VIRTUAL THREADS ГЛАЗАМИ GOROUTINE





Самойлов Сергей

Compute Storage,
MWS Cloud Platform

Кто я и зачем я

- Я техлид команды ComputeStorage
- Мы делаем самый-самый core облака — блочные устройства
- Я отвечаю за kotlin/go-направление



Почему не Erlang, не Kotlin, не Node?

Go — язык, который Oracle назвала по имени в тексте JEP 444

Прямая цитата в JEP 444

Oracle написала: "successful in other languages (e.g., goroutines in Go and processes in Erlang)". Erlang тут для полноты. Go — практический пример

Та же проблема, те же инструменты

Go и Java — оба мейнстримные production-языки. Оба решают web-серверы, микросервисы, I/O. Одна задача — конкурентность под нагрузкой

14 лет production-данных

Go 1.0 вышел в 2012, и с тех пор GMP в production у Google, Cloudflare, HashiCorp. Это не теория — это проверенное решение

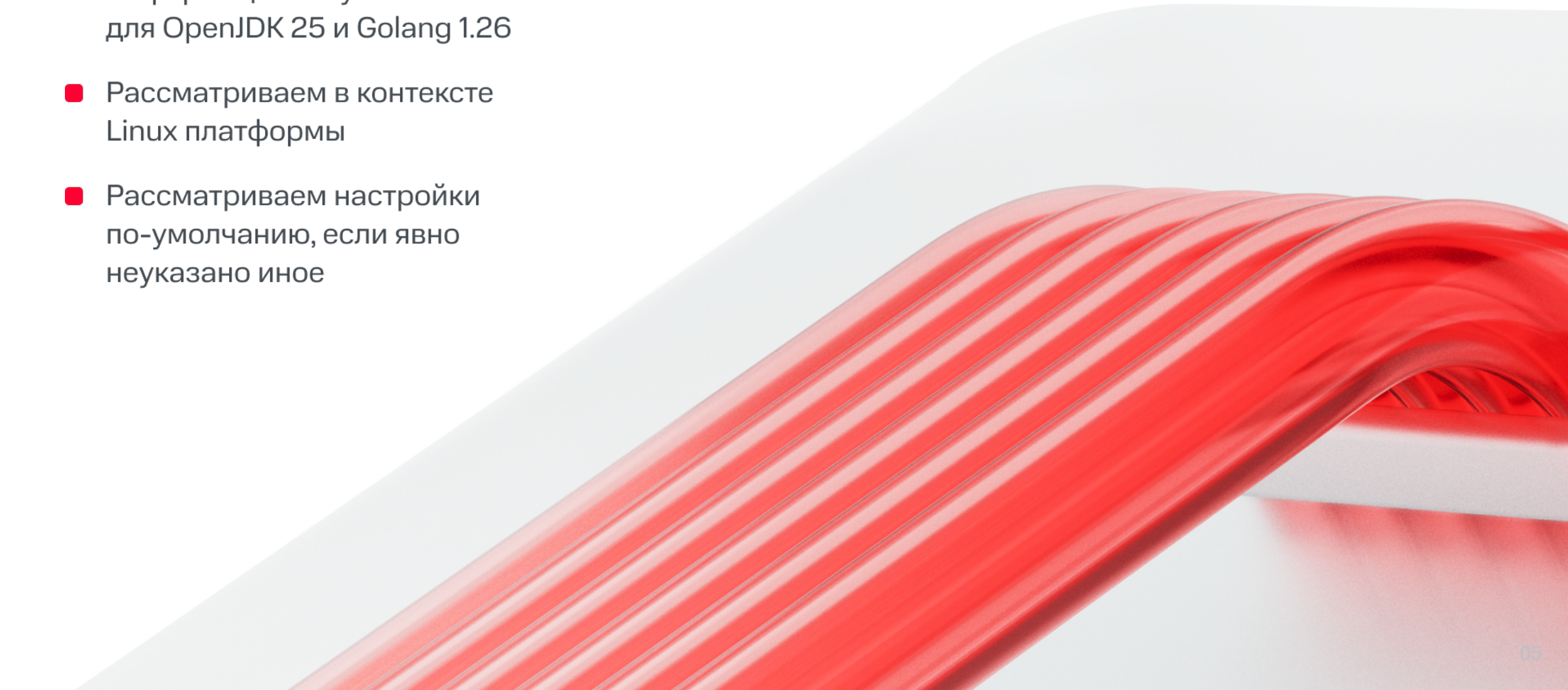
Go — это «что могло быть иначе»

Сравнивая Go GMP и Java VT, мы понимаем, что значит решить с нуля vs решить в legacy. Это делает Java-архитектуру понятнее

Через призму Go мы сможем легче понять, какие задачи стоят перед виртуальными потоками

Дисклеймеры

- Информация актуальная для OpenJDK 25 и Golang 1.26
- Рассматриваем в контексте Linux платформы
- Рассматриваем настройки по-умолчанию, если явно неуказано иное



А какую проблему мы решаем?

ЗАНЯТОСТЬ АДРЕСНОГО ПРОСТРАНСТВА

10 GB VAS

OS Thread стек × 10K

~10 MB VAS

Virtual Thread × 10K

~20 MB VAS

Goroutine × 10K

СТАРТОВЫЙ СТЕК ОДНОГО ПОТОКА

1 MB

OS Thread (Linux) — зарезервировано в адресном пространстве

~1 KB

Virtual Thread — Continuation-объект в heap JVM

2 KB

Goroutine — стек в heap, copy-on-grow

Context switch при блокировке: OS thread ~1–10 μs vs Virtual Thread ~100–200 ns — разница на порядок
На CPU-bound коде этого выигрыша нет: без блокировок переключений не бывает. Вернёмся к этому в бенчмарках.
"The JDK's current implementation caps throughput well below what hardware can support." — JEP 444

Но это виртуальная память!

Резерв стека не равен занятой RAM. Но поток ОС требует ресурсов.

1 000 000 потоков ОС ≈

16,384 ГБ

реальной RAM — только стеки ядра

Расчёт при 16 КиБ стека ядра на поток; без остальных расходов. Системные лимиты могут остановить создание раньше

