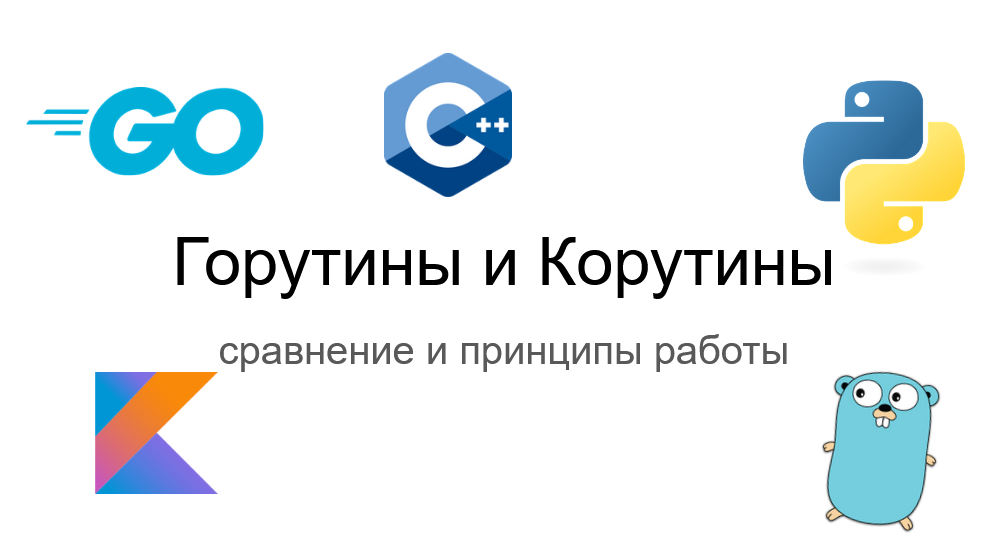
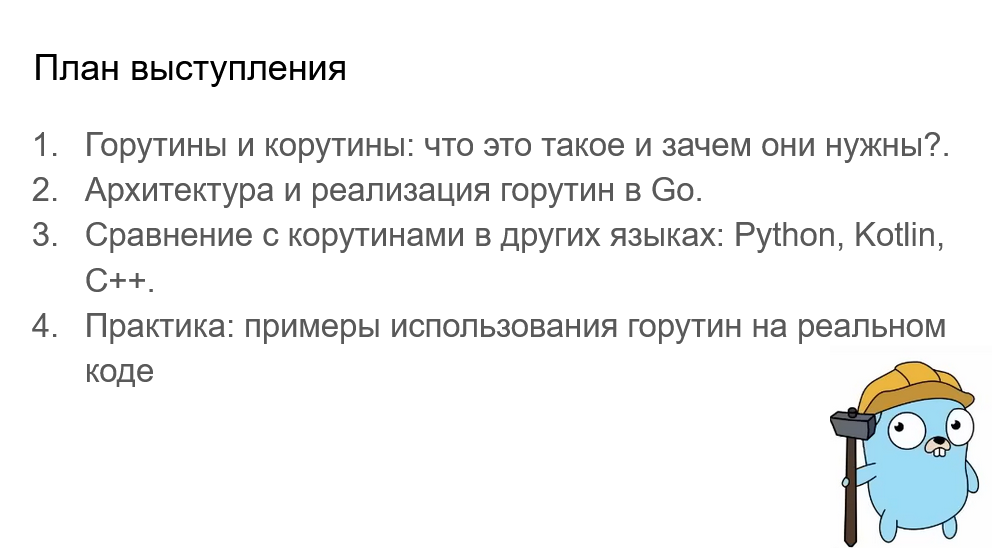
### **0. Вступление**



Всем привет меня зовут Роман! Сегодня я расскажу вам о том, что такое горутины и сравню их с корутинами



Вот подробный план выступления:

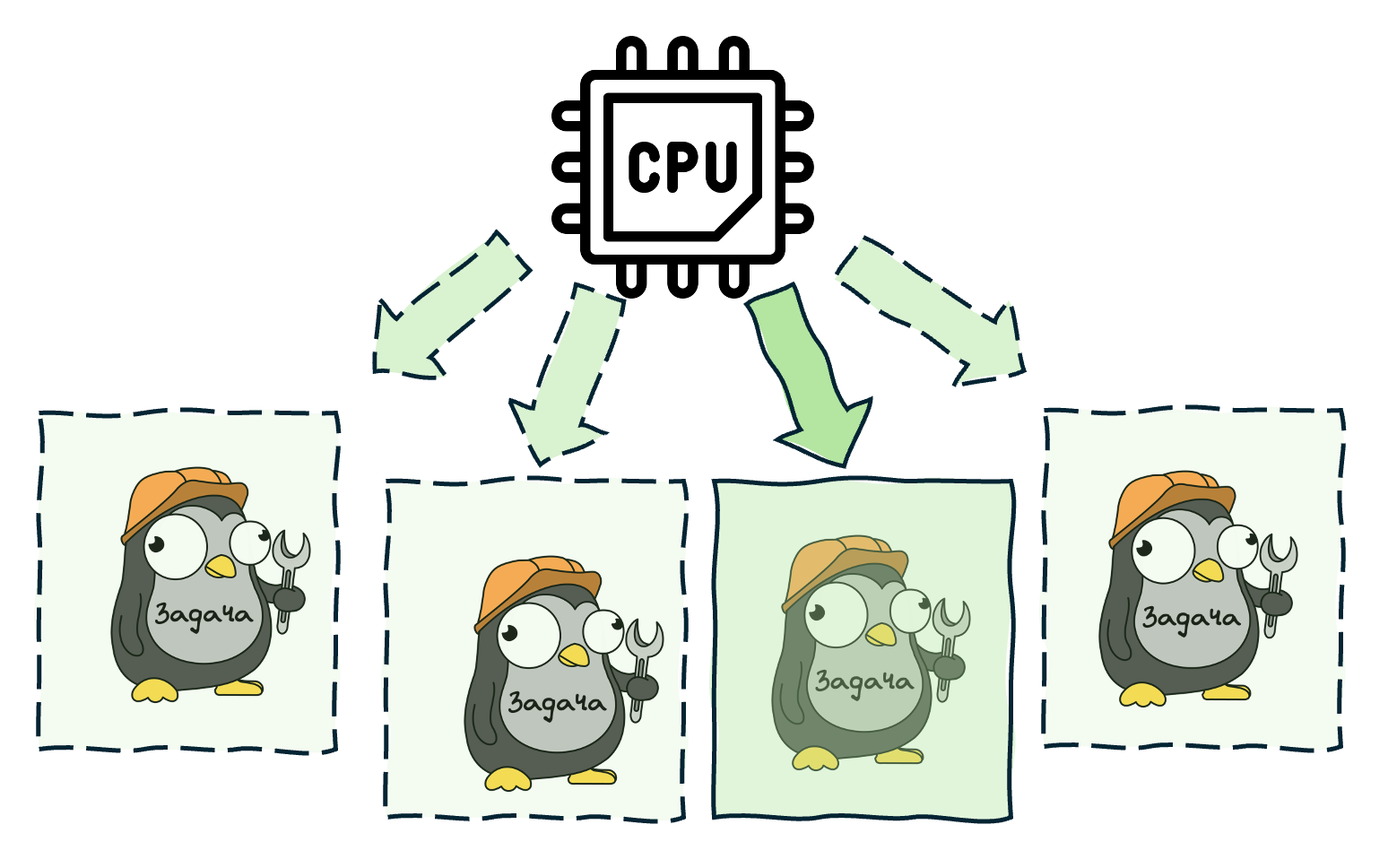
1. Классические потоки в ОС и их планировщик
2. Горутины и корутины: что это такое и зачем они нужны?.
3. Архитектура и реализация горутин в Go.
4. Сравнение с корутинами в других языках: Python, Kotlin, C++.
5. Области применения горутин.

