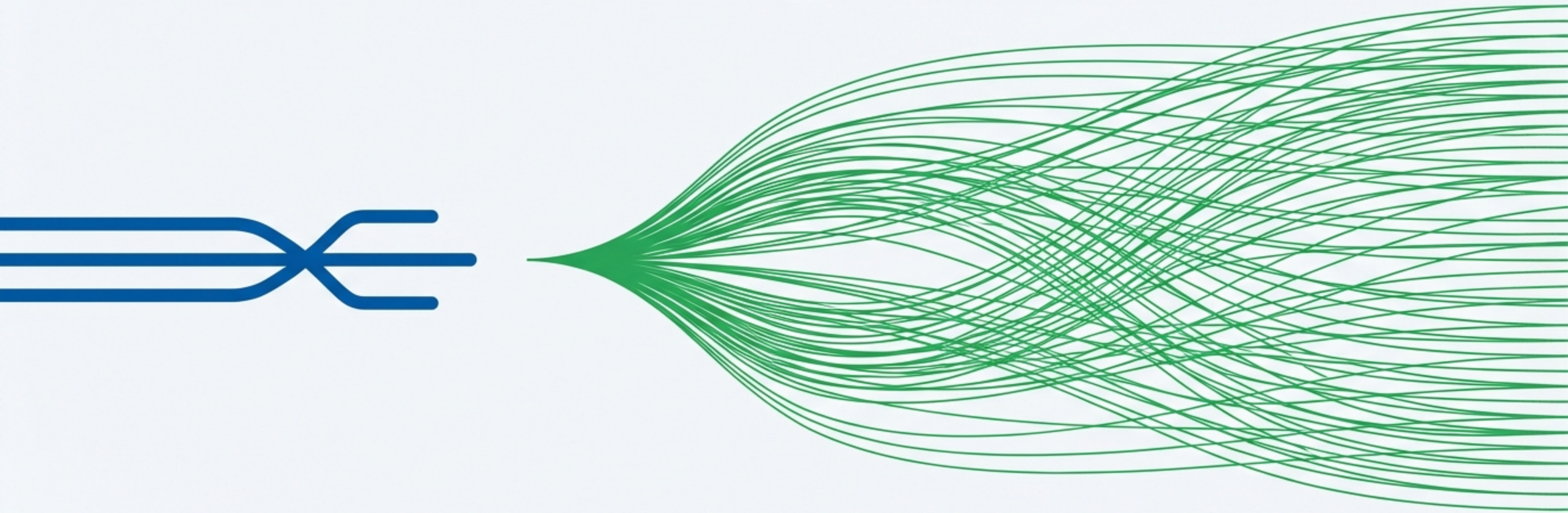
The New Era of Java Concurrency

Unlocking Massive Throughput with Virtual Threads



From scarcity to abundance. How JDK 21+ redefines scalability.

The Universal Bottleneck: Why Your Server Can't Scale

Little's Law: The governing principle

$$\text{Throughput} = \frac{\text{Concurrency}}{\text{Latency}}$$

To increase throughput, you must either decrease latency or increase concurrency.

A Concrete Example

A server with an average latency of 50ms.

With **10** concurrent threads:

`10 / 0.050s = 200 requests/sec`

To reach **2000** requests/sec:

`You need 100 concurrent threads.`

To keep up, the number of threads must grow as throughput grows.
But traditional Java threads are a **scarce and heavy resource**.

The Workarounds: A Legacy of Complexity

To overcome the thread limit, we turned to non-blocking asynchronous code. This solved one problem but created others.

Non-blocking via `CompletableFuture`

```
public CompletableFuture<ABC> abc(String id) {
  return supplyAsync(() -> api.a(id), executor)
      .thenCompose(a -> api.b(a))
      .thenCompose(b -> api.c(a, b))
      .thenApply(c -> new ABC(a, b, c));
}
```

Non-blocking with Reactive Programming

```
public Mono<ABC> abc(int id) {
  return api.a(id)
      .flatMap(a -> api.b(a).flatMap(
                b -> api.c(a, b).map(
                    c -> new ABC(a, b, c))));
}
```

This style is powerful but goes against the grain of the Java platform and the developer's mental model.

