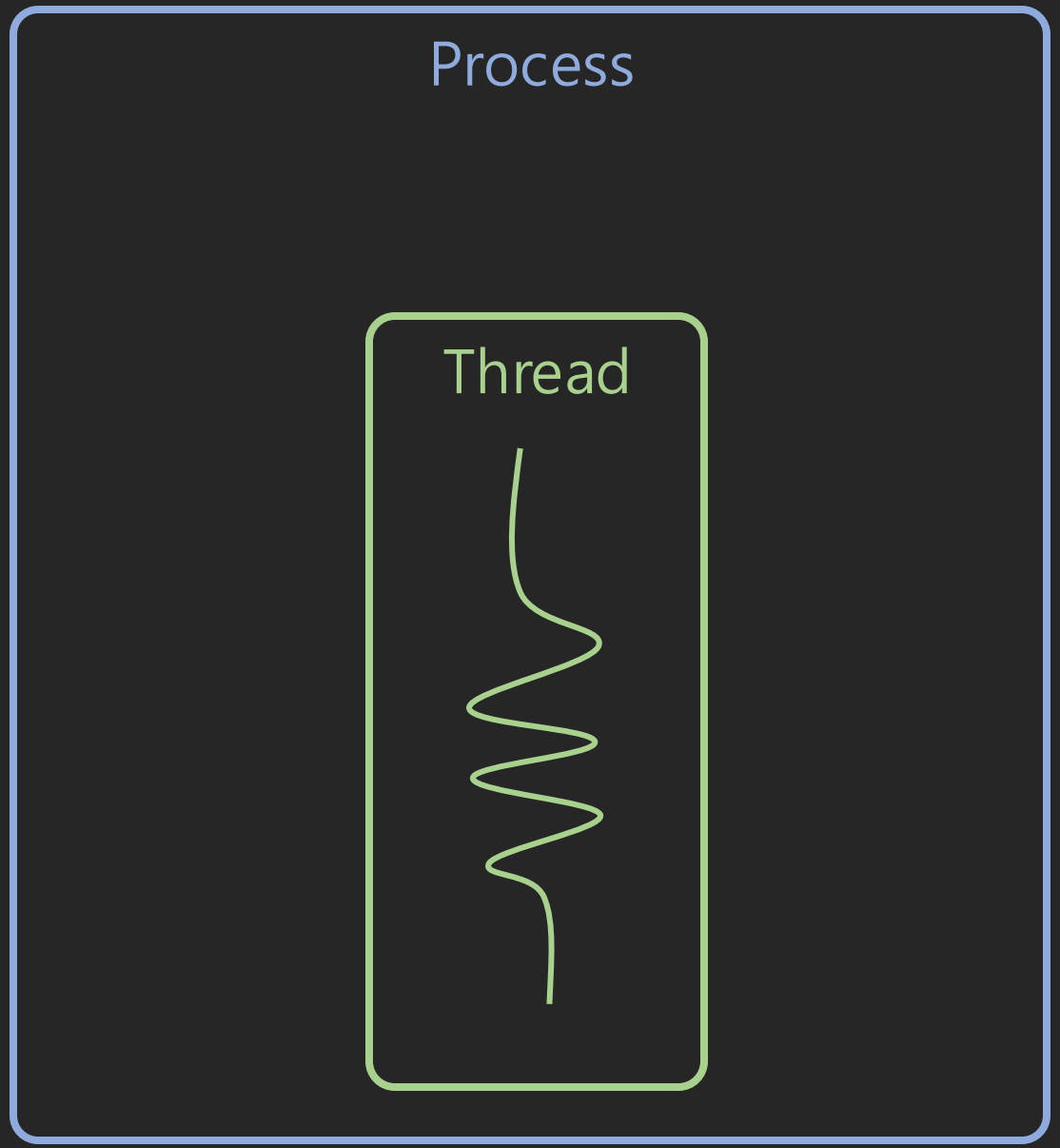
Game Changer in Java Concurrency : Virtual Thread (a.k.a Fiber)

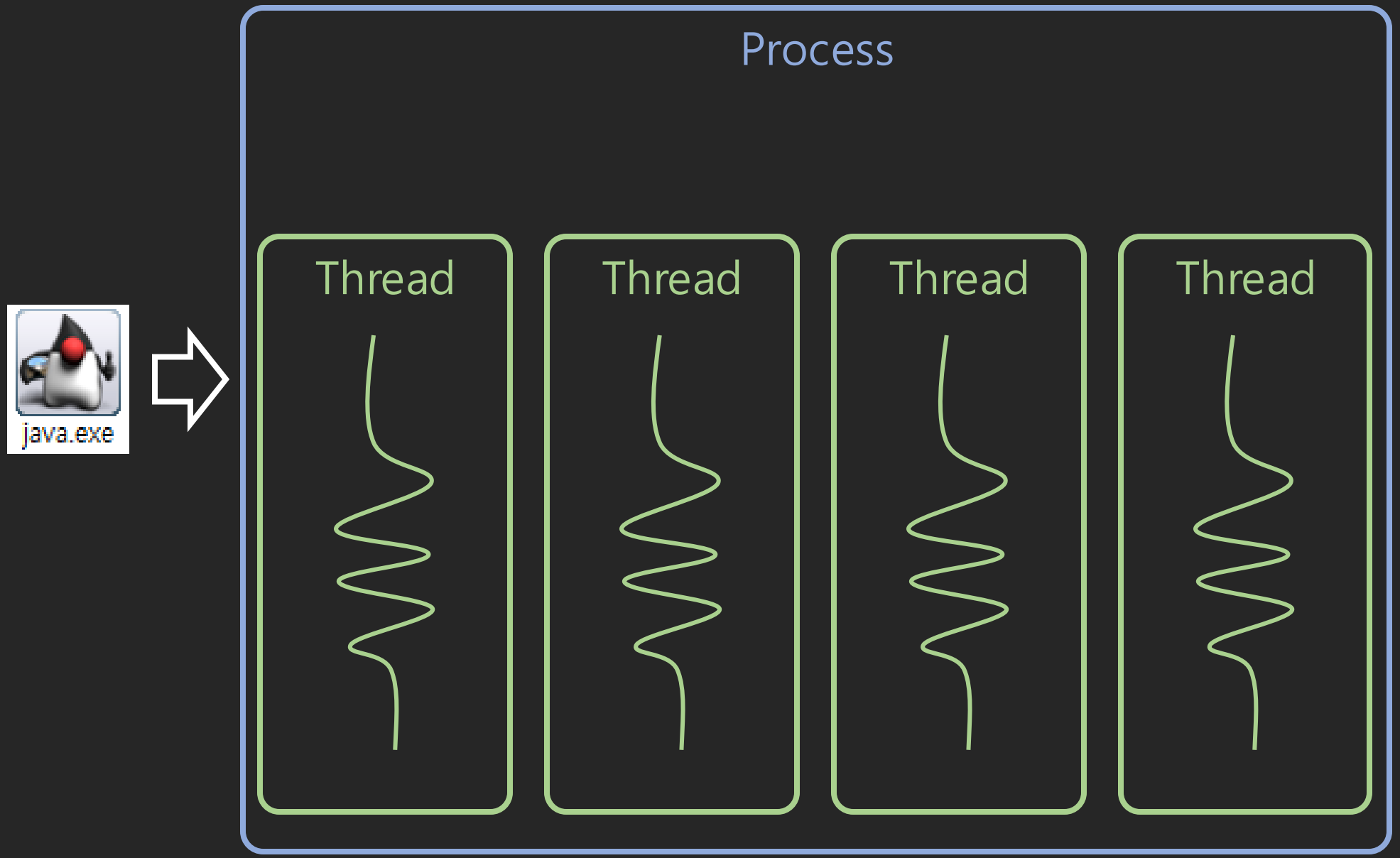
Richard HC
Order Fulfillment

Program? Process? Thread?



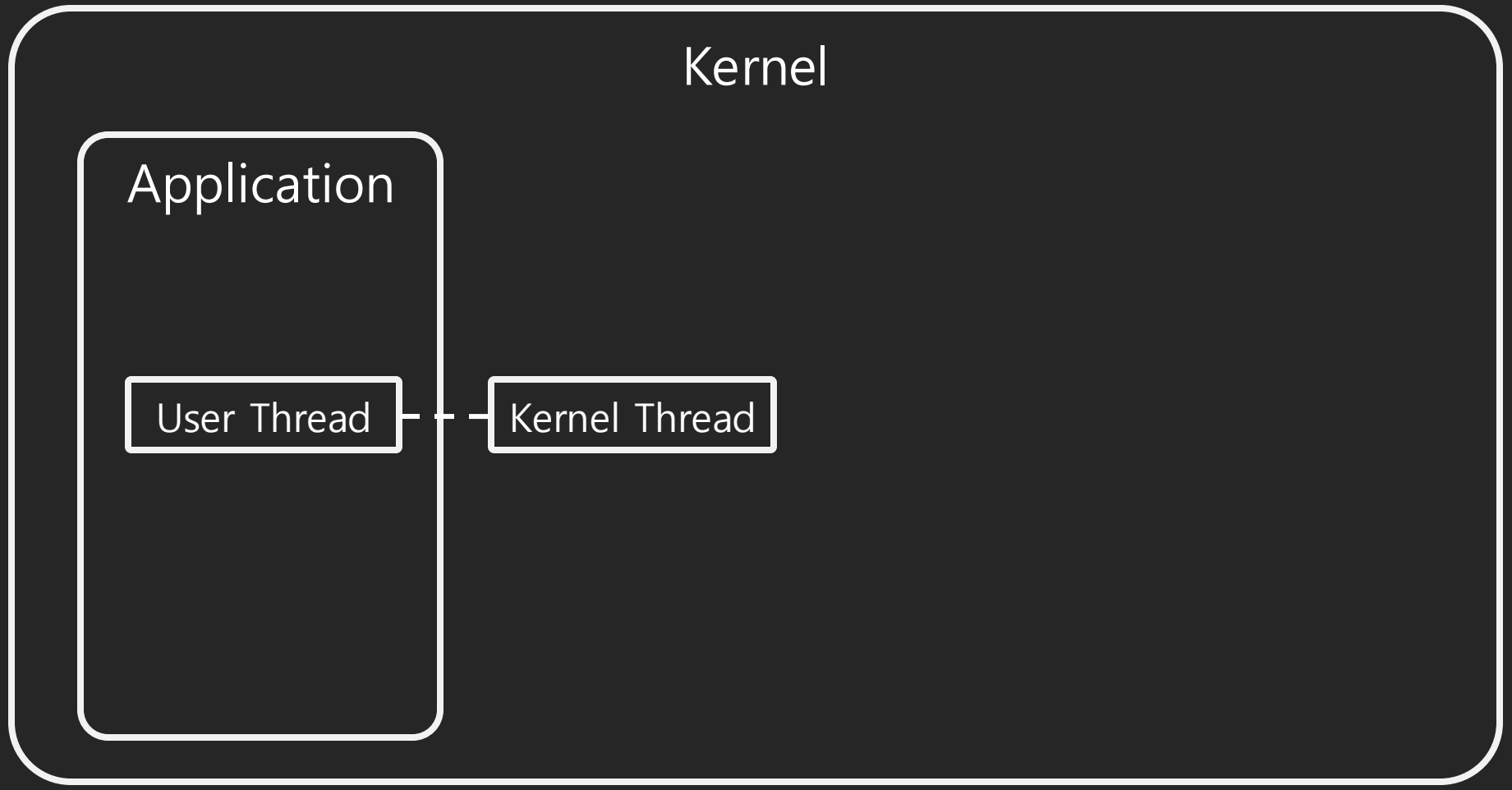
https://www.geeksforgeeks.org/difference-between-process-and-thread/
https://www.tutorialspoint.com/what-is-process-control-block-pcb
https://www.geeksforgeeks.org/process-table-and-process-control-block-pcb/

Program? Process? Thread?

