

Виртуальные войны обедающих философов

**Joker
2025**



Владимир Красильщик / Pragmatic Java Developer

01.tech

О себе

С 2003 г.

разработка на java за \$\$\$

С 2013 г.

доклады за



С 2018 г.

преподавание за \$\$\$ +



01.tech

Команде 7 лет, 1K+

10M+ пользователей

Node.js, Java, Kotlin

Латинская Америка, Азия,
Африка, острова Тихого
океана

AWS -> "своё облако"

10K+ RPS, 1K+ Finance
TPS

Мотивация к появлению этого доклада

vert.x против классической многопоточности в JVM
Владимир Красильщик (Независимый эксперт)

HL Saint HighLoad⁺⁺ 2019

Выводы

- ▶ У стандартной java concurrency есть плюсы, минусы и проблемы
- ▶ Есть альтернативные подходы, например, Actor Model
- ▶ Actor Model проще Java Memory Model
- ▶ vert.x — Actor Model, полиглотный, реактивный, создан для highload
- ▶ vert.x — решает классические задачи на многопоточность
- ▶ java concurrency или альтернатива — квалификация и производительность

28

HL Saint HighLoad⁺⁺

Miran data-center Nixys ZABBIX infobip DataArt Selectel ЦФТ DELL EMC

34:56 / 42:51

https://youtu.be/BpjpPrH_0p0c

Мотивация к появлению этого доклада



Интро в vert.x



Решение классических задач
vert.x vs threads



Достоинства vert.x перед
классической java concurrency

Мотивация к появлению этого доклада

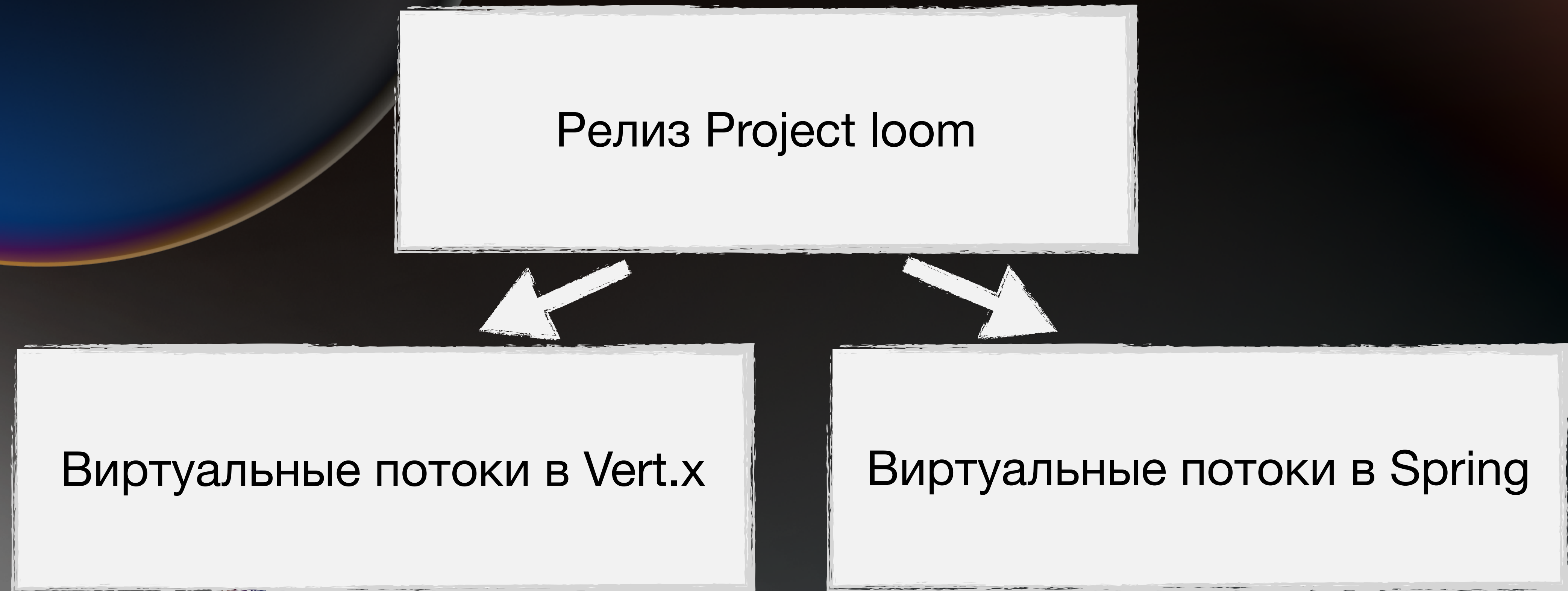


Бенчмарки



Масштабирование

Мотивация к появлению этого доклада



<https://spring.io/blog/2022/10/11/embracing-virtual-threads>

[https://vertx.io/docs/5.0.1/vertx-core/java/#virtual threads](https://vertx.io/docs/5.0.1/vertx-core/java/#virtual_threads)

Просто поставили `spring.threads.virtual.enabled=true` и ...

... и стало быстрее в 2 раза

... и ничего не изменилось

... и стало медленнее

Virtual threads are not faster threads — they do not run code any faster than platform threads. They exist to provide scale (higher throughput), not speed (lower latency). There can be many more of them than platform threads, so they enable the higher concurrency needed for higher throughput according to Little's Law.

<https://openjdk.org/jeps/444>

To put it another way, virtual threads can significantly improve application throughput when

- *The number of concurrent tasks is high (more than a few thousand)*
- *The workload is not CPU-bound, since having many more threads than processor cores cannot improve throughput in that case.*