The Hidden Costs of Asynchronous Code

The asynchronous style breaks the simple thread-per-request model Java was built on, leading to significant challenges.

- **Difficult to Read & Write:** The code flow is non-linear and callback-heavy.

- **Fragmented Execution:** Each stage of a request might run on a different thread.

- **Lost Context:** Stack traces become nearly useless, showing the scheduler's internals instead of the application's business logic.

- **Debugging Nightmare:** Stepping through request-handling logic with a debugger is practically impossible.

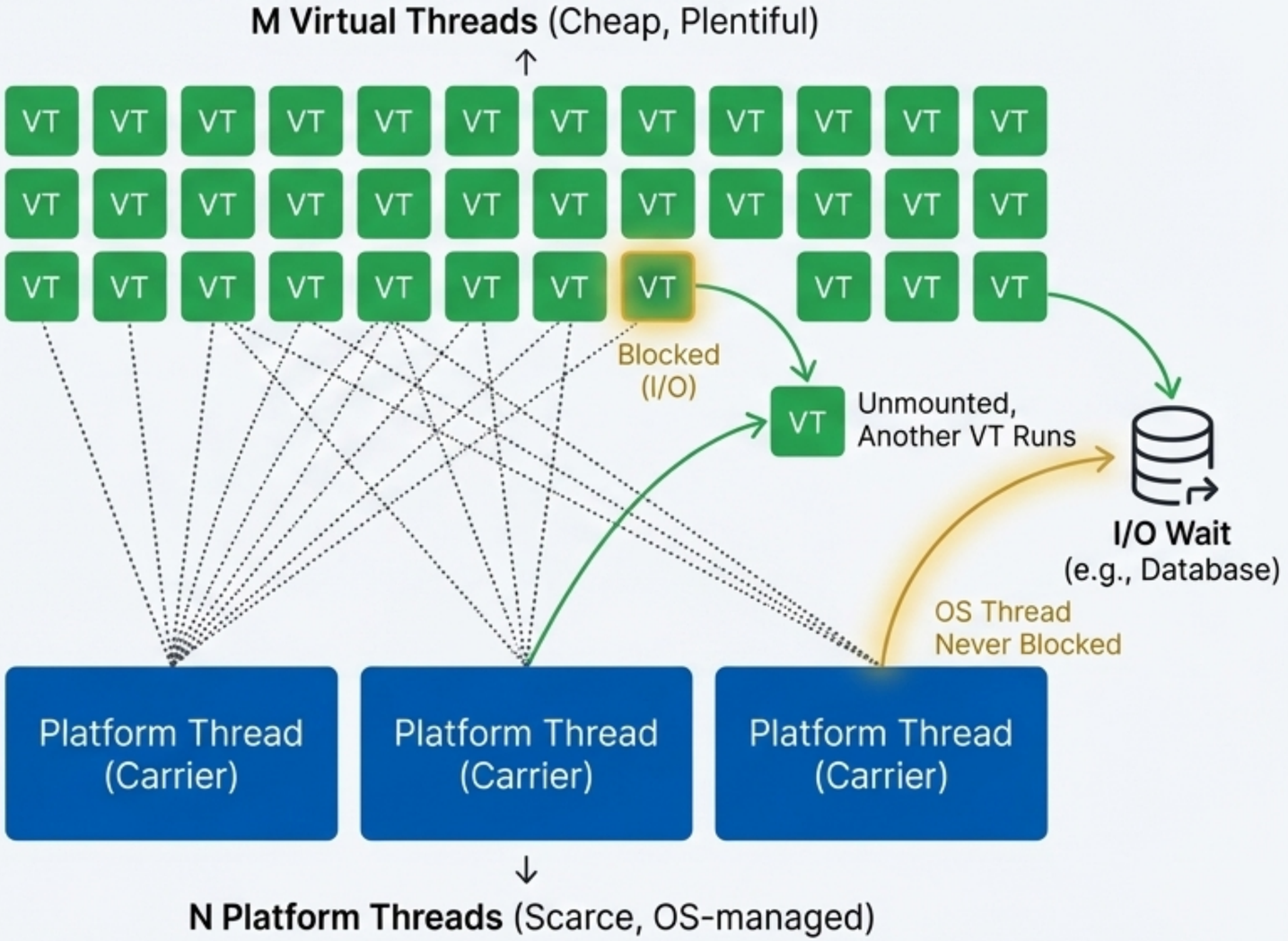
The Solution: Virtual Threads Have Arrived

