидания результата длительной операции – *опрос*. В этом случае вы периодически интересуетесь, завершилась ли операция. Для очень коротких операций такой способ иногда применяется, но в общем случае эта идея неудачна.

Вполне возможно, что вы уже применяли асинхронный код в своих программах. Всякий раз, запуская новый поток или пользуясь классом `ThreadPool`, вы пишете асинхронную программу, потому что текущий поток может продолжать заниматься другими вещами. Если вам доводилось создавать веб-страницы, из которых пользова-

тель может обращаться к другим страницам, то такой код был асинхронным, потому, что внутри веб-сервера нет потока, ожидающего, когда пользователь закончит ввод данных. Кажется очевидным, но вспомните о консольном приложении, которое запрашивает данные от пользователя с помощью метода `Console.ReadLine()`, и вам станет понятно, как мог бы выглядеть альтернативный блокирующий дизайн веб-приложений. Да, такой дизайн был бы кошмаром, но всё же он возможен.

Для асинхронного кода характерна типичная трудность: как узнать, когда операция завершилась? Ведь только после этого можно приступить к обработке ее результатов. В блокирующем коде все тривиально – следующая строка помещается сразу после вызова длительной операции. Но в асинхронном мире так сделать нельзя, потому что размещенная в этом месте строка почти наверняка будет выполнена раньше, чем асинхронная операция завершится.

Для решения этой проблемы придуман целый ряд приемов, позволяющих выполнить код по завершении фоновой операции:

- включить нужный код в состав самой операции, после кода, составляющего ее основное назначение;
- подписаться на событие, генерируемое по завершении;
- передать делегат или лямбда-функцию, которая должна быть выполнена по завершении (*обратный вызов*).

Если код, следующий за асинхронной операцией, необходимо выполнить в конкретном потоке (например, в потоке пользовательского интерфейса в программе на базе *WinForms* или *WPF*), то приходится ставить операцию в очередь этого потока. Всё это очень утомительно.

Чем так хорош асинхронный код?

Асинхронный код освобождает поток, из которого был запущен. И это очень хорошо по многим причинам. Прежде всего, потоки потребляют ресурсы компьютера, а чем меньше расходуется ресурсов, тем лучше. Часто существует лишь один поток, способный выполнить определенную задачу (например, поток пользовательского интерфейса) и, если не освободить его быстро, то приложение перестанет реагировать на действия пользователя. Мы еще вернемся к этой теме в следующей главе.

Но самым важным мне представляется тот факт, что асинхронное выполнение открывает возможность для параллельных вычислений.

Вы можете структурировать программу по-новому, реализовав мелкоструктурный параллелизм, но не жертвуя простотой и удобством сопровождения. Этот вопрос мы будем исследовать в главе 10.

Что такое async?

В версии C# 5.0 Microsoft добавила механизм, предстающий в виде двух новых ключевых слов: `async` и `await`.

Этот механизм опирается на ряд нововведений в .NET Framework 4.5, без которых был бы бесполезен.

Механизм async встроен в компилятор и без поддержки с его стороны не мог бы быть реализован в библиотеке. Компилятор преобразовывает исходный код, то есть действует примерно по тому же принципу, что в случае лямбда-выражений и итераторов в предыдущих версиях C#.

Эта возможность существенно упрощает *асинхронное* программирование, избавляя от необходимости использовать сложные приемы, как то было в предыдущих версиях языка. С ее помощью можно написать всю программу целиком в асинхронном стиле.

В этой книгу я буду называть словом **асинхронный** общий стиль программирования, упростившийся после появления в C# механизма **async**. На C# всегда можно было писать асинхронные программы, но это требовало значительных усилий со стороны программиста.

Что делает async?

Механизм async дает простой способ выразить, что должна делать программа по завершении длительной асинхронной операции. Метод, помеченный ключевым словом `async`, компилятор преобразует так, что асинхронный код выглядит очень похоже на блокирующий эквивалент. Ниже приведен простой пример блокирующего метода для загрузки веб-страницы.

```
private void DumpWebPage(string uri)
{
 WebClient webClient = new WebClient();
 string page = webClient.DownloadString(uri);
 Console.WriteLine(page);
}
```

А вот эквивалентный ему асинхронный метод.

```
private async void DumpWebPageAsync(string uri)
{
 WebClient webClient = new WebClient();
 string page = await webClient.DownloadStringTaskAsync(uri);
 Console.WriteLine(page);
}
```

Похожи, не правда ли? Но под капотом они сильно отличаются. Второй метод помечен ключевым словом `async`. Это обязательное условие для всех методов, в которых используется ключевое слово `await`. Еще мы добавили к имени метода суффикс `Async`, чтобы соблюсти общепринятое соглашение.

