Почему они не умеют писать многопоточные программы

Многопоточное программирование ничем принципиально не отличается от написания событийно-ориентированных графических пользовательских интерфейсов и даже от создания простых последовательных приложений. Здесь действуют все важные правила, касающиеся инкапсуляции, разделения ответственности, слабого связывания и т.д. Но многие разработчики с трудом пишут многопоточные программы именно потому, что пренебрегают этими правилами. Вместо этого они пытаются применять на практике гораздо менее важные знания о потоках и синхронизационных примитивах, почерпнутые в текстах о многопоточном программировании для начинающих.

Итак, что же это за правила

Иной программист, столкнувшись с проблемой, думает: «А, точно, надо применить регулярные выражения». И вот у него уже две проблемы — Джейми Завински.

Иной программист, столкнувшись с проблемой, думает: «А, точно, применю-ка я здесь потоки». И вот у него уже десять проблем — Билл Шиндлер.

Слишком многие программисты, берущиеся писать многопоточный код, попадают впросак, как герой баллады Гёте «*Ученик чародея*». Программист научится создавать пучок потоков, которые в принципе работают, но рано или поздно они выходят из-под контроля, и программист не знает, что делать.

Но в отличие от волшебника-недоучки несчастный программист не может надеяться на приход могучего чародея, который взмахнет волшебной палочкой и восстановит порядок. Вместо этого программист идет на самые неприглядные уловки, пытаясь справиться с постоянно возникающими проблемами. Результат всегда одинаков: получается чрезмерно усложненное, ограниченное, хрупкое и ненадежное приложение. В нем постоянно сохраняется угроза взаимной блокировки и существуют другие опасности, свойственные плохому многопоточному коду. Я уже не говорю о необъяснимых аварийных завершениях, плохой производительности, неполных или некорректных результатах работы.

Возможно, вы задавались вопросом: а почему это происходит? Распространено такое

ошибочное мнение: «Многопоточное программирование очень сложное». Но это не так. Если многопоточная программа ненадежна, то она обычно барахлит по тем же причинам, что и некачественные однопоточные программы. Просто программист не следует основополагающим, давно известным и проверенным методам разработки. Многопоточные программы лишь кажутся более сложными, так как чем больше параллельных потоков работают неправильно, тем больший беспорядок они учиняют — и гораздо быстрее, чем это сделал бы один поток.

Заблуждение о «сложности многопоточного программирования» широко распространилось из-за тех разработчиков, которые профессионально сложились на написании однопоточного кода, впервые столкнулись с многопоточностью и не справились с ней. Но вместо того, чтобы пересмотреть свои предубеждения и привычные приемы работы, они упрямо фиксят то, что никак не хочет работать. Оправдываясь за ненадежный софт и сорванные сроки, эти люди твердят одно и то же: «многопоточное программирование очень сложное».

Обратите внимание: выше я говорю о типичных программах, в которых используется многопоточность. Действительно, существуют сложные многопоточные сценарии — как и сложные однопоточные. Но они встречаются нечасто. Как правило, на практике от программиста не требуется ничего сверхъестественного. Мы перемещаем данные, преобразуем их, время от времени выполняем те или иные вычисления и, наконец, сохраняем информацию в базе данных или отображаем ее на экране.

Нет ничего сложного в усовершенствовании среднестатистической однопоточной программы и превращении ее в многопоточную. По крайней мере не должно быть. Сложности возникают по двум причинам:

- программисты не умеют применять простые, давно известные проверенные методы разработки;
- большинство сведений, излагаемых в книгах по многопоточному программированию, технически верны, но совершенно неприменимы при решении прикладных задач.

Самые важные концепции программирования универсальны. Они в равной степени применимы к однопоточным и к многопоточным программам. Программисты, тонущие в водовороте потоков, просто не усвоили важных уроков, еще когда осваивали однопоточный код. Я могу это утверждать потому, что такие разработчики совершают в многопоточных и однопоточных программах одни и те же фундаментальные ошибки.

Пожалуй, самый важный урок, который следовало выучить за шестидесятилетнюю историю программирования, формулируется так: *глобальное изменяемое состояние — зло*. Настоящее зло. О программах, зависящих от глобального изменяемого состояния, сравнительно сложно рассуждать, а в целом они отличаются ненадежностью, поскольку существует слишком много способов изменения состояния. Выполнена масса

исследований, подтверждающих этот общий принцип, существуют бесчисленные паттерны проектирования, основная цель которых — реализовать тот или иной способ сокрытия данных. Чтобы ваши программы были более предсказуемыми, старайтесь в максимальной степени устранить в них изменяемое состояние.

