Содержание

Вступительное слово

Книга, которую вы держите в руках, весьма любопытна. Это книга по JavaScript, открывающаяся примером, написанным на C, а речь в ней пойдет о многопоточности в языке, который явным образом объявлен однопоточным. В ней приведены интереснейшие примеры того, как и когда следует намеренно блокировать цикл событий, хотя эксперты на протяжении многих лет убеждали вас никогда так не поступать. А заканчивается она прекрасным списком причин, по которым не стоит использовать описанные механизмы, и подводных камней, подстерегающих на этом пути. Но я считаю, что эту книгу должен прочитать любой JavaScript-разработчик вне зависимости от того, где он предполагает развертывать и исполнять свой код.

Помогая компаниям создавать более эффективные и производительные приложения на JavaScript для Node.js, я часто должен был делать паузу и обсуждать с разработчиками распространенные заблуждения об этом языке программирования. Например, однажды мне встретился программист, имеющий большой опыт разработки на Java и .NET, который доказывал, что создание нового обещания в JavaScript очень напоминает создание потока в Java (это не так) и что обещания позволяют JavaScript выполнять код параллельно (не позволяют). В другой раз человек создал приложение для Node.js, которое запускало больше тысячи одновременных рабочих потоков, и никак не мог понять, почему не наблюдает ожидаемого повышения производительности при тестировании на машине, оснащенной всего восьмью процессорными ядрами. Урок понятен: многопоточность, конкурентность и параллелизм – все еще мало знакомые и трудные темы для очень большого процента JavaScript-разработчиков.

Борьба с этими заблуждениями и заставила меня (вместе с коллегой и членом технического руководящего комитета Node.js Маттео Коллина) создать семинар Broken Promises, посвященный основам асинхронного программирования на JavaScript, – мы учили команды разработчиков, как правильно рассуждать о порядке выполнения кода и хронометраже различных событий. Она же побудила меня заняться проектом с открытым исходным кодом Piscina (вместе с соразработчиком ядра Node.js Анной Хеннингсен), который предлагает учитывающую передовые практики реализацию модели пула потоков поверх рабочих потоков в Node.js. Но все это помогает решить лишь часть проблемы.

В этой книге Брайан и Томас квалифицированно описывают основы многопоточной разработки вообще и искусно иллюстрируют, как различные среды выполнения JavaScript, а именно браузеры и Node.js, допускают параллельные вычисления на языке, который не содержит никаких встроенных механизмов для этого. Поскольку ответственность за поддержку многопоточности возложена на среды выполнения, а между средами есть масса различий, браузеры и платформы типа Node.js реализуют многопоточность по-разному. И, хотя API похожи, рабочий поток в Node.js на деле совсем не

то, что веб-исполнитель в браузере. Поддержка разделяемых исполнителей, веб-исполнителей и сервисных исполнителей стала почти универсальной во всех браузерах, а рабочие потоки в Node.js существуют уже несколько лет, но все равно для JavaScript-разработчиков это относительно новые концепции. Но, где бы ни исполнялся ваш код, эта книга станет для вас источником прозрения и важной информации. Но самое главное – авторы точно объясняют, почему вообще следует «заморачиваться» многопоточностью в JavaScript-приложениях.

— Джеймс Снелл,
член технического руководящего комитета Node.js

Предисловие

Брайан и я (Томас) впервые встретились в Сан-Франциско на моем собеседовании при приеме на работу в филиал японской компании по разработке мобильных игр DeNA. Казалось, что большая часть руководства была готова отказать, но, после того как вечером в тот же день мы вдвоем заявились на неформальную встречу сообщества Node.js, Брайан убедил их сделать мне предложение.

Работая в DeNA, мы с Брайаном занимались написанием повторно используемых модулей для Node.js, чтобы группы разработчиков могли создавать игровые серверы из готовых компонентов, отвечающих целям игры. Мы всегда измеряли производительность, а обучение команд методам производительного программирования было частью нашей работы; наши серверы были объектами постоянного и пристального внимания разработчиков из индустрии, которая традиционно опиралась на C++.

Нам довелось работать вместе и в других местах. Одним из них стал небольшой стартап в области безопасности под названием Intrinsic, где в нашу задачу входило укрепление приложений для Node.js на таком всеобъемлющем и мелкоструктурном уровне, что я сомневаюсь, найдется ли в мире еще один подобный продукт. Оптимизация производительности также стояла там на одном из первых мест, поскольку заказчики не хотели терять ни грана пропускной способности. Мы потратили много часов на прогон тестов производительности, корпение над пламенными диаграммами и копание в потрохах Node.js. Если бы модуль рабочих потоков был доступен во всех версиях Node.js, поддержки которых требовал заказчик, то мы, безусловно, включили бы его в продукт.

Мы работали вместе и не по найму. Один из таких примеров – NodeSchool SF (https://nodeschool.io/sanfrancisco/), где мы бесплатно учили других использовать JavaScript и писать программы для Node.js. Мы также не раз выступали на одних и тех же конференциях и встречах по интересам.

Оба автора питают страсть к JavaScript и Node.js, а также к преподаванию этих предметов и развеиванию заблуждений. Осознав, как остро не хватает документации по созданию многопоточных JavaScript-приложений, мы поняли, что нужно делать. Эта книга родилась из нашего желания не только рассказывать другим о возможностях JavaScript, но также доказать, что платформы типа Node.js ничуть не хуже любых других с точки зрения создания высокопроизводительных служб, в полной мере задействующих доступное оборудование.

Для кого написана эта книга

Идеальный читатель этой книги – инженер, у которого за плечами несколько лет работы с JavaScript, но который, возможно, никогда не писал многопоточных приложений даже на таких традиционно поддерживающих многопоточ-

ность языках, как C++ или Java. Мы включили пример прикладного кода на C, как «всеобщем» многопоточном языке, но не предполагаем, что читатель знаком с ним или хотя бы понимает написанный на нем код.