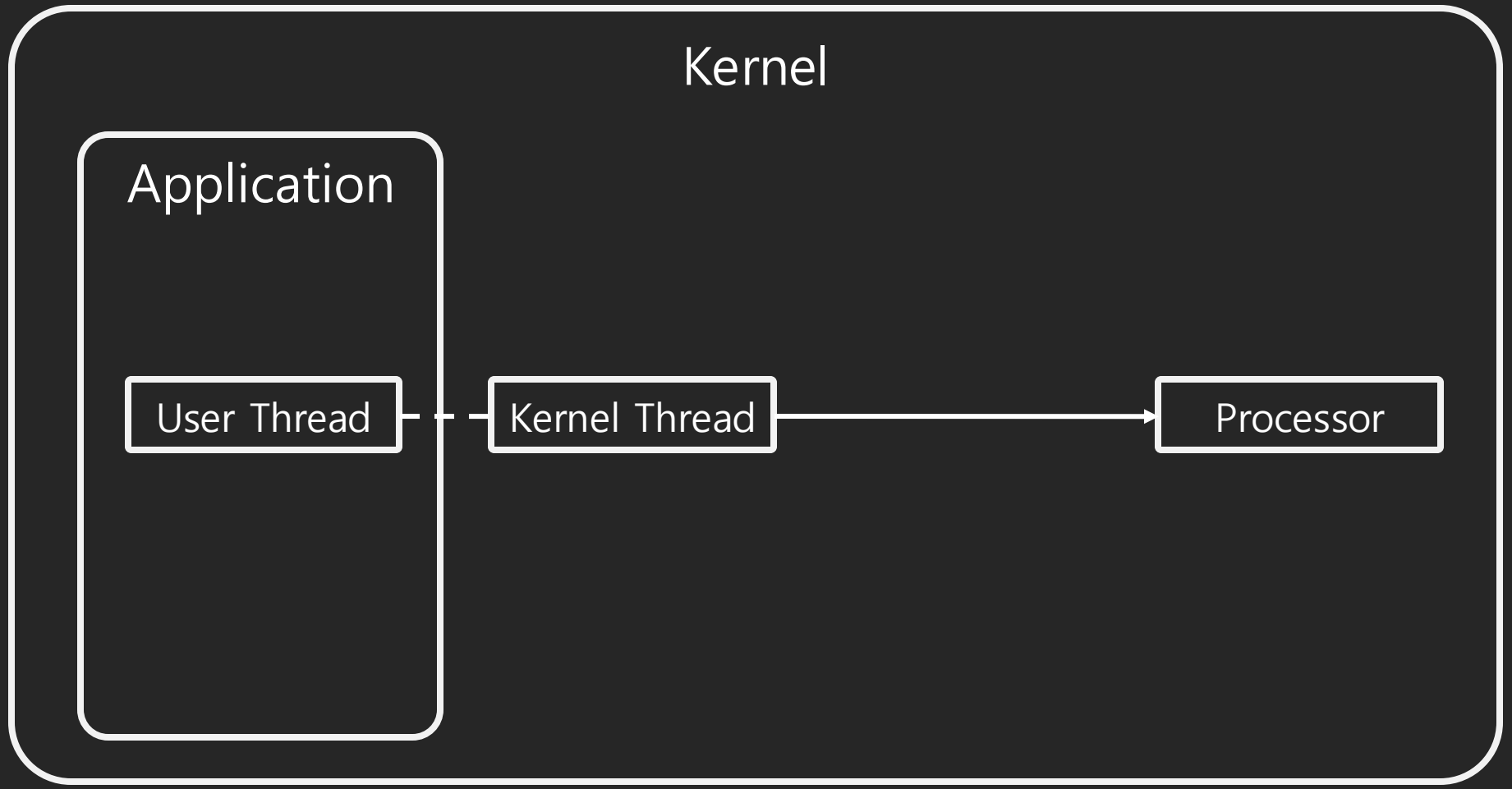


Kernel-level Thread
vs
User-level Thread

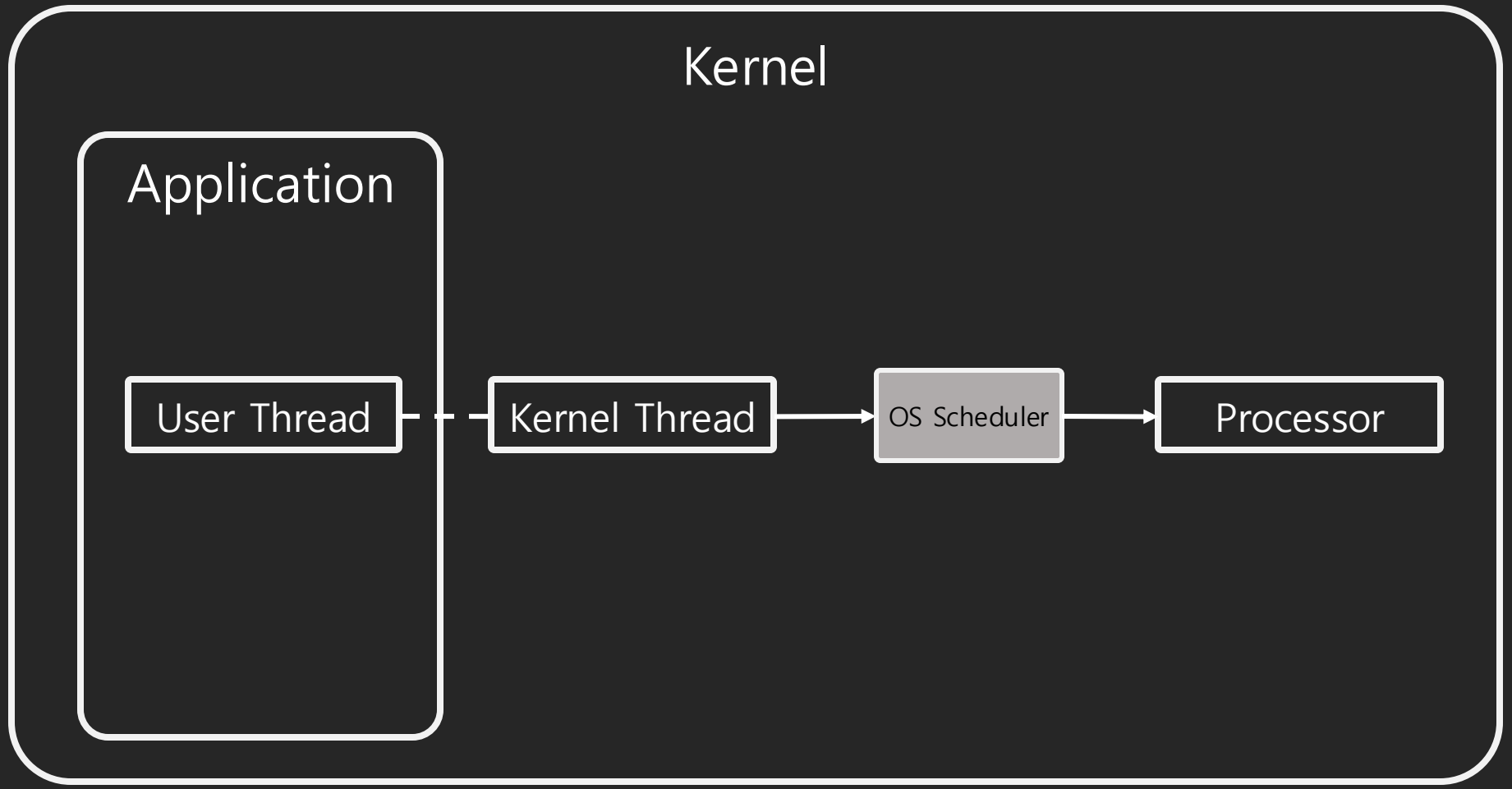
Kernel-level Thread vs User-level Thread



Kernel-level Thread vs User-level Thread



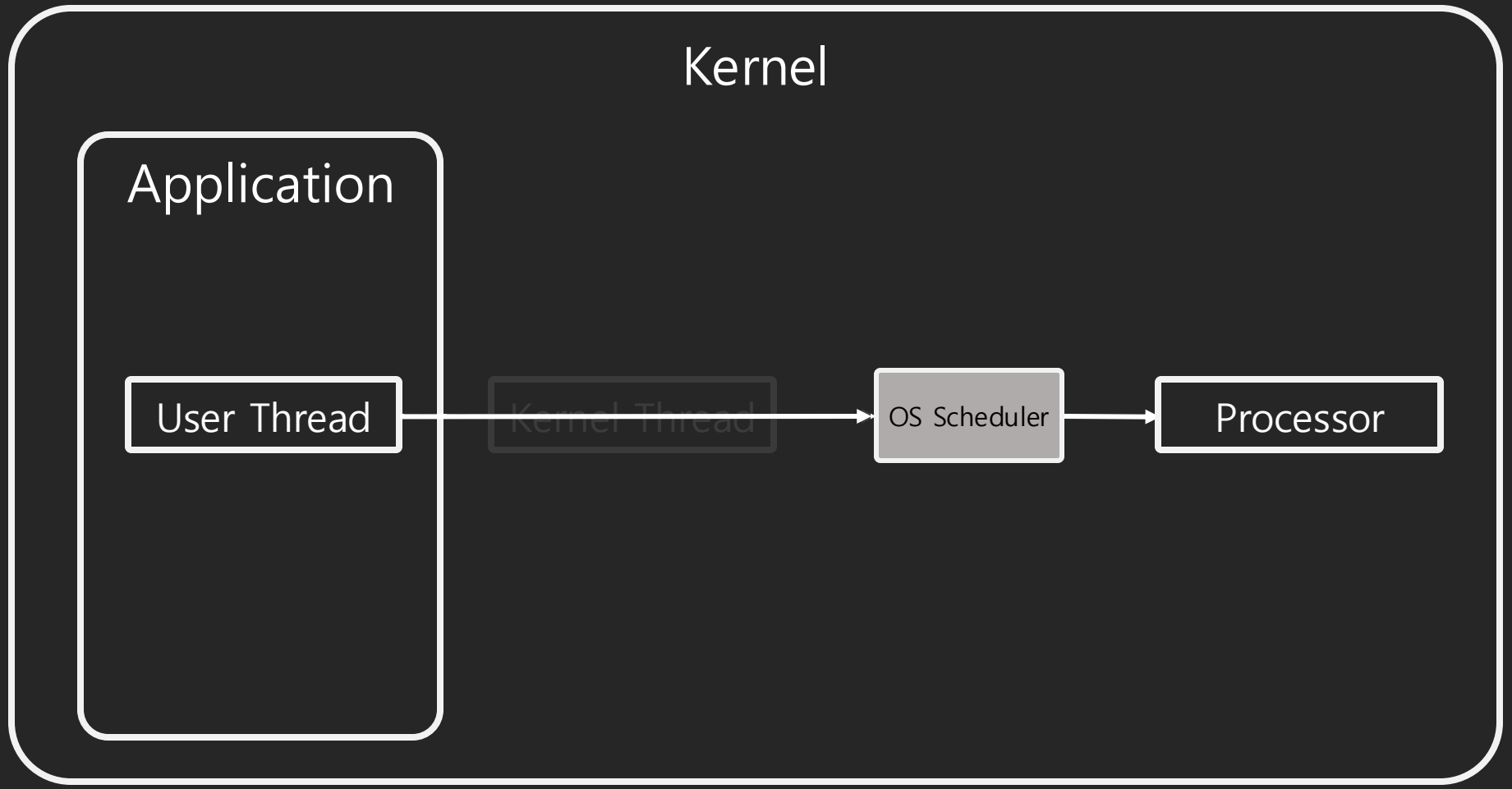
Kernel-level Thread vs User-level Thread