Наибольший интерес представляет ключевое слово `await`. Видя его, компилятор переписывает метод. Точная процедура довольно сложна, поэтому пока я приведу лишь ее не вполне корректное описание, которое, на мой взгляд, полезно для понимания простых случаев.

1. Весь код после `await` переносится в отдельный метод.
2. Новый вариант метода `DownloadString` называется `DownloadStringTaskAsync`. Он делает то же самое, что исходный, но асинхронно.
3. Это означает, что мы можем передать ему сгенерированный метод, который будет вызываться по завершении операции. Делается это с помощью некоторых магических манипуляций, о которых я расскажу ниже.
4. Когда загрузка страницы завершится, будет вызван наш код, которому передается загруженная строка `string`; в данном случае мы просто выводим ее на консоль.

```
private void DumpWebPageAsync(string uri)
{
 WebClient webClient = new WebClient();
 webClient.DownloadStringTaskAsync(uri) <- magic(SecondHalf);
}

private void SecondHalf(string awaitedResult)
{
 string page = awaitedResult;
 Console.WriteLine(page);
}
```

Что происходит в вызывающем потоке, когда он исполняет такой код? По достижении вызова `DownloadStringTaskAsync` начинается загрузка страницы. Но не в текущем потоке. В нем вызванный метод

сразу возвращает управление. Что делать дальше, решает вызывающая программа. К примеру, поток пользовательского интерфейса мог бы продолжить обработку действий пользователя. Или просто завершиться, освободив ресурсы. В любом случае мы написали асинхронный код!

Async не решает все проблемы

Механизм async намеренно спроектирован так, чтобы максимально напоминать блокирующий код. Мы можем рассматривать длительные или удаленные операции, как будто они выполняются локально и быстро, увеличив производительность за счет асинхронности.

Однако он не дает вам совсем забыть о том, что на самом деле операция выполняется в фоновом режиме и происходит обратный вызов. Необходимо иметь в виду, что многие средства языка в асинхронном режиме ведут себя по-другому, частности:

- исключения и блоки try-catch-finally;
- возвращаемые методами значения;
- потоки и контекст;
- производительность.

Не зная, что происходит в действительности, вы не сможете ни понять смысл неожиданных сообщений об ошибках, ни воспользоваться отладчиком для их исправления.

ГЛАВА 2. Зачем делать программу асинхронной

Асинхронное программирование – вещь важная и полезная, но почему именно важная, зависит от вида приложения. Некоторые преимущества проявляются всегда, но особенно значимы в приложениях, которые вы, возможно, не имели в виду. Поэтому рекомендую прочитать эту главу, чтобы лучше понимать контекст в целом.

Приложения с графическим интерфейсом пользователя для настольных компьютеров

К приложениям для настольных компьютеров предъявляется одно важное требование – они должны реагировать на действия пользователя. Исследования в области человеко-машинного интерфейса показывают, что пользователь не обращает внимания на медленную работу программы, если она откликается на его действия и – желательно – имеет индикатор хода выполнения.

Но если программа зависает, то пользователь недоволен. Обычно зависания связаны с тем, что программа перестает реагировать на действия пользователя во время выполнения длительной операции, будь то медленное вычисление или операция ввода/вывода, например обращение к сети.

Все каркасы для организации пользовательского интерфейса в C# работают в одном потоке. Это относится и к WinForms, и к WPF, и к Silverlight. Только этот поток может управлять содержимым окна, распознавать действия пользователя и реагировать на них. Если он занят или блокирован дольше нескольких десятков миллисекунд, то пользователь сочтет, что приложение «тормозит».

Асинхронный код, даже написанный вручную, позволяет потоку пользовательского интерфейса вернуться к своей основной обязанности – опросу *очереди сообщений* и реагированию на появляющиеся в ней события. Он также может анимировать ход выполнения задачи, а в последних версиях Windows еще и наведение мыши на различные объекты. То и другое служит для пользователя наглядным подтверждением того, что программа работает.