Если у вас есть опыт работы с такими языками, отлично – и тогда эта книга поможет понять предлагаемый JavaScript эквивалент конструкциям, знакомым по другим языкам. С другой стороны, если вы писали код только на JavaScript, то и тогда эта книга для вас. Мы включили информацию разного уровня: ссылки на низкоуровневые API, высокоуровневые паттерны и множество технических деталей, заполняющих пространство между тем и другим.

Цели

Пожалуй, самая важная цель этой книги – принести сообществу благую весть о том, что на JavaScript можно писать многопоточные приложения. Традиционно считалось, что JavaScript-код занимает только одно ядро, и действительно в Twitter и на разных форумах полно постов такого толка. Назвав книгу «Многопоточный JavaScript», мы надеемся полностью развенчать миф о том, будто JavaScript-приложения ограничены единственным ядром.

На более конкретном уровне цель книги – научить читателя нескольким аспектам написания многопоточных JavaScript-приложений. Прочитав книгу до конца, вы будете понимать различные API веб-исполнителей в браузерах, их сильные и слабые стороны и когда какой использовать. Что до Node.js, вы узнаете о модуле рабочих потоков и сможете сравнить его API с тем, что имеется в браузере.

В книге рассматриваются два подхода к построению многопоточных приложений: с использованием передачи сообщений и разделяемой памяти. Дочитав книгу, вы будете знать, какие API используются для реализации того и другого, когда отдать предпочтение одному подходу, а когда другому и в каких ситуациях их можно сочетать. У вас даже будете шанс потренироваться в создании некоторых высокоуровневых паттернов на основе этих подходов.

Графические выделения

В книге применяются следующие графические выделения.

Курсив
: Новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

`Моноширинный`
: Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные среды, предложения и ключевые слова языка.

Моноширинный полужирный
: Команды и иные строки, которые следует вводить буквально.

Моноширинный курсив
: Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.


Так обозначается совет или рекомендация.


Так обозначается замечание общего характера.


Так обозначается предупреждение или предостережение.

О ПРИМЕРАХ КОДА

Дополнительные материалы (примеры кода, упражнения и т. д.) можно скачать по адресу https://github.com/MultithreadedJSBook/code-samples.

Если у вас возникнет технический вопрос или затруднение в использовании примеров кода, можете отправить сообщение на адрес bookquestions@oreilly.com.

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 из книг издательства O'Reilly 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: «Multithreaded JavaScript by Thomas Hunter II and Bryan English (O'Reilly). Copyright 2022 Thomas Hunter II and Bryan English, 978-1-098-10443-6».

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

Благодарности

Эта книга стала возможной благодаря подробным техническим рецензиям, написанным следующими лицами.

Анна Хеннингсен (@addaleax)
: В настоящее время работает в команде инструментов разработки MongoDB в Германии. Анна была одним из самых активных соразработчиков ядра

Node.js на протяжении последних пяти лет и принимала участие в реализации рабочих потоков для этой платформы. Она одержима страстью к Node.js и его сообществу.

Шу-ю Гуо (@_shu)

Шу работает над реализацией и стандартизацией JavaScript. Он входит в комитет TC39, является одним из редакторов спецификации ECMAScript и автором модели памяти. В настоящее время работает над движком Google V8, главным источником реализации и стандартов языковых средств JavaScript. До того работал в Mozilla и агентстве Bloomberg.

Фернандо Ларраньяга (@xabadu)

Фернандо – инженер и соразработчик ПО с открытым исходным кодом. Несколько лет возглавлял сообщества JavaScript и Node.js в Южной Америке и в США. В настоящее время занимает должность старшего инженера-программиста в компании Square и является организатором неформальных встреч NodeSchool SF. А на предыдущих местах работы в других крупных технологических компаниях, например Twilio и Groupon, он разрабатывал приложения Node.js уровня предприятия и занимался масштабированием веб-приложений, используемых миллионами пользователей, начиная с 2014 года.

Об авторах

Томас Хантер II участвовал в разработке десятков сервисов Node.js и работал в компании, занимающейся обеспечением безопасности Node.js. Он выступал на нескольких конференциях по Node.js и JavaScript, имеет сертификат JSNSD/JSNAD и является организатором NodeSchool SF.

Брайан Инглиш разрабатывает проекты с открытым исходным кодом на JavaScript и Rust, занимался крупными корпоративными системами, оснащением инструментальными средствами и безопасностью на уровне приложений. В настоящее время работает старшим инженером на проектах с открытым исходным кодом в компании Datadog. Использовал Node.js в рабочих и личных проектах чуть ли не сразу после его появления. Также является соразработчиком ядра Node.js и внес немалый вклад в различные аспекты Node.js, принимая участие в некоторых рабочих группах.

Глава 1

Введение

Когда-то компьютеры были куда проще. Мы не хотим сказать, что их было легко использовать или что писать для них код не составляло труда, но концептуально возни с ними было гораздо меньше. Типичный ПК 80-х годов прошлого века имел один восьмиразрядный процессор и не бог весть сколько памяти. Как правило, в каждый момент времени могла работать только одна программа. Даже операционная система (в современной терминологии) не работала одновременно с программой, взаимодействующей с пользователем.

Но шло время, и люди возжелали запускать сразу несколько программ – так родилась многозадачность. Это дало возможность операционным системам исполнять одновременно несколько программ и переключаться между ними. Программы могли сами решать, когда следует уступить процессор операционной системе, чтобы та могла выполнить другую программу. Такой подход называется *кооперативной*, или *невытесняющей*, *многозадачностью*.

В системе с невытесняющей многозадачностью если какая-то программа по ошибке или намеренно не уступала процессор, то никакая другая программа не могла выполниться. Такие помехи работе других программ были нежелательны, поэтому в конечном итоге операционные системы перешли на *вытесняющую многозадачность*. В этой модели операционная система решала, какой программе и в какой момент выделить процессор, и использовала для этого собственные представления о справедливом планировании, не отдавая решения о переключении процессора самим программам. Сегодня почти во всех операционных системах используется именно этот подход, даже в многоядерных компьютерах, потому что обычно исполняемых программ больше, чем процессорных ядер.