ЧИСЛО ПОТОКОВ

`pids.max`

Спящий поток тоже занимает место в лимите

РЕАЛЬНАЯ ПАМЯТЬ

`memory.max`

Стек ядра и структуры. Используемые страницы пользовательского стека

РЕЗЕРВ АДРЕСОВ

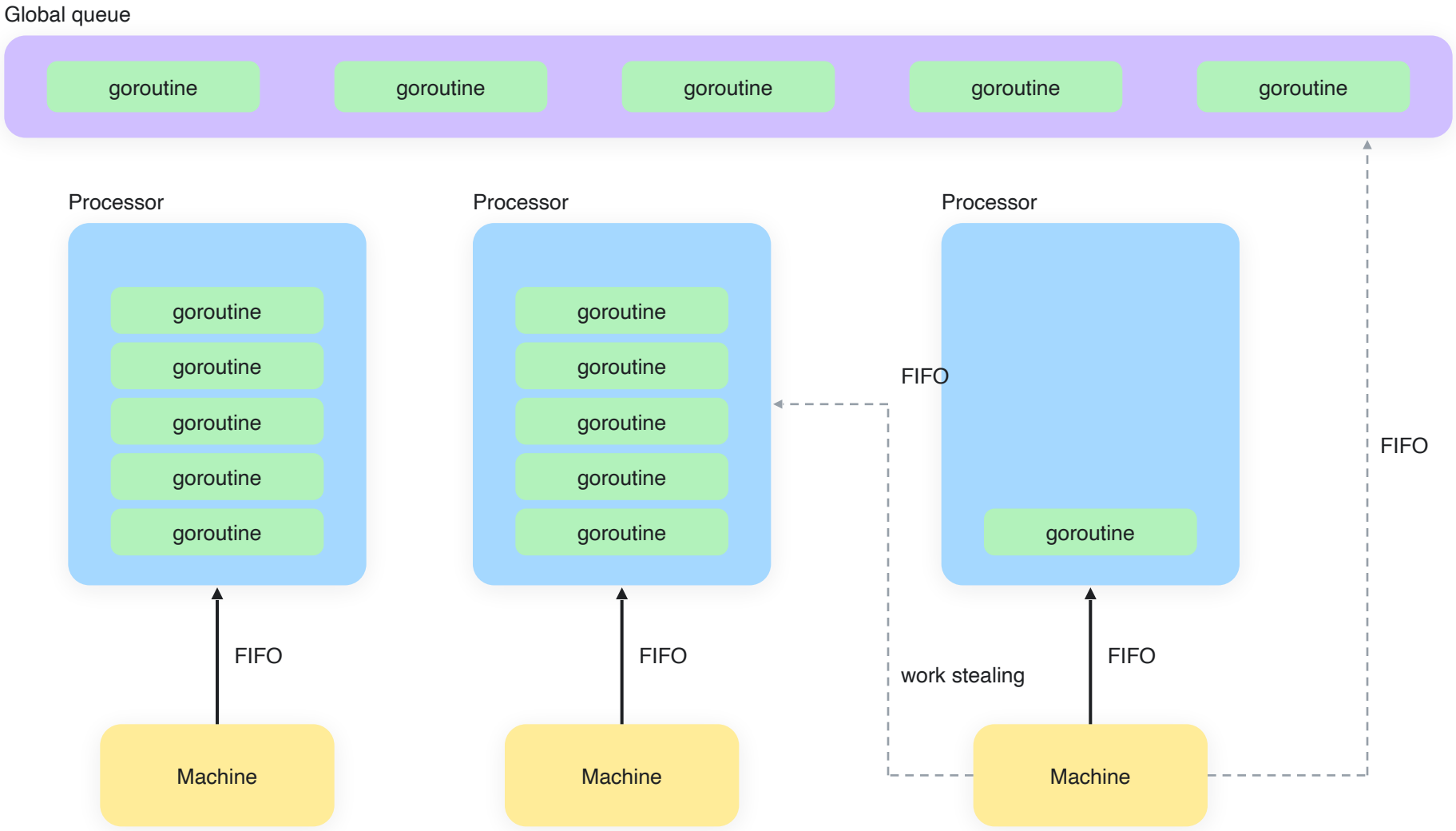
`RLIMIT_AS`

Если задан, учитывает резерв стеков в адресном пространстве процесса

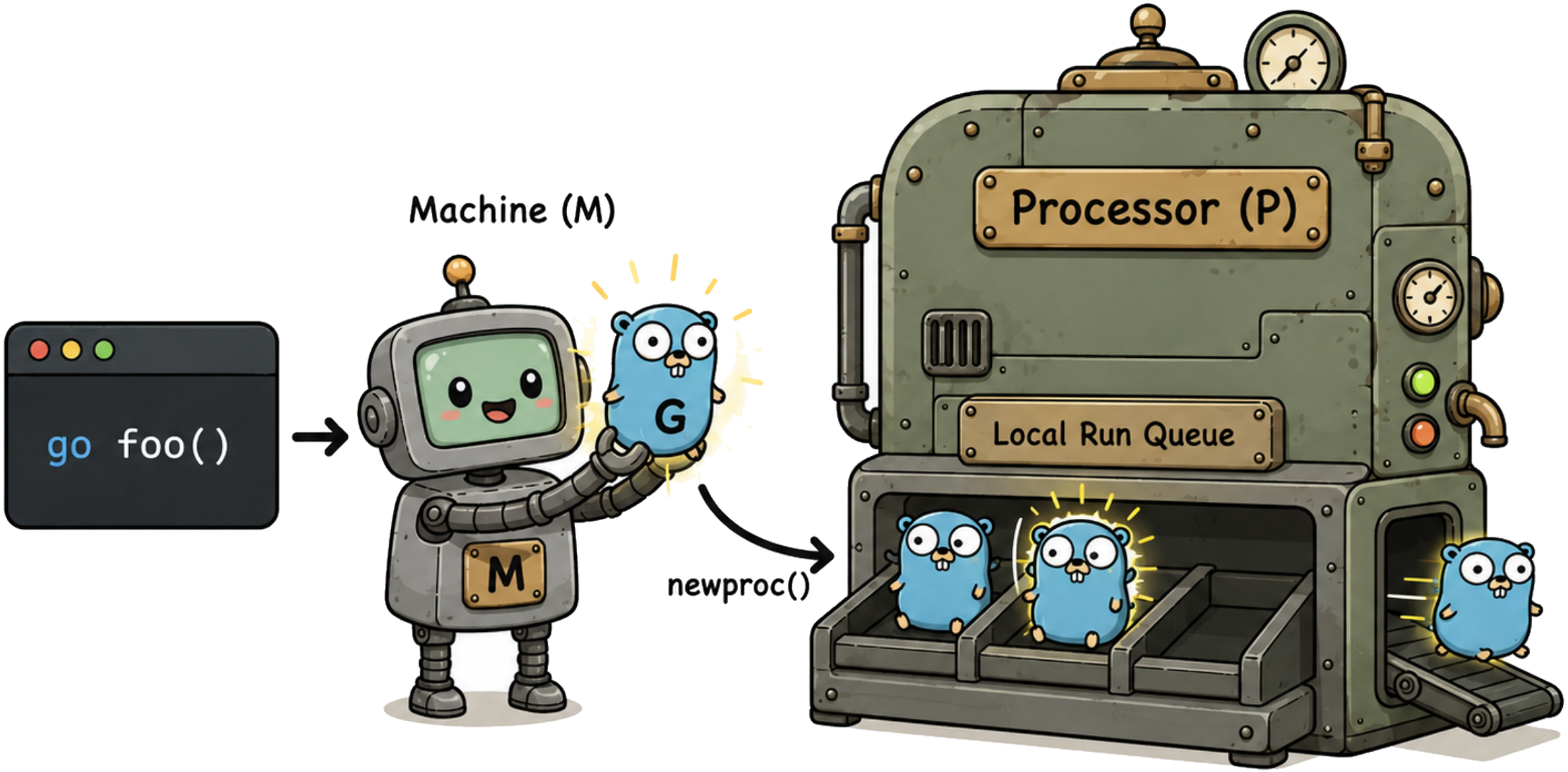
Но ведь есть крайне успешная модель в Go!



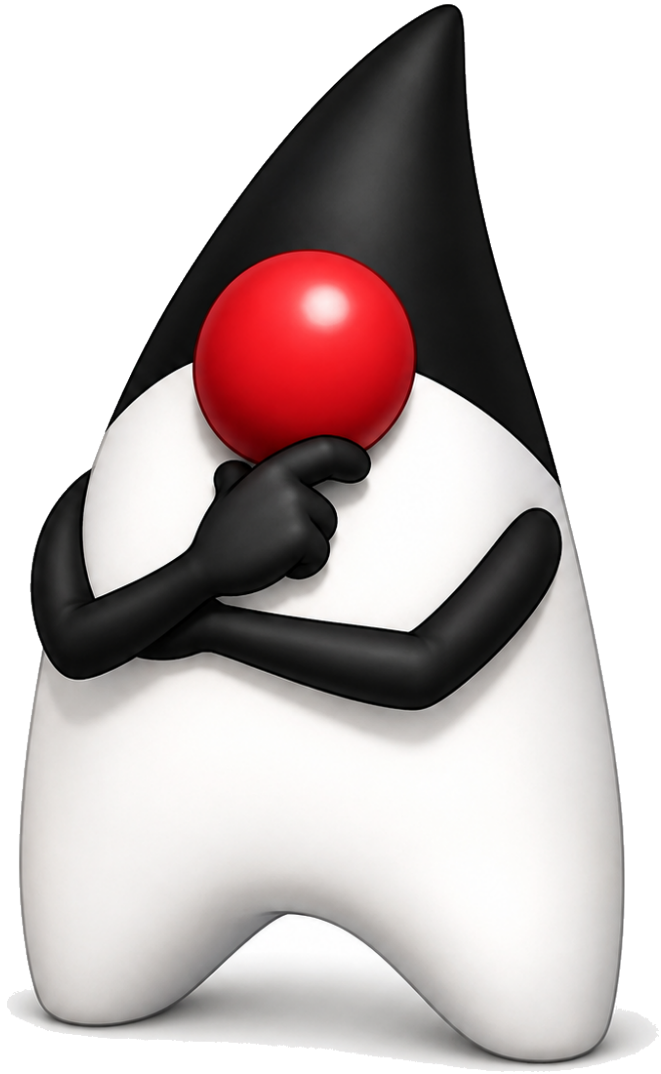
GMP модель


Global queue
goroutine
goroutine
goroutine
goroutine
goroutine
Processor
goroutine
goroutine
goroutine
goroutine
goroutine
Processor
goroutine
goroutine
goroutine
goroutine
goroutine
Processor
goroutine
FIFO
FIFO
FIFO
FIFO
work stealing
Machine
Machine
Machine

А что происходит под капотом?

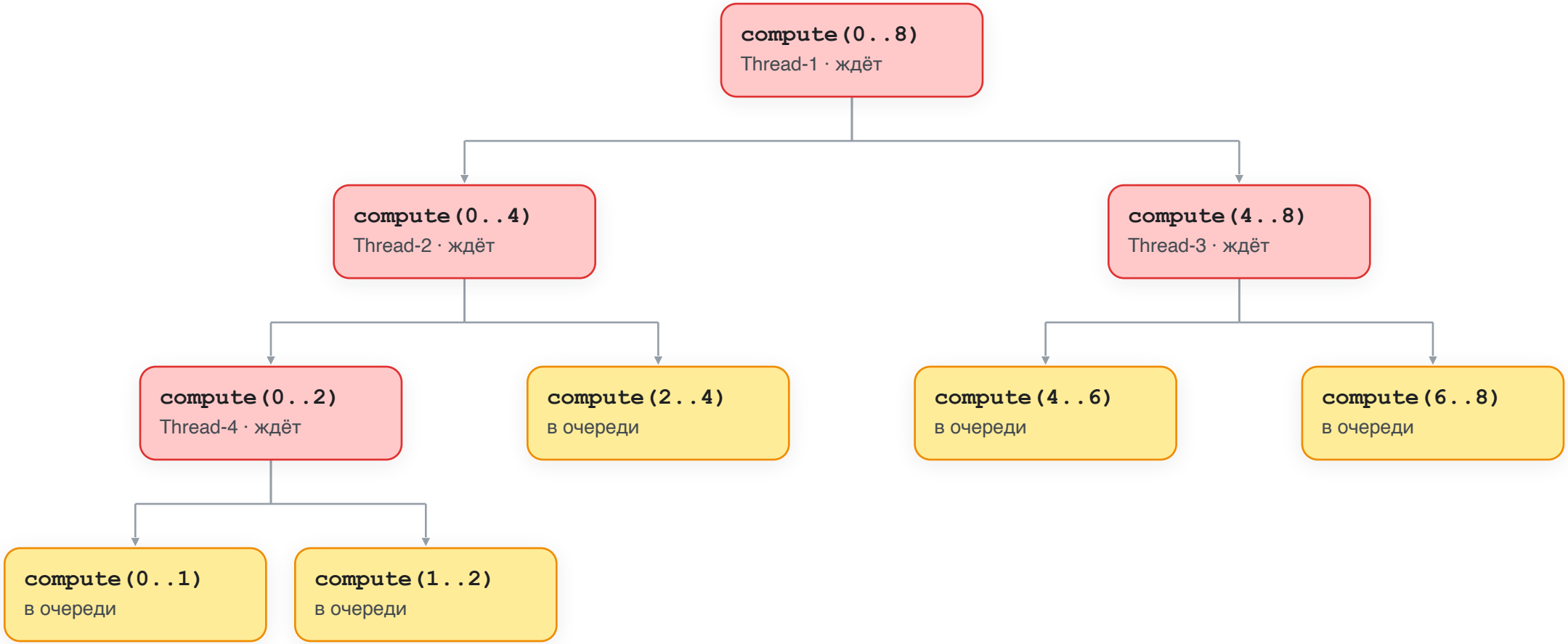


Что нам делать в Java?

