Virtuelle Threads

Nur nicht den Faden verlieren!

Java Forum Stuttgart 2024
Christian Schuster
mgm technology partners GmbH

Warum virtuelle Threads?

Warum Threads?

Thread (Informatik)

In der Informatik bezeichnet **Thread** [θɹɛd] (englisch *thread*, ‚Faden', ‚Strang') – auch **Aktivitätsträger** oder **leichtgewichtiger Prozess** genannt – einen *Ausführungsstrang* oder eine *Ausführungsreihenfolge* in der Abarbeitung eines Programms. Ein Thread ist Teil eines Prozesses.

- Weil wir vieles gleichzeitig machen wollen!
- Beispiele:
  - "klassisches" Java-Programm: Main-Thread, GC-Threads, etc.
  - Rich Client: Event-Thread, UI-Thread, Worker-Threads
  - Servlet-Container: Thread-Pool für Requests, verschiedene Hilfs-Threads
  - Parallele Streams: Thread-Pool

Warum Thread-Pools?

- Pooling: Vorhalten von Ressourcen, deren Bereitstellung teuer ist
- Erzeugung von Betriebssystem-Thread ist teuer
- Thread-Pool üblicherweise im Vorfeld allokiert
- Ein Thread arbeitet nacheinander mehrere Tasks ab
- Problem: Isolation der Tasks

Tasks: Was tun sie?

- Spektrum zwischen zwei Extremen

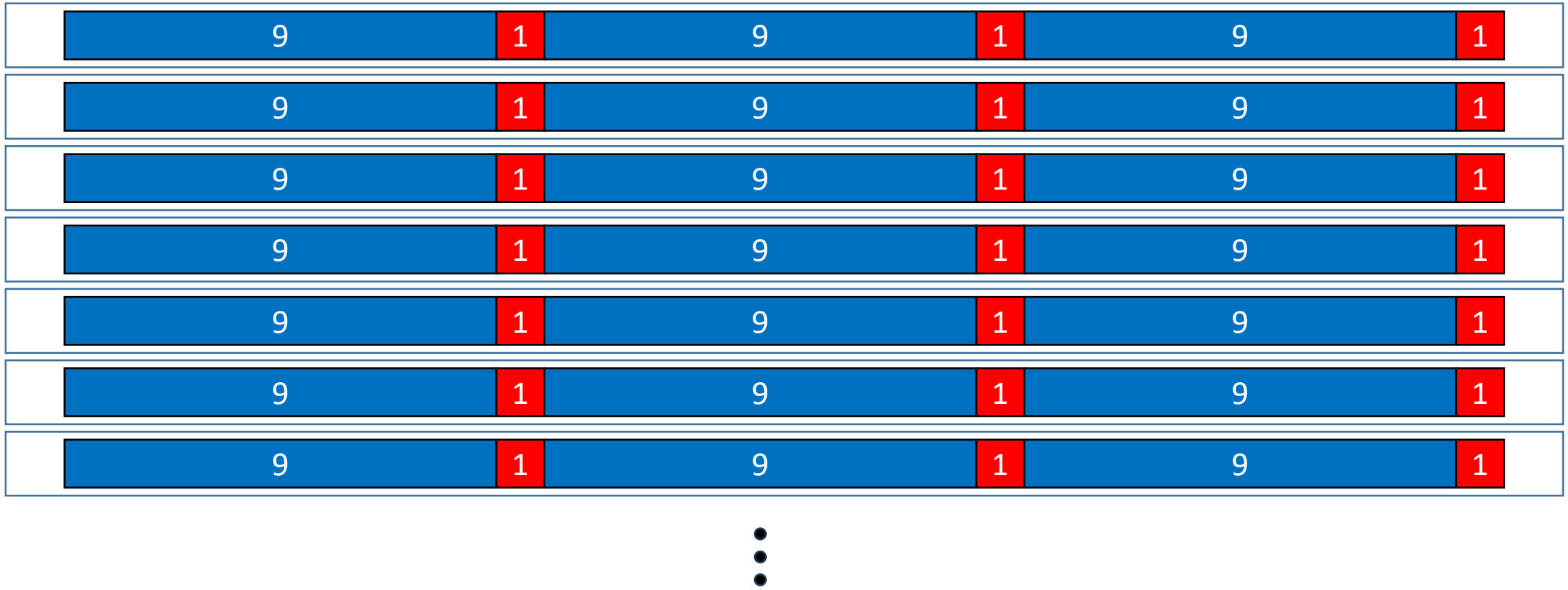
- I/O-lastig: Task wartet nur darauf, dass etwas passiert
- CPU-lastig: Task nutzt ständig einen Prozessor(kern)