Выполнение нескольких задач одновременно исключительно полезно и программистам, и пользователям. До изобретения потоков одна программа (т. е. один *процесс*) не могла выполнять сразу несколько задач. Если программисту все же нужны были конкурентные задачи, то он должен был либо разбить задачу на несколько меньших частей и самостоятельно планировать их выполнение внутри процесса, либо запускать отдельные задачи в разных процессах и организовать их взаимодействие.

Даже в наши дни в некоторых высокоуровневых языках для выполнения нескольких задач одновременно рекомендуется запускать дополнительные процессы. В таких языках, как Ruby и Python, имеется *глобальная блокировка*

интерпретатора (global interpreter lock – *GIL*), которая означает, что в каждый момент времени может работать только один поток. Хотя управление памятью при этом упрощается, сама идея многозадачности теряет привлекательность, и программисты предпочитают использовать несколько процессов.

До сравнительно недавнего времени JavaScript тоже был языком, в котором имелся единственный механизм многозадачности: разбить задачу на части и запланировать выполнение этих частей в будущем, а в случае Node.js – запускать дополнительные процессы. Обычно мы разбиваем код на асинхронные блоки, применяя обратные вызовы или обещания. Типичный фрагмент кода, написанный в таком духе, показан в примере 1.1, где операции организованы в виде обратных вызовов или `await`.

Пример 1.1 ❖ Два варианта написания типичного фрагмента асинхронного JavaScript-кода

```
readFile(filename, (data) => {
  doSomethingWithData(data, (modifiedData) => {
    writeFile(modifiedData, () => {
      console.log('done');
    });
  });
});

// или

const data = await readFile(filename);
const modifiedData = await doSomethingWithData(data);
await writeFile(filename);
console.log('done');
```

Сегодня во всех основных средах выполнения JavaScript имеется доступ к потокам, и, в отличие от Ruby и Python, нет никакой блокировки GIL, из-за которой они становятся практически бесполезными при выполнении счетных задач. Есть, правда, другие ограничения, например не использовать совместно один объект JavaScript в нескольких потоках (по крайней мере, не напрямую). Но все равно потоки полезны JavaScript-разработчикам, когда нужно огородить счетные задачи, отделив их от остальных. В браузере существуют также специальные потоки, которым доступна не такая функциональность, как у главного потока. Как все делается, мы подробно рассмотрим в последующих главах, но чтобы вы могли составить первоначальное представление, в примере 1.2 показано, как просто создать новый поток и обработать сообщение в браузере.

Пример 1.2 ❖ Запуск потока в браузере

```
const worker = new Worker('worker.js');
worker.postMessage('Hello, world');

// worker.js
self.onmessage = (msg) => console.log(msg.data);
```

Цель этой книги – исследовать и объяснить поток JavaScript как концепцию и элемент программирования. Вы узнаете, как их использовать и, что еще важнее, когда это делать. Не каждую проблему можно решить с помощью потоков. И даже не каждую счетную задачу следует решать с помощью потоков. Работа разработчиков ПО в том и заключается, чтобы оценивать проблемы и инструменты и выбирать наиболее подходящие решения. А наша цель – снабдить вас еще одним инструментом и достаточной информацией о том, как и когда применять его.

Что такое потоки?

Во всех современных операционных системах единицы исполнения вне ядра организованы в процессы и потоки. Разработчики могут использовать процессы и потоки и механизмы их взаимодействия, чтобы добавить в проект конкурентность. В системах с несколькими процессорами это также означает добавление параллелизма.

При выполнении любой программы, например Node.js или редактора кода, вы создаете процесс. Это означает, что код загружается в область памяти, выделенную этому и только этому процессу, и программа не сможет адресовать никакую другую область памяти, не запросив у ядра дополнительную память или отображение памяти. Если не создать дополнительных потоков или процессов, то в каждый момент времени можно будет выполнять только одну машинную *команду*, причем порядок команд предписан кодом программы. Можете считать, что команда – это единица кода, что-то вроде строки кода. (На самом деле команда обычно соответствует одной строке ассемблерного кода!)

Программа может запускать дополнительные процессы, имеющие собственную область памяти. Эти процессы не разделяют память между собой (если только она не была отображена с помощью специальных системных вызовов) и имеют свои собственные указатели команд, т. е. в одно и то же время выполняют разные команды. Если процессы выполняются на одном ядре, то процессор должен переключаться между процессами – временно приостанавливать выполнение одного и возобновлять выполнение другого.

Программа может также запускать потоки, а не полноценные процессы. Поток во многом напоминает процесс, но разделяет память с процессом, которому принадлежит. Процесс может создать много потоков, и у каждого будет свой указатель команд. Все сказанное о выполнении процессов относится и к потокам. Но, поскольку потоки разделяют память, им проще сообща использовать код и другие объекты. Поэтому для организации конкурентности потоки полезнее процессов, правда, ценой усложнения программирования, о чем мы будем говорить ниже в этой книге.

Типичный способ воспользоваться преимуществами потоков, например взять на себя математические операции, загружающие процессор, – создать дополнительный поток или пул потоков, освободив главный поток для взаимодействия с пользователями или другими программами и поручив ему

просто проверять в бесконечном цикле наличие запросов на новое взаимодействие. Многие классические веб-серверы, в частности Apache, именно так обрабатывают большое количество HTTP-запросов. В результате схема работы программы выглядит так, как на рис. 1.1. В этой модели данные HTTP-запроса передаются рабочему потоку для обработки, а когда ответ будет готов, он передается главному потоку, который возвращает его пользовательскому агенту.