<https://openjdk.org/jeps/444>

Ожидания от доклада

Ожидания от доклада

1

Измерение и сравнение
производительности решений

Ожидания от доклада

1

Измерение и сравнение
производительности решений

2

Виртуальные потоки против
платформенных

Ожидания от доклада

1

Измерение и сравнение
производительности решений

2

Виртуальные потоки против
платформенных

3

ReentrantLock против
synchronized

Ожидания от доклада

1

Измерение и сравнение
производительности решений

2

Виртуальные потоки против
платформенных

3

ReentrantLock против
synchronized

4

Sleep vs active waiting
vs blocking call

Ожидания от доклада

1

Измерение и сравнение
производительности решений

2

Виртуальные потоки против
платформенных

3

ReentrantLock против
synchronized

4

Sleep vs active waiting
vs blocking call

5

jmh vs jlbh

Ожидания от доклада

1

Измерение и сравнение
производительности решений

2

Виртуальные потоки против
платформенных

3

ReentrantLock против
synchronized

4

Sleep vs active waiting
vs blocking call

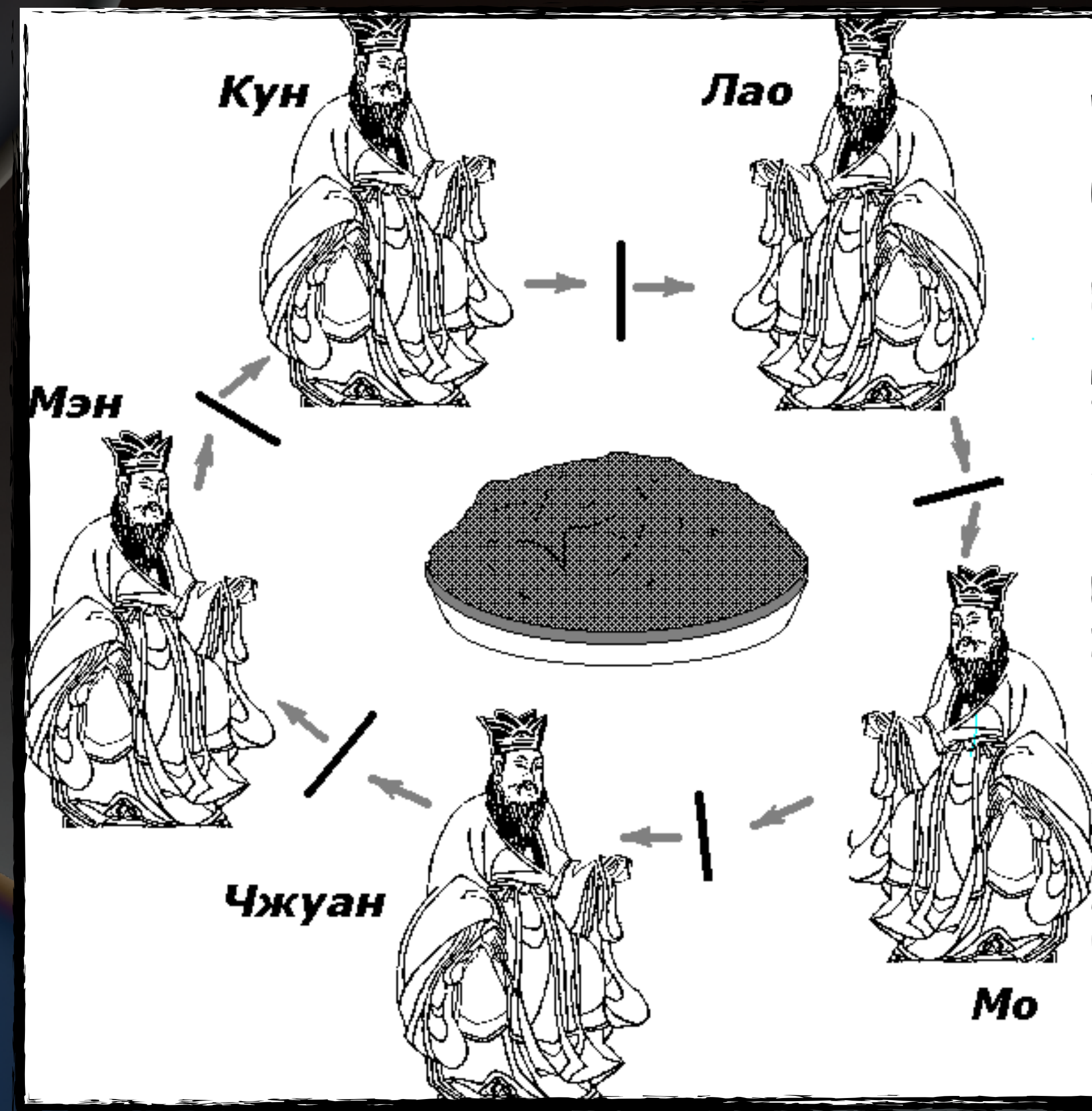
5

jmh vs jlbh

6

Анализ результатов,
неожиданные находки и
полезные выводы

Задача об обедающих философах



Почему обедающие философы?

Почему обедающие философы?

Хорошо масштабируются

Почему обедающие философы?

Хорошо масштабируются

Похожи на явную синхронизацию в
enterprise приложениях

Почему обедающие философы?

Хорошо масштабируются

Похожи на явную синхронизацию в
enterprise приложениях

Похожи на скрытую синхронизацию в
stdlib и 3rd parties

Обедающие философы, structured concurrency

```
CyclicBarrier barrier = new CyclicBarrier(1 + PHILOSOPHERS_COUNT);           // 1

ThreadFactory factory = Thread.ofPlatform().factory();                       // 2

try (ShutdownOnSuccess<Integer> scope = new ShutdownOnSuccess<>(null, factory)) { // 3
    for (int i = 0; i < PHILOSOPHERS_COUNT; i++) {
        Chopstick first = chopsticks.get(i);
        Chopstick second = chopsticks.get(i != 0 ? i - 1 : PHILOSOPHERS_COUNT - 1);
        scope.fork(new ReentrantPhilosopher(i, first, second, barrier));      // 4
    }
    barrier.await();                                                           // 5
    scope.join();                                                              // 6
}
```


Обедающие философы, structured concurrency

```
CyclicBarrier barrier = new CyclicBarrier(1 + PHILOSOPHERS_COUNT);           // 1

ThreadFactory factory = Thread.ofPlatform().factory();                       // 2

try (ShutdownOnSuccess<Integer> scope = new ShutdownOnSuccess<>(null, factory)) { // 3
    for (int i = 0; i < PHILOSOPHERS_COUNT; i++) {
        Chopstick first = chopsticks.get(i);
        Chopstick second = chopsticks.get(i != 0 ? i - 1 : PHILOSOPHERS_COUNT - 1);
        scope.fork(new ReentrantPhilosopher(i, first, second, barrier));      // 4
    }
    barrier.await();                                                           // 5
    scope.join();                                                              // 6
}
```


Обедающие философы, structured concurrency

```
CyclicBarrier barrier = new CyclicBarrier(1 + PHILOSOPHERS_COUNT); // 1

ThreadFactory factory = Thread.ofPlatform().factory(); // 2

try (ShutdownOnSuccess<Integer> scope = new ShutdownOnSuccess<>(null, factory)) { // 3
    for (int i = 0; i < PHILOSOPHERS_COUNT; i++) {
        Chopstick first = chopsticks.get(i);
        Chopstick second = chopsticks.get(i != 0 ? i - 1 : PHILOSOPHERS_COUNT - 1);
        scope.fork(new ReentrantPhilosopher(i, first, second, barrier)); // 4
    }
    barrier.await(); // 5
    scope.join(); // 6
}
```


Обедающие философы, structured concurrency

```
CyclicBarrier barrier = new CyclicBarrier(1 + PHILOSOPHERS_COUNT); // 1

ThreadFactory factory = Thread.ofPlatform().factory(); // 2

try (ShutdownOnSuccess<Integer> scope = new ShutdownOnSuccess<>(null, factory)) { // 3
    for (int i = 0; i < PHILOSOPHERS_COUNT; i++) {
        Chopstick first = chopsticks.get(i);
        Chopstick second = chopsticks.get(i != 0 ? i - 1 : PHILOSOPHERS_COUNT - 1);
        scope.fork(new ReentrantPhilosopher(i, first, second, barrier)); // 4
    }
    barrier.await(); // 5
    scope.join(); // 6
}
```


Обедающие философы, structured concurrency

```
CyclicBarrier barrier = new CyclicBarrier(1 + PHILOSOPHERS_COUNT); // 1

ThreadFactory factory = Thread.ofPlatform().factory(); // 2

try (ShutdownOnSuccess<Integer> scope = new ShutdownOnSuccess<>(null, factory)) { // 3
    for (int i = 0; i < PHILOSOPHERS_COUNT; i++) {
        Chopstick first = chopsticks.get(i);
        Chopstick second = chopsticks.get(i != 0 ? i - 1 : PHILOSOPHERS_COUNT - 1);
        scope.fork(new ReentrantPhilosopher(i, first, second, barrier)); // 4
    }
    barrier.await(); // 5
    scope.join(); // 6
}
```

Обедающие философы, structured concurrency

```
CyclicBarrier barrier = new CyclicBarrier(1 + PHILOSOPHERS_COUNT); // 1

ThreadFactory factory = Thread.ofPlatform().factory(); // 2

try (ShutdownOnSuccess<Integer> scope = new ShutdownOnSuccess<>(null, factory)) { // 3
    for (int i = 0; i < PHILOSOPHERS_COUNT; i++) {
        Chopstick first = chopsticks.get(i);
        Chopstick second = chopsticks.get(i != 0 ? i - 1 : PHILOSOPHERS_COUNT - 1);
        scope.fork(new ReentrantPhilosopher(i, first, second, barrier)); // 4
    }
    barrier.await(); // 5
    scope.join(); // 6
}
```


Философ на ReentrantLock

```
public class ReentrantPhilosopher implements Callable<Integer> {  
    public Integer call() {  
        barrier.await(); // 1  
        while (!Thread.currentThread().isInterrupted() // 2  
            && MAX_EAT_ATTEMPTS > myEatingAttempts) { // 3  
            firstChopstick.lock();  
            try {  
                secondChopstick.lock();  
                try {  
                    eat(myEatingAttempts); // 4  
                } finally {  
                    secondChopstick.unlock();  
                }  
            } finally {  
                firstChopstick.unlock();  
            }  
        }  
        return myEatingAttempts;  
    }  
}
```