### **1. Классические потоки в ОС и их планировщик**

Для глубинного понимания, **почему вообще появились корутины и горутины, и что это такое ?** Давайте сначала разберёмся, **что не так с классическими потоками (threads) и как они устроены.**

### **Как устроена параллельная работа компьютера**

Как вы наверняка понимаете, наш компьютер может выполнять несколько действий одновременно (параллельно) только в том случае, если у его процессора есть несколько ядер.

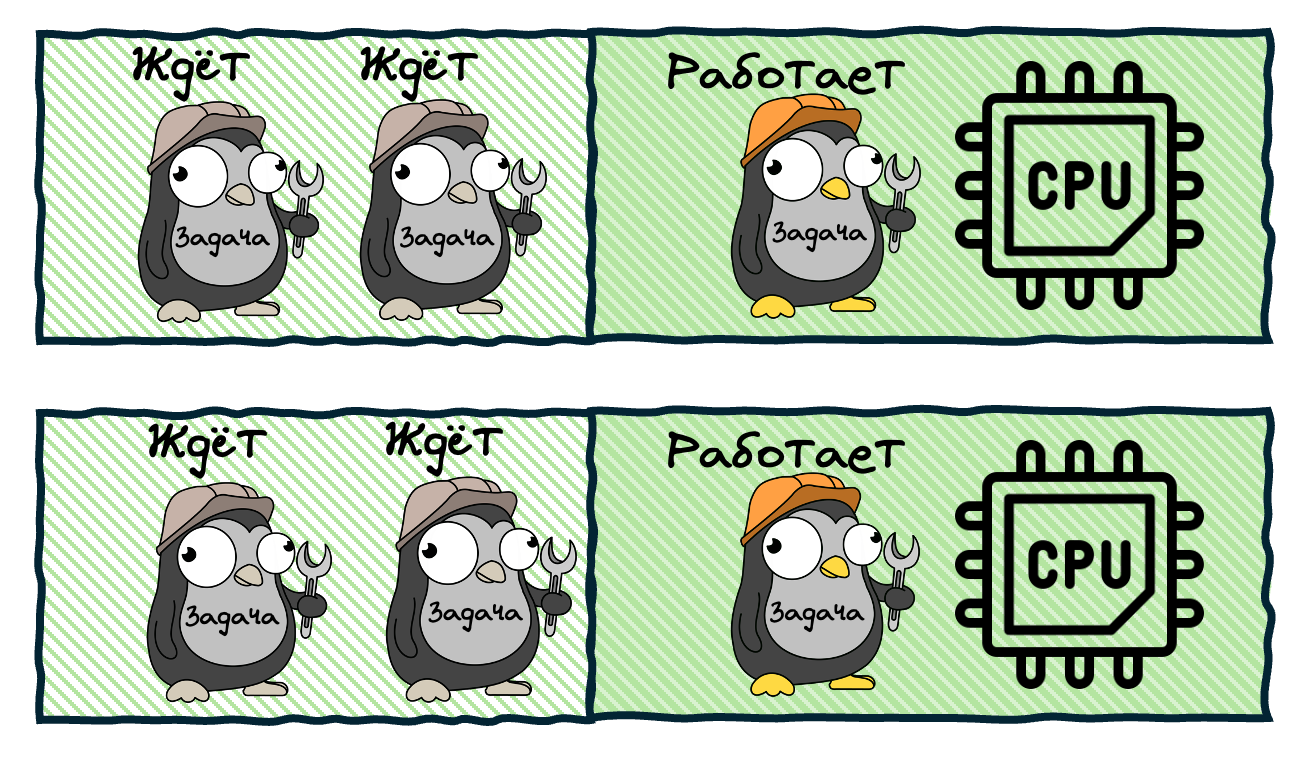


Но есть хорошие новости — процессор это очень быстрая штука, поэтому он даже с одним ядром может выполнять огромное количество действий почти одновременно, быстро переключаясь между ними. К примеру, даже на компьютере с одноядерным процессором вы можете "одновременно" смотреть мои ролики на YouTube, писать код, запускать его и выполнять ряд других задач. В каждый момент времени процессор будет выполнять лишь одну задачу, но он так быстро будет переключаться между ними, что вам будет казаться, будто сразу все.



А кто же занимается переключением задач и решает, какие из них будут выполняться на ядре, а какие подождут? Занимается этим планировщик операционной системы, и управляет он тредами.

Тред (поток) - это последовательность команд, которые выполняются в рамках одного процесса. Говоря простыми словами, это та работа, которая выполняется на процессоре, либо ждёт своей очереди.