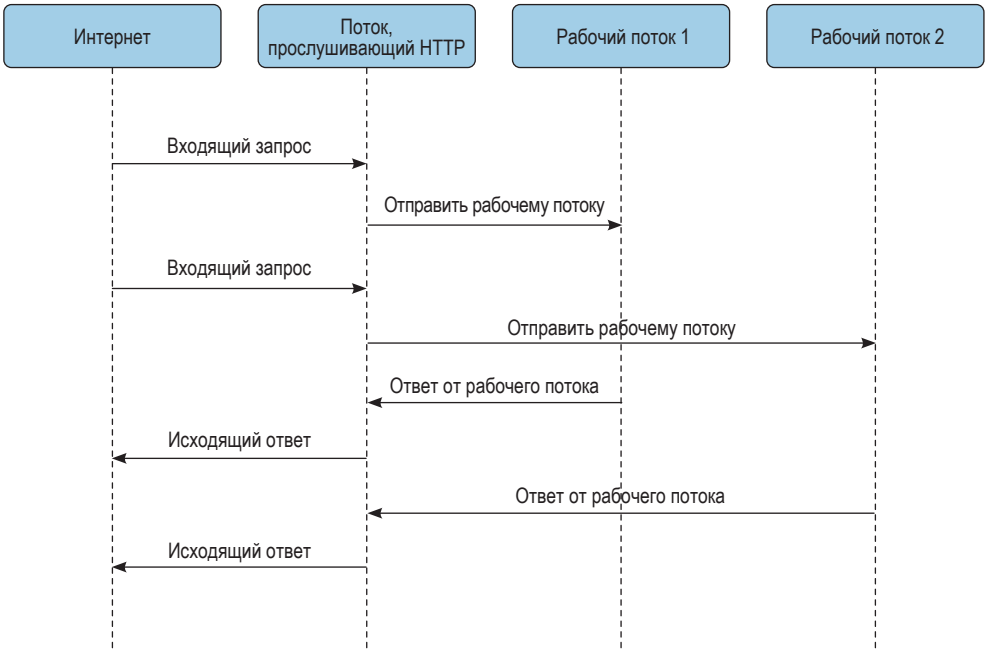

Рис. 1.1 ❖ Возможное использование рабочих потоков в HTTP-сервере

Чтобы от потоков была какая-то польза, они должны взаимодействовать. Это означает, например, что они должны уметь ждать возникновения событий в других потоках и получать от них данные. Как уже было сказано, потоки пользуются общей памятью, а с помощью дополнительных примитивов можно построить систему передачи сообщений между потоками. Часто такого рода конструкции доступны на уровне языка или платформы.

Конкурентность и параллелизм

Важно различать конкурентность и параллелизм, потому что они довольно часто возникают при многопоточном программировании. Эти термины тесно связаны, но в зависимости от обстоятельств могут означать похожие, но не вполне совпадающие вещи. Начнем с определений.

Конкурентность
Задачи перекрываются во времени.

Параллелизм
Задачи выполняются строго одновременно.

На первый взгляд кажется, что это одно и то же, но примите во внимание, что задачи могут быть разбиты на более мелкие части, которые чередуются во времени. В таком случае конкурентности можно добиться и без параллелизма, потому что интервалы времени, в течение которых задачи работают, могут перекрываться. Чтобы можно было говорить о параллельном выполнении задач, необходимо, чтобы они работали *строго одновременно*. В общем случае это означает, что они должны выполняться на разных процессорных ядрах в одно и то же время.

Рассмотрим рис. 1.2. Мы видим две задачи, работающие конкурентно и параллельно. В первом случае в каждый момент времени выполняется только одна задача, но на протяжении всего периода выполнение несколько раз переключается между задачами. Это значит, что интервалы работы двух задач перекрываются, что согласуется с определением конкурентности. Во втором случае обе задачи работают одновременно, т. е. параллельно. Поскольку интервалы их работы перекрываются, то они работают еще и конкурентно. Таким образом, из параллелизма следует конкурентность.

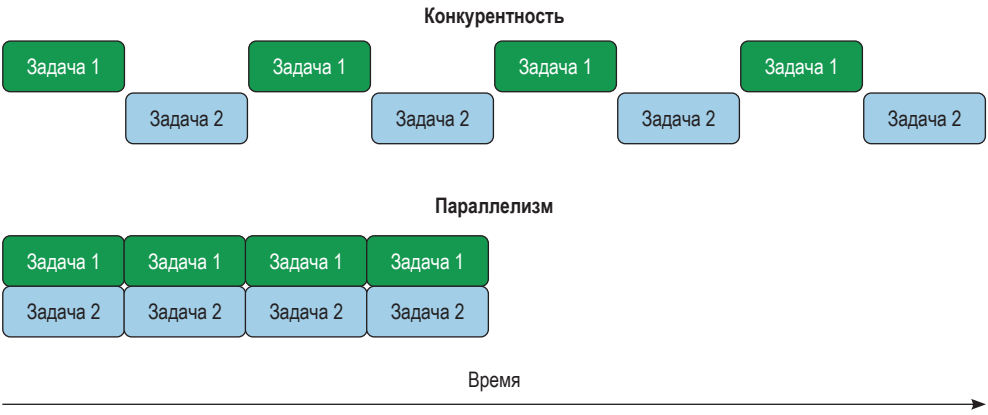

Рис. 1.2 ❖ Конкурентность и параллелизм

Потоки не обеспечивают параллелизм автоматически. Для этого необходимо, чтобы в системе было несколько процессорных ядер, а планировщик операционной системы решил запускать потоки на разных ядрах. Если в системе всего одно ядро или ядер меньше, чем потоков, то несколько потоков будут вынуждены работать на одном ядре конкурентно, а система будет переключать процессор между ними. Кроме того, в языках с блокировкой GIL, каковыми являются, в частности, Ruby и Python, потокам явно запрещено предоставлять параллелизм, потому что в среде может выполняться только одна команда в каждый момент времени.

Важно также принимать во внимание хронометраж, потому что обычно потоки включаются в программу ради повышения производительности. Если система допускает только конкурентность, потому что оснащена одним процессорным ядром или все ядра уже загружены другими задачами, то увеличение числа потоков может не принести желаемой выгоды. Более того, из-за накладных расходов на синхронизацию и контекстное переключение между потоками может даже оказаться, что программа стала работать медленнее. Всегда измеряйте производительность своего приложения в тех условиях, в которых оно предположительно будет работать. Только так вы сможете убедиться, дает ли многопоточная модель программирования какие-нибудь преимущества.