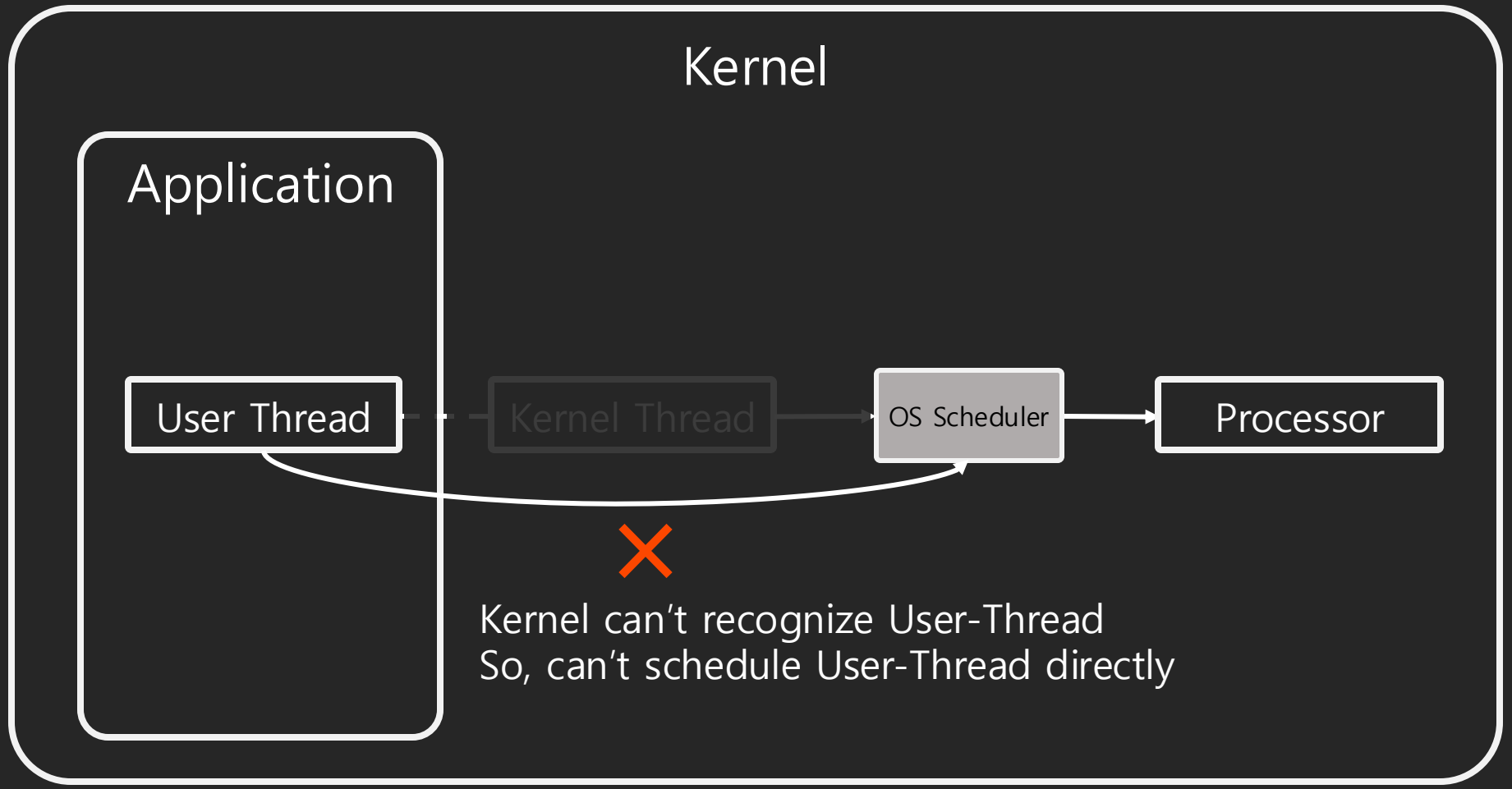
Kernel-level Thread vs User-level Thread

But, Why User-Level Thread
must be mapped with Kernel-Level Thread ?

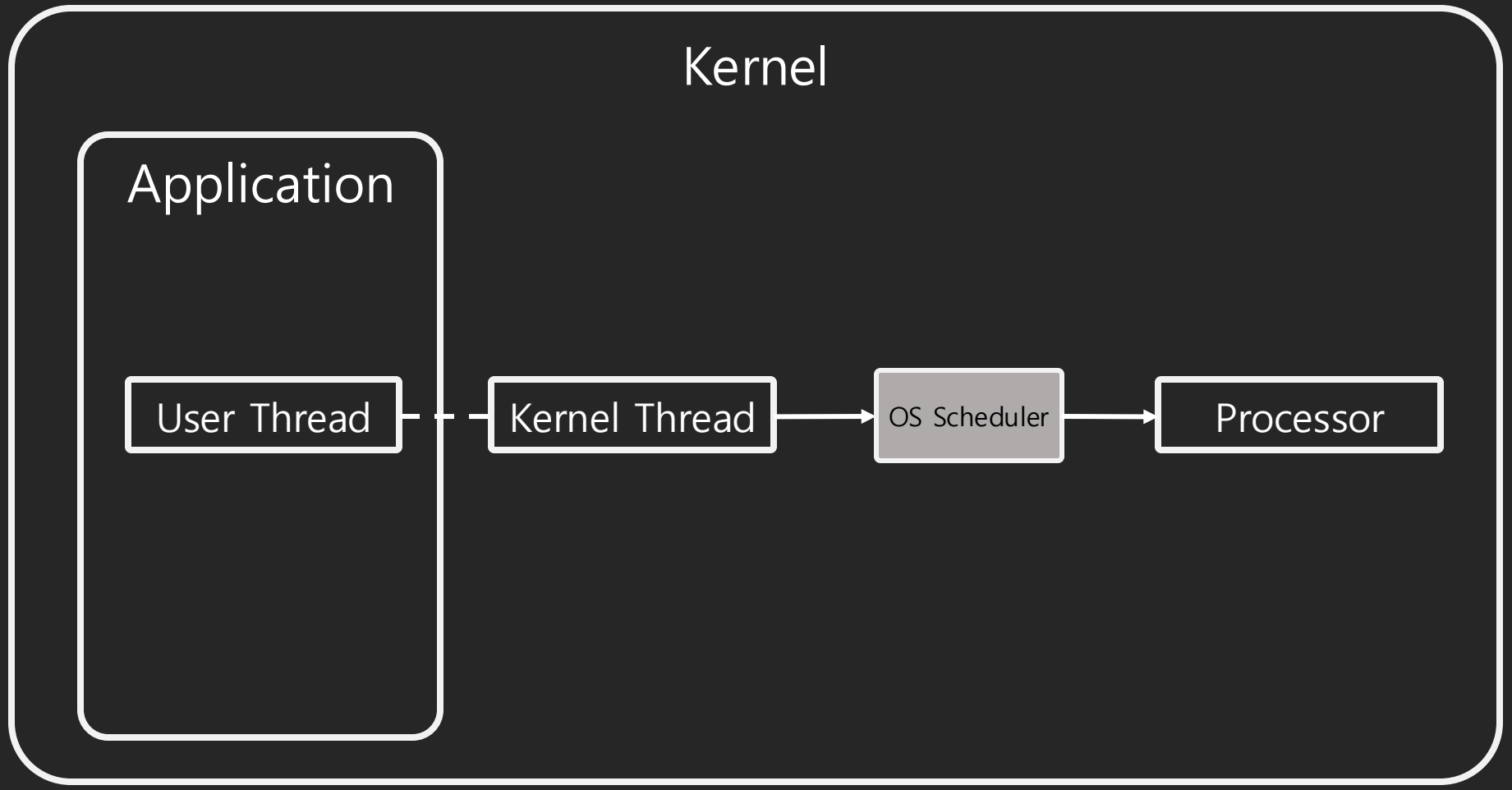
Kernel-level Thread vs User-level Thread



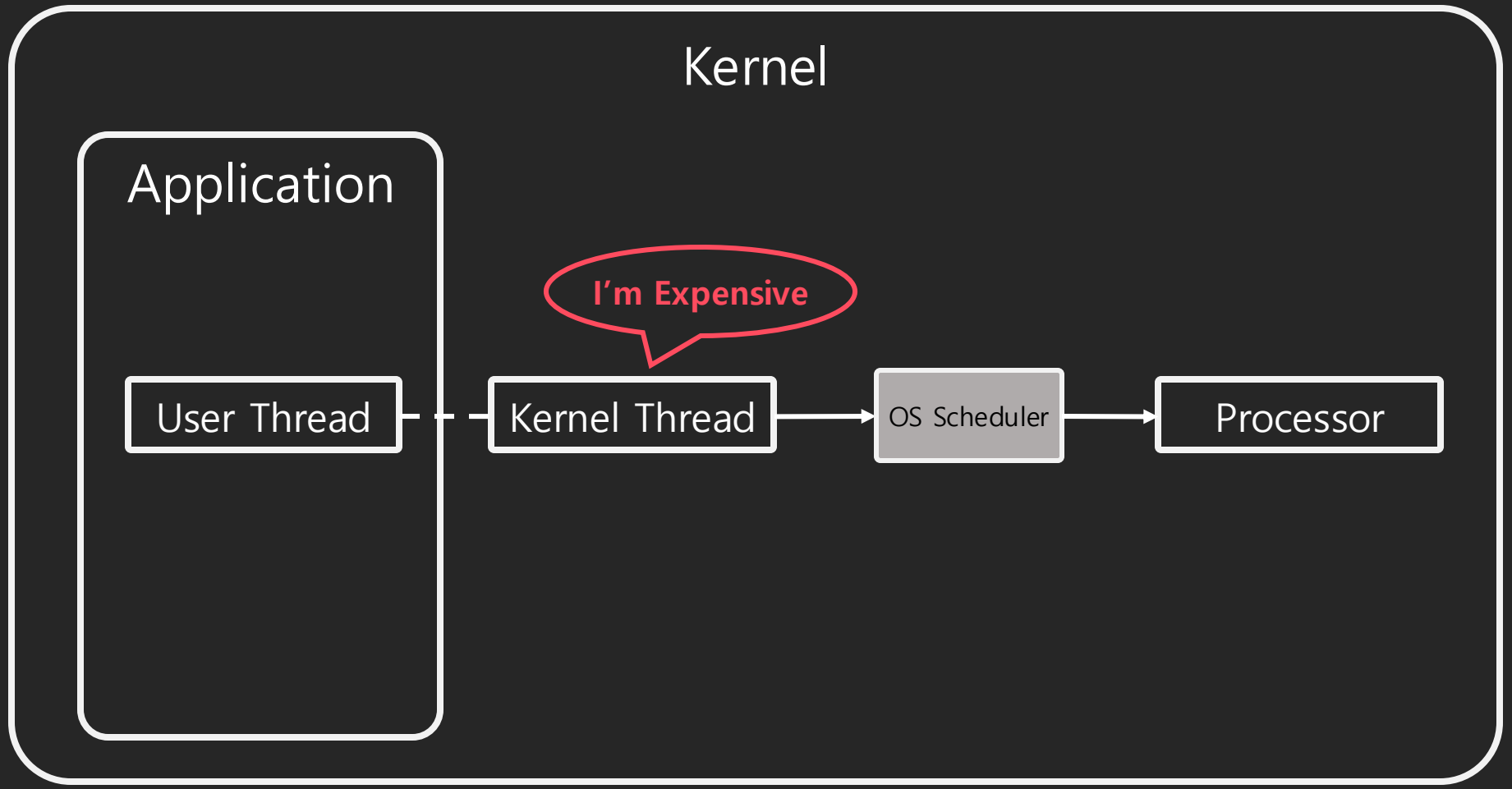
Kernel-level Thread vs User-level Thread