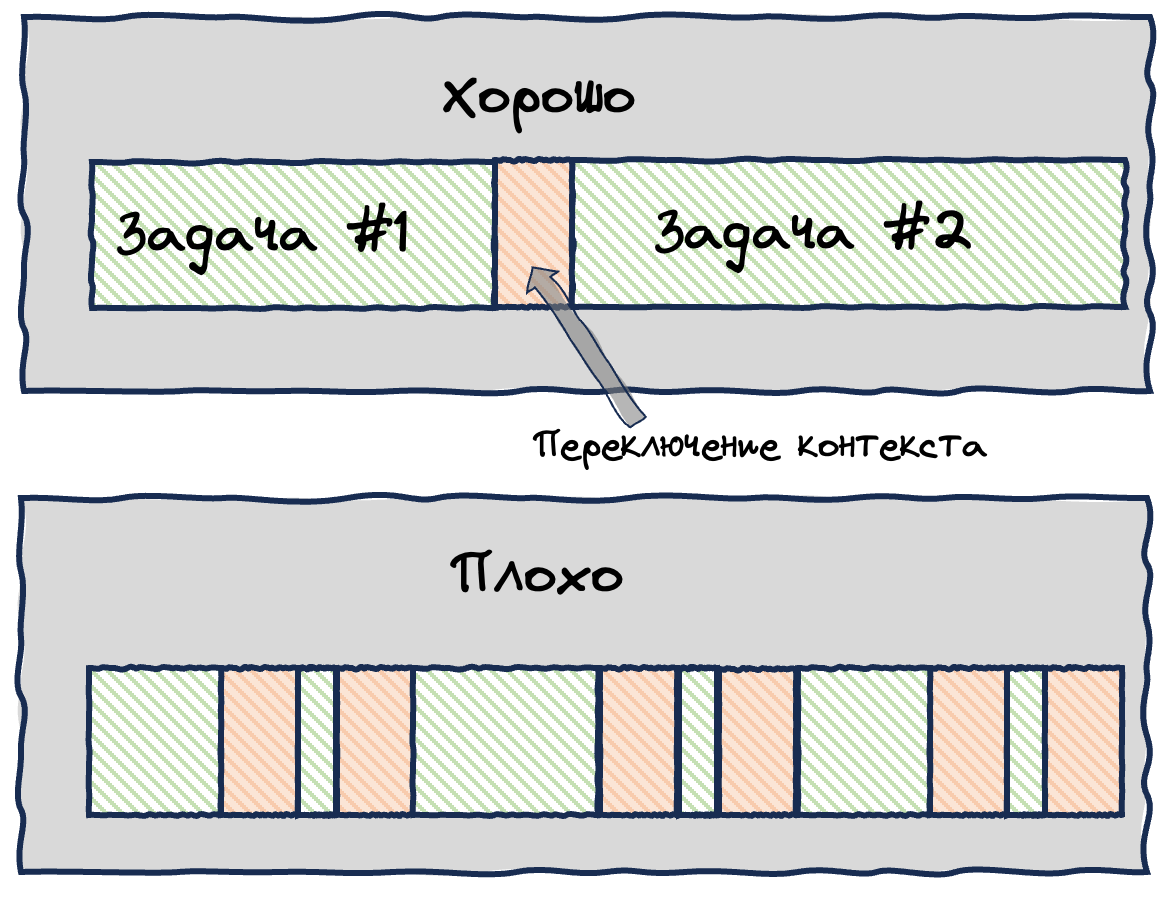


Каждый тред может находиться в одном из трёх состояний:

* **Executing**: выполняется прямо сейчас на одном из ядер
* **Runnable**: готов к выполнению, дожидается своей очереди
* **Waiting**: не готов к выполнению, так как ждёт какого-то события. Например, это может быть связано с i/o операцией, взаимодействием с ОС (syscall) и др.

Планировщик ОС может в произвольные моменты времени переключать треды — отключать выполняющиеся от ядра процессора и заменять их Runnable тредами. Это называется переключением контекста. Мы никак не можем повлиять на решения планировщика, как и не можем строить какие-то предсказания на этот счёт — все решения он принимает самостоятельно. Такой подход называется [вытесняющей многозадачностью](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%82%D0%B5%D1%81%D0%BD%D1%8F%D1%8E%D1%89%D0%B0%D1%8F_%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C)

*Самое главное, что здесь нужно понимать: треды, это дорого.* Давайте разберёмся почему.



Во-первых, переключение контекста, занимает довольно много времени: нужно обновить данные в кэшах процессора, сохранить состояние треда и др. Чем больше у нас тредов в состоянии Runnable, тем чаще будет переключаться контекст, и тем медленнее будет работать программа. К примеру, если переключений слишком много, то всем они суммарно могут занимать столько же времени, сколько само выполнение задач (а то и больше):

Во-вторых, каждый тред отъедает существенную часть памяти нашего компьютера: стек каждого треда часто может занимать до пары мегабайт (в нём хранятся локальные переменные, цепочки вызова функций и др.)

Теперь, зная что треды действительно дорогие, мы приходим к пониманию того, что хотим создавать их как можно меньше. Чем меньше тредов, тем реже переключается контекст, тем меньше памяти мы тратим.

Значит, нам нужна **лёгкая многозадачность** — без избыточной нагрузки и без всей той головной боли, что идёт в комплекте с потоками. Так и появились **корутины** и **горутины**.