Философ на synchronized

```
public class SynchronizedPhilosopher implements Callable<Integer> {  
  
    public Integer call() {  
        barrier.await();  
        while (!currentThread().isInterrupted()  
            && MAX_EAT_ATTEMPTS > myEatingAttempts) {  
            synchronized (firstChopstick) {  
                synchronized (secondChopstick) {  
                    eat(myEatingAttempts);  
                }  
            }  
        }  
        return myEatingAttempts;  
    }  
}
```


Тестовое окружение

- 1) macOS 15.5, 2.6 GHz 6-Core Intel Core i7, 16 GB, APPLE SSD AP0512N
- 2) openjdk-24.0.1, --enable-preview
- 3) 4064 max native threads
- 4) 1K конкурентных философов
- 5) 10K максимальных попыток поесть
- 6) jmh: 1 fork, 3 warmup, 7 iterations
- 7) jlbh: 1op/s, 21 warmup, 3 run, 7 iterations, 90 percentile

junit, NOOP

```
@RepeatedTest(200)
void test_reentrant_lock_philosophers_with_platform_threads() {
    test_reentrant_lock_philosophers_internal(Thread.ofPlatform().factory());
}

@RepeatedTest(200)
void test_reentrant_lock_philosophers_with_virtual_threads() {
    test_reentrant_lock_philosophers_internal(Thread.ofVirtual().factory());
}

@RepeatedTest(200)
void test_synchronized_philosophers_with_virtual_threads() {
    test_synchronized_philosophers_internal(Thread.ofVirtual().factory());
}

@RepeatedTest(200)
void test_synchronized_philosophers_with_platform_threads() {
    test_synchronized_philosophers_internal(Thread.ofPlatform().factory());
}
```


junit, NOOP, results

SynchronizedPhilosophersTest		22.17 s
■	test_synchronized_philosophers_with_platform_threads()	21.06 s
■	test_synchronized_philosophers_with_virtual_threads()	1.11 s
ReentrantLockPhilosophersTest		28.51 s
■	test_reentrant_lock_philosophers_with_platform_threads()	27.97 s
■	test_reentrant_lock_philosophers_with_virtual_threads()	541 ms

junit, NOOP, results

SynchronizedPhilosophersTest

22.17 s

■ test_synchronized_philosophers_with_platform_threads()

21.06 s

■ test_synchronized_philosophers_with_virtual_threads()

1.11 s

ReentrantLockPhilosophersTest

28.51 s

■ test_reentrant_lock_philosophers_with_platform_threads()

27.97 s

■ test_reentrant_lock_philosophers_with_virtual_threads()

541 ms

junit, NOOP, results

SynchronizedPhilosophersTest

22.17 s

■ test_synchronized_philosophers_with_platform_threads()

21.06 s

■ test_synchronized_philosophers_with_virtual_threads()

1.11 s

ReentrantLockPhilosophersTest

28.51 s

■ test_reentrant_lock_philosophers_with_platform_threads()

27.97 s

■ test_reentrant_lock_philosophers_with_virtual_threads()

541 ms

jmh, NOOP

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
public class NoopPhilosophersBenchmark {
    @Benchmark
    public void reentrant_noop_philosophers_with_virtual_threads(Blackhole blackhole) {
        test_philosophers_internal(Thread.ofVirtual().factory(), ReentrantPhilosopher::from);
    }
    @Benchmark
    public void reentrant_noop_philosophers_with_platform_threads(Blackhole blackhole) {
        test_philosophers_internal(Thread.ofPlatform().factory(), ReentrantPhilosopher::from);
    }
    @Benchmark
    public void synchronized_noop_philosophers_with_virtual_threads(Blackhole blackhole) {
        test_philosophers_internal(Thread.ofVirtual().factory(), SynchronizedPhilosopher::from);
    }
    @Benchmark
    public void synchronized_noop_philosophers_with_platform_threads(Blackhole blackhole) {
        test_philosophers_internal(Thread.ofPlatform().factory(), SynchronizedPhilosopher::from);
    }
}
```


jmh, NOOP, results

Benchmark		Score	Error	Units
reentrant	virtual	2.088	0.122	ms/op
synchronized	virtual	2.235	0.100	ms/op
synchronized	platform	114.038	16.327	ms/op
reentrant	platform	137.708	13.519	ms/op

jmh, NOOP, results



Benchmark		Score	Error	Units
reentrant	virtual	2.088	0.122	ms/op
synchronized	virtual	2.235	0.100	ms/op
synchronized	platform	114.038	16.327	ms/op
reentrant	platform	137.708	13.519	ms/op

Альтернативы NOOP

Альтернативы NOOP

1

```
Thread.sleep()
```


Альтернативы NOOP

1

`Thread.sleep()`

2

Активное ожидание:

`while(true){...}`

Альтернативы NOOP

1

`Thread.sleep()`

2

Активное ожидание:
`while(true){...}`

3

Блокирующий вызов:
- диск
- сеть
- `BlockingQueue`

Альтернативы NOOP

1

`Thread.sleep()`

2

Активное ожидание:
`while(true){...}`

3

Блокирующий вызов:
- диск
- сеть
- ~~BlockingQueue~~

Альтернативы NOOP

Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1
Round trip within same datacenter	500,000	0.5
Read 1 MB sequentially from disk	5,000,000	5
Read 1 MB sequentially from 1Gbps network	10,000,000	10
Disk seek	10,000,000	10
TCP packet round trip between continents	150,000,000	150

<https://gist.github.com/jboner/2841832>

<https://static.googleusercontent.com/media/sre.google/en//static/pdf/rule-of-thumb-latency-numbers-letter.pdf>

Альтернативы NOOP

Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1
Round trip within same datacenter	500,000	0.5
Read 1 MB sequentially from disk	5,000,000	5
Read 1 MB sequentially from 1Gbps network	10,000,000	10
Disk seek	10,000,000	10
TCP packet round trip between continents	150,000,000	150

<https://gist.github.com/jboner/2841832>

<https://static.googleusercontent.com/media/sre.google/en//static/pdf/rule-of-thumb-latency-numbers-letter.pdf>

Альтернативы NOOP

Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1
Round trip within same datacenter	500,000	0.5
Read 1 MB sequentially from disk	5,000,000	5
Read 1 MB sequentially from 1Gbps network	10,000,000	10
Disk seek	10,000,000	10
TCP packet round trip between continents	150,000,000	150

16KB -> 16K nanos

<https://gist.github.com/jboner/2841832>

<https://static.googleusercontent.com/media/sre.google/en//static/pdf/rule-of-thumb-latency-numbers-letter.pdf>

jmh, sleeping

```
private static Consumer<Integer> eating(Blackhole blackhole) {  
    return (myEatingAttempts) → {  
        try {  
            Thread.sleep(Duration.ofNanos(16_000));  
            blackhole.consume(myEatingAttempts);  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        }  
    };  
}
```

jmh, sleeping, results

Benchmark		Score	Error	Units
reentrant	platform	619.487	57.005	ms/op
synchronized	platform	695.648	47.757	ms/op
synchronized	virtual	743.421	111.426	ms/op
reentrant	virtual	757.324	85.793	ms/op

jmh, active waiting

```
private static Consumer<Integer> eating(Blackhole blackhole) {  
    return (myEatingAttempts) → {  
        long startTimeInNanos = System.nanoTime();  
        long currentTimeInNanos;  
        do {  
            currentTimeInNanos = System.nanoTime();  
        } while (currentTimeInNanos < startTimeInNanos + 16_000);  
        blackhole.consume(currentTimeInNanos);  
        blackhole.consume(myEatingAttempts);  
    };  
}
```

jmh, active waiting, results

Benchmark		Score	Error	Units
synchronized	virtual	170.812	7.355	ms/op
reentrant	virtual	178.575	16.057	ms/op
synchronized	platform	303.367	12.389	ms/op
reentrant	platform	416.446	92.891	ms/op

jmh, disk URL