Tasks: Rechenbeispiel

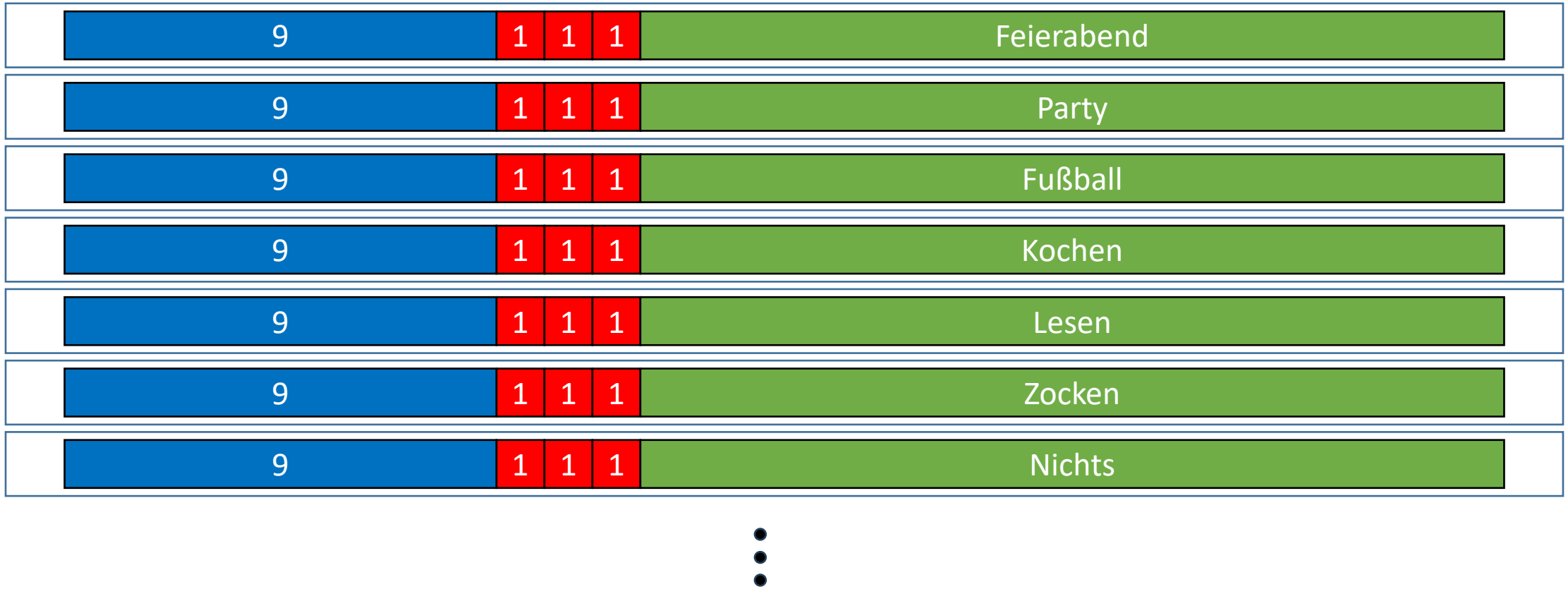
- Thread-Pool mit 12 Threads (= Anzahl logischer CPUs)
- 36 Tasks
- Jeder Task dauert 10 Sekunden
  - 9 Sekunden warten auf Antwort eines Webservices
  - eine Sekunde CPU-Zeit zur Verarbeitung der Antwort

- Wie lange dauert die Abarbeitung aller Tasks auf dem Thread-Pool?