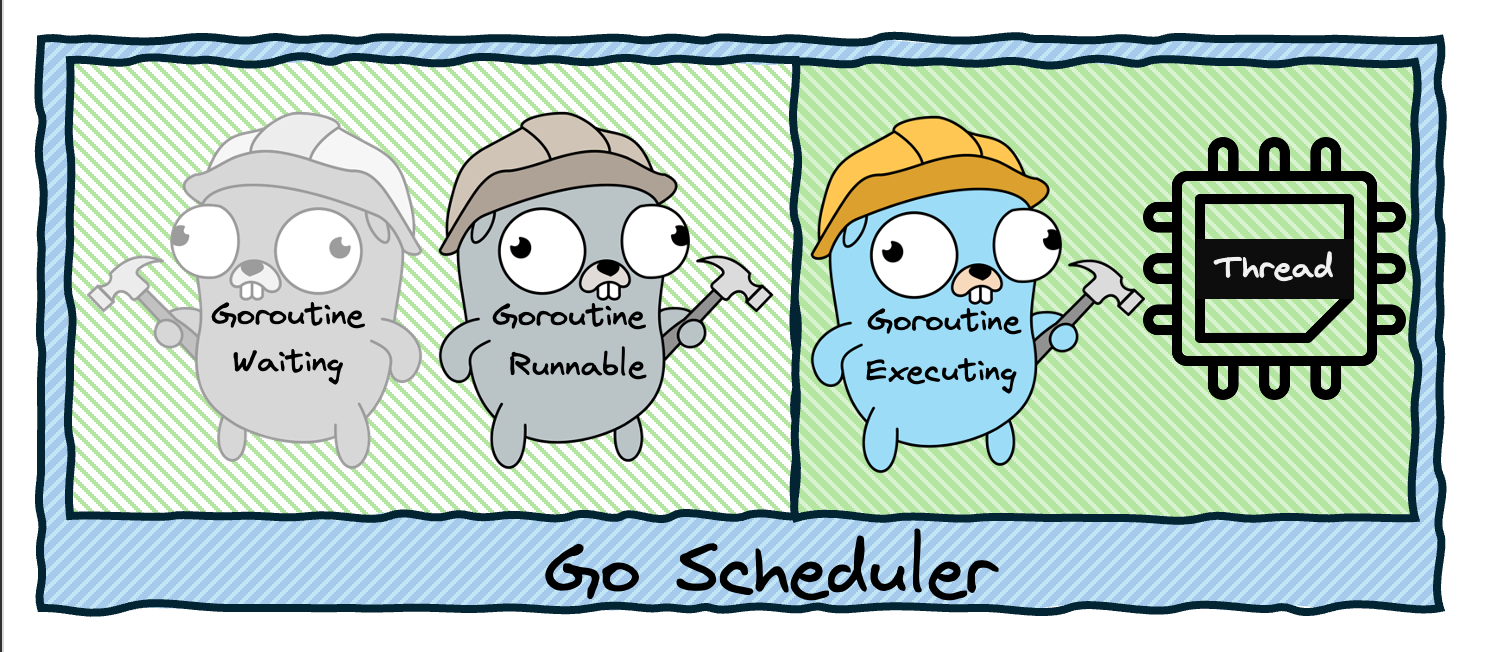
### **2.1. Горутины и корутины: что это такое и зачем они нужны?.**

#### **📜 Зарождение корутин:**

* **1958** — Мелвин Конвей вводит идею корутин: функции, которые могут приостанавливаться и возобновляться.
* **1960–70-е** — Simula, Modula-2: ранние реализации кооперативной многозадачности.
* **2000-е** — корутины становятся популярны в языках программирования: Python C#:
* **2007–2009** — Google работает над Go.
* **2009** — появление Go и **горутин**: ключевая фича языка, реализующая масштабируемую, но простую многозадачность.
* в Kotlin появляется поддержка корутин (suspend, structured concurrency).
* **2020** — C++20 включает корутины на уровне стандарта (co\_await, co\_yield).

### **3. Горутины**

### **3.1. Введение: Что такое горутина?**

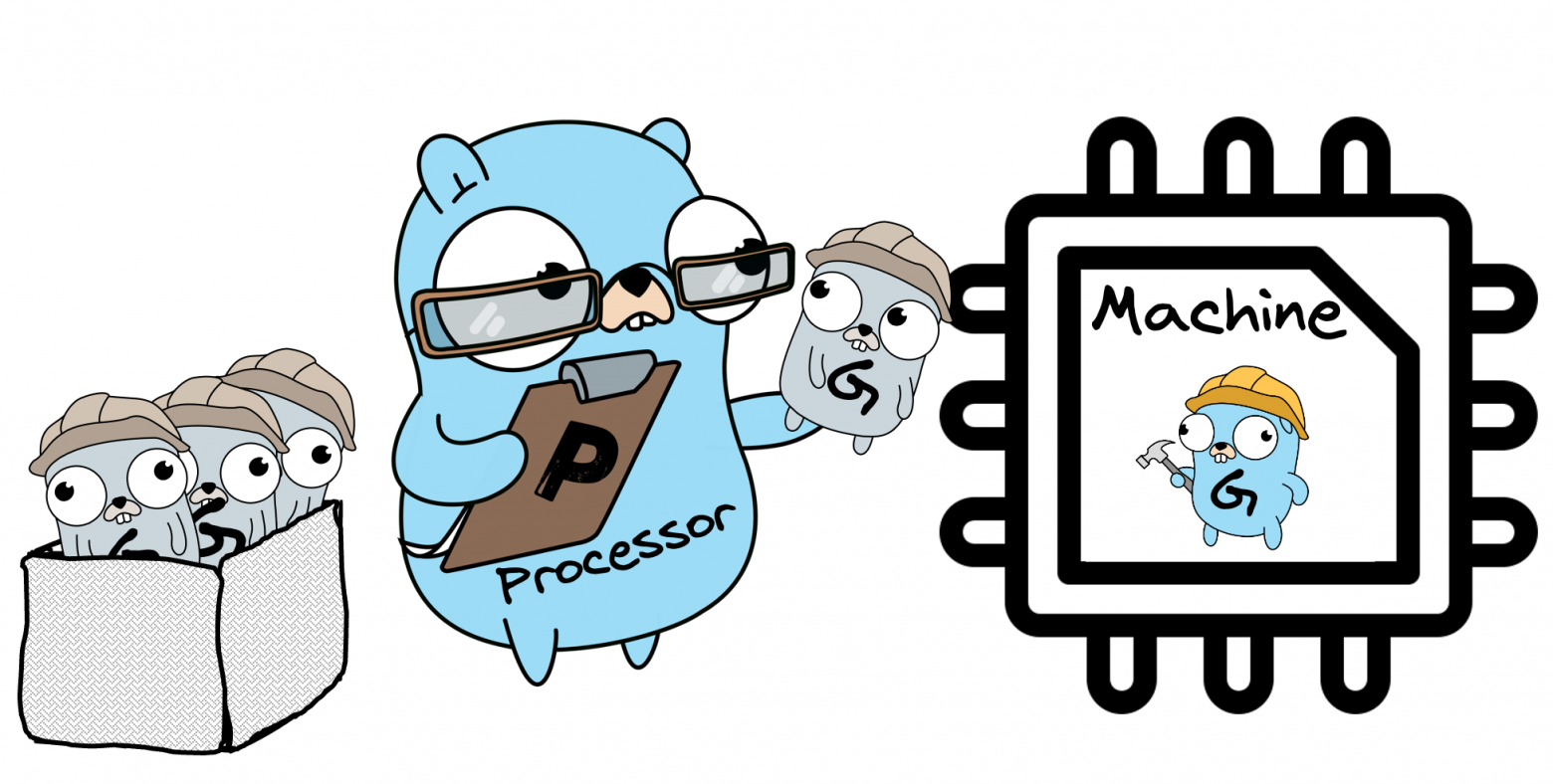
  
**Горутина** — это легковесный поток исполнения. Она запускается с помощью ключевого слова go, потребляет всего 2–4 КБ памяти на начальный стек и может масштабироваться до миллионов одновременных запусков.