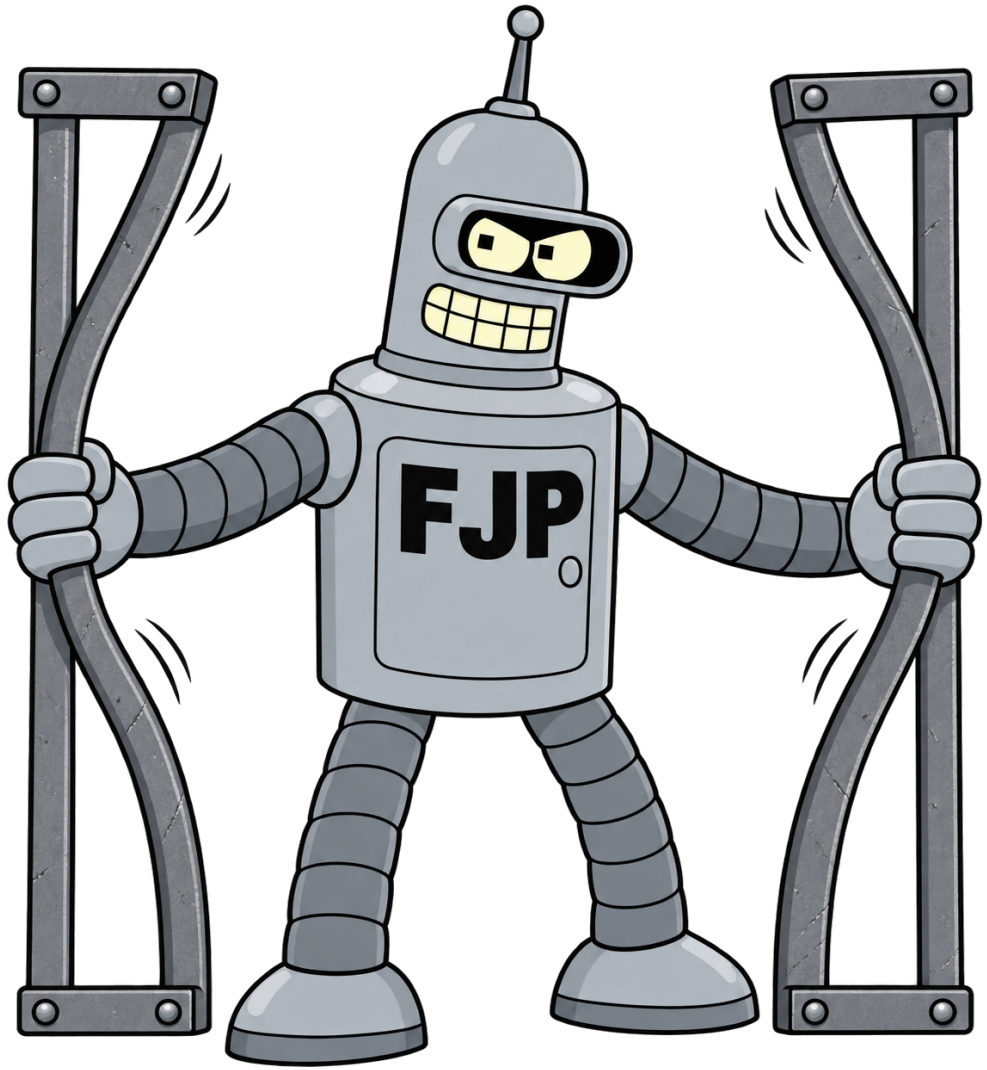


Вернемся в 2011 год

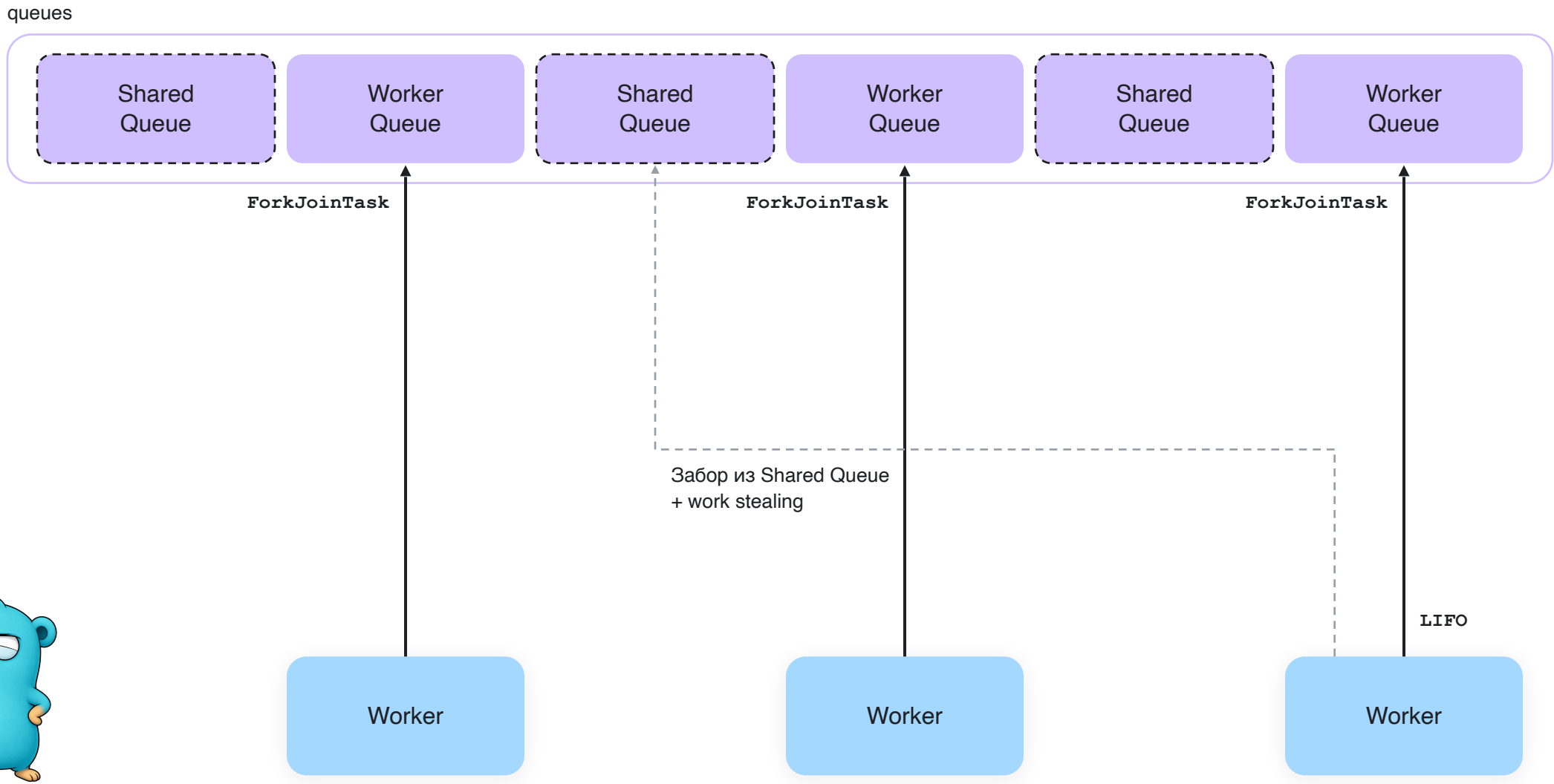
Обычный пул · Executors.newFixedThreadPool(4)



Встречайте FJP!



Как это работает





Ключевые правила FJP

fork()

кладёт подзадачу в СВОЮ очередь потока — без общего лока

join()

На join воркер может помочь выполнить работу, прежде чем перейти к ожиданию

LIFO для себя, FIFO для воров

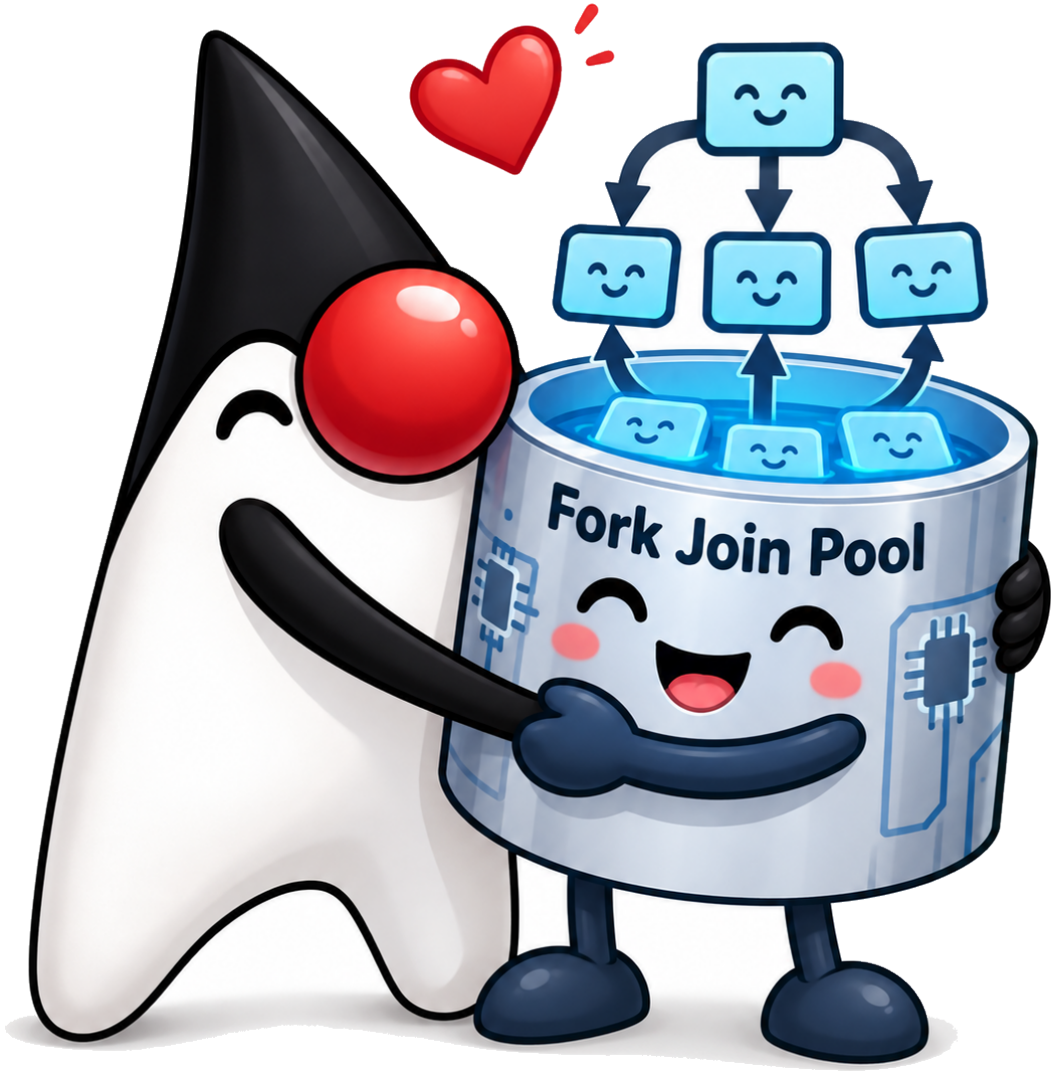
свои задачи — с горячего конца, украденные — с холодного

НИКТО НЕ ЖДЁТ

Если хотите знать больше о FJP



Но ведь это
то, что надо!



