Часть I

Основы асинхронного программирования и многопоточности

Первая часть этой книги охватывает асинхронное программирование и многопоточность на C#, объясняя, что это такое и как правильно их использовать. В этой части освещаются распространенные подводные камни и рассказывается, как их избежать.

Сначала мы рассмотрим понятия и терминологию многопоточности и асинхронного программирования, используемые в информатике в целом и в C# в частности (глава 1). Затем погрузимся в особенности асинхронного программирования с применением `async/await` на C# (главы 2 и 3). Потом обсудим многопоточность в C# (глава 4) и как многопоточность и асинхронное программирование работают вместе (глава 5). Наконец, мы поговорим о том, когда следует использовать `async/await` (глава 6) и как правильно использовать многопоточность (глава 7).

К концу части 1 вы научитесь писать корректный многопоточный код и правильно использовать `async/await`.

1 Асинхронное программирование и многопоточность

В этой главе

- ✓ Введение в многопоточность.
- ✓ Введение в асинхронное программирование.
- ✓ Совместное использование асинхронного программирования и многопоточности.

Как разработчики программного обеспечения мы часто стремимся сделать наши приложения более быстрыми, отзывчивыми и эффективными. Один из способов достичь этого — позволить компьютеру выполнять несколько задач одновременно, максимально используя существующие ядра процессора. Многопоточность и асинхронное программирование — это два подхода, которые обычно применяются для решения этой задачи.

Многопоточность позволяет компьютеру создавать видимость одновременного выполнения нескольких задач, даже когда их количество превышает количество ядер процессора. Асинхронное программирование, напротив, фокусируется на оптимизации использования процессора во время операций, которые обычно блокируют его, что гарантирует эффективное использование вычислительных ресурсов.

Возможность компьютера выполнять несколько задач одновременно чрезвычайно полезна. Она помогает поддерживать высокую отзывчивость приложений во время их работы и играет важную роль в работе высокопроизводительных

серверов, которые благодаря этой возможности получают способность взаимодействовать со множеством клиентов одновременно.

Оба подхода можно использовать для создания отзывчивых клиентских приложений или серверов, работающих с несколькими клиентами. А их сочетание может значительно повысить производительность и позволить серверам обрабатывать тысячи клиентов одновременно.

Эта глава познакомит вас с многопоточностью и асинхронным программированием и покажет, почему они важны. В оставшейся части книги будем говорить о том, как правильно их использовать в .NET и C#, и основное внимание уделим операторам `async/await` в C#. Вы узнаете, как работают эти подходы, научитесь правильно их использовать и обходить распространенные подводные камни.

1.1. Что такое многопоточность

Прежде чем начать рассматривать `async/await`, нам нужно понять, что такое многопоточность и асинхронное программирование. Для этого мы немного поговорим о веб-серверах и приготовлении пиццы. Начнем с пиццы (потому что она вкуснее веб-сервера).

В общих чертах процесс приготовления пиццы в заведениях, продающих еду навынос, обычно выглядит так.

1. Повар получает заказ.
2. Берет готовое тесто, формует его, добавляет соус, сыр и начинку.
3. Ставит пиццу в духовку и ждет, пока она приготовится (это самый долгий этап).
4. Затем повар делает еще несколько дел — достает пиццу из духовки, разрезает ее и кладет в коробку.
5. И наконец, передает пиццу доставщику.

Это не поваренная книга, поэтому наш алгоритм выпечки пиццы — метафора для одного из самых простых видов серверов — веб-сервера, обслуживающего статические файлы. В общих чертах работа простого веб-сервера выглядит так.

1. Сервер получает веб-запрос.
2. Исследует запрос, чтобы выяснить, что нужно сделать.
3. Читает файл (это самый долгий этап).
4. Выполняет дополнительную обработку (например, упаковывает содержимое файла).
5. Отправляет содержимое файла обратно браузеру.

В большей части главы мы будем игнорировать первый и последний шаги, потому что в большинстве веб-фреймворков (включая ASP.NET и ASP.NET Core) они обрабатываются самим фреймворком, а не нашим кодом. Мы кратко поговорим о них ближе к концу этой главы. Схема на рис. 1.1 иллюстрирует процесс обработки веб-запроса.

А теперь вернемся к пицце. В простейшем случае повар будет выполнять шаги по порядку, полностью заканчивая приготовление одной пиццы, прежде чем приступить к следующей. Пока пицца выпекается, повар будет просто стоять и ничего не делать (это полностью синхронная однопоточная версия процесса).