Однопоточный JavaScript

Исторически платформы, на которых работал JavaScript, вообще не поддерживали потоков, поэтому язык задумывался как однопоточный. Когда вы слышите от кого-то, что JavaScript однопоточный, говорящий имеет в виду именно это историческое наследие и стиль программирования, естественно тяготеющий к нему. Надо признать, что, вопреки названию книги, сам язык не содержит никаких встроенных средств для создания потоков. Это, впрочем, не должно вызывать удивления, потому что встроенных средств нет также для работы с сетью, устройствами, файловой системой или для выполнения системных вызовов. Даже такая важная функция, как `setTimeout()`, на самом деле не является принадлежностью JavaScript. Все это – посредством специальных API – предоставляет среда, в которую погружена виртуальная машина (ВМ), например Node.js или браузер.

Вместо использования потоков как примитива обеспечения конкурентности JavaScript-код чаще всего пишется в объектно-ориентированном стиле и исполняется в одном потоке. Возникающие события, например взаимодействие с пользователем или ввод–вывод, активируют выполнение функций, предварительно заданных как обработчики событий. Обычно эти функции называются *обратными вызовами*, и именно на них основано асинхронное программирование в Node.js и браузере. Даже при использовании обещаний или синтаксиса `async/await` в основе всего – глубоко или не очень глубоко внизу – лежат обратные вызовы. Важно понимать, что обратные вызовы не выполняются ни параллельно, ни наряду с каким-то другим кодом. Когда работает код обратного вызова, никакой другой код не работает. По-другому можно сказать, что в каждый момент времени активен только один стек вызовов.

Часто возникает ложное представление, будто операции выполняются параллельно, тогда как на самом деле они работают конкурентно. Например, допустим, что мы хотим открыть три файла, содержащих числа, *1.txt*, *2.txt* и *3.txt*, а затем сложить эти числа и напечатать результат. В Node.js можно написать код, показанный в примере 1.3.

Пример 1.3 ❖ Конкурентное чтение из файлов в Node.js

```
import fs from 'fs/promises';

async function getNum(filename) {
  return parseInt(await fs.readFile(filename, 'utf8'), 10);
}

try {
  const numberPromises = [1, 2, 3].map(i => getNum(`${i}.txt`));
  const numbers = await Promise.all(numberPromises);
  console.log(numbers[0] + numbers[1] + numbers[2]);
} catch (err) {
  console.error('Что-то не так:');
  console.error(err);
}
```

Чтобы выполнить этот код, сохраните его в файле *reader.js*. Проверьте, что текстовые файлы названы *1.txt*, *2.txt* и *3.txt* и что они содержат целые числа. Затем выполните команду `node reader.js`.

Мы воспользовались функцией `Promise.all()`, а значит, будем ждать, пока все три файла будут прочитаны и разобраны. При некотором воображении можно даже увидеть сходство с функцией `pthread_join()` из примера программы на C ниже в этой главе. Однако из того, что обещания создаются вместе и что мы ждем их совместного исполнения, еще не следует, что реализующий их код работает одновременно; можно лишь утверждать, что их временные интервалы перекрываются. Но указатель команд по-прежнему один, и в каждый момент времени выполняется только одна команда.

В отсутствие потоков JavaScript работает только с одной средой. Это означает, что существует один экземпляр ВМ, один указатель команд и один сборщик мусора. Говоря об указателе команд, мы имеем в виду, что интерпретатор JavaScript выполняет только одну команду в каждый момент времени. Но это не значит, что мы ограничены единственным глобальным объектом. И в браузере, и в Node.js в нашем распоряжении имеются области (realm) (https://262.ecma-international.org/11.0/#sec-code-realms).

Области можно рассматривать как экземпляры среды JavaScript, предоставляемые JavaScript-коду. Это значит, что каждая область получает свой глобальный объект со всеми его свойствами, например встроенным классом `Date`, объектом `Math` и пр. В Node.js глобальный объект называется `global`, а в браузерах `window`, но в современных версиях того и другого к нему можно также обращаться по имени `globalThis`.

В браузерах у каждого фрейма на веб-странице имеется область для всего содержащегося во фрейме JavaScript-кода. Поскольку у каждого фрейма своя копия `Object` и прочих примитивов внутри него, деревья наследования тоже разделены, поэтому оператор `instanceof` может работать не так, как вы ожидаете, если применяется к объектам из разных областей. Это показано в примере 1.4.

Пример 1.4 ❖ Объекты из внутреннего фрейма в браузере

```
const iframe = document.createElement('iframe');
document.body.appendChild(iframe);\
const FrameObject = iframe.contentWindow.Object; ❶

console.log(Object === FrameObject); ❷
console.log(new Object() instanceof FrameObject); ❸
console.log(FrameObject.name); ❹
```

❶ Глобальный объект внутри `iframe` доступен через свойство `contentWindow`.
❷ Здесь возвращается false, потому что `Object` внутри фрейма не совпадает с объектом из главного фрейма.
❸ `instanceof` возвращает `false`, как и следовало ожидать, поскольку это не один и тот же `Object`.
❹ Несмотря ни на что, конструкторы имеют одно и то же свойство `name`.

В Node.js области создаются функцией `vm.createContext()`, как показано в примере 1.5. В терминологии Node.js области называются контекстами (`Context`). Все правила и свойства, применимые к фреймам браузера, относятся и к контекстам, но в контексте отсутствует доступ к глобальным свойствам или еще чему-то, что может оказаться в области видимости файла Node.js. Если вам это нужно, то нужно вручную передать в контекст.

Пример 1.5 ❖ Объекты в новом контексте в Node.js

```
const vm = require('vm');
const ContextObject = vm.runInNewContext('Object'); ❶

console.log(Object === ContextObject); ❷
console.log(new Object() instanceof ContextObject); ❸
console.log(ContextObject.name); ❹
```

❶ Получить объекты из нового контекста можно с помощью функции `runInNewContext`.
❷ Возвращается false, потому что, как и во фреймах браузера, `Object` внутри контексте – не то же самое, что в главном контексте.
❸ Аналогично `instanceof` возвращает false.
❹ Как и раньше, у конструкторов имеется то же самое свойство `name`.

В любом случае важно помнить, что области пользуются одним и тем же указателем команд и что в каждый момент времени выполняется код только из одной области. Поэтому мы по-прежнему говорим об однопоточном выполнении.