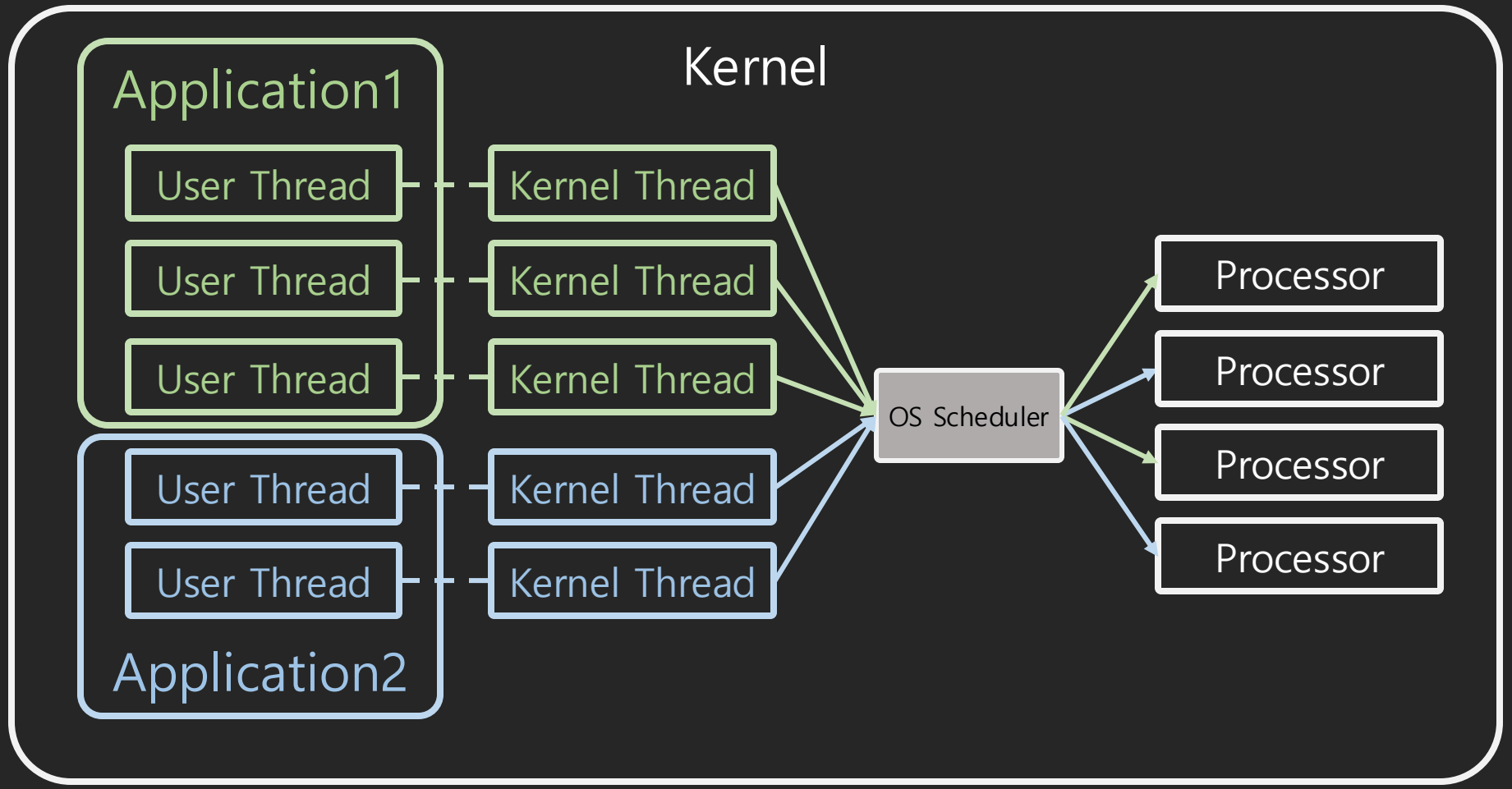
Kernel-level Thread vs User-level Thread



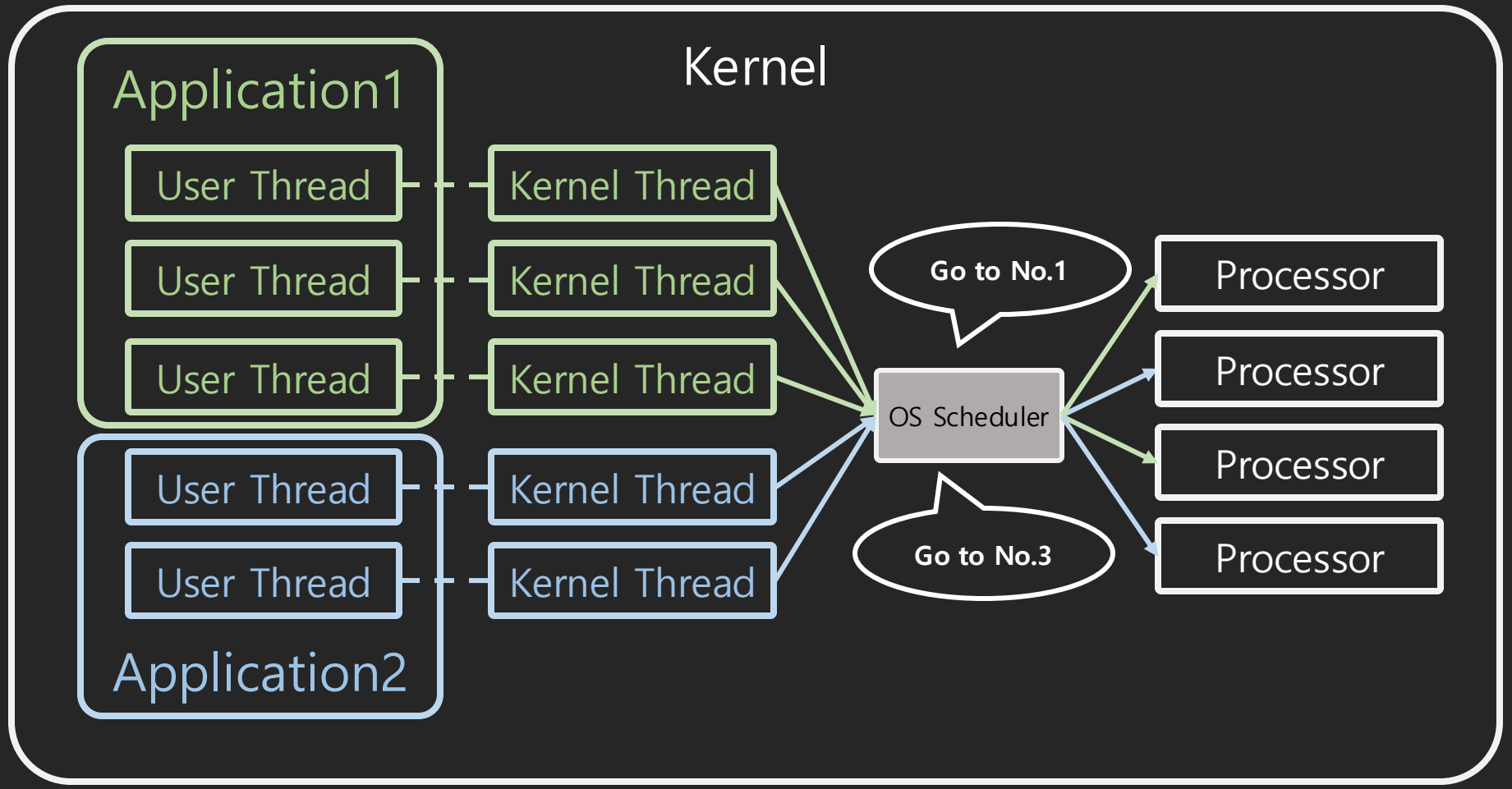
Kernel-level Thread vs User-level Thread



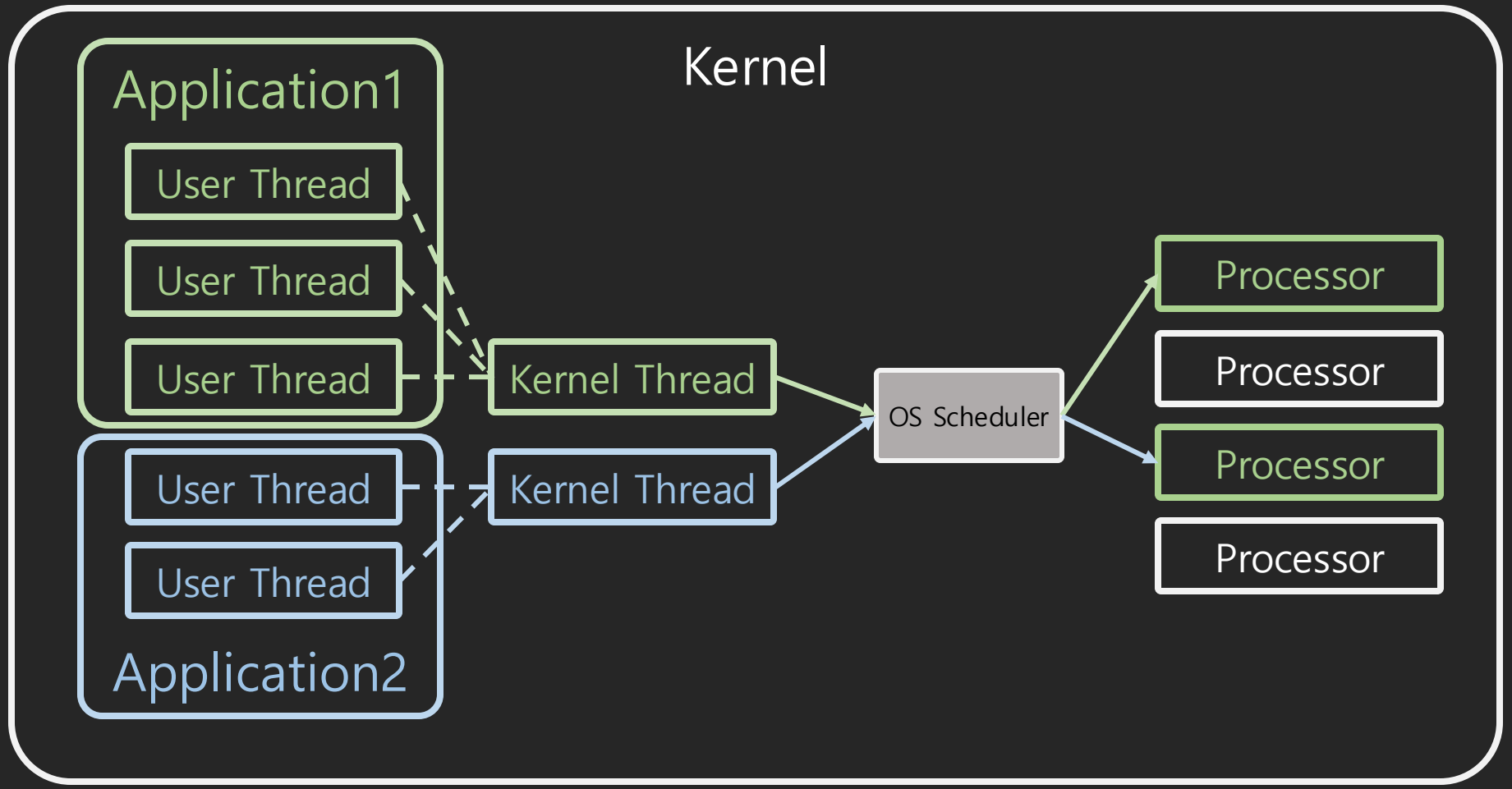
Kernel-level Thread vs User-level Thread