Рис. 1.1. Синхронная однопоточная обработка запроса

В мире веб-серверов в роли повара выступает центральный процессор. Код этого простого однопоточного веб-сервера последовательно выполняет необходимые операции, обрабатывая веб-запрос, и пока файл читается с диска, процессор простаивает и ничего не делает. В действительности операционная система приостанавливает ожидающий поток и передает процессор другой программе, но с точки зрения нашей программы это выглядит так, будто процессор простаивает.

Такая версия процесса имеет свои преимущества — она простая и легкая для понимания. Вы можете посмотреть на текущий шаг и с полной уверенностью сказать, на каком этапе процесса находитесь. Поскольку никакие два этапа не выполняются одновременно, они не могут мешать друг другу. Наконец, эта версия требует наименьшего объема памяти и потребляет меньше ресурсов, так как в каждый момент времени обрабатывает только один веб-запрос (или готовит только одну пиццу).

В то же время эта однопоточная синхронная версия процесса довольно расточительна, потому что большую часть времени повар/процессор ничего не делает и ждет, пока пицца приготовится в духовке (или файл загрузится с диска), и если наша пиццерия не обанкротится, то в какой-то момент мы начнем получать новые заказы быстрее, чем сможем их выполнять.

По этой причине было бы желательно, чтобы повар мог делать несколько пицц одновременно. Для этого можно использовать таймер, который будет подавать

звуковой сигнал каждые несколько секунд. Каждый раз, когда таймер подает сигнал, повар прекращает делать то, что он делал, записывая на бумажке, на чем он остановился, и начинает готовить новую пиццу или продолжает готовить предыдущую (игнорируя пиццы, выпекающиеся в духовке), пока таймер снова не подаст сигнал.

В этой версии повар пытается делать несколько дел одновременно, и каждое из этих дел называется *потоком*. Каждый поток — это последовательность операций, которые могут выполняться параллельно с другими похожими или отличающимися последовательностями.

Этот пример может показаться слишком наивным, потому что он явно неэффективен, и наш повар будет тратить слишком много времени, чтобы записать на бумажке, на чем он остановился, или прочитать с бумажки, чтобы понять, с какого места продолжить работу. Однако именно так работает многопоточность. Внутри процессора есть таймер, который сигнализирует, когда он должен переключиться на следующий поток, и при каждом переключении процессор сохраняет информацию о том, что он делал, и загружает состояние другого потока (это называется *переключением контекста*).

Например, когда код читает файл, поток не может продолжить работу, пока данные не извлечены из файла на диске. До этого момента мы говорим, что поток *заблокирован*. Выделение процессорного времени заблокированному потоку, очевидно, было бы расточительством, поэтому, когда поток начинает читать файл, операционная система переводит его в заблокированное состояние. Войдя в это состояние, поток немедленно освобождает процессор, чтобы им мог воспользоваться другой поток (возможно, в другой программе), и операционная система не будет выделять потоку процессорное время, пока он находится в заблокированном состоянии. Когда система закончит читать файл, она выведет поток из заблокированного состояния и тот снова сможет получать свои кванты процессорного времени.

Операции, которые могут привести к блокировке потока, называются *блокирующими*. Таковыми являются все операции доступа к файлам и сети. Блокирующими также являются все операции, взаимодействующие с чем-либо еще, кроме процессора и памяти, и все операции, ожидающие наступления некоторого события в другом потоке.

Вернемся в нашу пиццерию. В дополнение ко времени, необходимому на переключение между пиццами, повару нужно время, чтобы прочитать информацию и вернуться в то же самое место, где он был до переключения. Каждый поток в наших программах, даже если он не запущен, занимает некоторый объем памяти, поэтому создание очень большого количества потоков, каждый из которых выполняет блокирующую операцию (и бóльшую часть времени пребывает в заблокированном состоянии, не потребляя процессорного времени),

приведет к расточительному расходованию памяти. С увеличением числа потоков программа будет замедляться все больше из-за необходимости тратить все больше времени на управление потоками. В какой-то момент программа начнет тратить так много времени на управление потоками, что на выполнение полезной работы не останется времени или просто закончится память и произойдет сбой.