Скрытые потоки

Хотя ваш JavaScript-код, может быть, и работает – по крайней мере по умолчанию – в однопоточной среде, это еще не значит, что процесс, исполняющий ваш код, однопоточный. На самом деле в нем может быть много потоков, обеспечивающих эффективное и ничем не омрачаемое выполнение вашего кода. Утверждение о том, что Node.js – однопоточный процесс, является распространенным заблуждением.

В современных движках JavaScript, например V8, отдельные потоки используются для сборки мусора и других операций, которые необязательно должны синхронизироваться с выполнением JavaScript-кода. Кроме того, сами платформенные среды выполнения могут использовать дополнительные потоки для представления каких-то служб.

В Node.js библиотека `libuv` предоставляет независимый от ОС интерфейс асинхронного ввода–вывода, а поскольку не все системные интерфейсы ввода–вывода асинхронны, он пользуется пулом рабочих потоков, чтобы избежать блокировки программного кода, когда тот обращается к блокирующим API, например API файловой системы. По умолчанию запускается четыре таких потока, но это число можно изменить с помощью переменной среды `UV_THREADPOOL_SIZE`; максимальное число потоков равно 1024.

В системах Linux дополнительные потоки можно увидеть, выполнив команду `top -H` для данного процесса. В примере 1.6 запущен простой веб-сервер Node.js, а затем его PID передан команде `top`. Видно, что имеется семь различных потоков V8 и `libuv`, включая тот, в котором работает JavaScript-код. Можете попробовать то же самое с собственными программами для Node.js и даже попытаться изменить переменную среды `UV_THREADPOOL_SIZE` и посмотреть, изменится ли число потоков.

Пример 1.6 ❖ Распечатка `top`, из которой видны потоки в процессе Node.js

```
$ top -H -p 81862
top - 14:18:49 up 1 day, 23:18, 1 user, load average: 0.59, 0.82, 0.83
Threads:  7 total,  0 running, 7  sleeping,  0 stopped,  0 zombie
%Cpu(s): 2.2 us, 0.0 sy, 0.0 ni, 97.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 15455.1 total, 2727.9 free, 5520.4 used, 7206.8 buff/cache
MiB Swap:  2048.0 total, 2048.0 free,    0.0 used.     8717.3 avail Mem

  PID  USER    PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+ COMMAND
81862 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.03 node
81863 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81864 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81865 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81866 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81867 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
81868 bengl    20   0  577084  29272  25064 S   0.0   0.2  0:00.00 node
```

Браузеры тоже выполняют многие задачи, например отрисовку объектной модели документа (DOM), в потоках, отличных от того, где работает JavaScript-код. Эксперимент с `top -H` покажет такую же горстку потоков, как в случае Node.js. Современные браузеры даже идут дальше, используя несколько процессов, чтобы благодаря изоляции повысить уровень безопасности.

Важно помнить об этих дополнительных потоках, планируя ресурсы для своего приложения. Никогда не следует предполагать, что раз JavaScript – однопоточный язык, то и в JavaScript-приложении будет использоваться только один поток. Например, в производственных приложениях Node.js измеряйте количество запущенных потоков и планируйте соответственно. Не забывайте, что многие дополнительные модули в экосистеме Node.js запускают еще и собственные потоки, так что это упражнение следует выполнять для каждого приложения.

Потоки на C: обогатитесь с помощью криптовалюты Happycoin

Понятно, что потоки не уникальная особенность JavaScript. Они уже давно укоренились на уровне операционной системы и не зависят от языка. Посмотрим, как могла бы выглядеть многопоточная программа, написанная на C. Язык C – очевидный выбор, потому что именно написанный на C интерфейс лежит в основе большинстве реализаций потоков в языках высокого уровня, даже если их семантика выглядит иначе. Рассмотрим следующий алгоритм доказательства выполнения работы для простой и практически бесполезной криптовалюты Happycoin.

1. Сгенерировать случайное 64-разрядное целое число без знака.
2. Определить, является ли это число счастливым.
3. Если нет, это не Happycoin.
4. Если оно не делится на 10 000, это не Happycoin.
5. В противном случае это Happycoin.

Число называется счастливым, если в цикле замены числа суммой квадратов его цифр встретилась либо 1, либо ранее возникавшее число. В «Википедии» (https://en.wikipedia.org/wiki/Happy_number) это понятие определено точно, а также отмечено, что число возникает повторно тогда и только тогда, когда встречается 4. Вы, наверное, обратили внимание, что наш алгоритм неэффективен, потому что проверять делимость на 10 000 можно было бы до проверки того, является ли число счастливым. Это сделано намеренно, потому что мы хотим продемонстрировать высокую нагрузку.

Напишем простую программу на C, которая прогоняет алгоритм доказательства выполнения работы 10 000 000 раз, печатая все найденные Happycoin'ы и их количество.

Команду `cc` в шагах компиляции можно заменить на `gcc` или `clang`, в зависимости от того, что есть в вашей системе. В большинстве систем `cc` – псевдоним `gcc` или `clang`, поэтому мы остановимся на этом имени.

Пользователям Windows придется еще немного потрудиться, чтобы этот пример заработал в Visual Studio, потому что в нем используются POSIX-потоки (в соответствии со стандартом Portable Operating System Interface), а не потоки Windows, которые имеют с ними мало общего. Чтобы упростить выполнение примера в Windows, рекомендуем воспользоваться подсистемой Windows для Linux, тогда вы получите совместимую с POSIX среду.

С одним главным потоком

Создайте файл *happycoin.c* в каталоге *ch1-c-threads/*. Мы будем постепенно заполнять его кодом на протяжении этого раздела. Для начала добавьте код, показанный в примере 1.7.