Kernel-level Thread vs User-level Thread

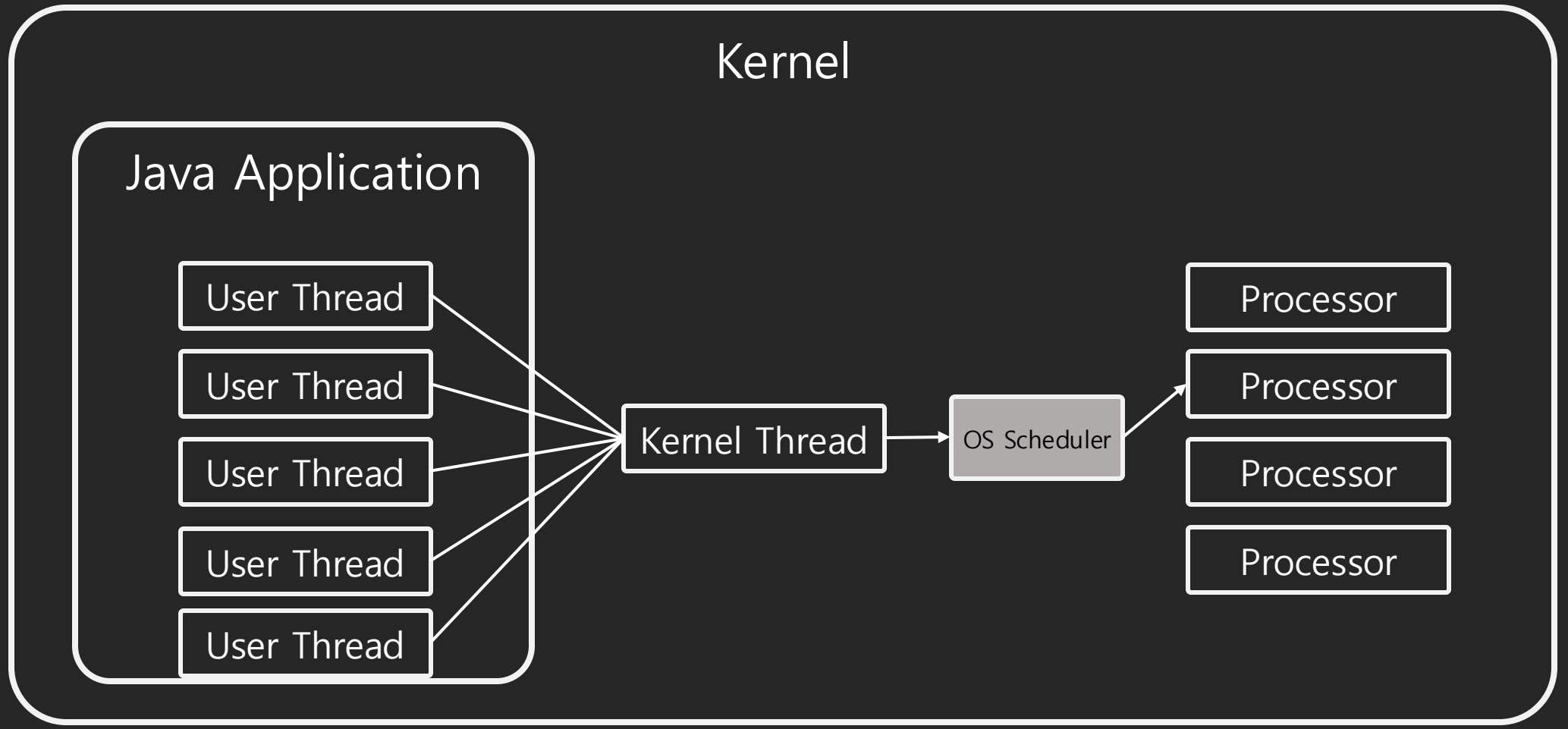


Kernel-level Thread vs User-level Thread



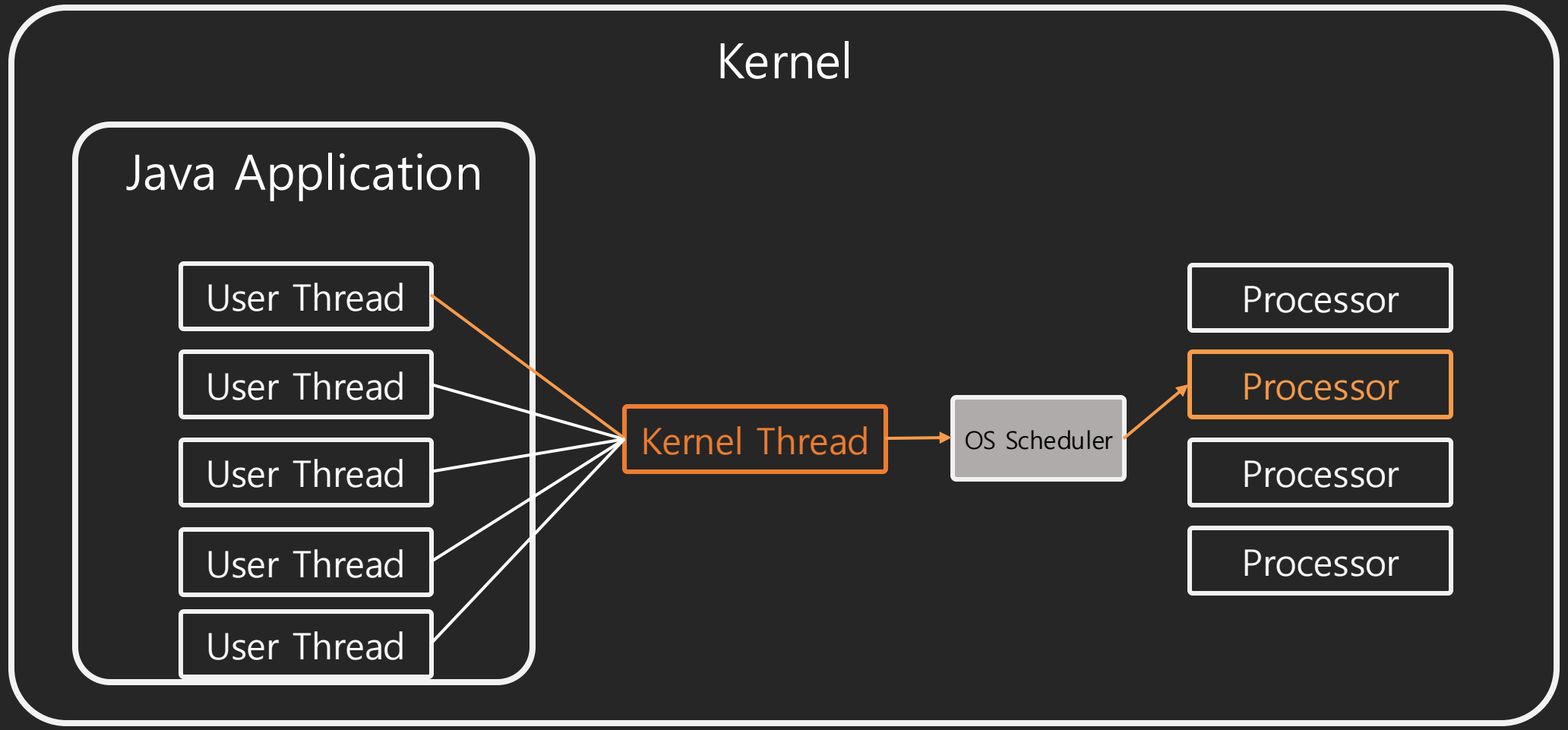
Java Thread Model

Green Thread



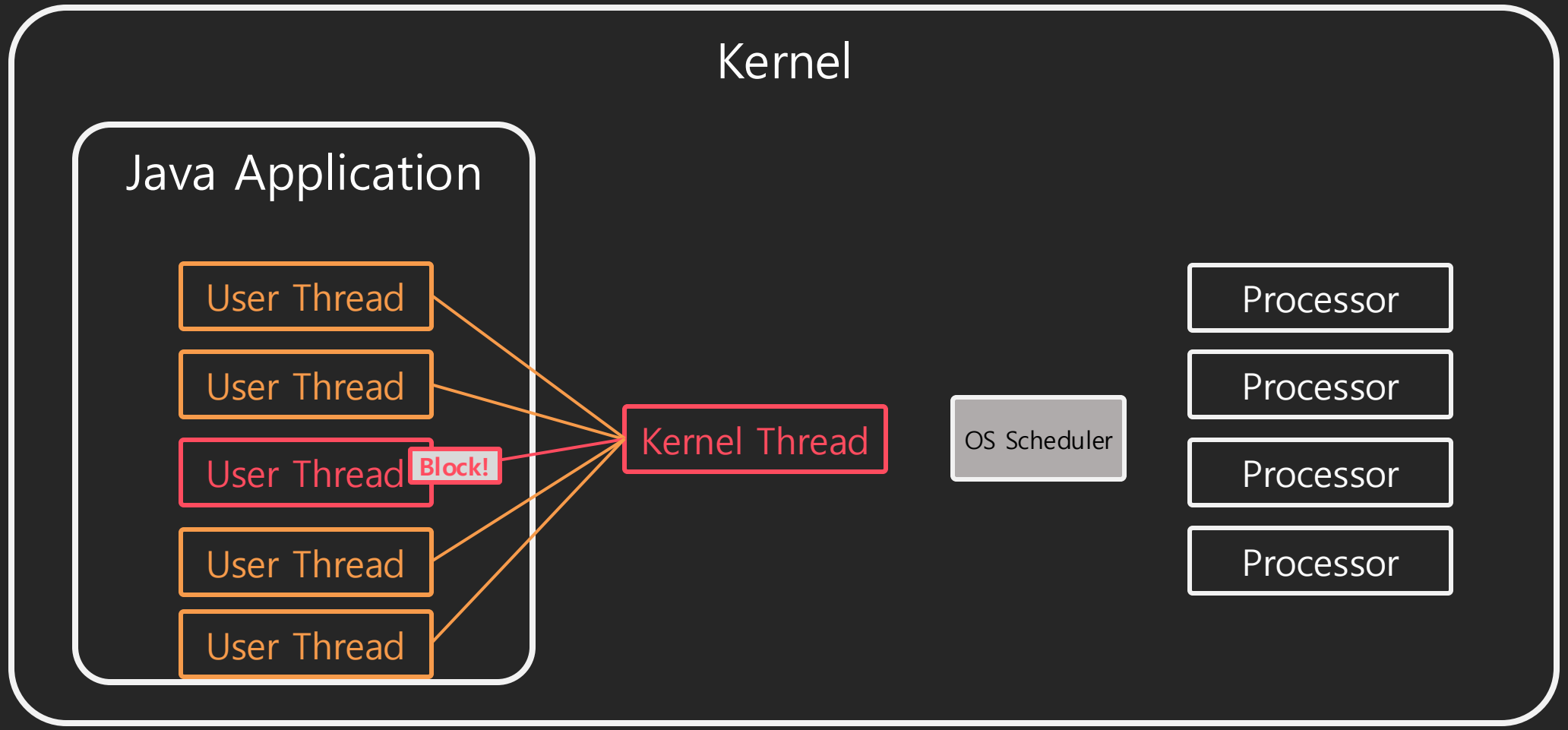
- User Thread and Kernel Thread are mapped many-to-one.
- At Java 1.1~1.2, Green Thread was used as default thread-model
- Since Java 1.3, Green Thread was abandoned