Горутины выполняются на тредах и могут находиться в трёх состояниях:

* **Waiting** — горутина чего-то ждёт.
* **Runnable** — готова к запуску.
* **Executing** — исполняется на каком-то треде.

Важно: **горутины — не системные потоки**. Они исполняются через встроенный планировщик Go.

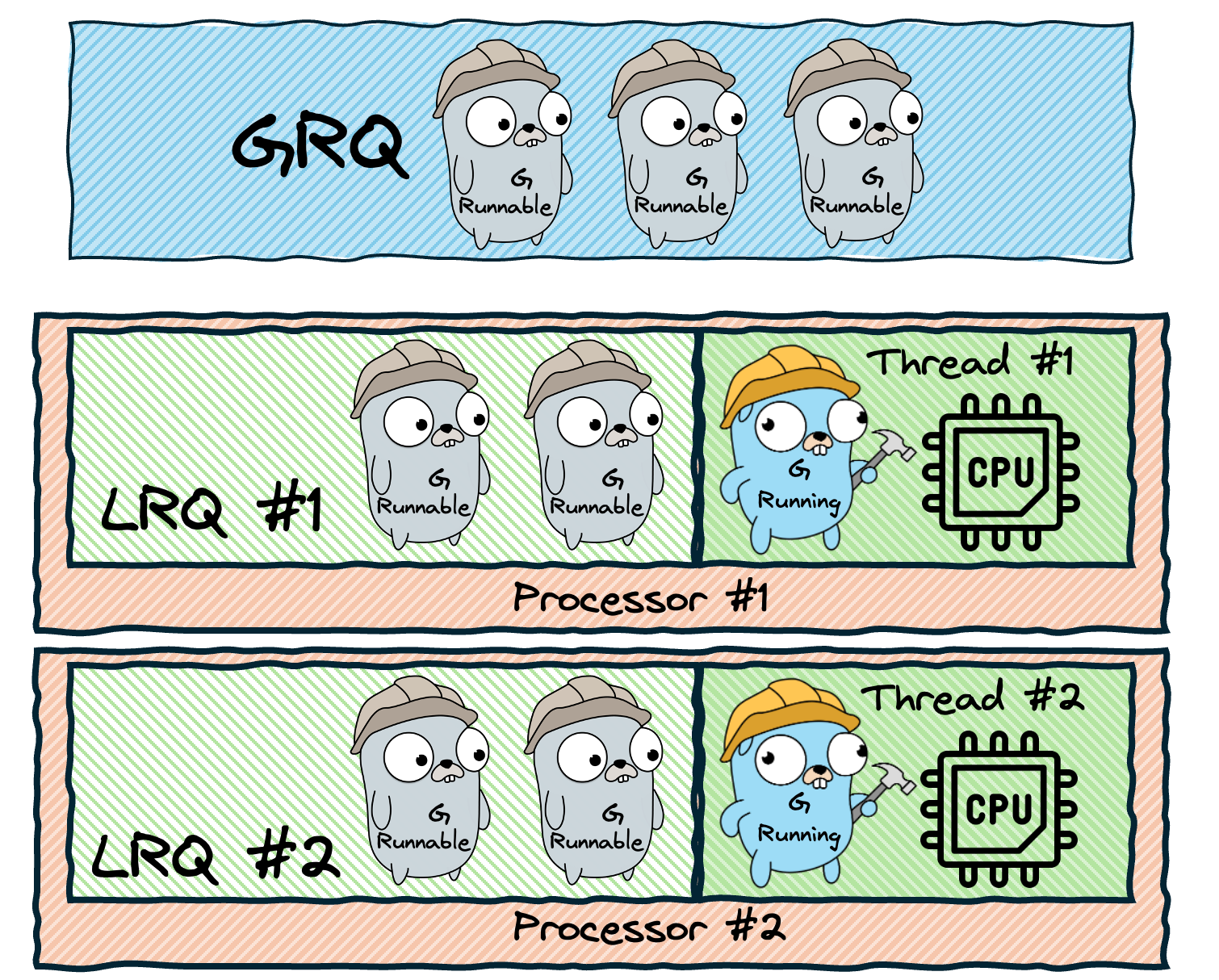
### **3.2. Планировщик Go и модель GMP**



Планировщик Go управляет горутинами так, чтобы эффективно использовать треды: создавать их как можно меньше и не позволять простаивать.  
Он реализован через модель **GMP**:

* **G (Goroutine)** — задача, которую нужно выполнить.
* **M (Machine)** — тред, исполняющий горутины.
* **P (Processor)** — объект, который управляет очередями горутин и назначает их M.

У каждого P есть своя локальная очередь, и помимо этого есть **глобальная очередь (Global Run Queue)**.  
 Она используется, если локальная очередь переполнена или пуста.