```
private static Consumer<Integer> eating(Blackhole blackhole) {  
    return (myEatingAttempts) → {  
        try (InputStream in = Path.of("16KB_file.txt")  
            .toFile()  
            .toURI()  
            .toURL()  
            .openStream()) {  
            byte[] bytes = in.readAllBytes();  
            blackhole.consume(bytes.length);  
            blackhole.consume(myEatingAttempts);  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    };  
}
```

jmh, disk URL, results

Benchmark		Score	Error	Units
reentrant	platform	585.240	94.912	ms/op
synchronized	platform	906.057	107.330	ms/op
reentrant	virtual	3610.834	744.468	ms/op
synchronized	virtual	17666.041	5157.029	ms/op

jmh, disk URL, results

Benchmark		Score	Error	Units
reentrant	platform	585.240	94.912	ms/op
synchronized	platform	906.057	107.330	ms/op
reentrant	virtual	3610.834	744.468	ms/op
synchronized	virtual	17666.041	5157.029	ms/op

jmh, FileInputStream

```
byte[] buffer = new byte[1024];
try (FileInputStream fis = new FileInputStream(new File("16KB_file.txt"))) {
    StringBuilder sb = new StringBuilder();
    int bytesRead;
    while ((bytesRead = fis.read(buffer)) != -1) {
        sb.append(new String(buffer, 0, bytesRead)).append("\n");
    }
    blackhole.consume(sb.length());
    blackhole.consume(myEatingAttempts);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```


jmh, FileInputStream, results

Benchmark		Score ± Error		Units
synchronized	platform	608,610	± 90,857	ms/op
reentrant	platform	542,769	± 27,046	ms/op
reentrant	virtual	51781,102	± 37847,544	ms/op
synchronized	virtual	64765,792	± 23449,206	ms/op

jmh, BufferedReader

```
try (BufferedReader bufferedReader =  
    new BufferedReader(new FileReader("16KB_file.txt"))) {  
    StringBuilder sb = new StringBuilder();  
    bufferedReader.lines().forEach(line → sb.append(line).append("\n"));  
    blackhole.consume(sb.length());  
    blackhole.consume(myEatingAttempts);  
} catch (IOException e) {  
    throw new RuntimeException(e);  
}
```


jmh, BufferedReader, results

Benchmark		Score ± Error	Units
reentrant	platform	934,681 ± 180,904	ms/op
synchronized	platform	1123,311 ± 108,851	ms/op
reentrant	virtual	8576,049 ± 1838,726	ms/op
synchronized	virtual	16701,601 ± 6824,735	ms/op

jmh, Scanner

```
try (Scanner scanner = new Scanner(new File("16KB_file.txt"))) {
    StringBuilder sb = new StringBuilder();
    while (scanner.hasNextLine()) {
        sb.append(scanner.nextLine()).append("\n");
    }
    blackhole.consume(sb.length());
    blackhole.consume(myEatingAttempts);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```


jmh, Scanner, results

Benchmark		Score ± Error	Units
synchronized	platform	3317,680 ± 592,888	ms/op
reentrant	platform	3797,203 ± 690,945	ms/op
reentrant	virtual	4008,147 ± 12499,175	ms/op
synchronized	virtual	2192,711 ± 1602,304	ms/op

jmh, Scanner, results

Benchmark		Score $\pm$ Error		Units
synchronized	platform	3317,680	$\pm$ 592,888	ms/op
reentrant	platform	3797,203	$\pm$ 690,945	ms/op
reentrant	virtual	4008,147	$\pm$ 12499,175	ms/op
synchronized	virtual	2192,711	$\pm$ 1602,304	ms/op

jmh, NIO, Files.readAllBytes

```
try {  
    byte[] bytes = Files.readAllBytes(Path.of("16KB_file.txt"));  
    blackhole.consume(bytes.length);  
    blackhole.consume(myEatingAttempts);  
} catch (IOException e) {  
    throw new RuntimeException(e);  
}
```

jmh, NIO, Files.readAllBytes, results

Benchmark		Score ± Error		Units
synchronized	platform	566.508	± 184.385	ms/op
reentrant	platform	677.767	± 46.845	ms/op
reentrant	virtual	1866.279	± 283.192	ms/op
synchronized	virtual	10129.351	± 1231.205	ms/op

jmh, NIO, Files.newBufferedReader

```
BufferedReader bufferedReader =  
    Files.newBufferedReader(Path.of("16KB_file.txt"));  
StringBuilder sb = new StringBuilder();  
String line;  
while ((line = bufferedReader.readLine()) != null) {  
    sb.append(line).append("\n");  
}
```

jmh, NIO, Files.newBufferedReader, results

Benchmark	Score ± Error	Units
synchronized platform	755,267 ± 83,059	ms/op
reentrant platform	792,351 ± 202,390	ms/op
reentrant virtual	1995,975 ± 384,531	ms/op
synchronized virtual	2176,248 ± 306,216	ms/op

jmh, NIO, FileChannel.map

```
try (FileChannel fc =
    new RandomAccessFile(new File("16KB_file.txt"), "rw").getChannel();) {
    int bufferSize = 8 * 1024;
    MappedByteBuffer mem = fc.map(FileChannel.MapMode.READ_ONLY, 0, bufferSize);
    byte[] dst = new byte[bufferSize];
    mem.get(dst, 0, bufferSize);
    blackhole.consume(new String(dst).length());
    blackhole.consume(myEatingAttempts);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

jmh, NIO, FileChannel.map, results

Benchmark		Score ± Error		Units
synchronized	platform	1872,335	± 4661,646	ms/op
reentrant	platform	1452,701	± 5599,624	ms/op
reentrant	virtual	103594,134	± 255428,092	ms/op
synchronized	virtual	1050017,245	± 27304315,889	ms/op

Может jmh уже не торт? jlbh в студию!

<https://github.com/OpenHFT/JLBH>

<http://www.rationaljava.com/2016/04/jlbh-introducing-java-latency.html>

jlbh (specjbb)

```
public class JlbhBenchmark implements JLBHTask {
    private JLBH jlbh; // 1
    private NanoSampler eating; // 2
    ...
    public static void main(String[] args) {
        JLBHOptions jlbh = new JLBHOptions()
            .warmUpIterations(21) // 3
            .iterations(7) // 4
            .throughput(1) // 5
            .runs(3) // 6
            .recordOSJitter(true) // 7
            .accountForCoordinatedOmission(false) // 8
            .jlbhTask(new JlbhBenchmark());
        new JLBH(jlbh).start();
    }
}
```


jlbh (specjbb)

```
public class JlbhBenchmark implements JLBHTask {
    private JLBH jlbh; // 1
    private NanoSampler eating; // 2
    ...
    public static void main(String[] args) {
        JLBHOptions jlbh = new JLBHOptions()
            .warmUpIterations(21) // 3
            .iterations(7) // 4
            .throughput(1) // 5
            .runs(3) // 6
            .recordOSJitter(true) // 7
            .accountForCoordinatedOmission(false) // 8
            .jlbhTask(new JlbhBenchmark());
        new JLBH(jlbh).start();
    }
}
```

jlbh (specjbb)

```
public class JlbhBenchmark implements JLBHTask {
    private JLBH jlbh; // 1
    private NanoSampler eating; // 2
    ...
    public static void main(String[] args) {
        JLBHOptions jlbh = new JLBHOptions()
            .warmUpIterations(21) // 3
            .iterations(7) // 4
            .throughput(1) // 5
            .runs(3) // 6
            .recordOSJitter(true) // 7
            .accountForCoordinatedOmission(false) // 8
            .jlbhTask(new JlbhBenchmark());
        new JLBH(jlbh).start();
    }
}
```


jlbh (specjbb)

```
@Override
public void init(JLBH jlbh) {
    this.jlbh = jlbh;
    this.eatingSampler = jlbh.addProbe("eating"); // 1
}
@Override
public void run(long startTimeNS) { // 2
    try {
        test_reading_philosophers(eatingSampler); // 3
        jlbh.sample(System.nanoTime() - startTimeNS); // 4
    } catch (InterruptedException | BrokenBarrierException e) {
        throw new RuntimeException(e);
    }
}
```