Core Definition

Virtual Threads are a lightweight thread implementation managed by the JDK, not the OS. They are cheap, plentiful, and designed for high-concurrency I/O-bound workloads.

The Paradigm Shift: M:N Scheduling

- A large number (M) of virtual threads are scheduled to run on a smaller number (N) of OS platform threads (called "carrier threads").
- When a virtual thread blocks on I/O, the JDK scheduler unmounts it and runs a different virtual thread on the same carrier. The OS thread is never blocked.



A Tale of Two Threads: By the Numbers

Parameter	Platform Threads	Virtual Threads
Origin	Wrapper for OS Thread	Managed by JDK
Stack Size	~1 MB (fixed)	Resizable (starts small)
Startup Time	> 1000 µs	1-10 µs
Context Switch	1-10 µs	~0.2 µs
Quantity	Thousands (< 5,000)	**Millions**

Virtual threads preserve the simple, familiar thread-per-request style while dramatically optimizing hardware utilization.

The Throughput Revolution in Action

Executing blocking tasks (`Thread.sleep(12s)`) with a fixed pool of 12 platform threads vs. a virtual thread executor.

10,000 TASKS

Platform Threads

~10,008 seconds

(~2.78 hours)

Virtual Threads

12 seconds

1,000,000 TASKS

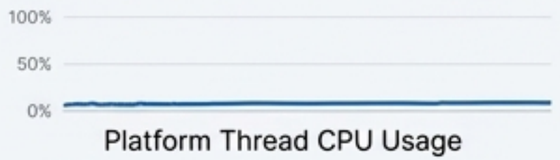
Platform Threads

Days to complete.
We didn't wait.

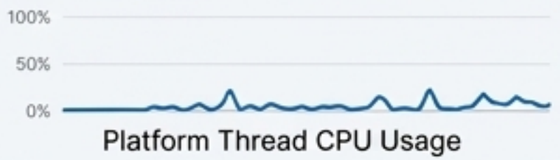
Virtual Threads

25 seconds

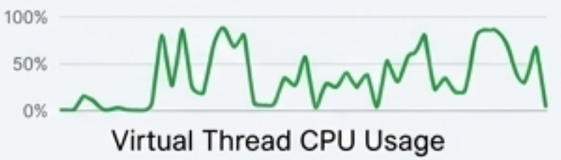
Visual Evidence



Platform Thread CPU Usage



Platform Thread CPU Usage



Virtual Thread CPU Usage

This is the core value proposition: Virtual Threads provide **throughput, not speed**. They don't run code faster; they run *more code concurrently*.

The Right Tool for the Right Job

✅ When to Use Virtual Threads

- **High number of concurrent tasks** (more than a few thousand).
- **Workloads are primarily I/O-bound** (e.g., microservices calling other services, database operations, message queues).

❌ When NOT to Use Virtual Threads

- **CPU-bound workloads**. Having more threads than processor cores offers no benefit.

Evidence: CPU-Intensive Test (Factorial Calculation)

Tasks	Platform Threads	Virtual Threads
1,000	241 seconds	256 seconds

For CPU-intensive work, virtual threads offer no performance advantage and may even have slight overhead. Stick to a pool of platform threads sized to the number of cores.

Putting Virtual Threads to Work: The Modern API

The Recommended Approach: `Executors.newVirtualThreadPerTaskExecutor()`

The simplest and most robust way to adopt virtual threads.