Даже при всей этой неэффективности многопоточный повар, прыгающий от одной пиццы к другой как сумасшедший, приготовит больше пицц за то же время, если только не окажется перегружен сверх меры и не даст сбой (я знаю, что повар не может сломаться, но это всего лишь метафора). Это объясняется тем, что однопоточный повар до этого большую часть своего времени проводил в ожидании, пока пицца испечется в духовке.

Как показано на рис. 1.2, поскольку у нас в процессоре есть только одно ядро (я знаю, что сейчас практически все компьютеры оснащены многоядерными процессорами, и мы скоро поговорим о них), мы не можем делать два дела одновременно. Все этапы приготовления выполняются один за другим и в действительности это не происходит параллельно, однако процессор может ожидать одновременно сколько угодно задач. Именно поэтому наша многопоточная версия смогла бы обработать три запроса за значительно меньшее время, чем потребовалось бы однопоточной версии для обработки двух.

Если внимательно посмотреть на рис. 1.2, то можно увидеть, что, хотя однопоточная версия обработала первый запрос быстрее, многопоточная версия завершила обработку всех трех до того, как однопоточная версия смогла завершить второй запрос. Этот пример иллюстрирует большое преимущество многопоточности, которое заключается в более эффективном использовании процессора в сценариях, где возможны периоды ожидания, а также показывает ее цену — немного дополнительных накладных расходов на каждом этапе пути.

До сих пор мы говорили об одноядерных процессорах, но все современные процессоры имеют несколько ядер. Как это меняет ситуацию?

1.2. Многоядерные процессоры

Многоядерные процессоры концептуально просты — это лишь группа одноядерных процессоров, размещенных на одном физическом кристалле.

Наличие восьмиядерного процессора эквивалентно наличию восьми поваров в нашей пиццерии. В предыдущем примере у нас был один повар, который мог делать только одно дело за раз, но притворялся, что делает несколько дел, быстро переключаясь между ними. Теперь у нас восемь поваров, каждый из которых может делать одно дело за раз (а все вместе — восемь дел одновременно), и каждый создает видимость, что делает несколько дел, быстро переключаясь между ними.