jlbh (specjbb)

```
try (ShutdownOnSuccess<Integer> scope = new ShutdownOnSuccess<>(null, factory)) {  
    philosophers.forEach(scope::fork);           // 1  
    barrier.await();                             // 2  
    long eatingStart = System.nanoTime();        // 3  
    scope.join();                                 // 4  
    eatingSampler.sampleNanos(System.nanoTime() - eatingStart); // 5  
}
```


jlbh (specjbb)

```
try (ShutdownOnSuccess<Integer> scope = new ShutdownOnSuccess<>(null, factory)) {  
    philosophers.forEach(scope::fork); // 1  
    barrier.await(); // 2  
    long eatingStart = System.nanoTime(); // 3  
    scope.join(); // 4  
    eatingSampler.sampleNanos(System.nanoTime() - eatingStart); // 5  
}
```

jlbh (specjbb)

```
...
...
----- BENCHMARK RESULTS (RUN 3) us -----
Run time: 11.02 s, distribution: NORMAL
Correcting for co-ordinated:false
Target throughput:1/s = 1 message every 1000000us
End to End: (7) 50/90 99/99.9 99.99 - worst was 544,210 / 760,220 760,220 / 760,220 760,220 - 760,220
eating with ReentrantLock+PlatformThreads (7) 50/90 99/99.9 99.99 - worst was 340,260 / 357,040 357,040 / 357,040 357,040 - 357,040
OS Jitter (32,782) 50/90 99/99.9 99.99 - worst was 13.62 / 29.2 87.2 / 1419 5,680 - 33,520
-----
----- SUMMARY (end to end) us -----
Percentile run1 run2 run3 % Variation
50.0: 896532.48 607125.50 544210.94 7.16
90.0: 1197473.79 881852.42 760217.60 9.64
99.0: 1197473.79 881852.42 760217.60 9.64
worst: 1197473.79 881852.42 760217.60 9.64
-----
----- SUMMARY (eating with ReentrantLock+PlatformThreads) us -----
Percentile run1 run2 run3 % Variation
50.0: 542113.79 345505.79 340262.91 1.02
90.0: 1026555.90 542113.79 357040.13 25.68
99.0: 1026555.90 542113.79 357040.13 25.68
worst: 1026555.90 542113.79 357040.13 25.68
-----
```


jlbh (specjbb)

```
...
...
----- BENCHMARK RESULTS (RUN 3) us -----
Run time: 11.02 s, distribution: NORMAL
Correcting for co-ordinated:false
Target throughput:1/s = 1 message every 1000000us
End to End: (7) 50/90 99/99.9 99.99 - worst was 544,210 / 760,220 760,220 / 760,220 760,220 - 760,220
eating with ReentrantLock+PlatformThreads (7) 50/90 99/99.9 99.99 - worst was 340,260 / 357,040 357,040 / 357,040 357,040 - 357,040
OS Jitter (32,782) 50/90 99/99.9 99.99 - worst was 13.62 / 29.2 87.2 / 1419 5,680 - 33,520
-----
----- SUMMARY (end to end) us -----
Percentile run1 run2 run3 % Variation
50.0: 896532.48 607125.50 544210.94 7.16
90.0: 1197473.79 881852.42 760217.60 9.64
99.0: 1197473.79 881852.42 760217.60 9.64
worst: 1197473.79 881852.42 760217.60 9.64
-----
----- SUMMARY (eating with ReentrantLock+PlatformThreads) us -----
Percentile run1 run2 run3 % Variation
50.0: 542113.79 345505.79 340262.91 1.02
90.0: 1026555.90 542113.79 357040.13 25.68
99.0: 1026555.90 542113.79 357040.13 25.68
worst: 1026555.90 542113.79 357040.13 25.68
-----
```

jlbh (specjbb)

```
...
...
----- BENCHMARK RESULTS (RUN 3) us -----
Run time: 11.02 s, distribution: NORMAL
Correcting for co-ordinated:false
Target throughput:1/s = 1 message every 1000000us
End to End: (7) 50/90 99/99.9 99.99 - worst was 544,210 / 760,220 760,220 / 760,220 760,220 - 760,220
eating with ReentrantLock+PlatformThreads (7) 50/90 99/99.9 99.99 - worst was 340,260 / 357,040 357,040 / 357,040 357,040 - 357,040
OS Jitter (32,782) 50/90 99/99.9 99.99 - worst was 13.62 / 29.2 87.2 / 1419 5,680 - 33,520
-----
SUMMARY (end to end) us -----
Percentile run1 run2 run3 % Variation
50.0: 896532.48 607125.50 544210.94 7.16
90.0: 1197473.79 881852.42 760217.60 9.64
99.0: 1197473.79 881852.42 760217.60 9.64
worst: 1197473.79 881852.42 760217.60 9.64
-----
SUMMARY (eating with ReentrantLock+PlatformThreads) us -----
Percentile run1 run2 run3 % Variation
50.0: 542113.79 345505.79 340262.91 1.02
90.0: 1026555.90 542113.79 357040.13 25.68
99.0: 1026555.90 542113.79 357040.13 25.68
worst: 1026555.90 542113.79 357040.13 25.68
-----
```


jlbh (specjbb)

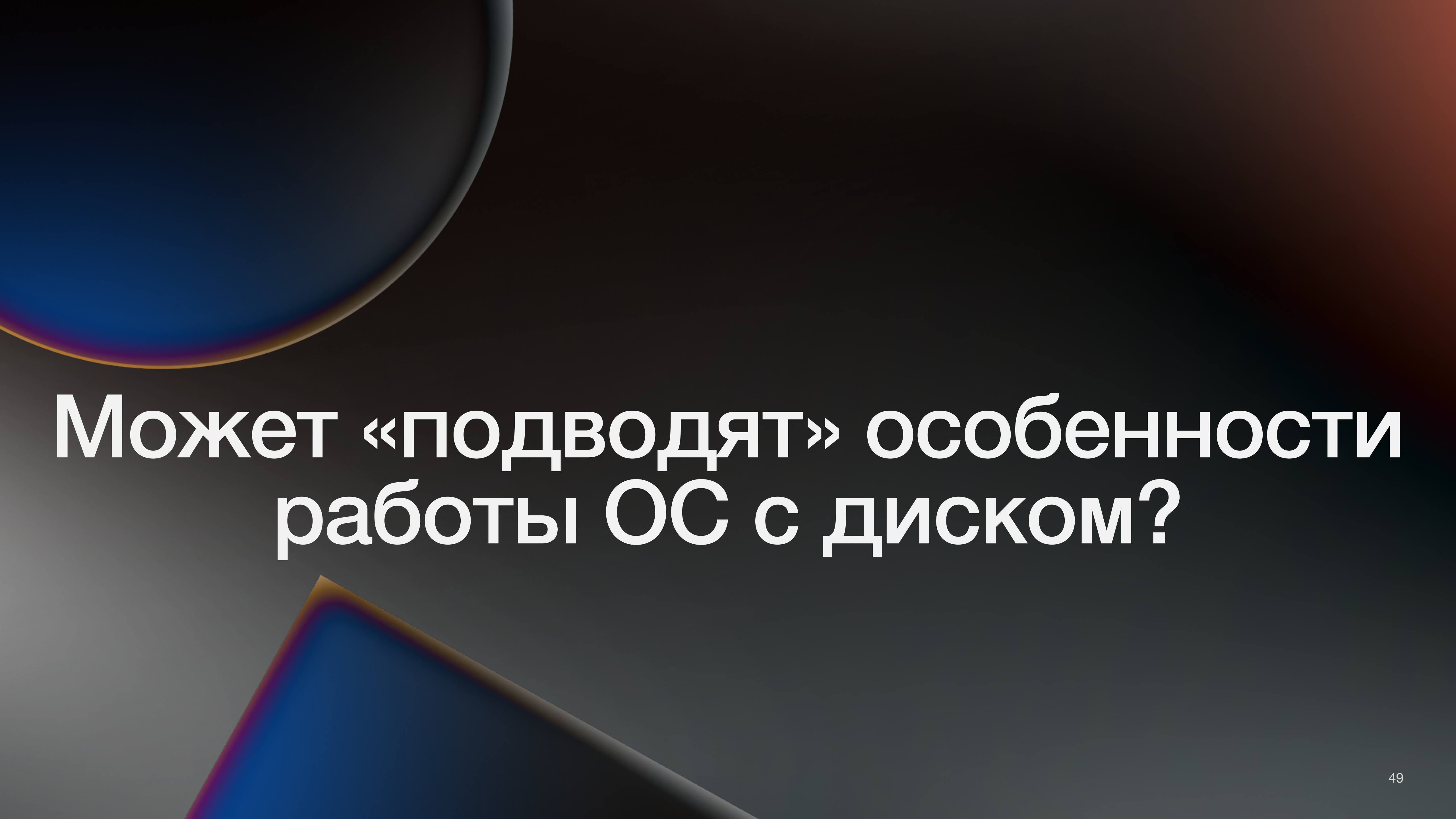
```
...
...
----- BENCHMARK RESULTS (RUN 3) us -----
Run time: 11.02 s, distribution: NORMAL
Correcting for co-ordinated:false
Target throughput:1/s = 1 message every 1000000us
End to End: (7) 50/90 99/99.9 99.99 - worst was 544,210 / 760,220 760,220 / 760,220 760,220 - 760,220
eating with ReentrantLock+PlatformThreads (7) 50/90 99/99.9 99.99 - worst was 340,260 / 357,040 357,040 / 357,040 357,040 - 357,040
OS Jitter (32,782) 50/90 99/99.9 99.99 - worst was 13.62 / 29.2 87.2 / 1419 5,680 - 33,520
-----
```