В однопоточной последовательной программе вероятность искажения данных прямо пропорциональна количеству компонентов, которые могут изменять эти данные.

Как правило, полностью избавиться от глобального состояния не удается, но в арсенале разработчика есть очень эффективные инструменты, позволяющие строго контролировать, какие компоненты программы могут изменять состояние. Кроме того, мы научились создавать ограничительные слои API вокруг примитивных структур данных. Поэтому мы хорошо контролируем, как изменяются эти структуры данных.

Проблемы глобального изменяемого состояния постепенно стали очевидными в конце 80-х и начале 90-х, с распространением событийно-ориентированного программирования. Программы больше не начинались «с начала» и не проходили единственный предсказуемый путь выполнения «до конца». У современных программ есть исходное состояние, после выхода из которого в них происходят события — в непредсказуемом порядке, с переменными временными интервалами. Код остается однопоточным, но уже становится асинхронным. Вероятность искажения данных возрастает именно потому, что порядок возникновения событий очень важен. Сплошь и рядом встречаются ситуации такого рода: если событие B происходит после события A, то все работает нормально. Но если событие A произойдет после события B, а между ними успеет вклиниться событие C, то данные могут быть искажены до неузнаваемости.

Если задействуются параллельные потоки, проблема еще больше усугубляется, так как сразу несколько методов могут одновременно оперировать глобальным состоянием. Становится невозможно судить о том, как именно изменяется глобальное состояние. Речь уже идет не только о том, что события могут происходить в непредсказуемом порядке, но и о том, что состояние нескольких потоков выполнения может обновляться *одновременно*. При асинхронном программировании вы, как минимум, можете гарантировать, что определенное событие сможет произойти не раньше, чем закончится обработка другого события. То есть можно с определенностью сказать, каким будет глобальное состояние на момент окончания обработки конкретного события. В многопоточном коде, как правило, невозможно сказать, какие события будут происходить параллельно, поэтому невозможно с определенностью описать глобальное состояние в любой момент времени.

Многопоточная программа с обширным глобальным изменяемым состоянием — это один из наиболее красноречивых известных мне примеров принципа неопределенности Гейзенберга. Невозможно проверить состояние программы, не изменив при этом ее поведение.

Когда я начинаю очередную филиппику о глобальном изменяемом состоянии (суть изложена в нескольких предыдущих абзацах), программисты закатывают глаза и уверяют меня, что все это им давно известно. Но если это вам известно — почему этого не скажешь по вашему коду? Программы нашпигованы глобальным изменяемым состоянием, а программисты удивляются, почему код не работает.

Неудивительно, что самая важная работа при многопоточном программировании происходит на этапе проектирования. Требуется четко определить, что должна делать программа, разработать для выполнения всех функций независимые модули, детально описать, какие данные требуются какому модулю, и определить пути обмена информацией между модулями (*Да, еще не забудьте подготовить красивые футболки для всех участников проекта. Первым делом. — прим. ред. в оригинале*). Этот процесс принципиально не отличается от проектирования однопоточной программы. Ключ к успеху, как и в случае с однопоточным кодом — ограничить взаимодействия между модулями. Если удастся избавиться от разделяемого изменяемого состояния, то проблемы совместного доступа к данным просто не возникнут.

Кто-то может возразить, что иногда нет времени на такое филигранное проектирование программы, которое позволит обойтись без глобального состояния. Я же считаю, что на это можно и нужно тратить время. Ничто не сказывается на многопоточных программах так губительно, как попытки справиться с глобальным изменяемым состоянием. Чем большим количеством деталей приходится управлять, тем выше вероятность, что ваша программа войдет в пике и рухнет.

В реалистичных прикладных программах должно существовать определенное разделяемое состояние, которое может изменяться. И вот тут у большинства программистов начинаются проблемы. Программист видит, что здесь требуется разделяемое состояние, обращается к многопоточному арсеналу и берет оттуда самый простой инструмент: универсальную блокировку (критическая секция, мьютекс или как это у них еще называется). Они, видимо, полагают, что взаимное исключение решит все проблемы с совместным доступом к данным.