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            // Your I/O-bound task here
            Thread.sleep(Duration.ofSeconds(1));
        });
    });
} // executor is automatically closed
```

Other Creation Methods

- `Thread.ofVirtual().start(...)`
- `Thread.startVirtualThread(...)`

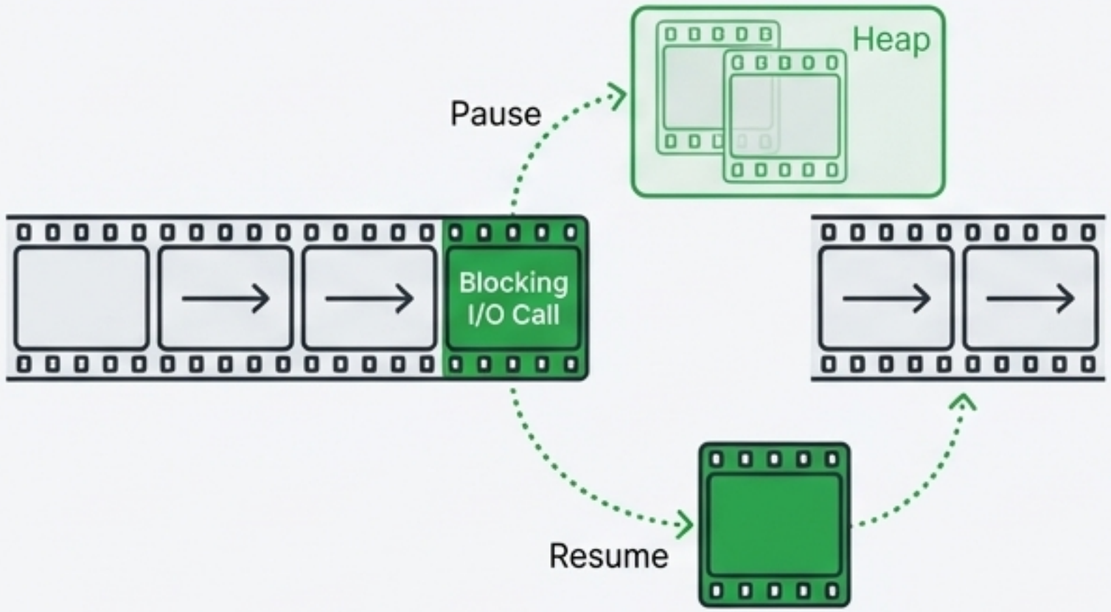
Utility Method

- `Thread.isVirtual()`

Under the Hood: Continuations and the JDK Scheduler

The Magic Revealed: Continuations

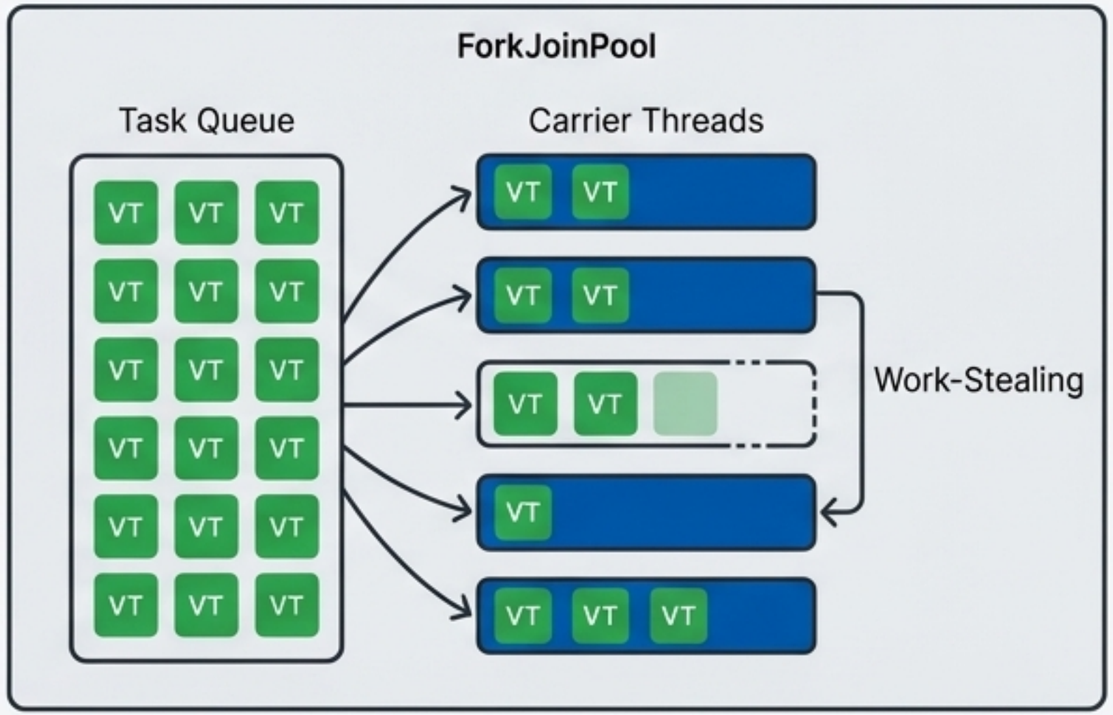
Virtual threads are built on a low-level JDK feature called **Continuations**. A Continuation allows a program to **pause** its execution and **resume** later from the exact same state.



⚠ **Important:** This is a JDK internal API, not for application developers to use directly.