Пример 1.7 ❖ *ch1-c-threads/happycoin.c*

```
#include <inttypes.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

uint64_t random64(uint32_t * seed) {
  uint64_t result;
  uint8_t * result8 = (uint8_t *)&result; ❶
  for (size_t i = 0; i < sizeof(result); i++) {
    result8[i] = rand_r(seed);
  }
  return result;
}
```

❶ **В этой строке используются указатели, которые, возможно, вам незнакомы, если опыт вашей работы ограничивается в основном JavaScript. В двух словах здесь происходит следующее: `result8` – это массив восьми 8-разрядных целых без знака, хранящийся в памяти, отведенной под 64-разрядное целое без знака `result`.**

Мы добавили несколько директив `include`, которые дают доступ к таким удобным вещам, как типы, функции ввода–вывода, а также для работы со временем и случайными числами. Поскольку алгоритм требует генерирования случайных 64-разрядных целых без знака (типа `uint64_t`), нам нужно восемь случайных байтов, которые `random64()` получает от вызова функции `rand_r()` в цикле. Так как `rand_r()` требует ссылки на начальное значение (seed), мы передаем его функции `random64()`.

Теперь добавим вычисление счастливого числа.

Пример 1.8 ❖ *ch1-c-threads/happycoin.c*

```
uint64_t sum_digits_squared(uint64_t num) {
  uint64_t total = 0;
  while (num > 0) {
    uint64_t num_mod_base = num % 10;
    total += num_mod_base * num_mod_base;
    num = num / 10;
  }
  return total;
}

bool is_happy(uint64_t num) {
  while (num != 1 && num != 4) {
    num = sum_digits_squared(num);
  }
  return num == 1;
}

bool is_happycoin(uint64_t num) {
  return is_happy(num) && num % 10000 == 0;
}
```

Для нахождения суммы квадратов цифр мы в функции `sum_digits_squared` пользуемся оператором взятия остатка `%`, чтобы выделить одну цифру (слева направо), а затем прибавить ее квадрат к сумме. Затем в цикле вызывается функция `is_happy`, до тех пор пока не получим 1 или 4. 1 означает, что число счастливое, а 4 – что цикл бесконечный и мы никогда не встретим 1. Наконец, в функции `is_happycoin()` мы проверяем, является ли число счастливым и делится ли оно на 10 000.

Обернем все это функцией `main()`, как показано в примере 1.9.

Пример 1.9 ❖ *ch1-c-threads/happycoin.c*

```
int main() {
  uint32_t seed = time(NULL);
  int count = 0;
  for (int i = 1; i < 10000000; i++) {
    uint64_t random_num = random64(&seed);
    if (is_happycoin(random_num)) {
      printf("%" PRIu64 " ", random_num);
      count++;
    }
  }
  printf("\ncount %d\n", count);
  return 0;
}
```

Первым делом нужно инициализировать генератор случайных чисел. В качестве начального значения вполне подойдет текущее время, поэтому получим его с помощью функции `time()`. Затем выполним 10 000 000 итераций цикла, на каждой из которых сначала получаем случайное число от `random64()`, а затем проверяем, является ли оно Happycoin'ом. Если да, то увеличиваем счетчик и печатаем число. Странный синтаксис `PRIu64` в вызове `printf()` необходим для правильного форматирования 64-разрядных целых без знака. По завершении цикла печатаем счетчик и завершаем программу.

Для компиляции и запуска программы наберите следующие команды, находясь в каталоге *ch1-c-threads*.

```
$ cc -o happycoin happycoin.c
$ ./happycoin
```

В одной строке будет напечатан список найденных Happycoin'ов, а в другой – их количество. Выглядеть это может так:

```
11023541197304510000 ... [ еще 167 чисел ] ... 770541398378840000
count 169
```

Выполнение программы занимает нетривиальное время – примерно 2 с на стандартном компьютере. Это тот случай, когда потоки могут ускорить работу, потому что многократно повторяется однотипная математическая операция.

Давайте превратим эту программу в многопоточную.

С четырьмя рабочими потоками

Заведем четыре потока, каждый из которых будет выполнять четверть итераций цикла, в котором генерируется случайное число и проверяется, является ли оно Happycoin'ом.

В POSIX C для управления потоками предназначены функции из семейства `pthread_*`. Функция `pthread_create()` создает поток. Ей передается функция, которая будет исполняться в этом потоке. Затем продолжается выполнение главного потока. Программа может дождаться завершения потока, вызвав для него функцию `pthread_join()`. Функции, исполняемой в потоке, созданном `pthread_create()`, можно передать аргументы, а возвращенное ей значение получить от `pthread_join()`.

В нашей программе мы инкапсулируем генерирование Happycoin'ов в функции `get_happycoins()`, именно ее будем запускать в потоках. Мы создадим четыре потока и сразу же начнем ждать их завершения. Получив от потока результаты, сохраним их, чтобы в конце можно было напечатать общий итог. Для передачи результатов заведем простую структуру `happy_result`.

Скопируйте существующий файл *happycoin.c* и назовите его *happycoin-threads.c*. В новый файл добавьте код из примера 1.10, разместив его под последней директивой `#include`.

Пример 1.10 ❖ *ch1-c-threads/happycoin-threads.c*

```
#include <pthread.h>

struct happy_result {
  size_t count;
  uint64_t * nums;
};
```

В первой строке включается файл `pthread.h`, который дает доступ к нужным нам функциям для работы с потоками. Затем определяется структура `struct happy_result`, в которой будут возвращаться значения из работающей в потоке функции `get_happycoins()`: массив найденных Happycoin'ов, представленный указателем, и их количество.

Теперь удалите функцию `main()` целиком, потому что мы собираемся заменить ее. Сначала добавьте функцию `get_happycoins()` из примера 11.1 – это тот код, который будет выполняться в рабочих потоках.

Пример 1.11 ❖ *ch1-c-threads/happycoin-threads.c*

```
void * get_happycoins(void * arg) {
  int attempts = *(int *)arg; ❶
  int limit = attempts/10000;
  uint32_t seed = time(NULL);
  uint64_t * nums = malloc(limit * sizeof(uint64_t));
  struct happy_result * result = malloc(sizeof(struct happy_result));
  result->nums = nums;
  result->count = 0;
  for (int i = 1; i < attempts; i++) {
    if (result->count == limit) {
```

```
      break;
    }
    uint64_t random_num = random64(&seed);
    if (is_happycoin(random_num)) {
      result->nums[result->count++] = random_num;
    }
  }
  return (void *)result;
}
```

❶ **Эта странная конструкция – приведение типа указателя – означает «обращайся с этим указателем как с указателем на `int` и дай мне значение этого `int`».**

Обратите внимание, что эта функция принимает один аргумент типа `void *` и возвращает одно значение типа `void *`. Функцию с такой сигнатурой ожидает получить `pthread_create()`, так что у нас просто нет выбора. Это означает, что мы должны привести аргументы к ожидаемым типам. Мы хотим передать количество попыток, поэтому приводим аргумент к типу `int`. Затем инициализируем начальное значение, как в предыдущем примере, но теперь это делается в потоковой функции, так что каждый поток получает разные начальные значения.