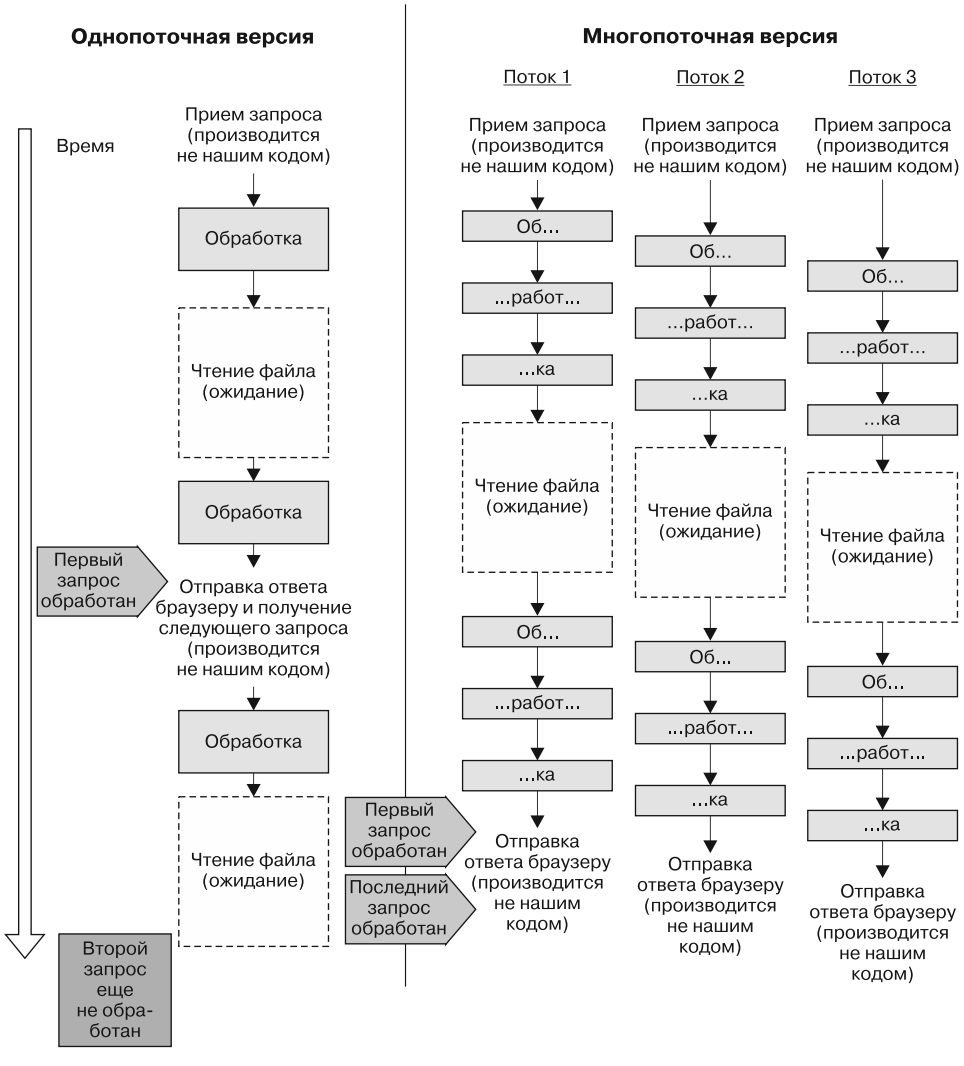

Рис. 1.2. Пример однопоточной и многопоточной обработки нескольких запросов

С точки зрения программного обеспечения при наличии многоядерного процессора, действительно, можно иметь несколько потоков, работающих по-настоящему одновременно. Когда у нас был одноядерный процессор, мы нарезали процесс на крошечные этапы и чередовали их, чтобы казалось, что они выполняются одновременно (хотя на самом деле в каждый конкретный момент времени выполнялся только один этап). Теперь, имея восьмиядерный процессор, мы по-прежнему нарезаем процесс на крошечные этапы и чередуем их, но при этом можем одновременно выполнять восемь таких этапов.

Теоретически восемь поваров способны приготовить больше пицц, чем один, однако несколько поваров могут непреднамеренно мешать друг другу работать. Например, они могут попытаться одновременно поставить пиццу в духовку или им может понадобиться использовать один и тот же нож для нарезания пиццы — чем больше поваров, тем выше вероятность, что это произойдет.

На рис. 1.3 показано, как выполняется та же многопоточная работа, что и на рис. 1.2, но на этот раз на двухъядерном процессоре (здесь показаны только два ядра, потому что схема с восемью ядрами была бы слишком большой, чтобы уместить ее на книжной странице).