Наличие только одного потока пользовательского интерфейса позволяет упростить синхронизацию. Если бы таких потоков было много, то один мог бы попытаться получить ширину кнопки в момент, когда другой занят размещением элементов управления. Чтобы избежать конфликтов, пришлось бы повсеместно расставлять блокировки; при этом производительность оказалась бы не лучше, чем в случае единственного потока.

Аналогия с кафе

Чтобы помочь вам разобраться, прибегну к аналогии. Если вы полагаете, что и так всё понимаете, можете спокойно пропустить этот раздел.

Представьте себе небольшое кафе, которое предлагает тосты на завтрак. Функции официантки выполняет сам хозяин. Он очень серьезно относится к качеству обслуживания клиентов, но об асинхронной обработке слыхом не слыхивал.

Поток пользовательского интерфейса как раз и моделирует действия хозяина кафе. Если в компьютере вся работа выполняется потоками, то в кафе – обслуживающим персоналом. В данном случае персонал состоит из одного-единственного человека, которого можно уподобить единственному потоку пользовательского интерфейса.

Первый посетитель заказывает тост. Хозяин берет ломтик хлеба, включает тостер и ждет, пока тот поджарит хлеб. Посетитель спрашивает, где взять масло, но хозяин его игнорирует – он всецело поглощен наблюдением за тостером, иначе говоря – блокирован. Через пять минут тост готов, и хозяин подает его посетителю. К этому моменту уже скопилась очередь, а посетитель обижен, что на него не обращают внимания. Ситуация далека от идеала.

Посмотрим, нельзя ли научить хозяина кафе действовать асинхронно.

Во-первых, необходимо, чтобы сам тостер мог работать асинхронно. При написании асинхронного кода мы должны позаботиться о

том, чтобы запущенная нами длительная операция могла выполнить обратный вызов по завершении. Точно так же, у тостера должен быть таймер, а поджаренный хлеб должен выскакивать с громким звуком, так чтобы хозяин заметил это.

Теперь хозяин может включить тостер и на время забыть о нем, вернувшись к обслуживанию клиентов. Точно так же, наш асинхронный код должен возвращать управление сразу после запуска длительной операции, чтобы поток пользовательского интерфейса мог реагировать на действия пользователя. Тому есть две причины:

- у пользователя остается впечатление, что интерфейс «отзывчивый», – клиент может попросить масло, и его не проигнорируют;
- пользователь может одновременно начать другую операцию – следующий клиент может изложить свой заказ.

Теперь хозяин кафе может одновременно обслуживать нескольких клиентов; единственным ограничением является количество тостеров и время, необходимое для подачи готового теста. Однако при этом возникают новые проблемы: необходимо помнить, кому какой тост предназначен. На самом деле, поток пользовательского интерфейса, вернувшись к обслуживанию очереди событий, ничего не помнит о том, завершения каких операций он ждет.

Поэтому мы должны связать с запускаемой задачей обратный вызов, который известит нас о том, что задача завершилась. Хозяину кафе достаточно прикрепить к тосту листочек с именем клиента. Нам же может потребоваться нечто более сложное, и в общем случае хотелось бы иметь возможность задавать произвольные инструкции, что делать по завершении задачи.

Последовав нашим советам, хозяин кафе стал работать полностью асинхронно, и его дело процветает. Довольны и клиенты. Ждать приходится меньше, и их просьбы больше не игнорируются. Надеюсь, эта аналогия помогла вам лучше понять, почему асинхронность так важна в приложениях с пользовательским интерфейсом.

Серверный код веб-приложения

У ASP.NET-приложений на веб-сервере нет ограничения на единственный поток, как в случае программ с пользовательским интерфейсом. И тем не менее асинхронное выполнение может оказаться весьма полезным, так как для таких приложений характерны длительные операции, особенно запросы к базе данных.

В зависимости от версии IIS может быть ограничено либо общее число потоков, обслуживающих веб-запросы, либо общее число одновременно обрабатываемых запросов. Если большая часть времени обработки запроса уходит на обращение к базе данных, то увеличение числа одновременно обрабатываемых запросов может повысить пропускную способность сервера.