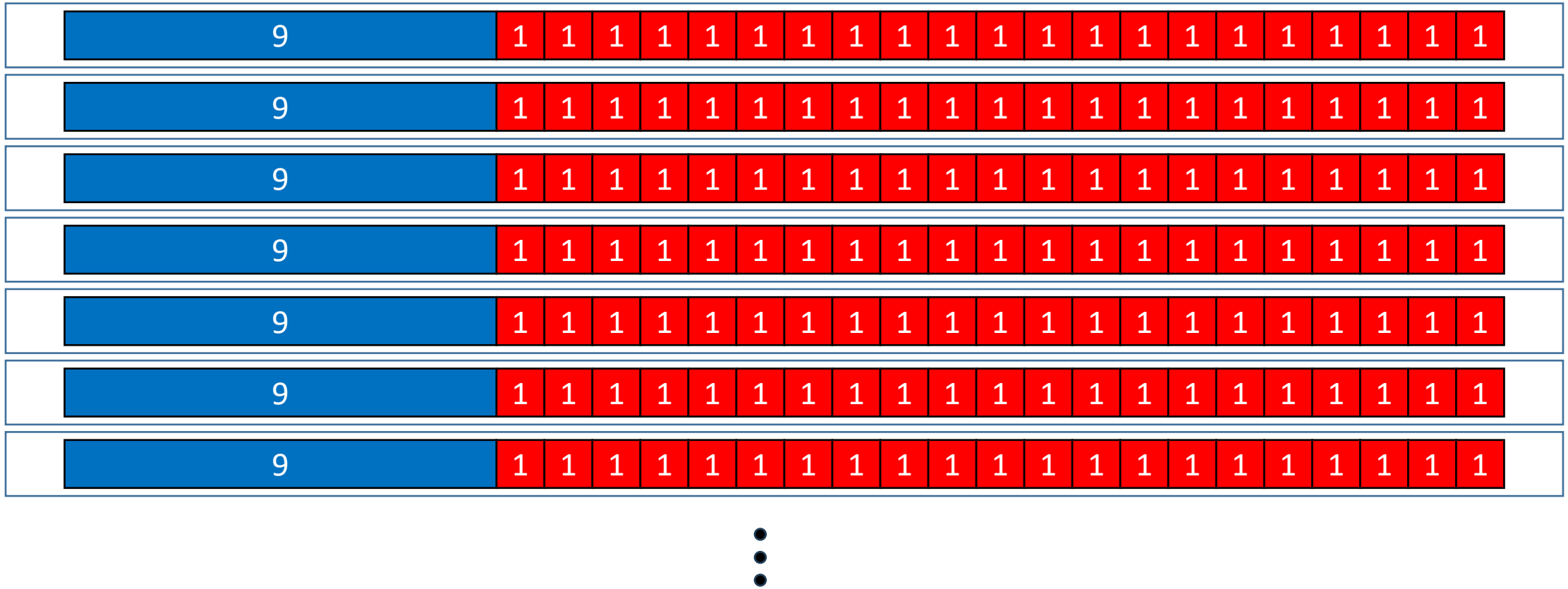
Tasks: Was wir gerade tun



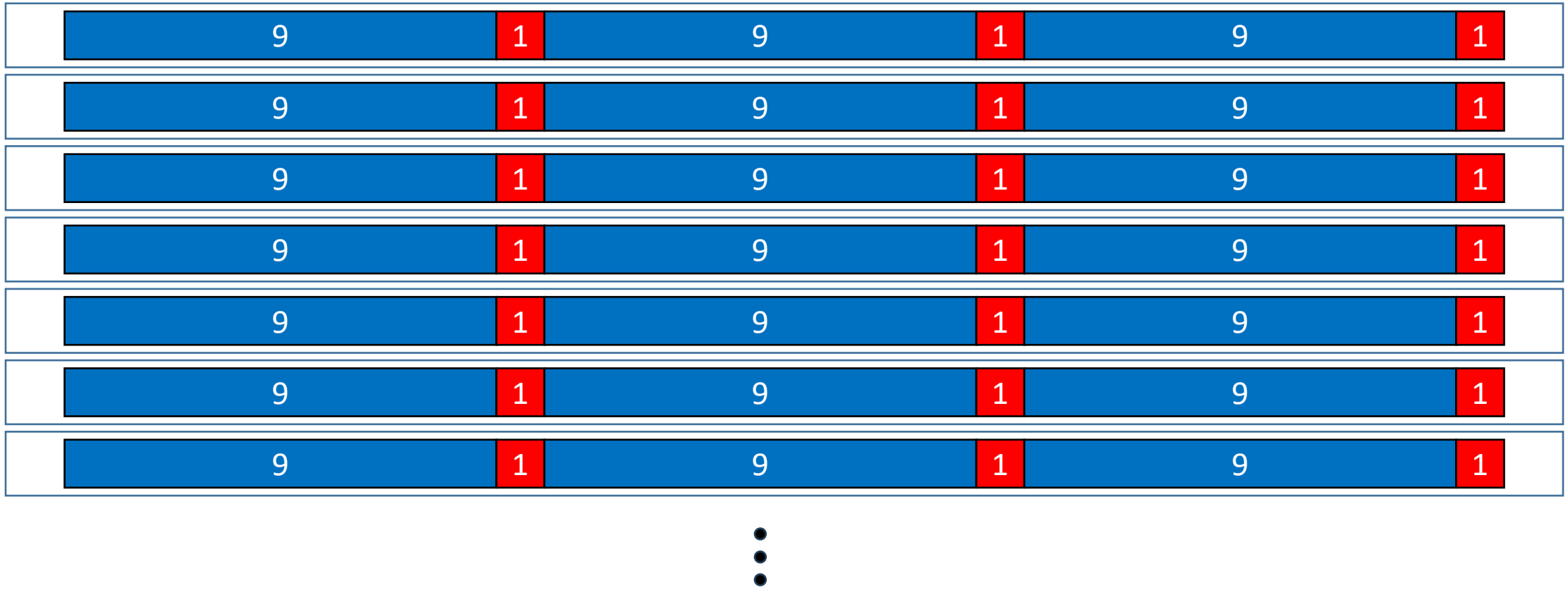
Tasks: Was wir eigentlich tun wollen



Tasks: Was wir letztlich tun werden



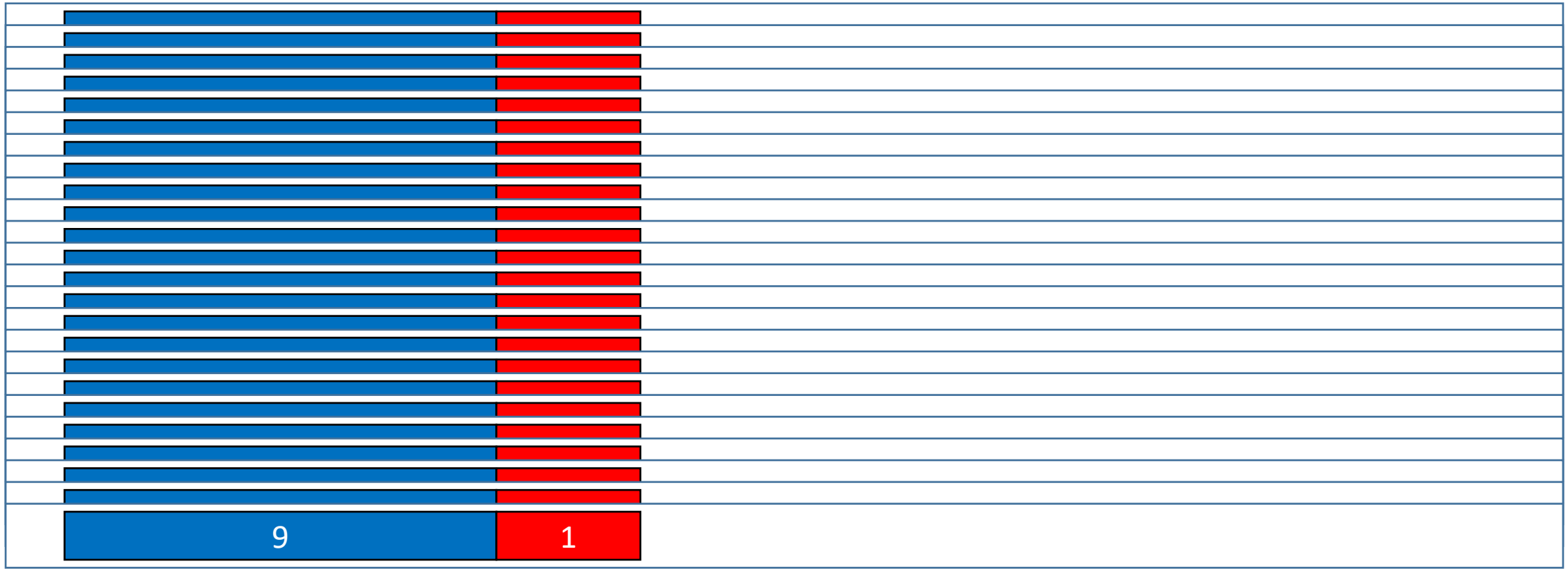
Tasks: Was wir gerade tun



Demo

36 Tasks (9+1 Sekunden), 12 Threads

"Optimierung": Mehr Threads



"Optimierung": Mehr Threads

- mehr gleichzeitig wartende Threads
- mehr gleichzeitig laufende Threads
- mehr Kontextwechsel zwischen Threads
- höherer Ressourcenverbrauch
- skaliert schlecht