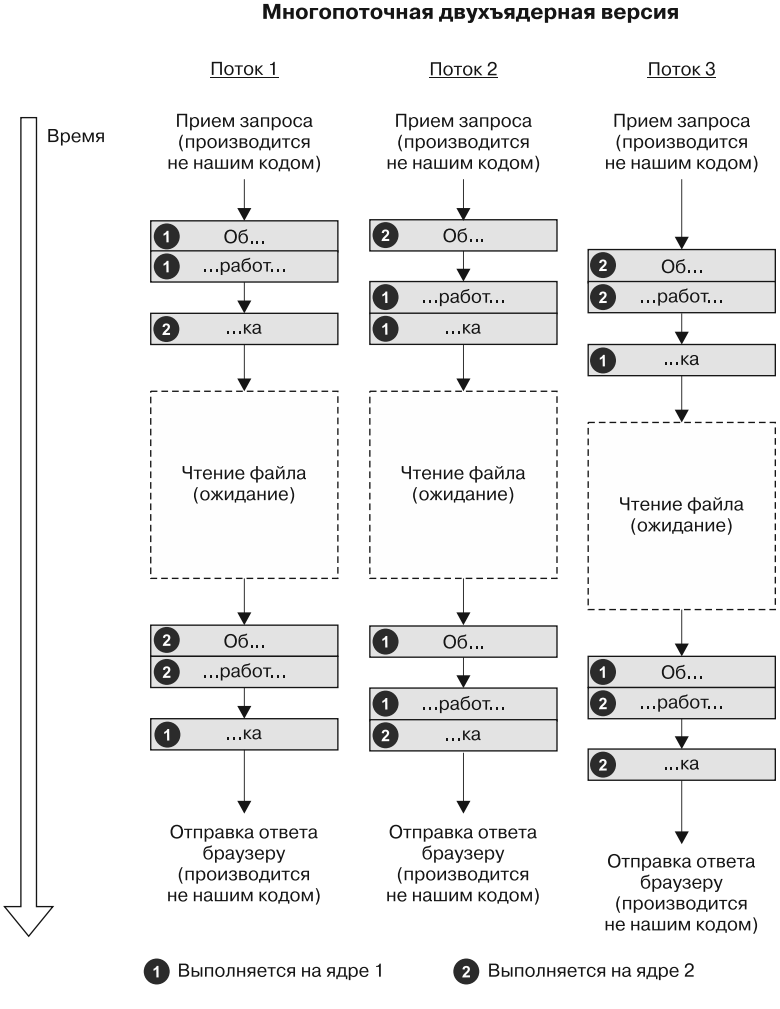

Рис. 1.3. Пример обработки трех запросов на двухъядерном процессоре

Обратите внимание, что по умолчанию между потоками и ядрами нет постоянной связи. Поток может произвольно перемещаться с одного ядра на другое (однако есть возможность привязать потоки для выполнения на определенных ядрах; этот прием так и называется — «привязка потоков», но пользоваться им следует только в особых случаях).

Двухъядерный процессор позволил вдвое сократить время обработки, по сравнению с одноядерной версией, но не повлиял на время, затраченное на ожидание. Так что, хотя мы и получили значительное ускорение, общее время сократилось менее чем вдвое. До сих пор основное улучшение производительности мы получали за счет выполнения других действий, ожидая, пока жесткий диск прочитает файл, но платили за это дополнительными накладными расходами и сложностью многопоточного программирования. Возможно, мы сможем сократить эти накладные расходы.

1.3. Асинхронное программирование

Вернемся в пиццерию и рассмотрим рациональное решение, которое раньше игнорировали: повар готовит одну пиццу, не останавливаясь и не переключаясь на другие, но, поставив пиццу в духовку, он может начать готовить следующую пиццу, а не просто сидеть и ждать. Позже, закончив очередной этап, повар может проверить, готова ли пицца в духовке, и если да, то вынуть ее, нарезать, положить в коробку и передать курьеру.

Это пример асинхронного программирования. Всякий раз, когда процессору нужно сделать что-то, что происходит вне процессора (например, прочитать файл), он отправляет задание внешнему компоненту (например, контроллеру диска) и просит этот компонент уведомить процессор, когда задание будет выполнено.

Асинхронная (или неблокирующая) версия обработки файла просто ставит операцию в очередь с помощью операционной системы (которая затем поставит ее в очередь с помощью контроллера диска) и немедленно возвращает управление, позволяя тому же потоку сделать что-то еще (рис. 1.4). Позже мы можем проверить, была ли операция завершена, и получить доступ к прочитанным данным.

Сравнив все схемы в этой главе, можно заметить, что эта однопоточная и асинхронная версия — самая быстрая из всех. Она завершает обработку первого запроса почти так же быстро, как однопоточная версия, а обработку последнего запроса — почти так же быстро, как двухъядерная многопоточная версия (и при этом не использует второе ядро), что делает ее самой производительной версией на данный момент.

Теперь взгляните на рис. 1.4. Схема на нем выглядит более беспорядочной, чем предыдущие, и ее сложнее читать, и это еще без указания того, что шаги

«последующей обработки» зависят от завершения операций чтения. Сложность чтения этой схемы объясняется тем, что на ней больше не виден весь процесс целиком. Обработка каждого запроса разбита на этапы, и, в отличие от примера с потоками, эти этапы не связаны друг с другом.