Когда поток блокирован в ожидании какого-то события, он не потребляет процессорное время. Но не следует думать, что он вообще не потребляет ресурсы сервера. На самом деле, любой поток, даже блокированный, потребляет два ценных ресурса.

Память

В Windows для каждого управляемого потока резервируется примерно один мегабайт виртуальной памяти. Если количество потоков исчисляется десятками, то это не страшно, но когда их сотни, то может превратиться в проблему. Если операционная система вынуждена выгружать память на диск, то возобновление потока резко замедляется.

Ресурсы планировщика

Планировщик операционной системы отвечает за выделение потокам процессоров. Планировщик должен рассматривать даже заблокированные потоки, иначе он не будет знать, когда они разблокируются. Это замедляет контекстное переключение, а, значит, и работу системы в целом.

В совокупности эти накладные расходы означают дополнительную нагрузку на сервер, а, стало быть, увеличивают задержку и снижают пропускную способность.

Помните: основная отличительная особенность асинхронного кода состоит в том, что поток, начавший длительную операцию, освобождается для других дел. В случае ASP.NET этот поток берется из пула потоков, поэтому после запуска длительной операции он сразу же возвращается в пул и может затем использоваться для обработки других запросов. Таким образом, для обработки одного и того же количества запросов требуется меньше потоков.

Еще одна аналогия: кухня в ресторане

Веб-сервер можно рассматривать как модель ресторана. Есть много клиентов, заказывающих еду, а кухня пытается обслужить все заказы как можно быстрее.

На нашей кухне много поваров, каждый из которых можно уподобить потоку. Повар готовит заказанные блюда, но в течение какого-то времени любое блюдо должно постоять на плите, а повару в этот момент делать нечего. Точно так же обстоит дело с веб-запросом, для обработки которого нужно обратиться к базе данных, – веб-сервер к этому отношения не имеет.

При блокирующей организации работ на кухне повар будет стоять у плиты, дожидаясь готовности блюда. Чтобы добиться точной аналогии с заблокированным потоком, которому не выделяется процессорное время, предположим, что повару ничего не платят за время, когда он ждет готовности. Быть может, в это время он читает газету.

Но даже если повару не нужно платить и для приготовления каждого блюда можно нанять нового повара, простаивающие в ожидании повара занимают место на кухне. В одну кухню не удастся впихнуть больше нескольких десятков поваров, иначе в ней будет трудно перемещаться, и вся работа станет.

Разумеется, асинхронная система работает куда лучше. Ставя блюдо на плиту или в духовку, повар помечает, что это за блюдо и на какой стадии приготовления оно находится, а затем начинает заниматься другим делом. Когда подойдет время снимать блюдо с плиты, это сможет сделать любой повар, – он и продолжит его готовить.

Этот подход может быть эффективно применен в веб-серверах. Несколько потоков могут справиться с обработкой того же количества одновременных запросов, для которого раньше требовались сотни потоков (а то и вообще не удавалось обработать). На самом деле, в некоторых каркасах для создания веб-серверов, например в *node.js*, отвергается сама идея о наличии нескольких потоков, и все запросы асинхронно обрабатываются единственным потоком. Зачастую при этом удается обработать больше запросов, чем может многопоточная, но блокирующая система. Точно так же, единственный повар в пустой кухне, хорошо организовавший свою работу, сможет приготовить больше блюд, чем сотня поваров, которые только и делают, что мешают друг другу или почитывают газетку.

Silverlight, Windows Phone и Windows 8

Проектировщики Silverlight, безусловно, знали о преимуществах асинхронного кода в приложениях с пользовательским интерфейсом, поэтому решили поощрить такой подход. Для этого они просто

изъяли из каркаса большую часть синхронных API. Так, например, веб-запросы можно отправлять только асинхронно.

Асинхронный код «заразен». Если где-то вызвать какой-нибудь асинхронный API, то и вся программа естественно становится асинхронной. Поэтому в Silverlight вы *обязаны* писать асинхронный код – альтернативы просто не существует. Возможно, и существует метод `Wait` или иной способ работать с асинхронным API синхронно, приостановив выполнение в ожидании обратного вызова. Но, поступая так, вы теряете все преимущества, о которых я говорил выше.