Как устроен виртуальный поток

```
public class Thread implements Runnable { ... }

abstract sealed class BaseVirtualThread extends Thread
        permits VirtualThread, ThreadBuilders.BoundVirtualThread {
    ...
}

final class VirtualThread extends BaseVirtualThread { ... }
```

`VirtualThread` — package-private. Снаружи его не видно: только **`Thread.ofVirtual()`** и **`Thread.isVirtual()`**

Да, но причём тут FJP?

```
final class VirtualThread extends BaseVirtualThread {
    private static final ForkJoinPool DEFAULT_SCHEDULER = createDefaultScheduler();
    private final Executor scheduler;
    ...
}
```

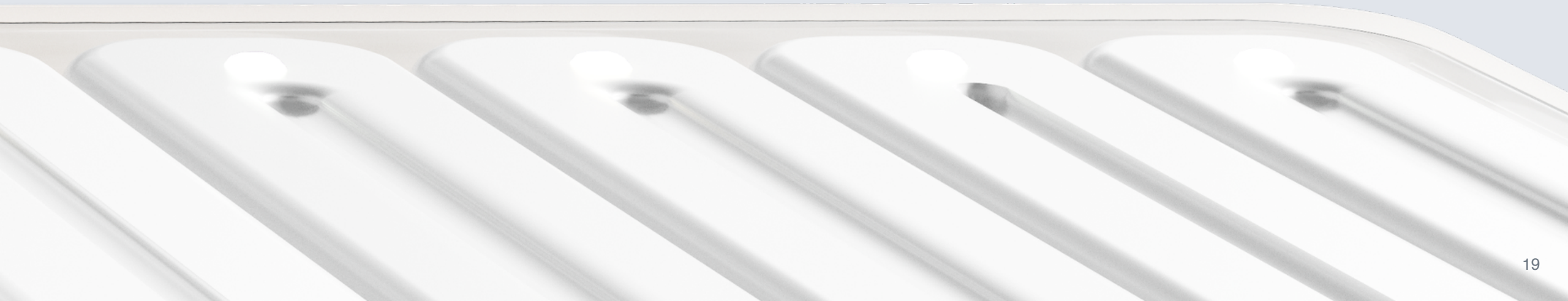
scheduler — куда отдавать задачу на исполнение. Тип — Executor: «что-то, у чего есть execute(Runnable)». По умолчанию — `DEFAULT_SCHEDULER`.

Disclaimer: asyncMode = true — очереди в режиме FIFO, а не LIFO как для fork/join.

```
// VirtualThread.createDefaultScheduler()
ForkJoinWorkerThreadFactory factory = pool -> new CarrierThread(pool);
// class CarrierThread extends ForkJoinWorkerThread
```

`carrier` — платформенный поток, который прямо сейчас исполняет виртуальный. Через призму Go это M.

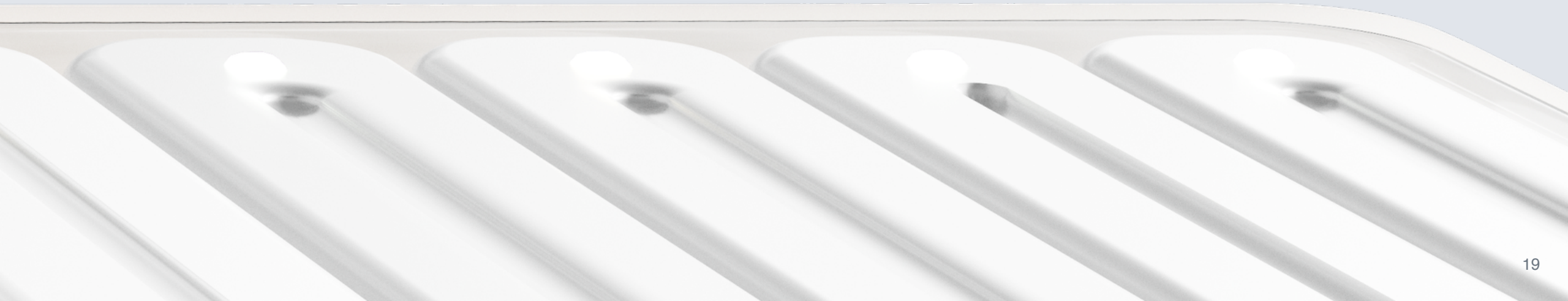
Но ведь этого мало!



Но ведь этого мало!

У нас ведь бывают:

01 Блокирующие операции
sleep, lock, очередь, Future.get()



Но ведь этого мало!

У нас ведь бывают:

01 Блокирующие операции
sleep, lock, очередь, Future.get()

02 IO-вызовы
сокеты, epoll



Но ведь этого мало!

У нас ведь бывают:

01 Блокирующие операции
sleep, lock, очередь, Future.get()

02 IO-вызовы
сокеты, epoll

03 Нативные вызовы
JNI, FFM — чужие кадры на стеке

Но ведь этого мало!

У нас ведь бывают:

01 Блокирующие операции
sleep, lock, очередь, Future.get()

02 IO-вызовы
сокеты, epoll

03 Нативные вызовы
JNI, FFM — чужие кадры на стеке

04 Долгие вычисления
CPU-bound цикл без единой блокировки

Блокирующие вызовы в Go

```
conn.Read(buf)          // данных в сокете нет
<-ch                   // канал пуст
mu.Lock()              // мьютекс занят
time.Sleep(d)          // просто ждём
```

Горутина не может продолжить. Что нужно делать рантайму?

Блокирующие вызовы в Go

```
conn.Read(buf)          // данных в сокете нет
<-ch                    // канал пуст
mu.Lock()               // мьютекс занят
time.Sleep(d)           // просто ждём
```

Горутина не может продолжить. Что нужно делать рантайму?

Наивно: заблокировать поток

10 000 соединений → 10 000 потоков ОС. Ровно та проблема, с которой начали

Блокирующие вызовы в Go

```
conn.Read(buf)          // данных в сокете нет
<-ch                    // канал пуст
mu.Lock()               // мьютекс занят
time.Sleep(d)           // просто ждём
```

Горутина не может продолжить. Что нужно делать рантайму?