Demo

36 Tasks (9+1 Sekunden), 36 Threads

Optimierung: Reactive Programming

- Publisher-Subscriber
- Pipelines von synchronen und asynchronen Operationen
- Paradigmenwechsel beim Umstieg von "imperativem" Java
- Backpressure: Subscriber kann Publisher bremsen
- schwierig zu debuggen
- Stacktraces wenig nützlich
- Verwendung von non-blocking I/O verpflichtend

Optimierung: Koroutinen

- verallgemeinerte Funktionen, die ihren Ablauf unterbrechen können
- verschiedene Programmiersprachen: Generatoren
- oft mit Kanälen zum Datenaustausch kombiniert
- benötigt geeignete Sprachkonstrukte
- für kooperative Tasks gut geeignet
- Coroutines@Kotlin & Goroutines@Go können Koroutinen abbilden
- in Java nicht einfach umsetzbar

Endlich: Virtuelle Threads

- JEP 425 (JDK 19), JEP 436 (JDK 20), JEP 444 (JDK 21)
- Teil von Project Loom
- seit JDK 19 als Preview-Feature verfügbar
- seit JDK 21 offiziell verfügbar
- basierend auf Continuations
- Plattform-Thread vs. virtueller Thread

Demo

36 Tasks (9+1 Sekunden), virtuelle Threads

Virtuelle Threads: Eigenschaften

- Java-Objekt ohne Betriebssystem-Äquivalent
- Stack des virtuellen Threads im Java-Heap
- Stack wächst und schrumpft bei Bedarf
- zur Ausführung an "Carrier-Thread" gebunden ("mount")
- kann (und muss) sich selbst vom Carrier-Thread lösen ("unmount")
- wird später wieder an (ggf. anderen) Carrier-Thread gebunden
- kein Zugriff auf Carrier-Thread über öffentliche API

Demo

Viele Threads

Virtuelle Threads: Kompatibilität

- Instanzen einer Subklasse von java.lang.Thread
- wie gewohnt zu debuggen
- brauchbare Stacktraces
- größtenteils wie "klassische" Threads verwendbar
  - Thread.stop() / .suspend() / .resume() → UnsupportedOperationException
  - Thread.setPriority() → No-Op
  - Thread.setDaemon(false) → IllegalArgumentException
  - nicht von ThreadMXBean unterstützt

Virtuelle Threads: Umdenken

- kein Pooling für virtuelle Threads
  - Millionen virtuelle Threads sind kein Problem
  - ein virtueller Thread pro Task/Request
- keine implizite Beschränkung anderer Ressourcen durch Thread-Pools
  - Beispiel: Webservice erlaubt 10 gleichzeitige Verbindungen
  - explizit zum Beispiel mit Semaphoren

Virtuelle Threads: Herausforderungen

- synchronized-Block bindet virtuellen Thread an Carrier-Thread
  - blockieren/warten in synchronized-Blöcken ist schlecht
  - kann einfach durch ReentrantLock ersetzt werden
- native Methode oder "Foreign Function" (JEP 454) ebenfalls
  - nicht anders möglich, aufgerufener Code außerhalb der JVM
- blockierende I/O-Operationen (syscalls) problematisch
  - Großteil des JDK ist asynchron
  - Rest: JDK kompensiert durch temporäre zusätzliche Carrier-Threads
- jstack zeigt virtuelle Threads nicht an
  - jcmd <pid> Thread.dump_to_file -format=json <file>

Demo

Virtual Thread Pinning

Virtuelle Threads: Unterstützung

- Tomcat 10.1
- Tomcat 11
  - org.apache.catalina.core.StandardVirtualThreadExecutor
- Spring Boot 3.2
  - spring.threads.virtual.enabled=true
- Hibernate 6.2.8 / 6.3.0
- pgjdbc 42.6.0
- MySQL Connector/J 9.0.0


Fragen?


Vielen Dank!