Количество проблем, которые могут возникнуть при такой единой блокировке, просто ошеломляет. Необходимо учесть и условия гонки, и проблемы пропускания (gating problems) при чрезмерно обширной блокировки, и вопросы, связанные со справедливостью распределения — вот лишь несколько примеров. Если же у вас несколько блокировок, в особенности если они вложенные, то также придется принять меры против взаимной блокировки, динамической взаимной блокировки, очередей на блокировку, а также исключить другие угрозы, связанные с параллелизмом. К тому же существуют и характерные проблемы одиночной блокировки.
Когда я пишу или проверяю код, я руководствуюсь практически безотказным железным правилом: *если вы сделали блокировку, то, по-видимому, где-то допустили ошибку*.

Это утверждение можно прокомментировать двумя способами:

1. Если вам понадобилась блокировка, то, вероятно, у вас присутствует глобальное изменяемое состояние, которое требуется защитить от параллельных обновлений. Наличие глобального изменяемого состояния — это недоработка, допущенная на этапе проектирования приложения. Пересмотрите и измените дизайн.
2. Правильно пользоваться блокировками нелегко, а локализовать баги, связанные с блокировкой, бывает невероятно сложно. Весьма вероятно, что вы будете использовать блокировку неправильно. Если я вижу блокировку, а программа при этом необычно себя ведет, то я первым делом проверяю код, зависящий от блокировки. И обычно нахожу в нем проблемы.

Обе эти интерпретации корректны.

Писать многопоточный код несложно. Но очень, очень сложно правильно использовать синхронизационные примитивы. Возможно, вам не хватает квалификации для правильного использования даже одной блокировки. Ведь блокировки и другие синхронизационные примитивы — это конструкции, возводимые на уровне всей системы. Люди, которые разбираются в параллельном программировании гораздо лучше вас, пользуются этими примитивами для построения конкурентных структур данных и высокоуровневых синхронизационных конструктов. А мы с вами, обычные программисты, просто берем такие конструкции и используем в нашем коде. Программист, пишущий приложения, должен использовать низкоуровневые синхронизационные примитивы не чаще, чем он делает непосредственные вызовы драйверов устройств. То есть практически никогда.

Пытаясь при помощи блокировок решать проблемы совместного доступа к данным, вы как будто тушите пожар жидким кислородом. Как и пожар, такие проблемы легче предотвратить, чем устранить. Если вы избавитесь от разделяемого состояния, то не придется и злоупотреблять синхронизационными примитивами.

Большинство того, что вы знаете о многопоточности, не имеет значения

В пособиях по многопоточности для начинающих вы узнаете, что такое потоки. Потом автор начнет рассматривать различные способы, которыми можно наладить параллельную работу этих потоков — например, расскажет о контроле доступа к разделяемым данным при помощи блокировок и семафоров, остановится на том, какие вещи могут произойти при работе с событиями. Подробно рассмотрит условные переменные, барьеры памяти, критические секции, мьютексы, volatile-поля и атомарные операции. Будут рассмотрены примеры того, как использовать эти низкоуровневые конструкции для выполнения всевозможных системных операций. Дочитав этот материал

до половины, программист решает, что уже достаточно знает обо всех этих примитивах и об их применении. В конце концов, если я знаю, как эта штука работает на системном уровне, то смогу таким же образом применить ее и на уровне приложения. Да?

Представьте себе, что вы рассказали подростку, как самому собрать двигатель внутреннего сгорания. Затем без всякого обучения вождению вы сажаете его за руль автомобиля и говорите: «Езжай»! Подросток понимает, как работает машина, но не имеет ни малейшего представления о том, как добраться на ней из точки A в точку B.

Понимание того, как потоки работают на системном уровне, обычно никак не помогает использовать их на уровне приложения. Я не утверждаю, что программистам не нужно учить все эти низкоуровневые детали. Просто не рассчитывайте, что сможете с ходу применить эти знания при проектировании или разработке бизнес-приложения.

Во вводной литературе на тему многопоточности (а также в соответствующих академических курсах) не следует изучать такие низкоуровневые конструкции. Нужно сосредоточиться на решении самых распространенных классов проблем и показать разработчикам, как такие задачи решаются при помощи высокоуровневых возможностей. В принципе большинство бизнес-приложений — это исключительно простые программы. Они считывают данные с одного или нескольких устройств ввода, выполняют какую-либо сложную обработку этих данных (например, в процессе работы запрашивают еще какие-то данные), а затем выводят результаты.