Silverlight для Windows Phone, как следует из названия, является разновидностью Silverlight. Правда, включены дополнительные API, небезопасные в среде браузера, например TCP-сокеты. Но и в этом случае предоставляются только асинхронные версии API, что заставляет вас писать асинхронный код. И уж для мобильного устройства, располагающего крайне ограниченными ресурсами, это вдвойне оправдано. Запуск дополнительных потоков весьма негативно отражается на времени работы аккумулятора.

И в Windows 8, технически не связанной с Silverlight, подход такой же. Количество различных API в этом случае гораздо больше, но для методов, которые могут выполняться дольше 50 мс, предоставляются только асинхронные версии.

Параллельный код

Современные компьютеры оснащаются несколькими процессорными ядрами, работающими независимо друг от друга. Желательно, чтобы программа могла задействовать имеющиеся ядра, но два ядра не могут писать в одну и ту же ячейку памяти, так как это чревато повреждением ее содержимого.

Быть может, было бы лучше применять *чистое* (иначе говоря, функциональное) программирование, при котором не существует побочных эффектов, то есть состояние памяти не изменяется. Это помогло бы полнее воспользоваться преимуществами параллелизма, но для некоторых программ неприемлемо. Пользовательским интерфейсам состояние необходимо. Базы данных сами по себе являются состоянием.

Стандартное решение предполагает использование взаимно исключающих блокировок (мьютексов) в случаях, когда несколько ядер потенциально могут обращаться к одной и той же ячейке памяти. Но тут есть свои проблемы. Часто бывает так, что программа захватывает

блокировку, а затем вызывает метод или генерирует событие, в котором захватывается другая блокировка. Иногда удерживать сразу две блокировки необязательно, но так код оказывается проще. В результате другим потокам приходится ждать освобождения блокировки, хотя они могли бы в это время заниматься полезной работой. Хуже того, иногда возникает ситуация, когда каждый поток ждет освобождения блокировки, занятой другим, а это приводит к взаимоблокировке (deadlock). Такие ошибки трудно предвидеть, воспроизводить и исправлять.

Одно из самых многообещающих решений – модель вычислений, основанная на *акторах*. При таком подходе каждый участок записываемой памяти принадлежит ровно одному актору. Единственный способ записать в эту память – отправлять актору-владельцу сообщения, которые он обрабатывает поочередно и, возможно, посылает сообщения в ответ. Но это как раз и есть асинхронное программирование. Запрос действия у актора – типичная асинхронная операция, поскольку мы можем заниматься другими вещами, пока не придет ответное сообщение. А значит, для программирования такой модели можно использовать механизм async, как мы и увидим в главе 10.

Пример

Рассмотрим пример приложения для ПК, которое откровенно нуждается в переходе на асинхронный стиль. Его исходный код можно скачать по адресу https://bitbucket.org/alexdavies74/faviconbrowser. Рекомендую сделать это (если вы не пользуетесь системой управления версиями Mercurial, то код можно скачать в виде zip-файла) и открыть решение в Visual Studio. Необходимо скачать ветвь `default`, содержащую синхронную версию.

Запустив программу, вы увидите окно с кнопкой. Нажатие этой кнопки приводит к отображению значков нескольких популярных сайтов. Для этого программа скачивает файл `favicon.ico`, присутствующий на большинстве сайтов (рис. 2.1).

Приглядимся к коду. Наиболее важная его часть – метод, который загружает значок и добавляет его на панель WrapPanel (отметим, что это приложение WPF).

```
private void AddAFavicon(string domain)
{
 WebClient webClient = new WebClient();
 byte[] bytes = webClient.DownloadData("http://"+domain+"/favicon.ico");
```

```
    Image imageControl = MakeImageControl(bytes);
    m_WrapPanel.Children.Add(imageControl);
  }
```

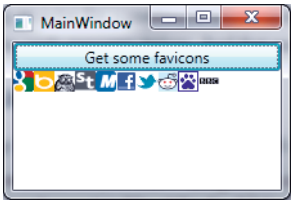

Рис. 2.1. Обозреватель значков сайтов

Обратите внимание, что эта реализация полностью синхронна. Поток блокируется на время скачивания значка. Вероятно, вы заметили, что в течение нескольких секунд после нажатия кнопки окно ни на что не реагирует. Как вы теперь понимаете, объясняется это тем, что поток пользовательского интерфейса блокирован и не обрабатывает события, генерируемые действиями пользователя.

В следующих главах мы преобразуем эту синхронную программу в асинхронную.