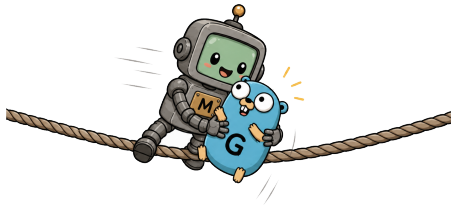
Наивно: заблокировать поток

10 000 соединений → 10 000 потоков ОС. Ровно та проблема, с которой начали

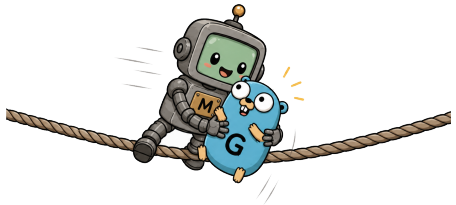
Go: снять горутину с потока

Поток остаётся свободен и берёт следующую горутину. Один вызов — gopark

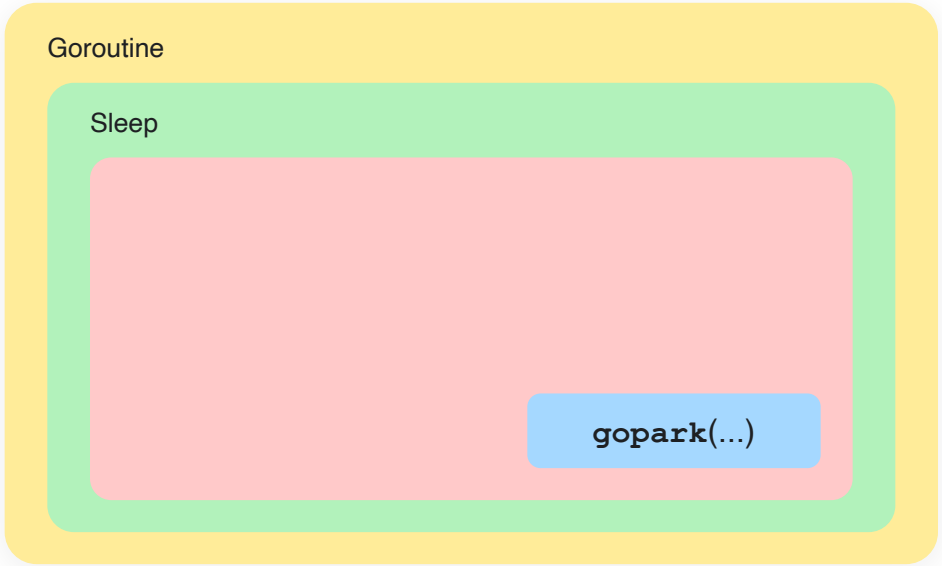
Как это работает



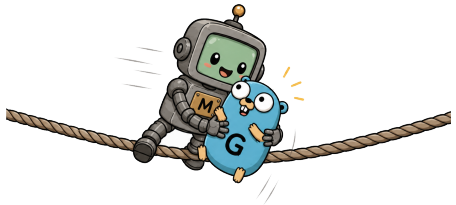
Как это работает



Machine



Как это работает

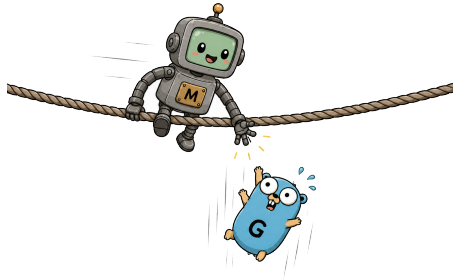


Machine

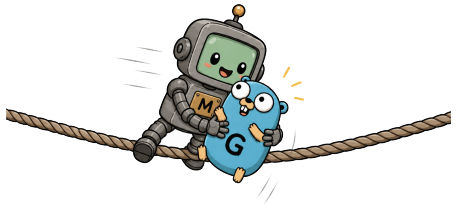
Goroutine

Sleep

```
gopark(...)
```



Как это работает



Machine

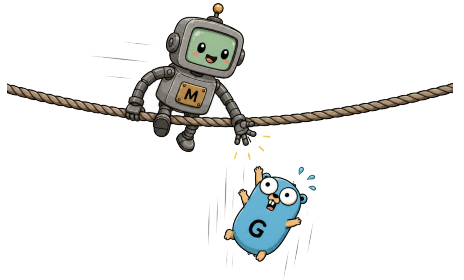
Goroutine

Sleep

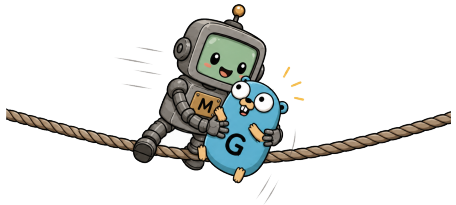
`gopark(...)`

timer

`ready(...)`



Как это работает



Machine

Goroutine

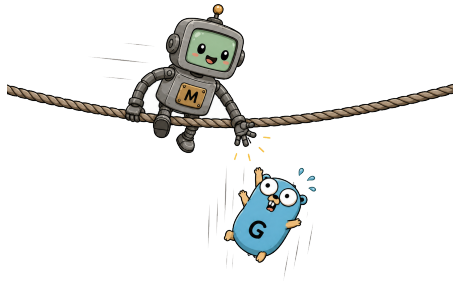
Sleep

`gopark(...)`

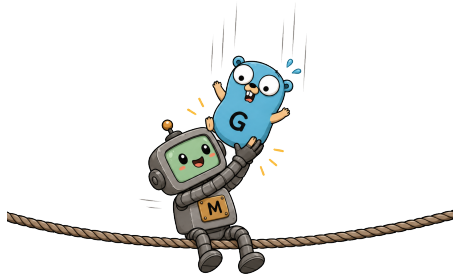
timer

`ready(...)`

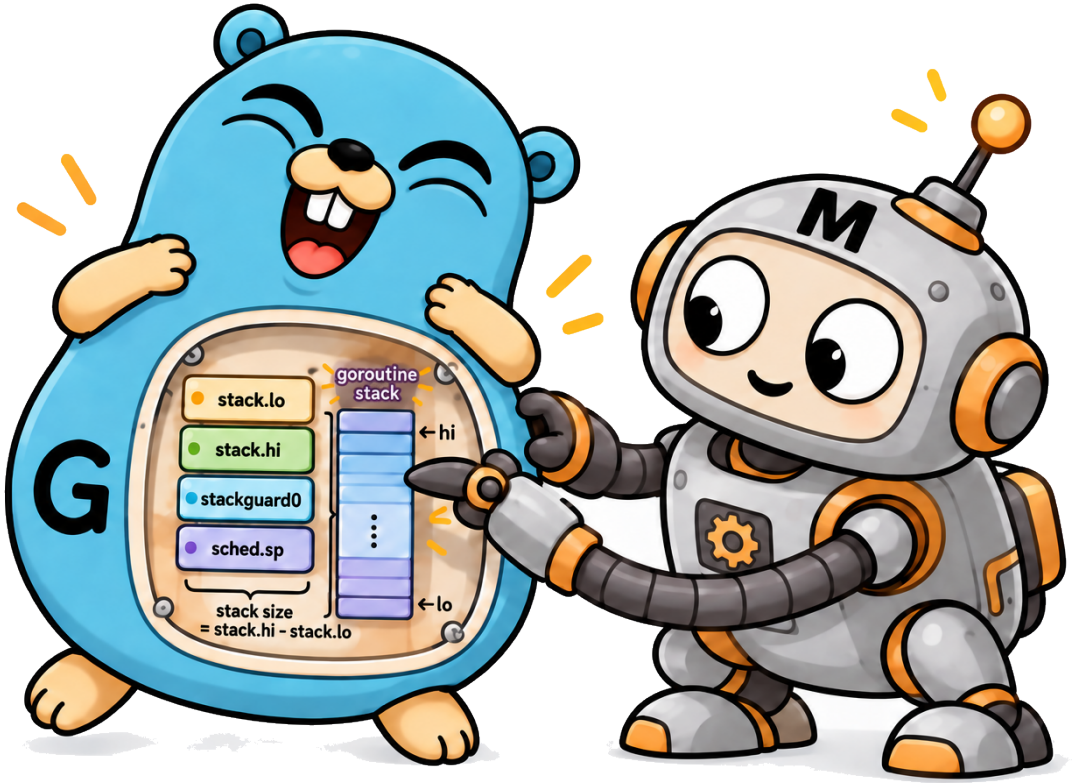








Но почему мы можем выбросить горутину



Итого: рецепт Go

Итого: рецепт Go

01 Свой стек: у каждой горутины — собственная память под кадры

Итого: рецепт Go

01 Свой стек: у каждой горутины — собственная память под кадры

02 gopark: смена указателя, стек остаётся на месте

Итого: рецепт Go

01 Свой стек: у каждой горутины — собственная память под кадры

02 gopark: смена указателя, стек остаётся на месте

03 goready: разбудить может любой G

Итого: рецепт Go

01 Свой стек: у каждой горутины — собственная память под кадры

02 gopark: смена указателя, стек остаётся на месте

03 goready: разбудить может любой G

Повторим на Java?!

Как нам решить похожую проблему в Java

Как нам решить похожую проблему в Java

~~Сделаем просто: доходим до метода sleep/lock/etc и снимаем с потока.~~



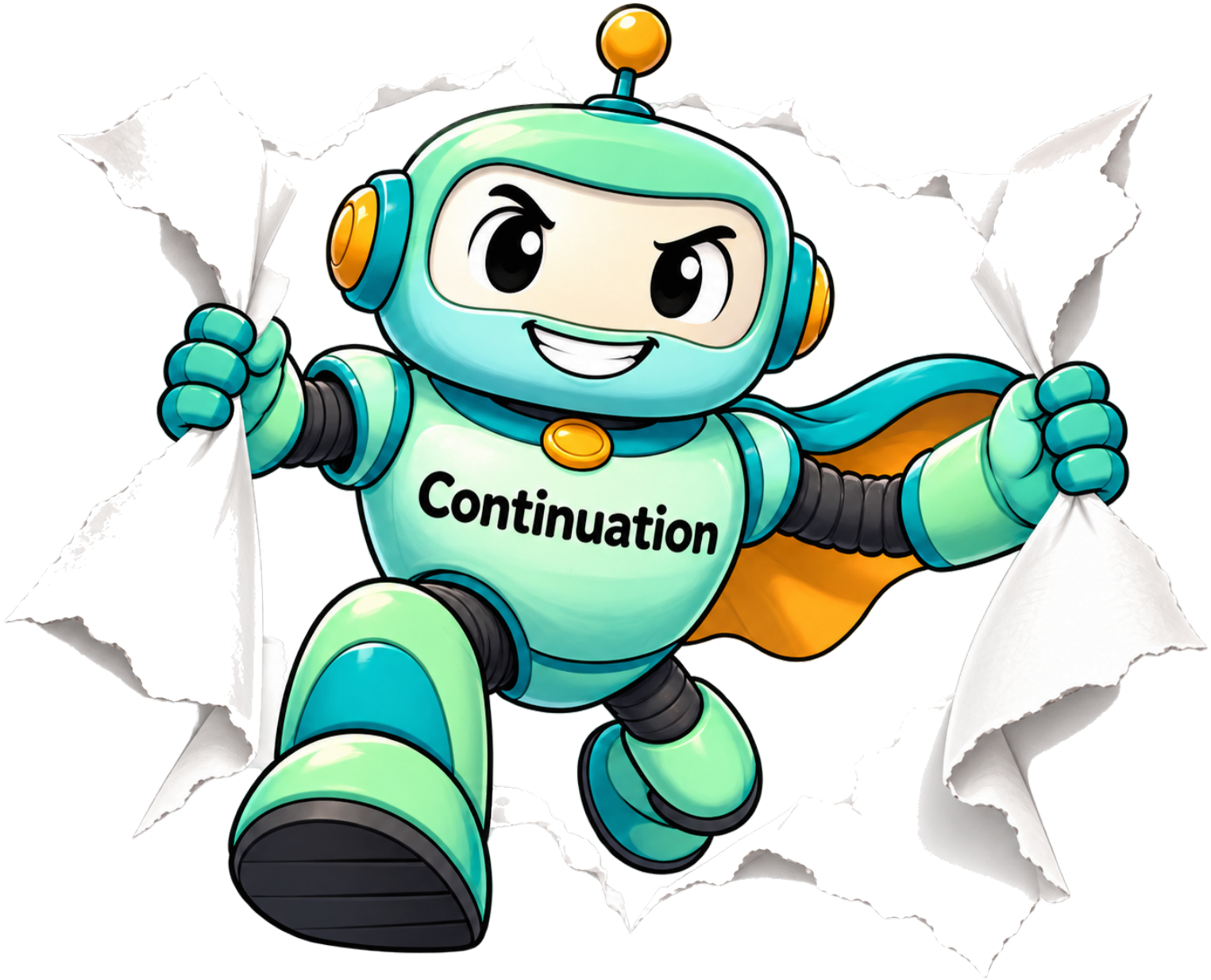
Как нам решить похожую проблему в Java

~~Сделаем просто: доходим до метода sleep/lock/etc и снимаем с потока.~~





> Обычный Java-метод нельзя приостановить посередине
>
> Его состояние — кадры на стеке потока ОС.
> Нет никакого dropg(), потому что нечего отвязывать: у задачи нет своего стека


Continuation

Встречайте Continuation

```
ContinuationScope scope = new ContinuationScope("demo");

Continuation cont = new Continuation(scope, () -> {
    System.out.println("A");
    Continuation.yield(scope);   // ← приостановились ЗДЕСЬ
    System.out.println("B");
});

cont.run();                  // печатает "A" и возвращает управление сюда
System.out.println("между");
cont.run();                  // печатает "B" — продолжили с того же места
```

Встречайте Continuation

```
ContinuationScope scope = new ContinuationScope("demo");

Continuation cont = new Continuation(scope, () -> {
    System.out.println("A");
    Continuation.yield(scope);   // ← приостановились ЗДЕСЬ
    System.out.println("B");
});

cont.run();                  // печатает "A" и возвращает управление сюда
System.out.println("между");
cont.run();                  // печатает "B" — продолжили с того же места
```

Встречайте Continuation

```
ContinuationScope scope = new ContinuationScope("demo");

Continuation cont = new Continuation(scope, () -> {
    System.out.println("A");
    Continuation.yield(scope);   // ← приостановились ЗДЕСЬ
    System.out.println("B");
});

cont.run();                    // печатает "A" и возвращает управление сюда
System.out.println("между");
cont.run();                    // печатает "B" — продолжили с того же места
```

Вернемся к VirtualThread

```
final class VirtualThread extends BaseVirtualThread {
    private static final ContinuationScope VTHREAD_SCOPE = new ContinuationScope("VirtualThreads");
    private static final ForkJoinPool DEFAULT_SCHEDULER = createDefaultScheduler();

    // scheduler and continuation
    private final Executor scheduler;          // 1. КТО его будет исполнять
    private final Continuation cont;           // 2. ЧТО исполнять + где хранится стек
    private final Runnable runContinuation;    // 3. задача-обёртка для планировщика

    // virtual thread state, accessed by VM
    private volatile int state;                // 4. машина состояний
    ...
}
```