The Scheduler

The JDK uses a work-stealing `ForkJoinPool` in FIFO mode to schedule virtual threads onto carrier platform threads. By default, the number of carrier threads equals the number of available processor cores.

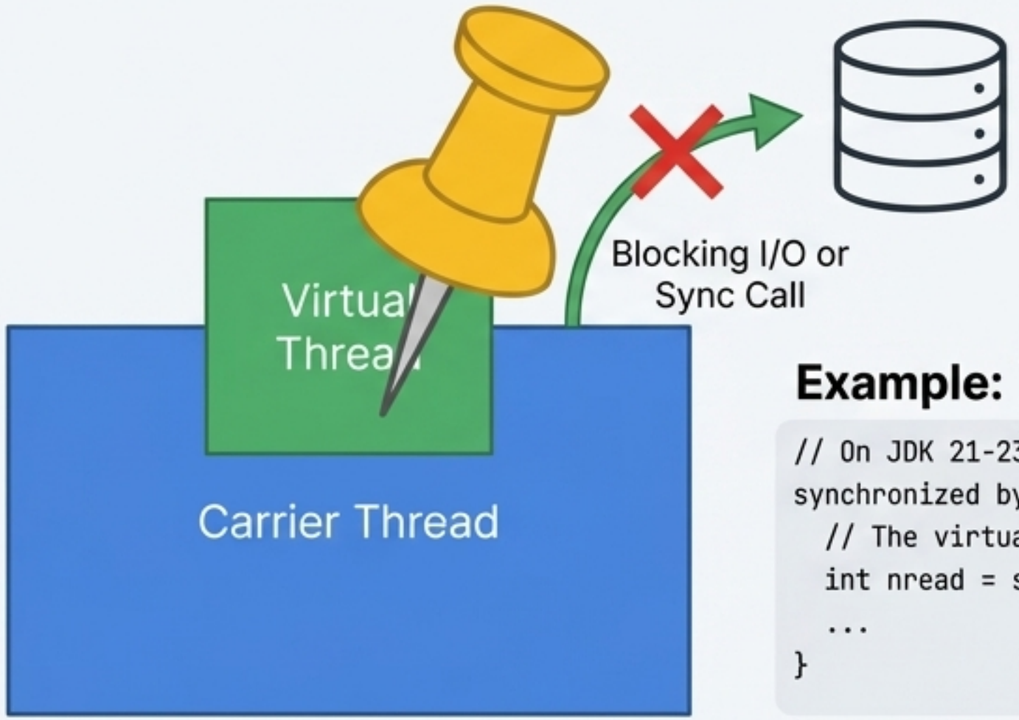


The Hidden Danger: When Virtual Threads Get “Pinned”

Pinning occurs when a virtual thread cannot be unmounted from its carrier thread during a blocking operation. This means the carrier thread (and its underlying OS thread) becomes blocked, defeating the purpose of virtual threads.

Primary Causes

1. Executing a `native` method or a Foreign Function call.
2. Executing code inside a `synchronized` block or method (on JDK versions prior to 24).



Example: `synchronized` Pinning