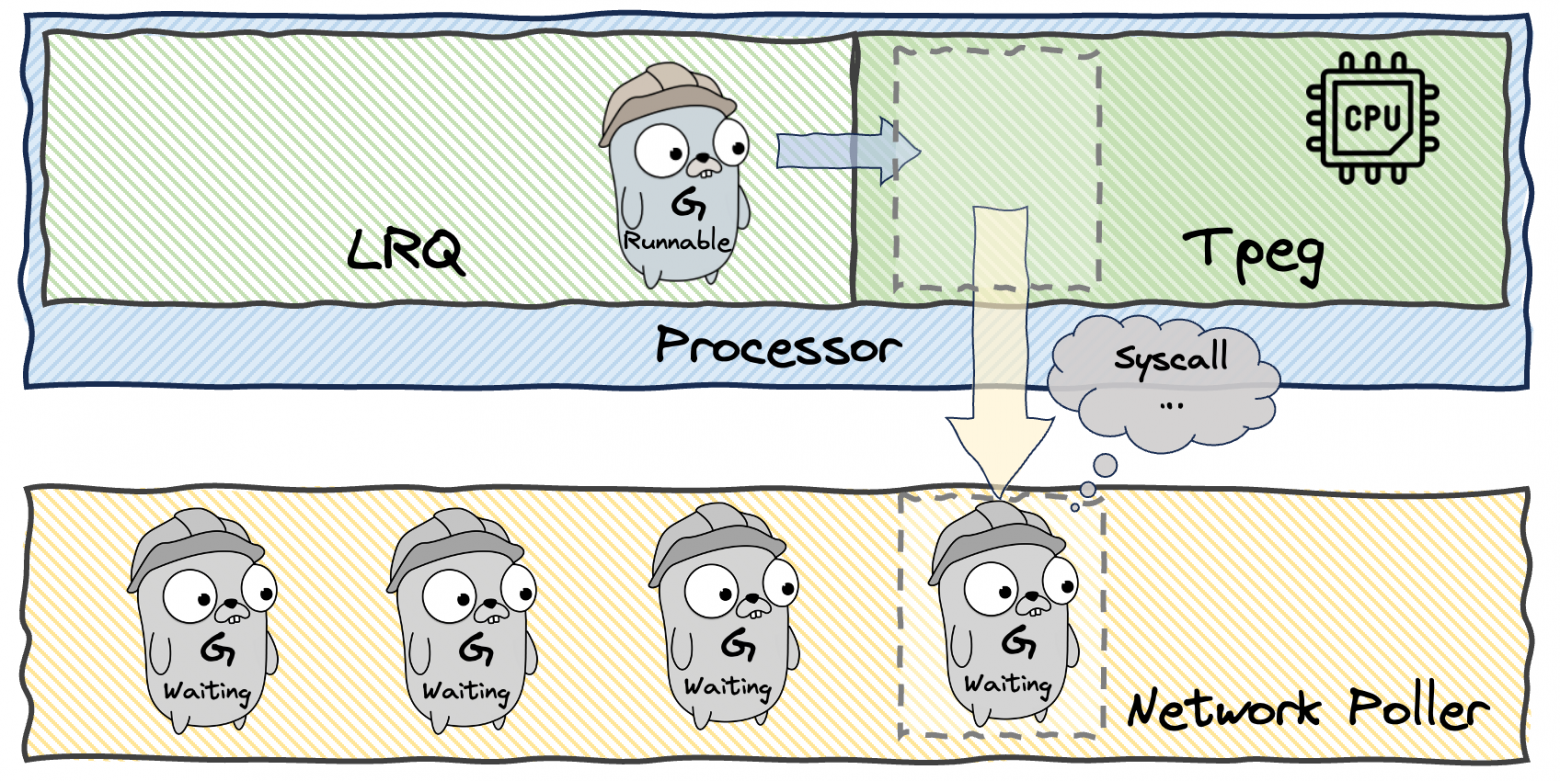


### **3.3. Как работают очереди: GRQ, LRQ и порядок проверки**

Вот как P выбирает следующую горутину:

1. **1 из 61 раза** проверяется **GRQ** (глобальная очередь).
2. Если там пусто — проверяется **LRQ** (локальная очередь).
3. Если и там пусто — P **пытается украсть** горутину у другого P.
4. Если не удалось — снова проверяется GRQ.
5. Если и она пуста — P **обращается к netpoller'у**, который обрабатывает завершенные I/O.

Почему именно 1/61? Это число подобрано экспериментально. Оно простое, уменьшает синхронизацию и сбалансировано по частоте.



### **4. Что делает netpoller?**

**Netpoller** — компонент рантайма Go, который:

* отслеживает асинхронные сетевые операции (например, read или write);
* позволяет **не блокировать потоки ОС**;
* работает поверх системных механизмов (epoll, kqueue, IOCP).

Как это выглядит:

1. Горутина вызывает net.Read() или net.Write().
2. Если возможно — операция регистрируется в системе (epoll и т.п.), горутина переводится в waiting.
3. Поток (M) и процессор (P) **освобождаются**.
4. Когда событие готово — ОС сообщает epoll → netpoller активирует горутину.

### **5. Sysmon и Handoff**

**Sysmon** — это фоновый процесс Go-рантайма.

Что он делает:

* Следит за тредами, заблокированными на системных вызовах.
* Если тред блокирован слишком долго (более **10мс**), он инициирует **handoff** — передачу P другому M.

Почему это важно? Потому что P содержит очередь задач. Если M блокирован, а P простаивает — теряется производительность.

### **6. Примеры системных вызовов**

Вот, какие бывают системные вызовы:

| **Название** | **Что делает** |
| --- | --- |
| read | читает данные |
| write | пишет данные |
| open | открывает файл |
| close | закрывает файл |
| fork | создаёт процесс |
| exec | запускает программу |
| mmap | отображает файл в память |
| exit | завершает программу |

### **7. Полный жизненный цикл горутины**

1. Пользователь вызывает go func().
2. Горутина добавляется в локальную или глобальную очередь.
3. P, прикреплённый к M, выбирает горутину:  
   * сначала из **локальной очереди**,
   * потом может **украсть** у других,
   * или взять из **netpoller'а**.
4. M исполняет горутину, пока она:  
   * не завершится,
   * не заблокируется,
   * или не будет вытеснена.
5. Если P без работы — он **засыпает**, пока его не разбудят.

### **8. Когда и как горутина "уступает"**

Чтобы приостанавливать горутину, нужны **безопасные точки проверки**:

* **Перед вызовом функций** (в стеке уже всё сохранено).
* **При блокирующих операциях** (например, syscall или таймер).

Как горутина поймёт, что ей пора уступить?

Через **флаг stackguard**. Если он установлен в stackPreempt, то при следующей проверке горутина уступает процессор.

### **9. Кто устанавливает stackPreempt?**

Наш старый знакомый — **Sysmon**!

Если горутина работает больше **10 мс**, Sysmon ставит ей stackguard = stackPreempt.  
 Такой подход называется **кооперативной многозадачностью** — когда задачи **сами** решают, когда уступить.

### **10. Проблема жадных горутин и решение в Go 1.14**

Что если горутина **никогда не уступает**?  
 Она:

* не вызывает функции,
* не делает системных вызовов,
* не проверяет stackguard.

Мы ничего не можем с ней сделать. Решение: **принудительное вытеснение** с помощью **сигналов ОС**.