Green Thread



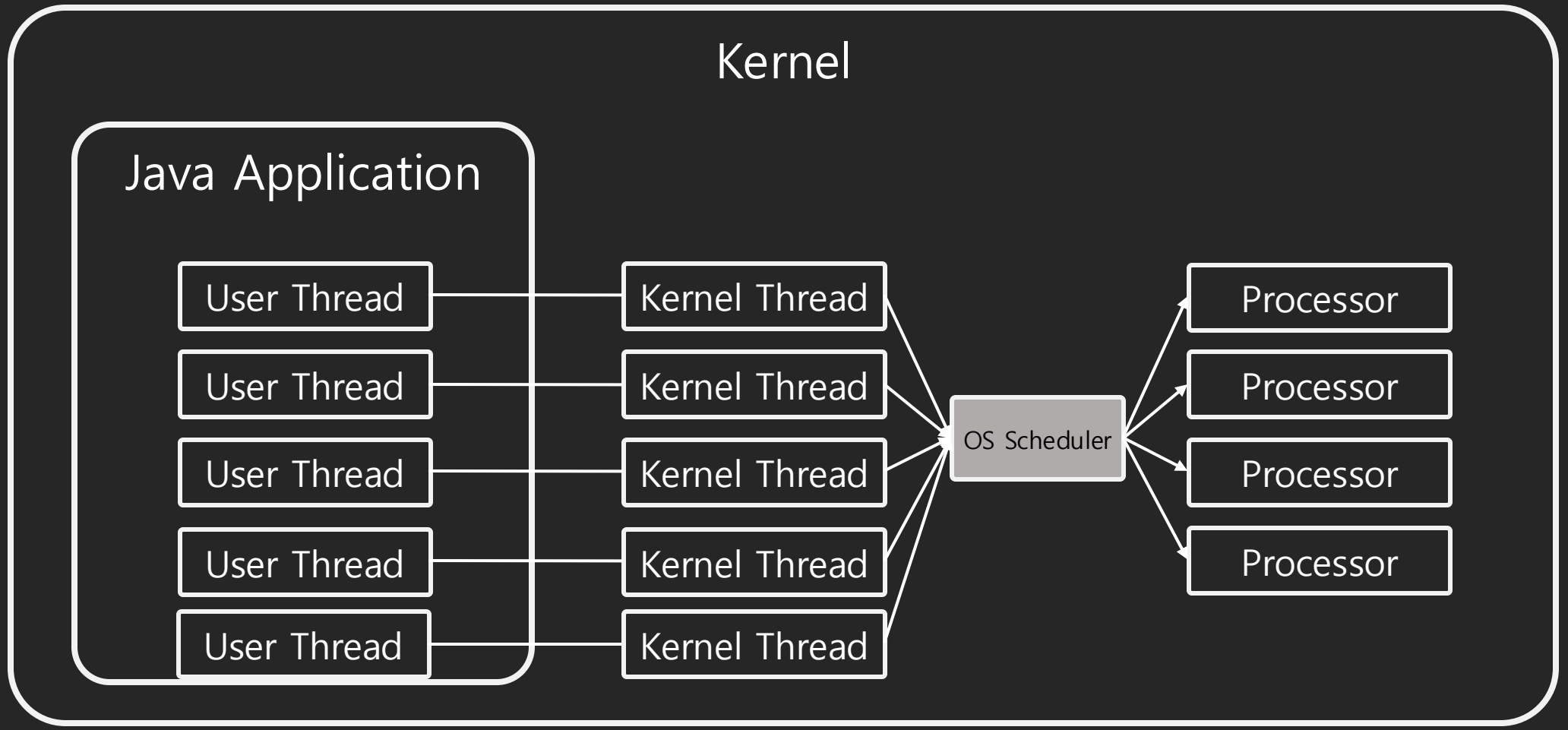
- Light-weight
  - Context-Switching happens on JVM, not Kernel
  - Only one Kernel Thread
- Since there is only one Kernel Thread, only one processor works even if it is a multi-processor

Green Thread



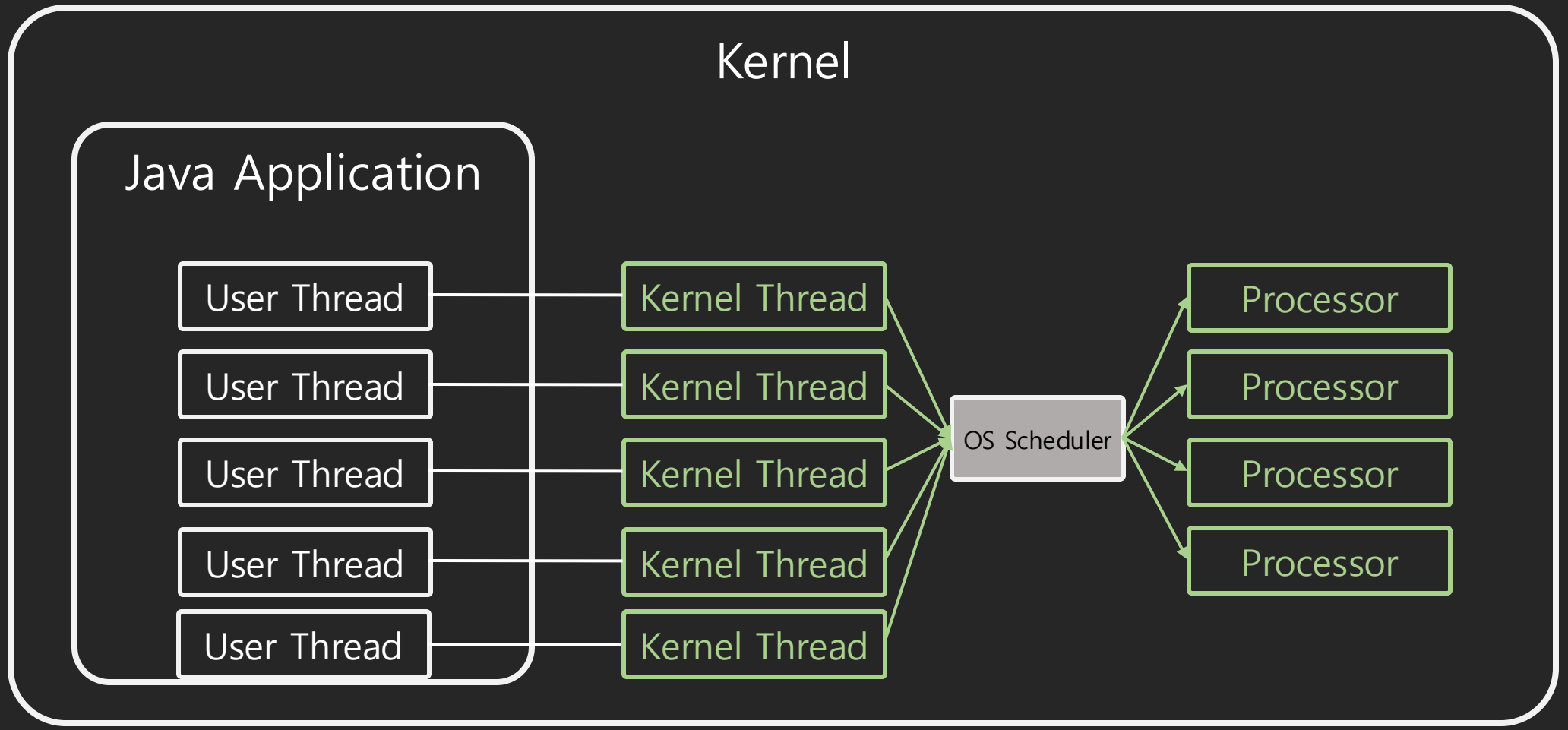
- If one thread is blocked, all threads are blocked.

Native Thread (Platform Thread)



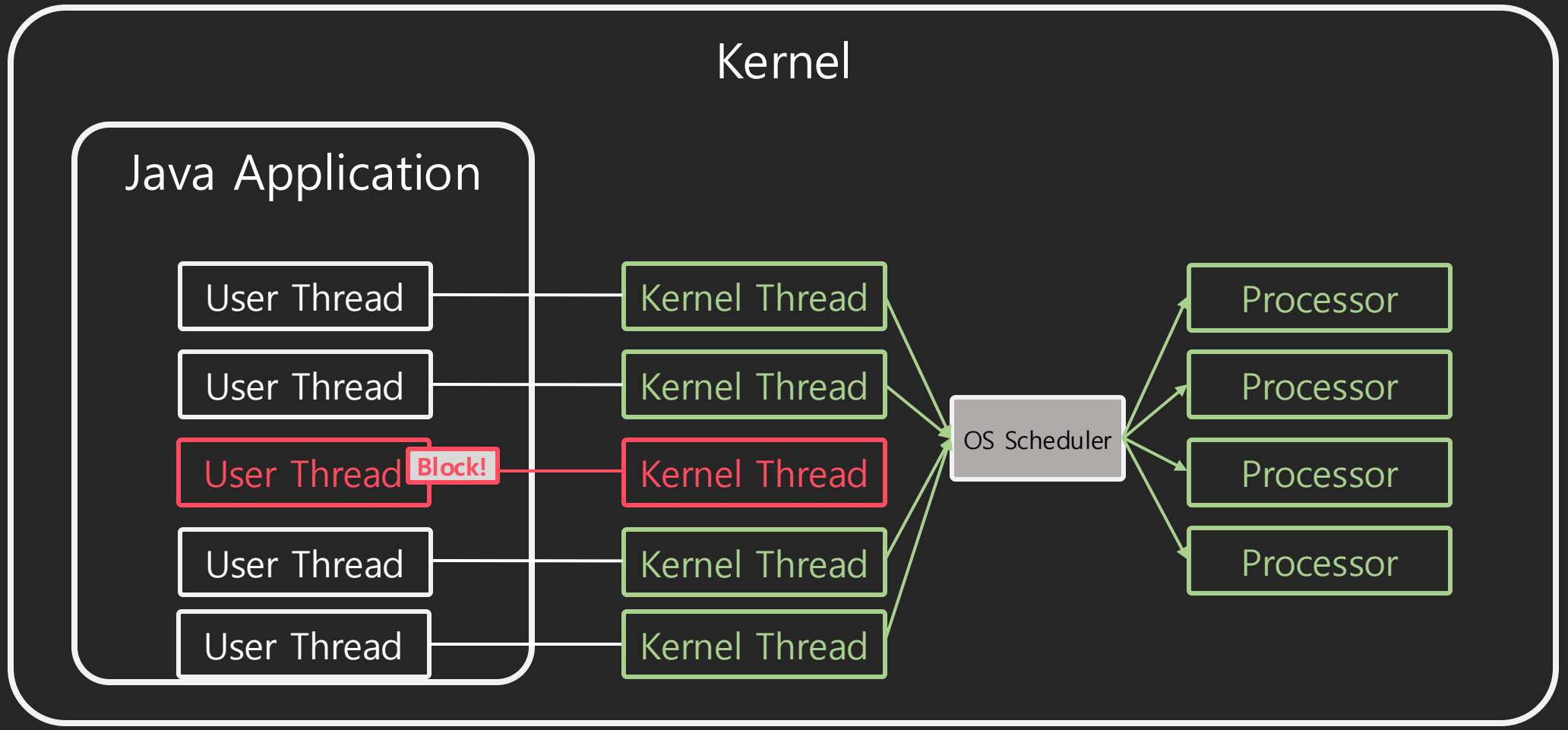
- User thread and kernel thread are mapped one-to-one.
- Appeared from Java 1.2 and used by default from Java 1.3.

Native Thread (Platform Thread)



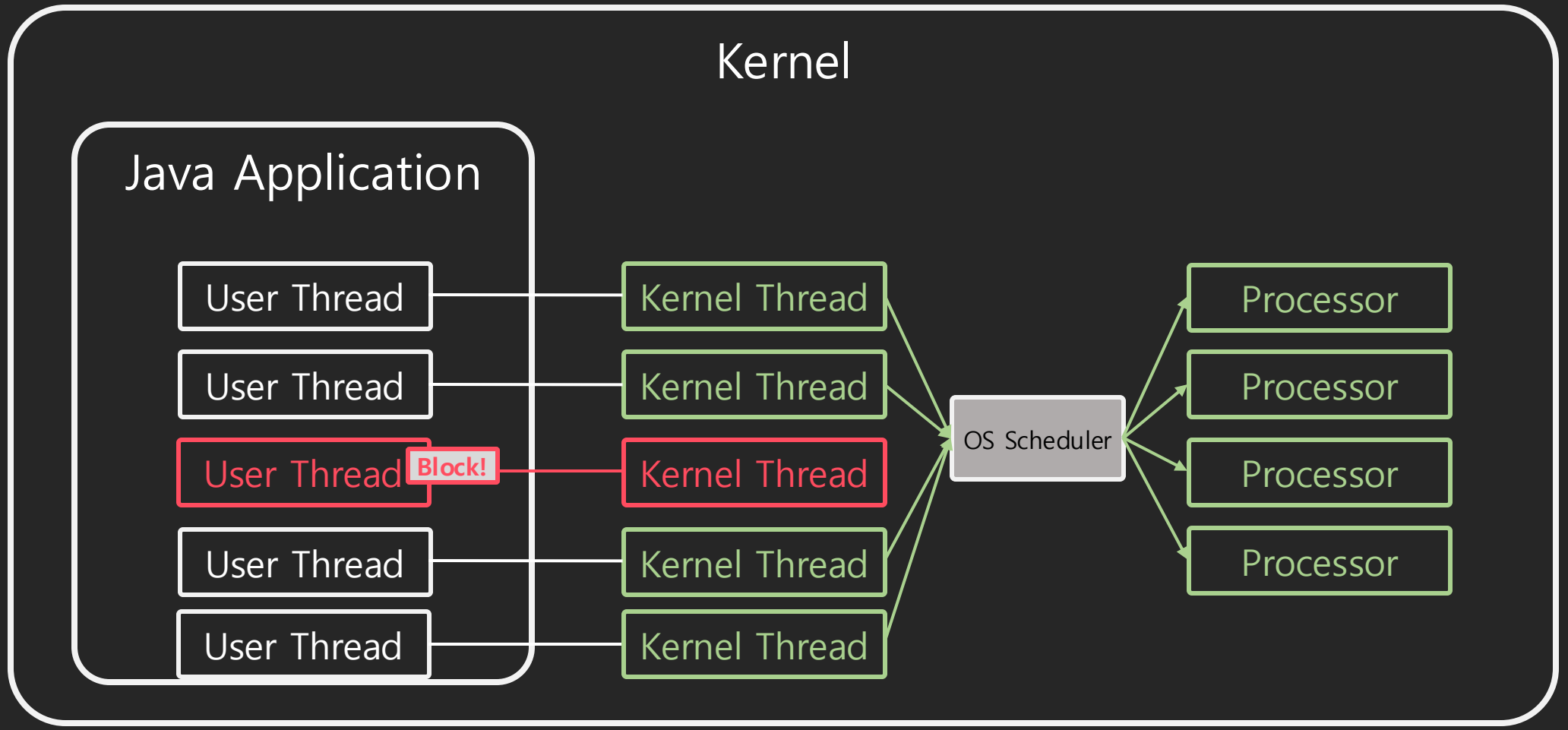
- Support parallelism as scheduling on multiple processors