----- SUMMARY (end to end) us -----				
Percentile	run1	run2	run3	% Variation
50.0:	896532.48	607125.50	544210.94	7.16
90.0:	1197473.79	881852.42	760217.60	9.64
99.0:	1197473.79	881852.42	760217.60	9.64
worst:	1197473.79	881852.42	760217.60	9.64

----- SUMMARY (eating with ReentrantLock+PlatformThreads) us -----				
Percentile	run1	run2	run3	% Variation
50.0:	542113.79	345505.79	340262.91	1.02
90.0:	1026555.90	542113.79	357040.13	25.68
99.0:	1026555.90	542113.79	357040.13	25.68
worst:	1026555.90	542113.79	357040.13	25.68

jlbh (specjbb), disk URL, results

jlbh benchmark		jmh: лучший run	jlbh: лучший 90%
reentrant	platform	585.240 ms	760.217 ms
synchronized	platform	906.057 ms	877.658 ms
reentrant	virtual	3610.834 ms	7138.705 ms
synchronized	virtual	17666.041 ms	14445.182 ms



Может «подводят» особенности
работы ОС с диском?

Альтернативы блокирующему чтению с диска

Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1
Round trip within same datacenter	500,000	0.5
Read 1 MB sequentially from disk	5,000,000	5
Read 1 MB sequentially from 1Gbps network	10,000,000	10
Disk seek	10,000,000	10
TCP packet round trip between continents	150,000,000	150

<https://gist.github.com/jboner/2841832>

<https://static.googleusercontent.com/media/sre.google/en//static/pdf/rule-of-thumb-latency-numbers-letter.pdf>

Альтернативы блокирующему чтению с диска

Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1
Round trip within same datacenter	500,000	0.5
Read 1 MB sequentially from disk	5,000,000	5
Read 1 MB sequentially from 1Gbps network	10,000,000	10
Disk seek	10,000,000	10
TCP packet round trip between continents	150,000,000	150

<https://gist.github.com/jboner/2841832>

<https://static.googleusercontent.com/media/sre.google/en//static/pdf/rule-of-thumb-latency-numbers-letter.pdf>

Альтернативы блокирующему чтению с диска

Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1
Round trip within same datacenter	500,000	0.5
Read 1 MB sequentially from disk	5,000,000	5
Read 1 MB sequentially from 1Gbps network	10,000,000	10
Disk seek	10,000,000	10
TCP packet round trip between continents	150,000,000	150

16K nanos -> 1.6KB

<https://gist.github.com/jboner/2841832>

<https://static.googleusercontent.com/media/sre.google/en//static/pdf/rule-of-thumb-latency-numbers-letter.pdf>

jmh, http

```
private static Consumer<Integer> eating(Blackhole blackhole) {  
    return (myEatingAttempts) → {  
        try (var in = URI.create("http://127.0.0.1:" + port + "/payload")  
            .toURL()  
            .openStream()) {  
            int length = in.readAllBytes().length;  
            blackhole.consume(length);  
            blackhole.consume(myEatingAttempts);  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    };  
}
```

jmh, http, results

Benchmark	Score $\pm$ Error	Units
synchronized platform	1126.106 $\pm$ 177.126	ms/op
reentrant platform	1195.433 $\pm$ 211.757	ms/op
reentrant virtual	1668.852 $\pm$ 1178.063	ms/op
synchronized virtual	6924.583 $\pm$ 1950.337	ms/op

jmh, http, results

Benchmark	Score ± Error	Units
synchronized platform	1126.106 ± 177.126	ms/op
reentrant platform	1195.433 ± 211.757	ms/op
reentrant virtual	1668.852 ± 1178.063	ms/op
synchronized virtual	6924.583 ± 1950.337	ms/op

Может «подводит» особенность реализации `URL.openStream()`?


```

void handle(Request request, Response response) {
    var url1 = ...
    var url2 = ...

    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        var future1 = executor.submit(() -> fetchURL(url1));
        var future2 = executor.submit(() -> fetchURL(url2));
        response.send(future1.get() + future2.get());
    } catch (ExecutionException | InterruptedException e) {
        response.fail(e);
    }
}

String fetchURL(URL url) throws IOException {
    try (var in = url.openStream()) {
        return new String(in.readAllBytes(), StandardCharsets.UTF_8);
    }
}

```



```

void handle(Request request, Response response) {
    var url1 = ...
    var url2 = ...

    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        var future1 = executor.submit(() -> fetchURL(url1));
        var future2 = executor.submit(() -> fetchURL(url2));
        response.send(future1.get() + future2.get());
    } catch (ExecutionException | InterruptedException e) {
        response.fail(e);
    }
}

String fetchURL(URL url) throws IOException {
    try (var in = url.openStream()) {
        return new String(in.readAllBytes(), StandardCharsets.UTF_8);
    }
}

```


jmh, okhttp

```
URL url = URI.create("http://127.0.0.1:" + port + "/payload").toURL();
Request request = new Request.Builder().url(url).build();
try (Response response = client.newCall(request).execute()) {
    String responseAsString = response.body().string();
    blackhole.consume(responseAsString);
    blackhole.consume(myEatingAttempts);
}
```

jmh, okhttp, results

Benchmark		Score $\pm$	Error	Units
reentrant	virtual	21.009 $\pm$	2.779	ms/op
synchronized	virtual	21.153 $\pm$	3.176	ms/op
synchronized	platform	115.089 $\pm$	8.478	ms/op
reentrant	platform	133.544 $\pm$	61.621	ms/op

jmh, okhttp, results



Benchmark		Score	±	Error	Units
reentrant	virtual	21.009	±	2.779	ms/op
synchronized	virtual	21.153	±	3.176	ms/op
synchronized	platform	115.089	±	8.478	ms/op
reentrant	platform	133.544	±	61.621	ms/op