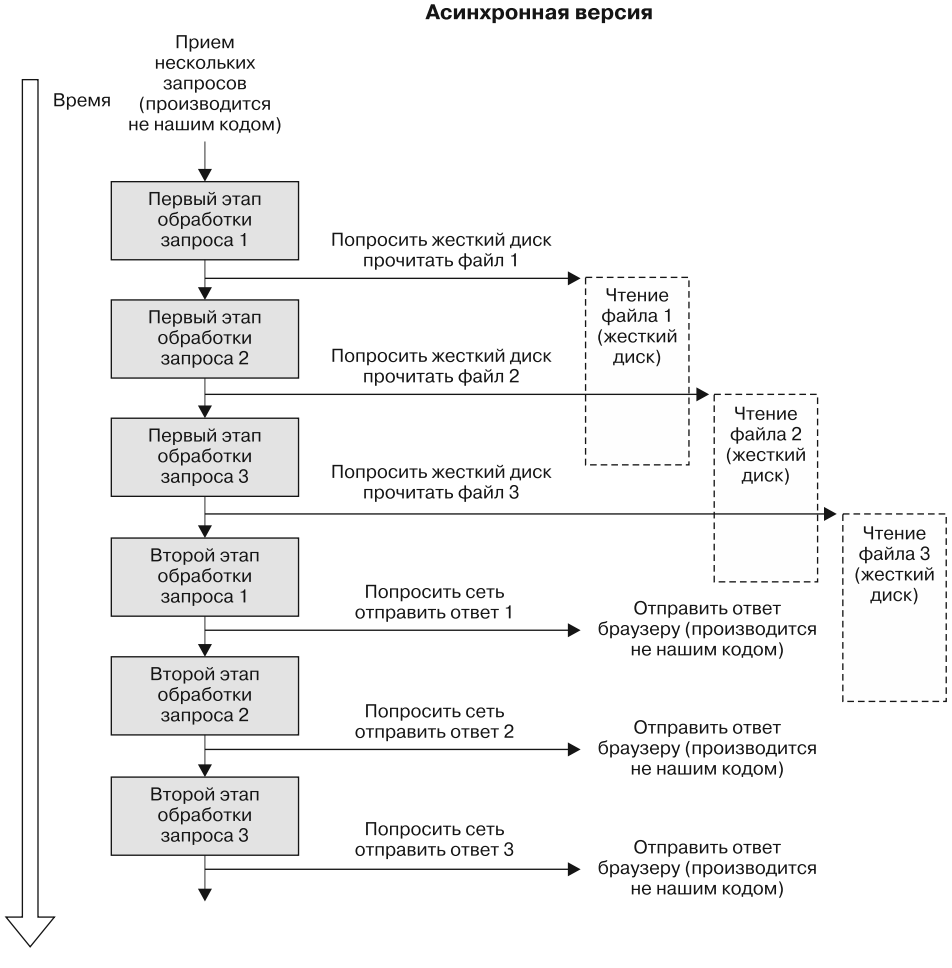

Рис. 1.4. Обработка трех запросов однопоточным асинхронным веб-сервером

Вот почему, несмотря на широкое использование многопоточности, асинхронное программирование до появления `async/await` выбирали только те, кто создавал высокопроизводительные серверы (или использовал среды, где не было другого выбора, например JavaScript). Как показано на рис. 1.4, код приходилось разбивать на части, которые пишутся по отдельности, что затрудняло создание кода

и еще больше — его понимание. Так было до тех пор, пока в C# не появились операторы `async`/`await`, позволяющие писать асинхронный код так, как если бы это был обычный синхронный код.

На рис. 1.4 также показано, что я использую такие же асинхронные техники как для чтения файла, так и для отправки ответа браузеру. Это потому, что первый и последний этапы в последовательности обработки веб-запроса — «прием веб-запроса» и «отправка ответа браузеру» — выполняются в основном сетевой картой, а не процессором, подобно тому как чтение файла производится жестким диском, и потому они могут выполняться асинхронно, не заставляя процессор ждать.

Даже с использованием одной лишь многопоточности, без применения приемов асинхронного программирования, вполне можно написать серверы, способные обрабатывать низкие и средние нагрузки, запуская отдельный поток для обслуживания каждого соединения. Однако, если вам нужно создать сервер, который может обрабатывать тысячи соединений одновременно, накладные расходы на работу такого количества потоков замедлят сервер до такой степени, что он не сможет обрабатывать запросы или вообще выйдет из строя.

Выше мы говорили об асинхронном программировании как о способе избежать многопоточности, но без многопоточности невозможно использовать всю мощь многоядерных процессоров. Давайте посмотрим, можно ли использовать многопоточность и асинхронное программирование вместе, чтобы добиться еще большей производительности.

1.4. Совместное использование многопоточности и асинхронного программирования

В последний раз вернемся к примеру с пиццерией. Мы можем еще больше усовершенствовать процесс приготовления пиццы: не заставлять повара периодически проверять духовку, а оснастить духовку звуковым сигналом, который будет подаваться по завершении выпекания пиццы. Когда духовка подаст звуковой сигнал, повар сможет прервать свою текущую работу, вынуть пиццу, положить ее в коробку, передать курьеру, а затем вернуться к прерванной работе.

Программный эквивалент — при запуске асинхронной операции попросить операционную систему уведомить программу запуском функции обратного вызова, зарегистрированной при запуске асинхронной операции. Эта функция обратного вызова должна запускаться в новом потоке (на самом деле в потоке из пула; о пулах потоков мы поговорим далее в этой книге), потому что сам вызывающий поток не ждет и в данный момент делает что-то другое. Вот почему асинхронность и многопоточность хорошо работают вместе.