Выделив достаточно памяти для массива и структуры `happy_result`, мы входим в такой же цикл, что в функции `main()` в однопоточной версии, но на этот раз помещаем результаты в структуру, а не печатаем их. По завершении цикла мы возвращаем указатель на структуру, предварительно приведя его к типу `void *` в соответствии с сигнатурой функции. Именно так информации передается главному потоку, который будет ее интерпретировать.

Мы продемонстрировали одно из важнейших свойств потоков, отличающих их от процессов, – использование общей памяти. Если бы мы использовали не потоки, а процессы и какой-то механизм *межпроцессного взаимодействия* (interprocess communication – IPC) для обратной передачи результатов, то не смогли бы просто передать адрес памяти главному процессу, потому что главный процесс не имеет доступа к памяти рабочего. Из-за виртуализации памяти этот адрес мог бы указывать в какое-то произвольное место главного процесса. Поэтому вместо указателя мы должны были бы передать главному процессу все значение целиком по каналу IPC, что повлекло бы за собой накладные расходы. Но, коль скоро мы используем потоки, а не процессы, можно передать просто указатель, потому что в главном потоке он будет указывать туда же, куда и в рабочем.

Но у разделяемой памяти есть свои недостатки. В нашем случае рабочий поток никак не использует память, после того как передал ее главному. Но так бывает не всегда. Чаще всего приходится аккуратно управлять доступом потоков к разделяемой памяти с помощью механизмов синхронизации, иначе результаты могут быть непредсказуемы. Как это работает в JavaScript, мы подробно рассмотрим в главах 4 и 5.

Теперь обернем все это функцией `main()`, как показано в примере 1.12.

Пример 1.12 ❖ *ch1-c-threads/happycoin-threads.c*

```
#define THREAD_COUNT 4

int main() {
```

```
  pthread_t thread [THREAD_COUNT];

  int attempts = 10000000/THREAD_COUNT;
  int count = 0;
  for (int i = 0; i < THREAD_COUNT; i++) {
    pthread_create(&thread[i], NULL, get_happycoins, &attempts);
  }
  for (int j = 0; j < THREAD_COUNT; j++) {
    struct happy_result * result;
    pthread_join(thread[j], (void **)&result);
    count += result->count;
    for (int k = 0; k < result->count; k++) {
      printf("%" PRIu64 " ", result->nums[k]);
    }
  }
  printf("\ncount %d\n", count);
  return 0;
}
```

Сначала мы объявляем все четыре потока в виде массива в стеке. Затем делим число итераций (10 000 000) на число потоков. Результаты мы будем передавать функции `get_happycoins()` в качестве аргумента, как показано в первом цикле, где `pthread_create()` создает потоки. В следующем цикле мы ждем завершения каждого потока с помощью `pthread_join()`. После этого можем напечатать результаты, собранные от всех потоков, и общее число Happycoin'ов – точно так же, как в однопоточной версии.

В этой программе есть утечка памяти. Одна из трудностей многопоточного программирования на C и некоторых других языках состоит в том, что очень легко запутаться, где и когда память выделяется и где и когда ее нужно освобождать. Попробуйте модифицировать код, так чтобы вся выделенная из кучи память гарантированно освобождалась перед выходом из программы.

Откомпилируйте и запустите новую версию программы, выполнив следующие команды из каталога *ch1-c-threads*.

```
$ cc -pthread -o happycoin-threads happycoin-threads.c
$ ./happycoin-threads
```

Результат будет выглядеть примерно так:

```
2466431682927540000 ... [ еще 154 числа ] ... 15764177621931310000
count 156
```

Вы, конечно, заметили, что результат похож на полученный ранее[1]. И наверное, обратили внимание, что программа работает немного быстрее. На стандартном компьютере она выполняется примерно 0.8 с. Это, безусловно, не в четыре раза быстрее из-за начальных накладных расходов в главном

[1] Счетчики в двух версиях различаются, но это несущественно, потому что счетчик зависит от того, сколько случайных чисел оказались Happycoin'ами. В любых двух прогонах счетчики будут различаться.

потоке, да и печать результатов тоже чего-то стоит. Можно было бы печатать результаты сразу после их получения в рабочем потоке, но тогда они могли бы наложиться друг на друга, потому что ничто не мешает двум потокам печатать в одно и то же место одновременно. Путем отправки результатов в главный поток мы координируем печать результатов, избегая мусора на экране.

Это иллюстрация главного преимущества и одного из недостатков многопоточного кода. С одной стороны, полезно разбивать счетные задачи на части, работающие параллельно. С другой стороны, необходимо гарантировать, что события надлежащим образом синхронизированы, иначе могут возникнуть загадочные ошибки. Добавляя потоки в программу на любом языке, нужно внимательно следить за их правильным использованием. И, как всегда при попытке ускорить программу, необходимо проводить измерения. Никому не нужна лишняя сложность, обусловленная многопоточностью, если это не приносит приложению никакой выгоды.

Любой язык программирования, поддерживающий потоки, предоставляет какие-то механизмы для создания и уничтожения потоков, передачи сообщений между ними и взаимодействия с данными, разделяемыми потоками. Эти механизмы могут выглядеть по-разному, потому что различаются языки и парадигмы, а равно и модели параллельного программирования. Познакомившись с тем, как выглядит многопоточная программа на языке низкого уровня C, перейдем к JavaScript. На поверхности здесь все устроено немного иначе, но, как мы увидим, принципы остаются теми же.