Сводный бенчмарк java concurrency

synchronized	noop	virtual	3.402	±	0.554	ms/op
reentrant	noop	virtual	3.786	±	0.375	ms/op
reentrant	ok_http	virtual	21.268	±	0.866	ms/op
synchronized	ok_http	virtual	21.993	±	2.141	ms/op
synchronized	ok_http	platform	120.036	±	15.304	ms/op
reentrant	ok_http	platform	137.073	±	32.623	ms/op
synchronized	noop	platform	171.570	±	23.510	ms/op
reentrant	active	virtual	198.108	±	17.251	ms/op
synchronized	active	virtual	201.018	±	17.320	ms/op
reentrant	noop	platform	213.722	±	25.279	ms/op
synchronized	active	platform	455.479	±	55.032	ms/op
reentrant	sleeping	platform	609.585	±	59.735	ms/op
synchronized	sleeping	platform	821.443	±	92.263	ms/op
reentrant	sleeping	virtual	872.009	±	134.314	ms/op
reentrant	active	platform	947.969	±	790.412	ms/op
synchronized	sleeping	virtual	1191.576	±	501.751	ms/op

Сводный бенчмарк java concurrency

NOOP виртуальные
философы на 2 порядка
эффективней
платформенных

synchronized	noop	virtual	3.402	±		
reentrant	noop	virtual	3.786	±		
reentrant	ok_http	virtual	21.268	±		
synchronized	ok_http	virtual	21.993	±		
synchronized	ok_http	platform	120.036	±	15.304	ms/op
reentrant	ok_http	platform	137.073	±	32.623	ms/op
synchronized	noop	platform	171.570	±	23.510	ms/op
reentrant	active	virtual	198.108	±	17.251	ms/op
synchronized	active	virtual	201.018	±	17.320	ms/op
reentrant	noop	platform	213.722	±	25.279	ms/op
synchronized	active	platform	455.479	±	55.032	ms/op
reentrant	sleeping	platform	609.585	±	59.735	ms/op
synchronized	sleeping	platform	821.443	±	92.263	ms/op
reentrant	sleeping	virtual	872.009	±	134.314	ms/op
reentrant	active	platform	947.969	±	790.412	ms/op
synchronized	sleeping	virtual	1191.576	±	501.751	ms/op

Сводный бенчмарк java concurrency

Активно ждать на виртуальных потоках в 2 раза лучше чем на платформенных

synchronized	noop	virtual	3.402	±		
reentrant	noop	virtual	3.786	±		
reentrant	ok_http	virtual	21.268	±		
synchronized	ok_http	virtual	21.993	±		
synchronized	ok_http	platform	120.036	±	15.304	ms/op
reentrant	ok_http	platform	137.073	±	32.623	ms/op
synchronized	noop	platform	171.570	±	23.510	ms/op
reentrant	active	virtual	198.108	±	17.251	ms/op
synchronized	active	virtual	201.018	±	17.320	ms/op
reentrant	noop	platform	213.722	±	25.279	ms/op
synchronized	active	platform	455.479	±	55.032	ms/op
reentrant	sleeping	platform	609.585	±	59.735	ms/op
synchronized	sleeping	platform	821.443	±	92.263	ms/op
reentrant	sleeping	virtual	872.009	±	134.314	ms/op
reentrant	active	platform	947.969	±	790.412	ms/op
synchronized	sleeping	virtual	1191.576	±	501.751	ms/op

Сводный бенчмарк java concurrency

Активно ждать в 2 раза
выгоднее чем спать

synchronized	noop	virtual	3.402	±			
reentrant	noop	virtual	3.786	±			
reentrant	ok_http	virtual	21.268	±			
synchronized	ok_http	virtual	21.993	±			
synchronized	ok_http	platform	120.036	±	15.304	ms/op	
reentrant	ok_http	platform	137.073	±	32.623	ms/op	
synchronized	noop	platform	171.570	±	23.510	ms/op	
reentrant	→ active	virtual	198.108	±	17.251	ms/op	
synchronized	→ active	virtual	201.018	±	17.320	ms/op	
reentrant	noop	platform	213.722	±	25.279	ms/op	
synchronized	→ active	platform	455.479	±	55.032	ms/op	
reentrant	sleeping	platform	← 609.585	±	59.735	ms/op	
synchronized	sleeping	platform	← 821.443	±	92.263	ms/op	
reentrant	sleeping	virtual	← 872.009	±	134.314	ms/op	
reentrant	→ active	platform	947.969	±	790.412	ms/op	
synchronized	sleeping	virtual	← 1191.576	±	501.751	ms/op	

**А что если философов будет
не 1К, а 1000К?**

Масштабирование, virtual, NOOP

1K	10K	reentrant	1.952	±	0.188	ms/op
1K	10K	synchronized	2.184	±	0.439	ms/op
10K	10K	synchronized	13.453	±	2.037	ms/op
10K	10K	reentrant	13.790	±	0.852	ms/op
10K	100K	synchronized	20.490	±	0.973	ms/op
10K	100K	reentrant	21.520	±	2.153	ms/op
10K	1000K	synchronized	70.700	±	1.221	ms/op
10K	1000K	reentrant	72.035	±	1.424	ms/op
100K	10K	synchronized	276.019	±	92.038	ms/op
100K	10K	reentrant	284.643	±	35.604	ms/op
100K	100K	synchronized	255.395	±	102.263	ms/op
100K	100K	reentrant	297.571	±	37.570	ms/op
100K	1000K	reentrant	351.140	±	102.412	ms/op
100K	1000K	synchronized	382.512	±	60.712	ms/op
1000K	10K	synchronized	3847.501	±	197.755	ms/op
1000K	10K	reentrant	4307.880	±	402.627	ms/op
1000K	100K	reentrant	3993.247	±	530.418	ms/op
1000K	100K	synchronized	4089.308	±	576.735	ms/op
1000K	1000K	synchronized	3792.618	±	352.711	ms/op
1000K	1000K	reentrant	3931.608	±	460.112	ms/op

Масштабирование, virtual, NOOP

Философы

1K	10K	reentrant	1.952	±	0.188	ms/op
1K	10K	synchronized	2.184	±	0.439	ms/op
10K	10K	synchronized	13.453	±	2.037	ms/op
10K	10K	reentrant	13.790	±	0.852	ms/op
10K	100K	synchronized	20.490	±	0.973	ms/op
10K	100K	reentrant	21.520	±	2.153	ms/op
10K	1000K	synchronized	70.700	±	1.221	ms/op
10K	1000K	reentrant	72.035	±	1.424	ms/op
100K	10K	synchronized	276.019	±	92.038	ms/op
100K	10K	reentrant	284.643	±	35.604	ms/op
100K	100K	synchronized	255.395	±	102.263	ms/op
100K	100K	reentrant	297.571	±	37.570	ms/op
100K	1000K	reentrant	351.140	±	102.412	ms/op
100K	1000K	synchronized	382.512	±	60.712	ms/op
1000K	10K	synchronized	3847.501	±	197.755	ms/op
1000K	10K	reentrant	4307.880	±	402.627	ms/op
1000K	100K	reentrant	3993.247	±	530.418	ms/op
1000K	100K	synchronized	4089.308	±	576.735	ms/op
1000K	1000K	synchronized	3792.618	±	352.711	ms/op
1000K	1000K	reentrant	3931.608	±	460.112	ms/op

Масштабирование, virtual, NOOP

Философы	1K	10K	reentrant	1.952	± 0.188	ms/op
	1K	10K	synchronized	2.184	± 0.439	ms/op
	10K	10K	synchronized	13.453	± 2.037	ms/op
	10K	10K	reentrant	13.790	± 0.852	ms/op
	10K	100K	synchronized	20.490	± 0.973	ms/op
	10K	100K	reentrant	21.520	± 2.153	ms/op
	10K	1000K	synchronized	70.700	± 1.221	ms/op
Кормления	10K	1000K	reentrant	72.035	± 1.424	ms/op
	100K	10K	synchronized	276.019	± 92.038	ms/op
	100K	10K	reentrant	284.643	± 35.604	ms/op
	100K	100K	synchronized	255.395	± 102.263	ms/op
	100K	100K	reentrant	297.571	± 37.570	ms/op
	100K	1000K	reentrant	351.140	± 102.412	ms/op
	100K	1000K	synchronized	382.512	± 60.712	ms/op
	1000K	10K	synchronized	3847.501	± 197.755	ms/op
	1000K	10K	reentrant	4307.880	± 402.627	ms/op
	1000K	100K	reentrant	3993.247	± 530.418	ms/op
	1000K	100K	synchronized	4089.308	± 576.735	ms/op
	1000K	1000K	synchronized	3792.618	± 352.711	ms/op
	1000K	1000K	reentrant	3931.608	± 460.112	ms/op