Вернемся к VirtualThread

```
final class VirtualThread extends BaseVirtualThread {
    private static final ContinuationScope VTHREAD_SCOPE = new ContinuationScope("VirtualThreads");
    private static final ForkJoinPool DEFAULT_SCHEDULER = createDefaultScheduler();

    // scheduler and continuation
    private final Executor scheduler;          // 1. КТО его будет исполнять
    private final Continuation cont;           // 2. ЧТО исполнять + где хранится стек
    private final Runnable runContinuation;    // 3. задача-обёртка для планировщика

    // virtual thread state, accessed by VM
    private volatile int state;                // 4. машина состояний
    ...
}
```

Что внутри runContinuation

> «The virtual thread is mounted on the current thread before the task runs or continues. It unmounts when the task completes or yields.»

```
private void runContinuation() {
    // carrier — только платформенный поток
    if (Thread.currentThread().isVirtual()) throw new WrongThreadException();
    // STARTED | UNPARKED | UNBLOCKED | YIELDED  →  RUNNING
    if (!compareAndSetState(initialState, RUNNING)) return;

    mount();                                    // связать VT с carrier
    try {
        cont.run();                             // исполнять до yield или конца
    } finally {
        unmount();                              // отвязать
        if (cont.isDone()) afterDone(); else afterYield();
    }
}
```

Что внутри runContinuation

> «The virtual thread is mounted on the current thread before the task runs or continues. It unmounts when the task completes or yields.»

```
private void runContinuation() {
    // carrier — только платформенный поток
    if (Thread.currentThread().isVirtual()) throw new WrongThreadException();
    // STARTED | UNPARKED | UNBLOCKED | YIELDED  →  RUNNING
    if (!compareAndSetState(initialState, RUNNING)) return;

    mount();                                    // связать VT с carrier
    try {
        cont.run();                            // исполнять до yield или конца
    } finally {
        unmount();                             // отвязать
        if (cont.isDone()) afterDone(); else afterYield();
    }
}
```

Что внутри runContinuation

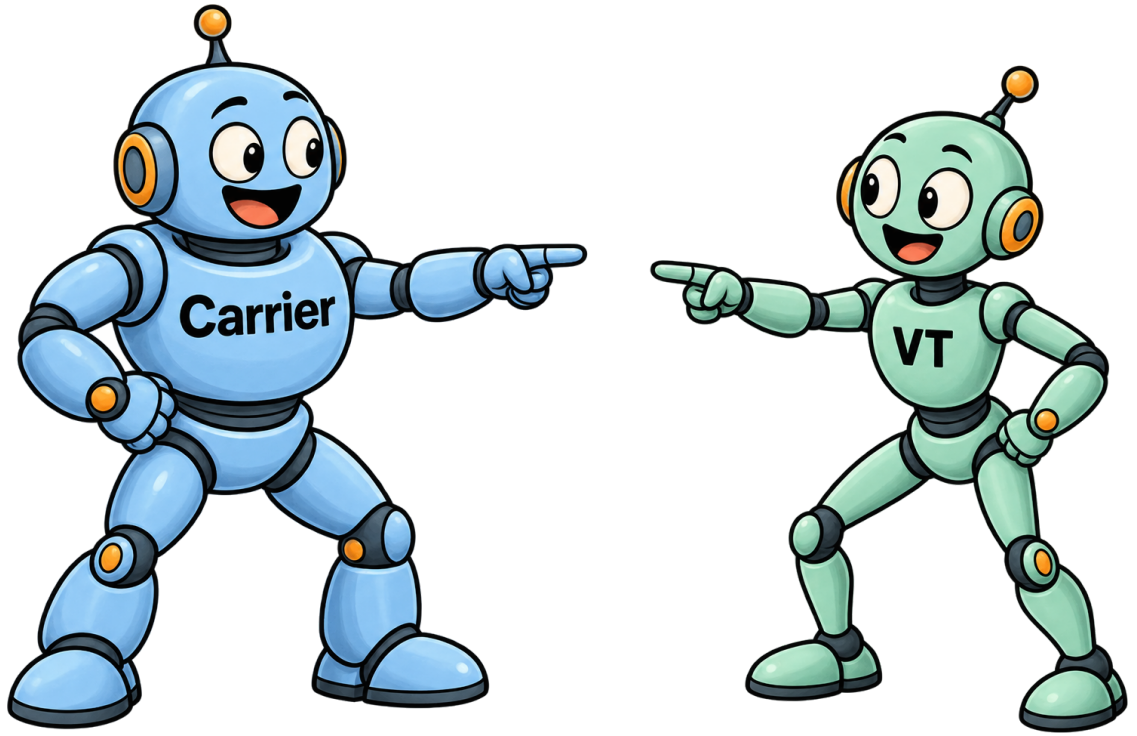
> «The virtual thread is mounted on the current thread before the task runs or continues. It unmounts when the task completes or yields.»

```
private void runContinuation() {
    // carrier — только платформенный поток
    if (Thread.currentThread().isVirtual()) throw new WrongThreadException();
    // STARTED | UNPARKED | UNBLOCKED | YIELDED  →  RUNNING
    if (!compareAndSetState(initialState, RUNNING)) return;

    mount();                                    // связать VT с carrier
    try {
        cont.run();                             // исполнять до yield или конца
    } finally {
        unmount();                              // отвязать
        if (cont.isDone()) afterDone(); else afterYield();
    }
}
```

А что вообще такое mount?

```
private void mount() {
    Thread carrier = Thread.currentCarrierThread();
    setCarrierThread(carrier);      // VT → carrier
    // ...синхронизация флага interrupted
    carrier.setCurrentThread(this);  // carrier → VT
    // с этого момента Thread.currentThread() == VT
}
```



park: как VT сходит с carrier

```
private static void sleepNanos(long nanos) throws InterruptedException {
    ...
    if (currentThread() instanceof VirtualThread vthread) {
        vthread.sleepNanos(nanos);   // ← ветка для виртуального потока
    } else {
        sleepNanos0(nanos);          // ← native: реально морозит поток ОС
    }
    ...
}
```

Thread.sleep → sleepNanos → VirtualThread.sleepNanos → parkNanos → yieldContinuation → Continuation.yield

Java пошла почти тем же путём

ШАГ	GO	JAVA
Сохранить состояние	g.sched: SP / PC / BP — стек на месте	кадры копируются в куче
Освободить носитель	dropg() → schedule()	unmount() → возврат в runWorker
Где ждёт	нигде: ссылка в структуре события (таймер, канал, pollDesc)	нигде: ссылка в Poller / мониторе / таймере
Разбудить	goready: статус + runnext + wakep	unpark: статус + задача в очередь + signalWork
Продолжить	gogo(&g.sched) — вернуть регистры	mount() → cont.run()

park ≠ unmount: две независимые оси

	MOUNTED (НА CARRIER)	UNMOUNTED (CARRIER СВОБОДЕН)
выполняется	RUNNING	—
ждёт	PINNED	PARKED
готов, ждёт очереди	—	UNPARKED · YIELDED · UNBLOCKED

Комментарии авторов в VirtualThread.java:

```
RUNNING   = 2; // runnable-mounted
PARKED    = 4; // unmounted
PINNED    = 5; // mounted
UNPARKED = 9; // unmounted but runnable
```

НОРМАЛЬНЫЙ PARK

«Запаркован и размонтирован»: VT ждёт, carrier ушёл за следующей задачей

PINNED

«Запаркован, но остался на carrier». VT ждёт, и carrier ждёт вместе с ним. Это та самая клетка, в которой живёт вся боль виртуальных потоков

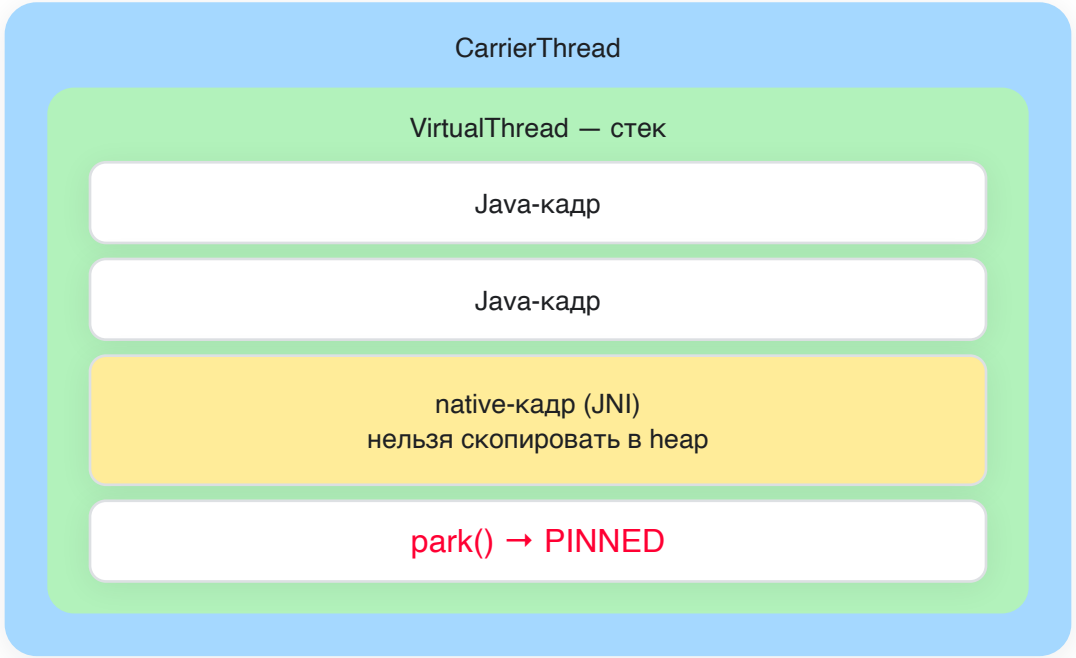
Нативные вызовы: когда unmount невозможен

Что пиннит VT к carrier

• Нативный кадр на стеке: JNI-вызов, FFM downcall. Кадры чужого ABI нельзя скопировать в StackChunk.
• Статический инициализатор класса.