```
// On JDK 21-23, this code PINS the virtual thread
synchronized byte[] getData() {
  // The virtual thread is now pinned to its carrier.
  int nread = socket.getInputStream().read(buf); // BLOCKS THE OS THREAD!
  ...
}
```

Frequent, long-duration pinning harms scalability and can lead to **thread starvation** or even **deadlocks**.

Defeating Pinning: Modern Solutions and Best Practices

The Best Solution: Upgrade Your JDK

As of JDK 24 (JEP 491), the JVM can unmount virtual threads even when they block inside a `synchronized` block.

This is a game-changer, allowing many existing libraries to work with virtual threads without modification.

For Older JDKs (21-23)

Replace `synchronized` blocks with `java.util.concurrent.locks.ReentrantLock`.

What About Native Methods?

Pinning still occurs for JNI / FFM calls. Be mindful of libraries that make extensive use of native code.

How to Detect Pinning

Use JDK Flight Recorder (JFR) and look for the `jdk.VirtualThreadPinned` event. This will help you identify problematic code sections.

Virtual Threads: The Essential Rulebook

Key Principles

- ✓ **Never pool virtual threads**. They are cheap and should be created per task. 🧵
- 👻 Virtual threads are always **daemon threads**.
- They have a fixed priority (`NORM_PRIORITY`).
- `Thread.currentThread()` returns the virtual thread, not the carrier.

Common Pitfalls & Limitations

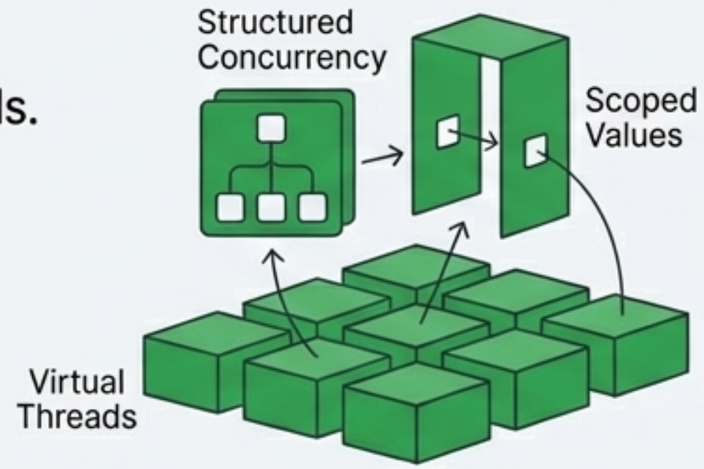
- ⚠ **GC Overhead**: Virtual thread stacks live on the heap, increasing GC work.
- 🗑 **G1 GC**: Does not support "humongous" stack chunk objects. Very deep call stacks can cause a `StackOverflowError` sooner.
- 🔍 **Debugging**: `jstack` thread dumps can be unreadable with millions of threads. New tooling is required.
- ☁ **Mixing Workloads**: Avoid mixing CPU-intensive and I/O-bound tasks in the same virtual thread executor.

The Journey Ahead: What's Next for Java Concurrency?

Built on Virtual Threads

The introduction of virtual threads enables new, higher-level concurrency models.

- ✓ **Structured Concurrency**: Treating groups of related tasks running in different threads as a single unit of work.
- ✓ **Scoped Values**: A robust alternative to thread-local variables for sharing data across threads.



Food for Thought: Evolving the Ecosystem

With virtual threads, is reactive programming still necessary for achieving high throughput?

What is the future role of `CompletableFuture`?

How do we manage shared resources in a world of millions of threads?

Virtual Threads are the start of a new chapter, making concurrent programming in Java simpler, more efficient, and more powerful than ever before.