Масштабирование, virtual, NOOP



Философы	1K	10K	reentrant	1.952	± 0.188	ms/op
	1K	10K	synchronized	2.184	± 0.439	ms/op
	10K	10K	synchronized	13.453	± 2.037	ms/op
	10K	10K	reentrant	13.790	± 0.852	ms/op
	10K	100K	synchronized	20.490	± 0.973	ms/op
	10K	100K	reentrant	21.520	± 2.153	ms/op
Кормления	10K	1000K	synchronized	70.700	± 1.221	ms/op
	10K	1000K	reentrant	72.035	± 1.424	ms/op
	100K	10K	synchronized	276.019	± 92.038	ms/op
	100K	10K	reentrant	284.643	± 35.604	ms/op
	100K	100K	synchronized	255.395	± 102.263	ms/op
	100K	100K	reentrant	297.571	± 37.570	ms/op
	100K	1000K	reentrant	351.140	± 102.412	ms/op
	100K	1000K	synchronized	382.512	± 60.712	ms/op
	1000K	10K	synchronized	3847.501	± 197.755	ms/op
	1000K	10K	reentrant	4307.880	± 402.627	ms/op
	1000K	100K	reentrant	3993.247	± 530.418	ms/op
	1000K	100K	synchronized	4089.308	± 576.735	ms/op
	1000K	1000K	synchronized	3792.618	± 352.711	ms/op
	1000K	1000K	reentrant	3931.608	± 460.112	ms/op

Масштабирование, virtual, okhttp

1K	10K	reentrant	16.639	ms/op
1K	10K	synchronized	12.299	ms/op
10K	10K	reentrant	153.864	ms/op
10K	10K	synchronized	182.394	ms/op
10K	100K	reentrant	150.171	ms/op
10K	100K	synchronized	174.926	ms/op
10K	1000K	reentrant	197.704	ms/op
10K	1000K	synchronized	178.236	ms/op
100K	10K	reentrant	1542.291	ms/op
100K	10K	synchronized	1766.490	ms/op
100K	100K	reentrant	1704.856	ms/op
100K	100K	synchronized	1838.388	ms/op
100K	1000K	reentrant	1703.520	ms/op
100K	1000K	synchronized	1908.303	ms/op
1000K	10K	reentrant	22066.961	ms/op
1000K	10K	synchronized	23521.333	ms/op
1000K	100K	reentrant	25421.230	ms/op
1000K	100K	synchronized	24288.554	ms/op
1000K	1000K	reentrant	26757.401	ms/op
1000K	1000K	synchronized	33848.830	ms/op

Масштабирование, virtual, okhttp

1K	10K	reentrant	16.639	±	78.606	ms/op
1K	10K	synchronized	12.299	±	23.337	ms/op
10K	10K	reentrant	153.864	±	75.793	ms/op
10K	10K	synchronized	182.394	±	536.973	ms/op
10K	100K	reentrant	150.171	±	474.616	ms/op
10K	100K	synchronized	174.926	±	202.954	ms/op
10K	1000K	reentrant	197.704	±	308.298	ms/op
10K	1000K	synchronized	178.236	±	333.246	ms/op
100K	10K	reentrant	1542.291	±	2511.288	ms/op
100K	10K	synchronized	1766.490	±	3443.229	ms/op
100K	100K	reentrant	1704.856	±	3530.813	ms/op
100K	100K	synchronized	1838.388	±	1682.690	ms/op
100K	1000K	reentrant	1703.520	±	3900.612	ms/op
100K	1000K	synchronized	1908.303	±	6017.006	ms/op
1000K	10K	reentrant	22066.961	±	32925.217	ms/op
1000K	10K	synchronized	23521.333	±	55721.010	ms/op
1000K	100K	reentrant	25421.230	±	192726.043	ms/op
1000K	100K	synchronized	24288.554	±	48552.237	ms/op
1000K	1000K	reentrant	26757.401	±	177259.781	ms/op
1000K	1000K	synchronized	33848.830	±	172931.769	ms/op

Масштабирование, virtual, okhttp

1K	10K	reentrant	16.639	±	78.606	ms/op
1K	10K	synchronized	12.299	±	23.337	ms/op
10K	10K	reentrant	153.864	±	75.793	ms/op
10K	10K	synchronized	182.394	±	536.973	ms/op
10K	100K	reentrant	150.171	±	474.616	ms/op
10K	100K	synchronized	174.926	±	202.954	ms/op
10K	1000K	reentrant	197.704	±	308.298	ms/op
10K	1000K	synchronized	178.236	±	333.246	ms/op
100K	10K	reentrant	1542.291	±	2511.288	ms/op
100K	10K	synchronized	1766.490	±	3443.229	ms/op
100K	100K	reentrant	1704.856	±	3530.813	ms/op
100K	100K	synchronized	1838.388	±	1682.690	ms/op
100K	1000K	reentrant	1703.520	±	3900.612	ms/op
100K	1000K	synchronized	1908.303	±	6017.006	ms/op
1000K	10K	reentrant	22066.961	±	32925.217	ms/op
1000K	10K	synchronized	23521.333	±	55721.010	ms/op
1000K	100K	reentrant	25421.230	±	192726.043	ms/op
1000K	100K	synchronized	24288.554	±	48552.237	ms/op
1000K	1000K	reentrant	26757.401	±	177259.781	ms/op
1000K	1000K	synchronized	33848.830	±	172931.769	ms/op



ReentrantLock, okhttp, virtual, jmh vs jlbh

reentrant	jmh	jlbh: лучший 90%
-----:	-----:	-----:
1K 10K	16 ± 78 ms	1700 ms (15%)
10K 10K	153 ± 75 ms	17414 ms (13%)
100K 10K	1542 ± 2511 ms	187099 ms (??%)
1000K 10K	22066 ± 32925 ms	1582292 ms (??%)

ReentrantLock, okhttp, virtual, jmh vs jlbh



reentrant		jmh		jlbh: лучший 90%	
-----:		-----:		-----:	
1K	10K	16 ±	78 ms	1700 ms	(15%)
10K	10K	153 ±	75 ms	17414 ms	(13%)
100K	10K	1542 ±	2511 ms	187099 ms	(??%)
1000K	10K	22066 ±	32925 ms	1582292 ms	(??%)

Выводы

Выводы

Виртуальные потоки для
I/O на высоком числе
конкурентных задач

1

Выводы

Виртуальные потоки для
I/O на высоком числе
конкурентных задач

1

Переключение контекстов
виртуальных потоков на
порядки быстрее

2

Выводы

Виртуальные потоки для
I/O на высоком числе
конкурентных задач

1

Переключение контекстов
виртуальных потоков на
порядки быстрее

2

Виртуальные потоки не
помогут если «плохой»
клиент I/O

3

Выводы

Виртуальные потоки для
I/O на высоком числе
конкурентных задач

1

Переключение контекстов
виртуальных потоков на
порядки быстрее

2

Виртуальные потоки не
помогут если «плохой»
клиент I/O

3

Structured concurrency
гораздо элегантнее
ExecutorService

4

Выводы

5

jlbh лучше подходит для измерения задач I/O и дает больше возможностей: fixed throughput, NanoSampler

Выводы

5

jlbn лучше подходит для измерения задач I/O и дает больше возможностей: fixed throughput, NanoSampler

6

Инженерный подход: фиксированное окружение, методология, итерации, эксперименты, несколько подходов к измерению, относительные числа

Спасибо за внимание! Вопросы?



<https://github.com/bob-the-dyer/joker-2025>

@vlakra

vladimir.krasilchik@gmail.com

01.tech