Зачастую такие программы отлично вписываются в модель «поставщик-потребитель», требующую применения всего трех потоков:

поток ввода считывает данные и помещает их в очередь ввода;
рабочий поток считывает записи из очереди ввода, обрабатывает их и помещает результаты в очередь вывода;
поток вывода считывает записи из очереди вывода и сохраняет их.

Три этих потока работают независимо, коммуникация между ними происходит на уровне очередей.

Хотя технически эти очереди можно считать зонами разделяемого состояния, на практике они представляют собой всего лишь коммуникационные каналы, в которых действует собственная внутренняя синхронизация. Очереди поддерживают работу сразу со многими производителями и потребителями, в них можно параллельно добавлять и удалять элементы.

Поскольку этапы ввода, обработки и вывода изолированы между собой, их реализацию несложно менять, не затрагивая остальной части программы. Пока не меняется тип данных, находящихся в очереди, можно по своему усмотрению выполнять рефакторинг

отдельных компонентов программы. Кроме того, поскольку в работе очереди участвует произвольное число поставщиков и потребителей, не составит труда добавить и другие производители/потребители. У нас могут быть десятки потоков ввода, записывающих информацию в одну и ту же очередь, либо десятки рабочих потоков, забирающих информацию из очереди ввода и переваривающих данные. В рамках одного компьютера такая модель хорошо масштабируется.

Но самое важное заключается в том, что современные языки программирования и библиотеки очень упрощают создание приложений по модели «производитель-потребитель». В .NET вы найдете параллельные коллекции и библиотеку TPL Dataflow. В Java есть сервис Executor, а также BlockingQueue и другие классы из пространства имен java.util.concurrent. В C++ есть библиотека Boost для работы с потоками и библиотека Thread Building Blocks от Intel. В Visual Studio 2013 от Microsoft появились асинхронные агенты. Подобные библиотеки также имеются в Python, JavaScript, Ruby, PHP и, насколько мне известно, во многих других языках. Вы сможете создать приложение вида «производитель-потребитель» при помощи любого из этих пакетов, ни разу не прибегая к блокировкам, семафорам, условным переменным или каким-либо другим синхронизационным примитивам.

В этих библиотеках свободно используются самые разные синхронизационные примитивы. Это нормально. Все перечисленные библиотеки написаны людьми, которые разбираются в многопоточности несравнимо лучше среднего программиста. Работа с подобной библиотекой практически не отличается от использования языковой библиотеки времени исполнения. Это можно сравнить с программированием на высокоуровневом языке, а не на ассемблере.

Модель «поставщик-потребитель» — всего один из многих примеров. В вышеперечисленных библиотеках содержатся классы, при помощи которых можно реализовать многие распространенные паттерны многопоточного проектирования, не вдаваясь при этом в низкоуровневые детали. Можно создавать масштабные многопоточные приложения, практически не заморачиваясь о том, как именно координируются потоки проходит синхронизация.

Работайте с библиотеками

Итак, создание многопоточных программ ничем принципиально не отличается от написания однопоточных синхронных программ. Важные принципы инкапсуляции и сокрытия данных универсальны, и их значение лишь возрастает, когда в работе участвует множество параллельных потоков. Если пренебрегать этими важными аспектами, то вас не спасут даже самые исчерпывающие знания низкоуровневого обращения с потоками.

Современным разработчикам приходится решать массу задач на уровне программирования приложений, бывает, что просто некогда задумываться о том, что

происходит на системном уровне. Чем затейливее становятся приложения, тем более сложные детали приходится скрывать между уровнями API. Мы занимаемся этим уже не один десяток лет. Можно утверждать, что качественное скрытие сложности системы от программиста — основная причина, по которой программисту удается писать современные приложения. Если уж на то пошло — разве мы не скрываем сложность системы, реализуя цикл сообщений пользовательского интерфейса, выстраивая низкоуровневые протоколы обмена информацией и т.д.?

С многопоточностью складывается аналогичная ситуация. Большинство многопоточных сценариев, с которыми может столкнуться средний программист бизнес-приложений, уже хорошо известны и качественно реализованы в библиотеках. Библиотечные функции отлично скрывают ошеломляющую сложность параллелизма. Этими библиотеками нужно научиться пользоваться точно так же, как вы пользуетесь библиотеками элементов пользовательского интерфейса, протоколами связи и многочисленными другими инструментами, которые просто работают. Оставьте низкоуровневую многопоточность специалистам — авторам библиотек, применяемых при создании прикладных программ.

Джим Мишель

Источник

Теги: многопоточность, перевод, программирование