Native Thread (Platform Thread)



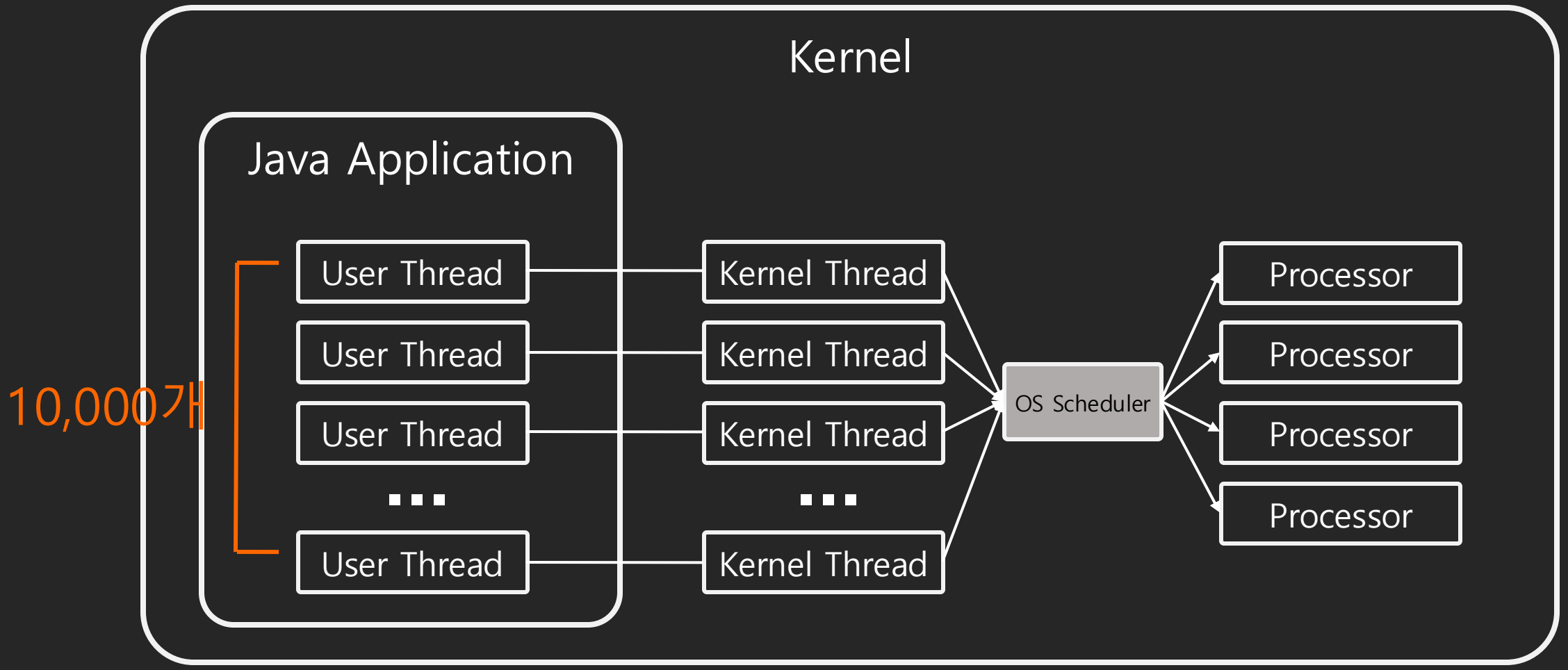
- Even if one thread is blocked, other threads continue to run as is.
- If we have enough resources (e.g. Memory), you can create as many as we want.

Native Thread (Platform Thread)



- Even if one thread is blocked, other threads continue to run as is.
- If we have enough resources (e.g. Memory), you can create as many as we want.

Native Thread (Platform Thread)



- Context-Switching happens on Kernel-level. So, expensive.

Native Thread (Platform Thread)

- Need a lot of Stack Memory
  - Q. Basically, how much stack memory is used per thread??

Native Thread (Platform Thread)

- Need a lot of Stack Memory
  - Q. Basically, how much stack memory is used per thread??

1MB / thread

200 threads = 200 MB
10,000 threads = 10GB

Native Thread (Platform Thread)

- Need a lot of Stack Memory
  - Q. Basically, how much stack memory is used per thread??

1MB / thread

200 threads = 200 MB
10,000 threads = 10GB

  - It can be depend on OS, JVM Vendor, and Java Version
    - Linux/MacOS/Oracle Solaris: 1MB
    - Windows can be different

  - We can adjust using JVM Optino '-Xss'
    - However, there is a minimum/maximum value. If it is out of bounds, an error occurs (this also seems to vary depending on the OS/JDK version)

```
➜  ~ java -Xss1k -version

The Java thread stack size specified is too small. Specify at least 208k
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
```

Configuring Stack Sizes in the JVM - Baeldung

Stack Memory Usage by Thread counts

1000 Threads
1GB Stack Mem.

```
yhc94@DESKTOP-5LQRGK9 MINGW64 ~/IdeaProjects/java-concurrency-practice (master)
$ jcmd 44816 VM.native_memory | grep stack
                            (stack: reserved=1080320KB, committed=80196KB)
```

2000 Threads
2GB Stack Mem.

```
yhc94@DESKTOP-5LQRGK9 MINGW64 ~/IdeaProjects/java-concurrency-practice (master)
$ jcmd 9640 VM.native_memory | grep stack
                            (stack: reserved=2104320KB, committed=156124KB)
```

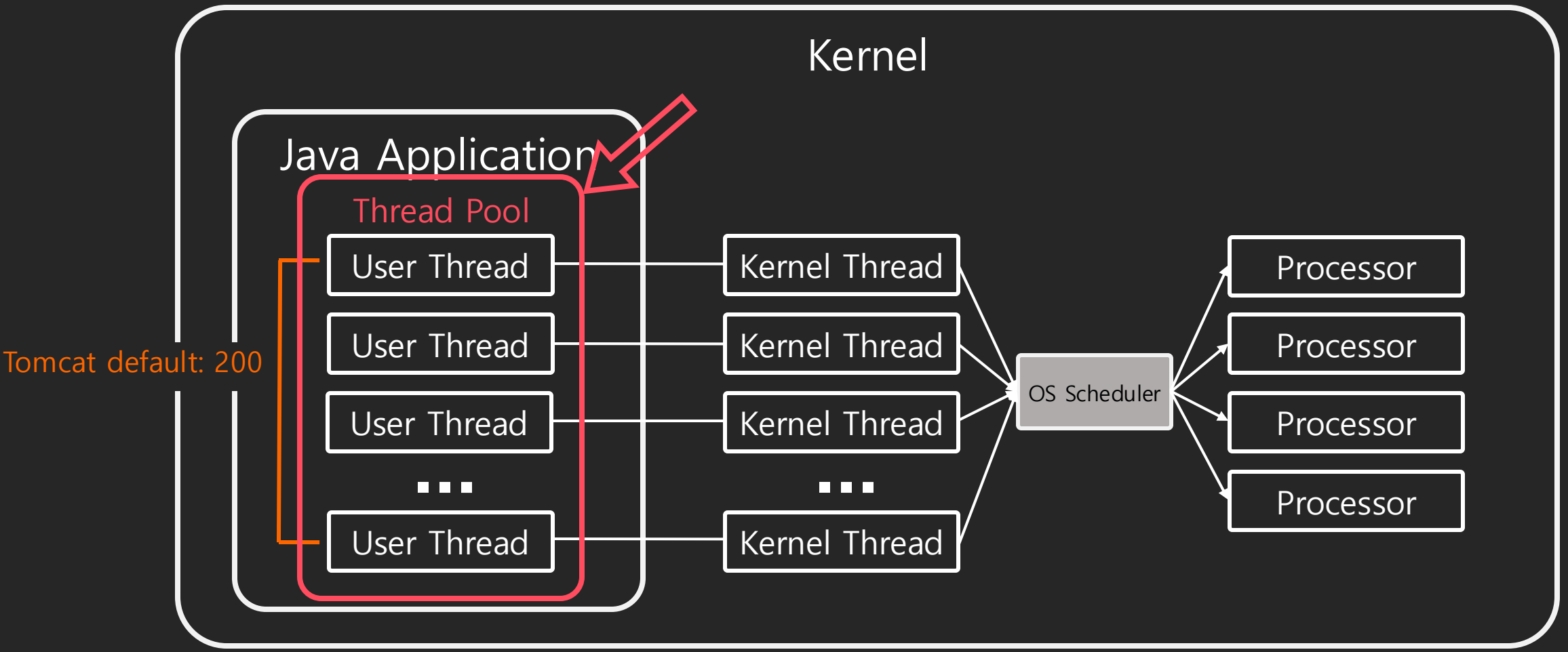
4000 Threads
4GB Stack Mem.

```
yhc94@DESKTOP-5LQRGK9 MINGW64 ~/IdeaProjects/java-concurrency-practice (master)
$ jcmd 29792 VM.native_memory | grep stack
                            (stack: reserved=4154368KB, committed=276560KB)
```

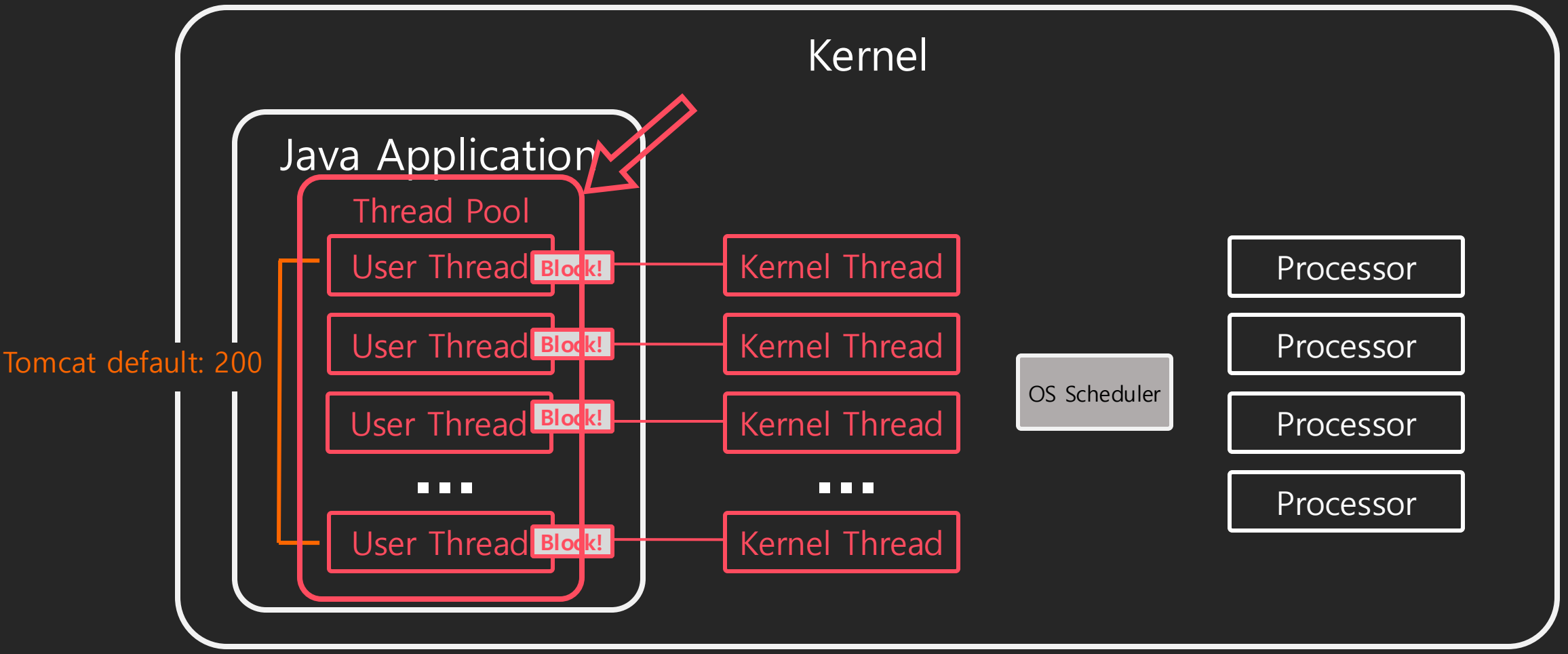
...