### **11. Сигналы и вытеснение**

ОС может прервать выполнение процесса через сигнал.  
Go использует **SIGURG** (редко используется), чтобы:

1. Прервать выполнение.
2. Вызвать специальный обработчик.
3. Сохранить состояние горутины.
4. Вернуть управление планировщику.

### **Заключение**

Таким образом, планировщик Go — это мощный и сложный механизм, который:

* эффективно управляет миллионами горутин;
* использует модель GMP и очереди;
* взаимодействует с системными механизмами через netpoller и Sysmon;
* реализует как **кооперативное**, так и **принудительное вытеснение**.

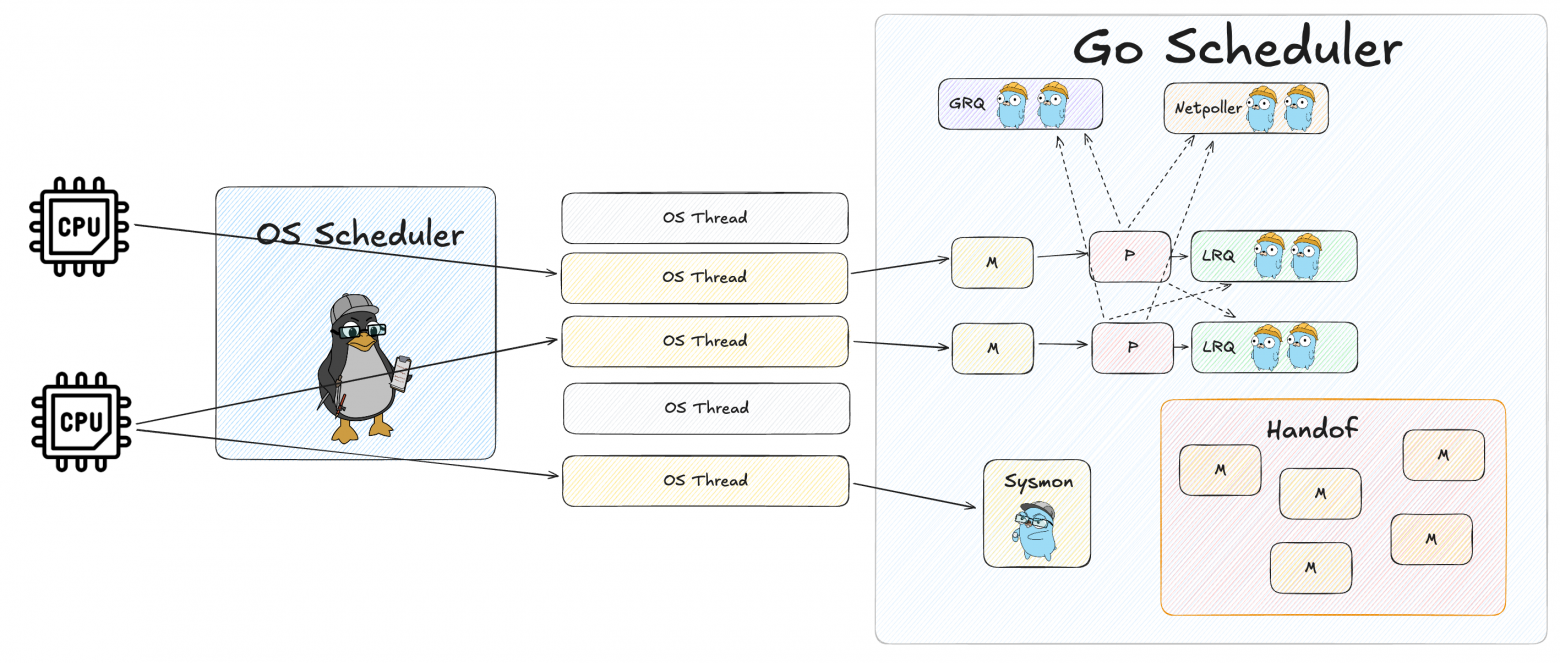


Схема разделена на **левую часть (OS Scheduler)** и **правую часть (Go Scheduler)**. Разберем всё по порядку:

### **🔧Слева: Классический планировщик ОС**

* **CPU** — физические ядра процессора.
* **OS Scheduler** — встроенный планировщик операционной системы, который управляет обычными потоками (threads).
* **OS Threads** — системные потоки, которые планировщик ОС ставит на выполнение на CPU.

📝 В обычной многопоточности всё управляется ОС, потоки дорогие, переключение между ними — тяжёлое.

* Основной ресурс для выполнения программ — ядра процессора
* Ядер мало, а работы у них много, поэтому вводится концепция тредов: треды выполняются на ядрах.
* Планировщик ОС управляет тредами и оптимизирует работу с ними таким образом, чтобы ядра не простаивали без работы.
* Треды ОС большие и страшные — не потому что их плохо спроектировали, а потому что на уровне ОС есть свои ограничения и особенности работы, поэтому тредов нам доступно мало.
* Тредов мало, а работы внутри нашей программы очень много, поэтому вводится концепция горутин: горутины выполняются на тредах

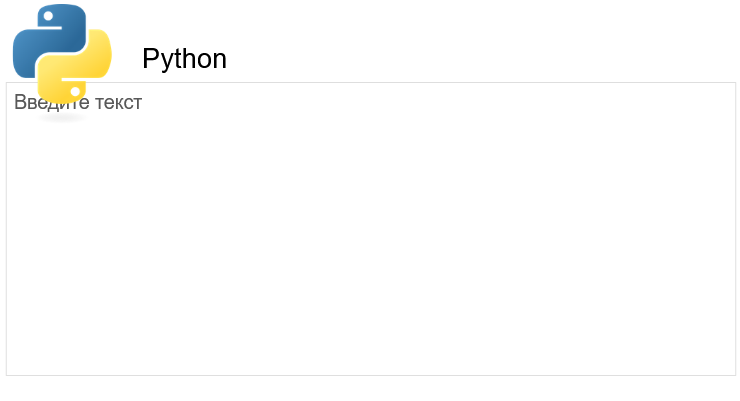
### **⚙️ Справа: Планировщик Go (M:P:G модель)**

* **M (Machine)** — это обёртка над **OS Thread**. Реальный поток ОС.
* **P (Processor)** — логический процессор Go. Он содержит локальную очередь (LRQ) горутин. У Go по умолчанию столько P, сколько доступно CPU.
* **G (Goroutine)** — горутина, легковесная задача.
* **GRQ (Global Run Queue)** — глобальная очередь горутин, если P не хватает задач.
* **Netpoller** — слушает сетевые события (например, когда данные пришли по сокету).
* **Sysmon** — системный монитор, следит за блокировками, “мертвыми” потоками и др.
* **HandOff** — механизм передачи задач между потоками, если например один P освобождается.