JEP 491, Java 24: synchronized больше не пиннит

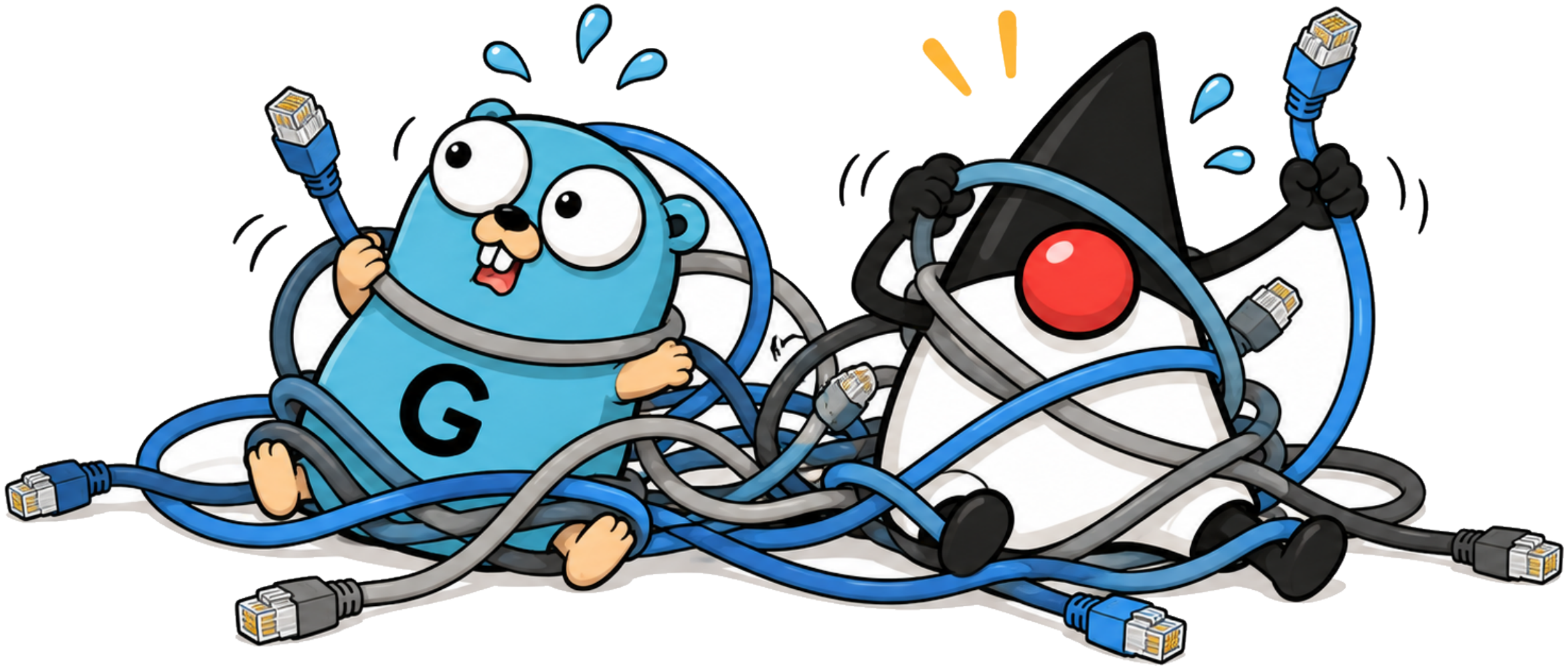
Владение монитором переехало с carrier на VT. Появились состояния BLOCKING / BLOCKED / UNBLOCKED и платформенный поток VirtualThread-unblocker.
Совет «не используйте synchronized с VT» устарел.



carrier занят, пока VT ждёт

Компенсация (+1 поток в FJP) есть не «при пиннинге», а только там, где JDK сам обработал вызов. Например Blocker.begin() в RandomAccessFile. Для вашего JNI компенсации нет: parkOnCarrierThread зовёт U.park напрямую. Carrier просто стоит

А что с сетью?



netpoller vs Poller

GO

netpoller

Не поток и не объект-деятель. Это epoll-дескриптор + pollDesc (карта fd → G).

epoll_wait зовёт кто угодно:
- любой M попутно в findRunnable
- дежурный M без P (ровно один)
- sysmon, если сеть последний опросы был > 10 мс

JAVA

sun.nio.ch.Poller

Та же ядерная половина: готовность копится в epoll независимо от нас. Но сбор централизован, и планировщик о сети не знает вообще.

Linux по умолчанию (VTHREAD_POLLERS):
- 1 платформенный MasterPoller
- N виртуальных SubPoller (N = min(2^⌊$\log_2$ cpus⌋, 32)), 1 на запись
- цикл: MasterPoller → SubPoller → прикладной VT

Вытеснение: Go может, Java — нет

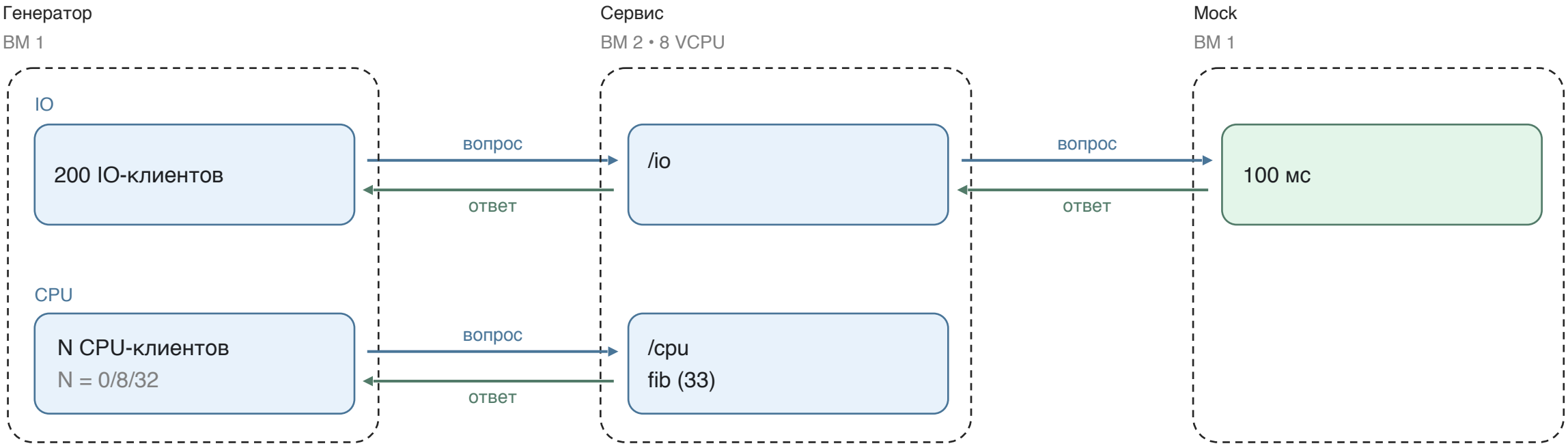
GO 1.14+

• sysmon — служебный поток-надзиратель, работает без P. Видит: горутина держит P дольше 10 мс → шлёт потоку сигнал SIGURG.
• Поток не останавливают — с него снимают горутину.

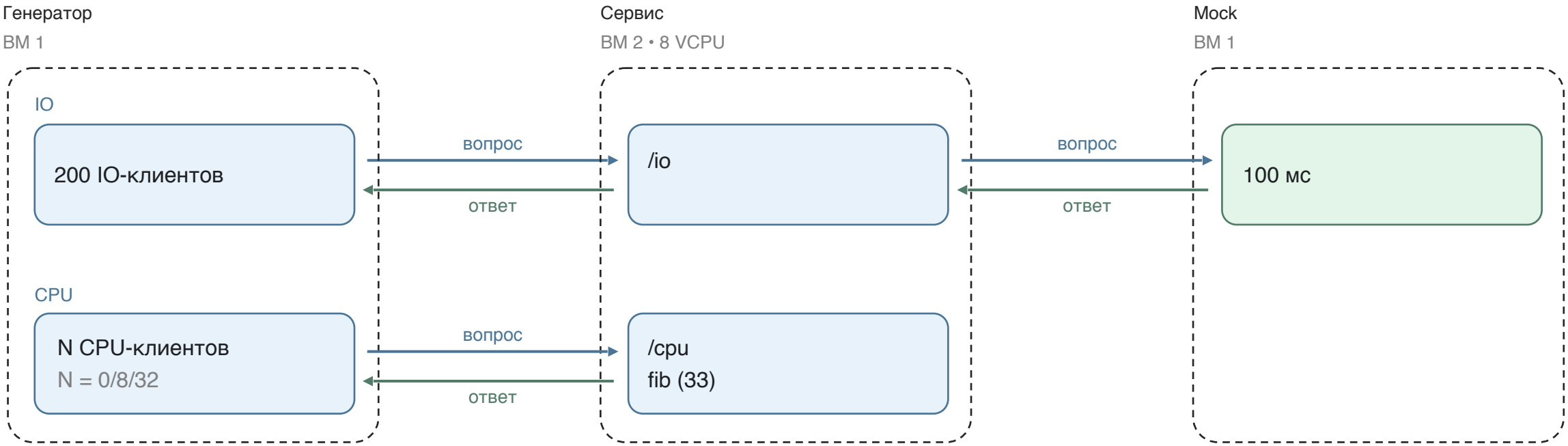
Java — никак

Аналога sysmon + SIGURG нет вообще. Планировщик не снимает вычисляющий VT с carrier по истечении временного кванта. Долгий цикл без блокировок и явного yield удерживает carrier до завершения

А давайте посчитаем

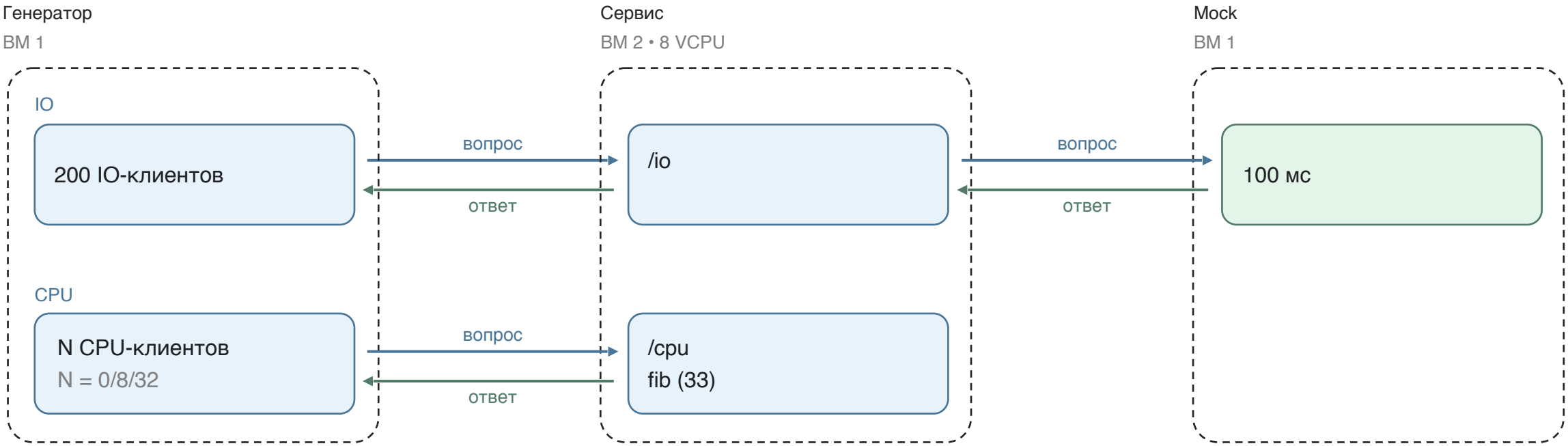


А давайте посчитаем



01 Генератор и сервис — на разных машинах: CPU генератора не конкурирует с CPU сервиса