# **Как устроены корутины в других языках.**

**Корутина** — это особая форма функции, которая может **приостанавливать свое выполнение** и **возобновляться позже**, при этом **сохраняя свое локальное состояние**. Она реализует **асинхронное управление потоком**, не создавая при этом полноценного потока (thread), а значит, обходится дешевле по ресурсам.



### **Python (asyncio)**

В Python корутина — это объект‑генератор, который поддерживает протокол \_\_await\_\_. Когда интерпретатор видит async def, он строит конечный автомат: каждая точка с await превращается в переход автомата. При исполнении корутина бежит до первого await, складывает в кадре (frame) все локальные переменные, позицию bytecode и отдаёт управление объекту Task.  
 Task передаёт работу однопоточному event‑loop’у. Цикл отслеживает ready‑список файловых дескрипторов через select/poll/epoll/kqueue и таймер‑кучу. Когда событие готово (сокет принёс данные, таймер истёк), соответствующий Task ставится в очередь, его coroutine‑frame снова активируется и выполняется до следующего await. Всё строго кооперативно: если корутина не вызывает await, она никому не уступит.

### 

### **Kotlin (kotlinx.coroutines)**

В Kotlin любая suspend‑функция превращается компилятором в класс с методом invokeSuspend. Этот класс хранит номер текущего состояния и все локальные переменные как поля. Каждая приостановка — это вызов скрытой функции suspendCoroutine или встроенного primitive’а, которая получает объект Continuation<T>. Эта continuation описывает «что делать после возобновления» и передаётся планировщику.

Планировщик живёт в библиотеке kotlinx.coroutines. Он основан на пуле рабочих потоков и work‑stealing‑очередях. Диспетчеры (Dispatchers.Default, IO, Main и т. д.) решают, где исполнять continuation: на CPU‑пуле, в отдельном IO‑пуле или прямо в текущем потоке. Управление может быть вытесняющим: coroutine A может возобновиться на другом потоке, пока coroutine B заблокирована на suspend‑точке. При этом действует structured concurrency: если родитель отменён, планировщик рекурсивно отменяет все дочерние continuations и выкидывает CancellationException.

### 

### **C++20 (языковые корутины)**

В C++ функция, содержащая co\_await, co\_yield или co\_return, компилируется в coroutine‑frame — структуру в динамической памяти, куда складываются параметры, локальные переменные и служебные поля. Компилятор генерирует тип‑обёртку с пользовательским promise\_type, который определяет, как и где хранится результат, что делать при исключении и т. п.

co\_await expr раскладывается на три шага через awaiter‑объект:

1. await\_ready() — нужно ли сразу продолжать.
2. await\_suspend(coroutine\_handle) — если нужна пауза, планировщик получает handle текущего frame и решает, когда возобновить.
3. await\_resume() — код после пробуждения.

Стандарт не специфицирует планировщик. Библиотеки (cppcoro, folly, liburing, asio) предоставляют свои: обычно это кооперативные очереди continuations, привязанные к thread‑pool’у или I/O‑уровню (io\_uring, epoll). Разработчик определяет, как кладутся coroutine\_handle’ы в очередь и каким потоком они вызываются.

Разница



### **Как работают корутины и горутины**

**В Go** горутины — это лёгкие, управляемые рантаймом "потоки", которые создаются с помощью ключевого слова go. За всем управлением стоит встроенный планировщик Go — он распределяет горутины между системными потоками, сам занимается вытеснением, балансировкой, блокировками. У горутин есть собственный стек, который может расти и сжиматься. Разработчику не нужно заботиться о сохранении состояния или переключении вручную — всё это делает рантайм. Это почти как потоки, только «умнее» и легче.

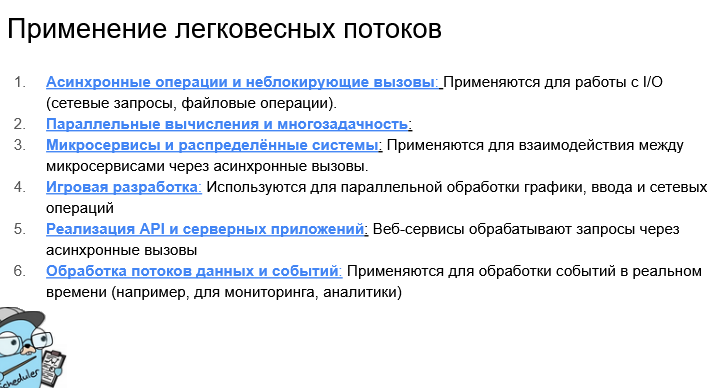
**В Python**, **Kotlin** и **C++20** корутины — это низкоуровневый механизм, который позволяет функции «приостанавливаться» и потом продолжаться с того же места. Когда вы пишете await, suspend или co\_await, происходит сохранение контекста (всех переменных и текущего места в коде). Возобновить такую функцию может только внешний планировщик — будь то asyncio в Python, диспетчеры kotlinx.coroutines в Kotlin или пользовательская очередь в C++.

### **Ключевая разница**

**Горутины в Go — это механизм + встроенный планировщик.** Они работают автономно, планируются вытесняюще, переключаются прозрачно, и рантайм Go сам всё контролирует.

**Корутины — это механизм без планировщика.** Они дают возможность «заморозить» функцию, но не решают, кто и когда её продолжит. Это делает внешний инструмент или код пользователя.

### **Применение**

****

1. **Асинхронные операции и неблокирующие вызовы**: Применяются для работы с I/O (сетевые запросы, файловые операции).
2. **Параллельные вычисления и многозадачность**:
3. **Микросервисы и распределённые системы**: Применяются для взаимодействия между микросервисами через асинхронные вызовы.
4. **Игровая разработка**: Используются для параллельной обработки графики, ввода и сетевых операций
5. **Реализация API и серверных приложений**: Веб-сервисы обрабатывают запросы через асинхронные вызовы
6. **Обработка потоков данных и событий**: Применяются для обработки событий в реальном времени (например, для мониторинга, аналитики)

### **Личное Применение:**