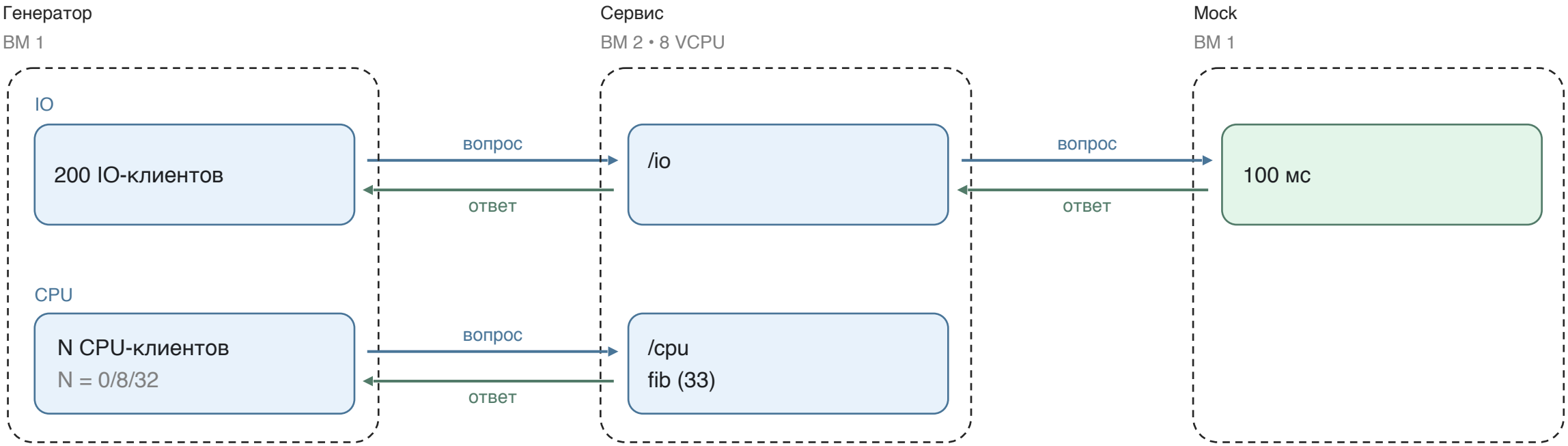
А давайте посчитаем



01 Генератор и сервис — на разных машинах: CPU генератора не конкурирует с CPU сервиса

02 JDK 25.0.3 · Go 1.26.4 · mock-backend GET /slow = flat 100 мс sleep

А давайте посчитаем

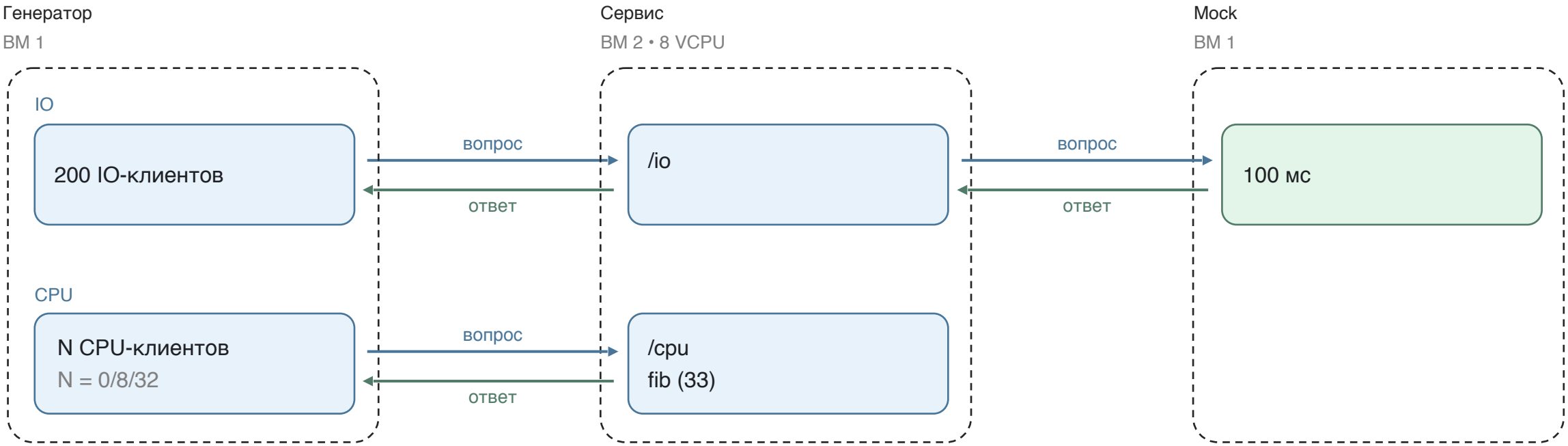


01 Генератор и сервис — на разных машинах: CPU генератора не конкурирует с CPU сервиса

02 JDK Temurin 25.0.3 · Go 1.26.4 · mock-backend GET /slow = flat 100 мс sleep

03 Флаги выровнены: JVM -Xms4g -Xmx4g -XX:+UseG1GC -XX:ActiveProcessorCount=8; Go GOMAXPROCS=8. Оба рантайма видят 8 ядер

А давайте посчитаем



01 Генератор и сервис — на разных машинах: CPU генератора не конкурирует с CPU сервиса

02 JDK Temurin 25.0.3 · Go 1.26.4 · mock-backend GET /slow = flat 100 мс sleep

03 Флаги выровнены: JVM -Xms4g -Xmx4g -XX:+UseG1GC -XX:ActiveProcessorCount=8; Go GOMAXPROCS=8. Оба рантайма видят 8 ядер

04 Closed loop тестирование - следующий запрос только после ответа прошлого

Четыре испытуемых

	СЕРВИС	КОНФИГУРАЦИЯ
A	`java-platform`	FixedThreadPool(200) на всё — CPU и I/O в одном пуле 
B	`java-virtual`	virtualThreadPerTaskExecutor() на всё, включая fib
D	`java-virtual-yeild`	перед каждой вычислительной операцией делаем yeild
E	`java-virtual-split`	сервер на VT, CPU → выделенный платформенный cpuPool=8; VT паркуется на Future.get() и unmount
F	`golang`	goroutine-per-request, fib + I/O inline, async preemption

Четыре испытуемых

	СЕРВИС	КОНФИГУРАЦИЯ
A	`java-platform`	FixedThreadPool(200) на всё — CPU и I/O в одном пуле
B	`java-virtual`	virtualThreadPerTaskExecutor() на всё, включая fib
D	`java-virtual-yeild`	перед каждой вычислительной операцией делаем yeild
E	`java-virtual-split`	сервер на VT, CPU → выделенный платформенный cpuPool=8; VT паркуется на Future.get() и unmount
F	`golang`	goroutine-per-request, fib + I/O inline, async preemption

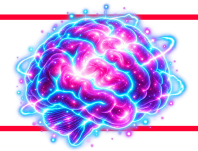




Четыре испытуемых

	СЕРВИС	КОНФИГУРАЦИЯ	
A	`java-platform`	FixedThreadPool(200) на всё — CPU и I/O в одном пуле	
B	`java-virtual`	virtualThreadPerTaskExecutor() на всё, включая fib	
D	`java-virtual-yeild`	перед каждой вычислительной операцией делаем yeild	
E	`java-virtual-split`	сервер на VT, CPU → выделенный платформенный cpuPool=8; VT паркуется на Future.get() и unmount	
F	`golang`	goroutine-per-request, fib + I/O inline, async preemption	

Четыре испытуемых

	СЕРВИС	КОНФИГУРАЦИЯ	
A	`java-platform`	FixedThreadPool(200) на всё — CPU и I/O в одном пуле	
B	`java-virtual`	virtualThreadPerTaskExecutor() на всё, включая fib	
D	`java-virtual-yeild`	перед каждой вычислительной операцией делаем yeild	
E	`java-virtual-split`	сервер на VT, CPU → выделенный платформенный cpuPool=8; VT паркуется на Future.get() и unmount	
F	`golang`	goroutine-per-request, fib + I/O inline, async preemption	

Четыре испытуемых

	СЕРВИС	КОНФИГУРАЦИЯ	
A	`java-platform`	FixedThreadPool(200) на всё — CPU и I/O в одном пуле	
B	`java-virtual`	virtualThreadPerTaskExecutor() на всё, включая fib	
D	`java-virtual-yeild`	перед каждой вычислительной операцией делаем yeild	
E	`java-virtual-split`	сервер на VT, CPU → выделенный платформенный cpuPool=8; VT паркуется на Future.get() и unmount	
F	`golang`	goroutine-per-request, fib + I/O inline, async preemption	

Методология: SPLIT

200 I/O-клиентов фиксировано + cpu (0, 8, 32) бьют одновременно, метрики раздельно по классу. CPU0 = чистый I/O-пол.

СЕРВИС, P99 I/O	CPU0	CPU8	CPU32	RPS CPU32	ERR
java-platform	102 мс	118 мс	153 мс	1614	0
java-virtual	102 мс	129 мс	4963 мс	72	0
java-virtual-yeild	102 мс	124 мс	434 мс	1088	0
java-virtual-split	102 мс	107 мс	105 мс	1969	0
golang	103 мс	122 мс	242 мс	1337	0

ВЫВОД

Вынос вычислений с carriers резко улучшил обслуживание I/O в этом стенде. При этом CPU-пропускная способность снизилась примерно на 9% — стоит показать обе стороны.

МОЙ ЛИЧНЫЙ ВЫВОД

Если CPU вынести в отдельный пул, java-virtual-split обгоняет даже Go: планированием CPU-потоков занимается ядро, а у ядра вытеснение точнее, чем 10 мс sysmon.

Что унести с собой

01 Использовать VT для большого числа ожидающих запросов
Ожидание сети и обычная парковка позволяют освободить carrier

02 Проверять влияние долгих вычислений на задержки соседних запросов
Если они мешают I/O, рассмотреть отдельный CPU-пул с ограниченной очередью

03 Различать причины задержки
VT может ждать событие, ждать свободный carrier или удерживать его.
Эти случаи требуют разных решений

04 Ограничивать доступ к реальным ресурсам
Количество VT не заменяет ограничения на соединения к БД, внешние сервисы и одновременно выполняемые вычисления