10,000 Threads
10GB Stack Mem.

Native Thread (Platform Thread) – Thread Pool

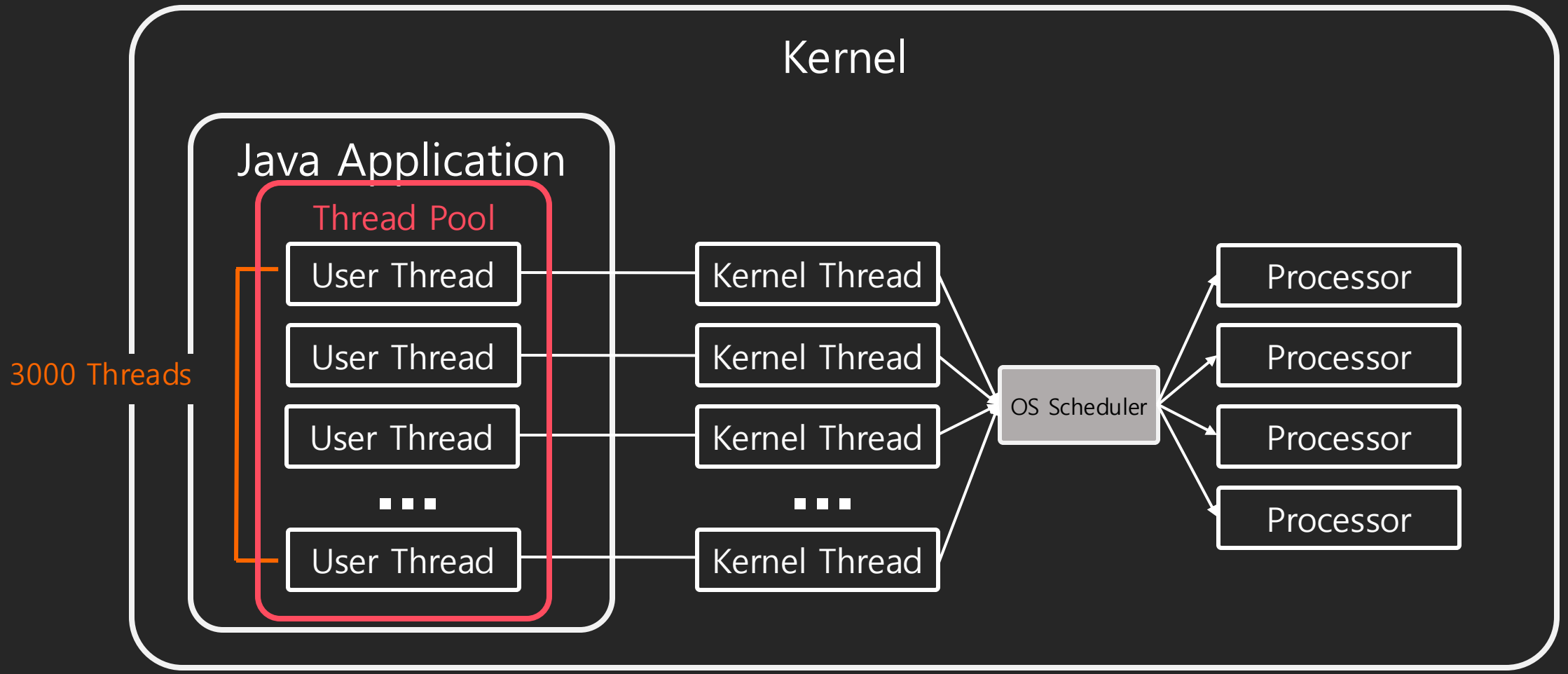


Native Thread (Platform Thread) – Thread Pool



- If we use Thread Pool to limit memory usage of Threads,
  if there are a lot of requests, Threads will be exhausted and bottleneck will occur.

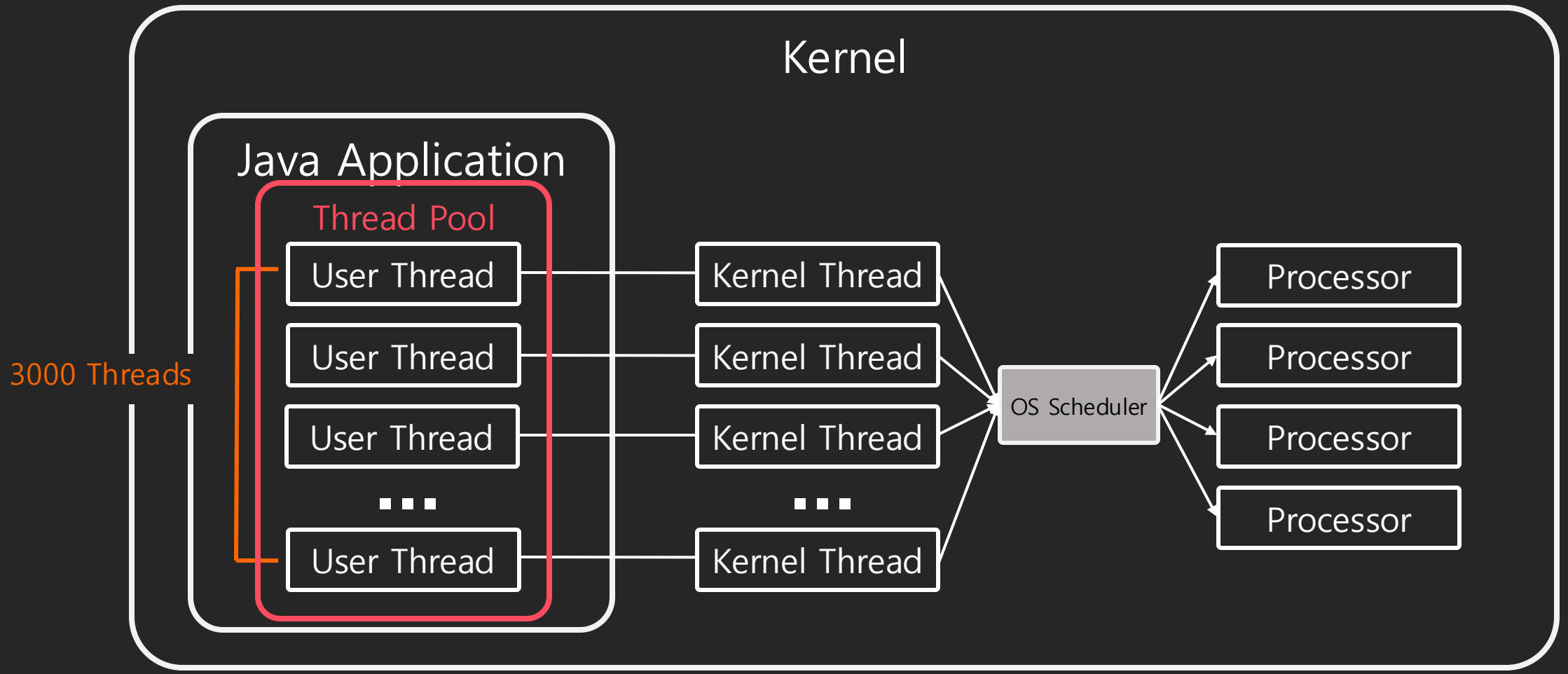
Native Thread (Platform Thread) – Example



Each instance: 3,000 threads = 1GB stack memory

Total: 45 instances -> 135GB stack memory

Native Thread (Platform Thread) – Example



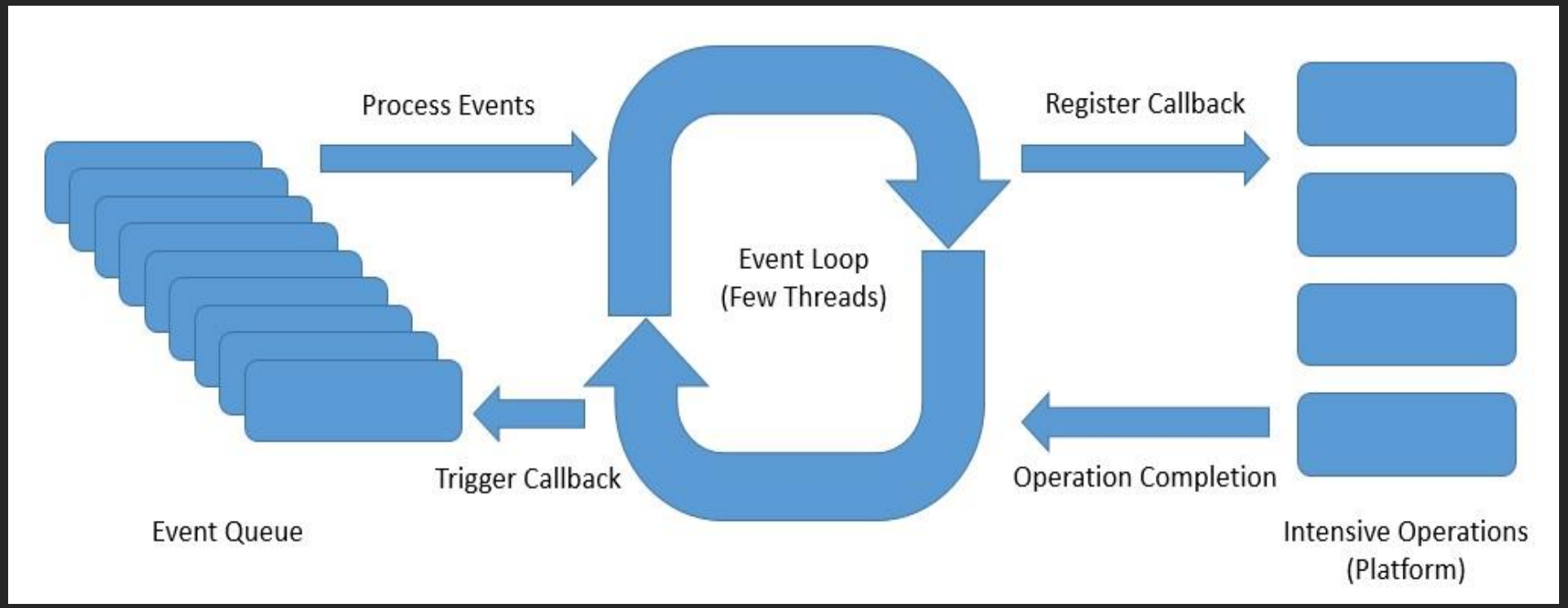
Each instance: 3,000 threads = 1GB stack memory

Total: 45 instances -> 135GB stack memory

Event-Loop

(feat. Netty, Event-Loop, Reactive Programming)

Event Loop



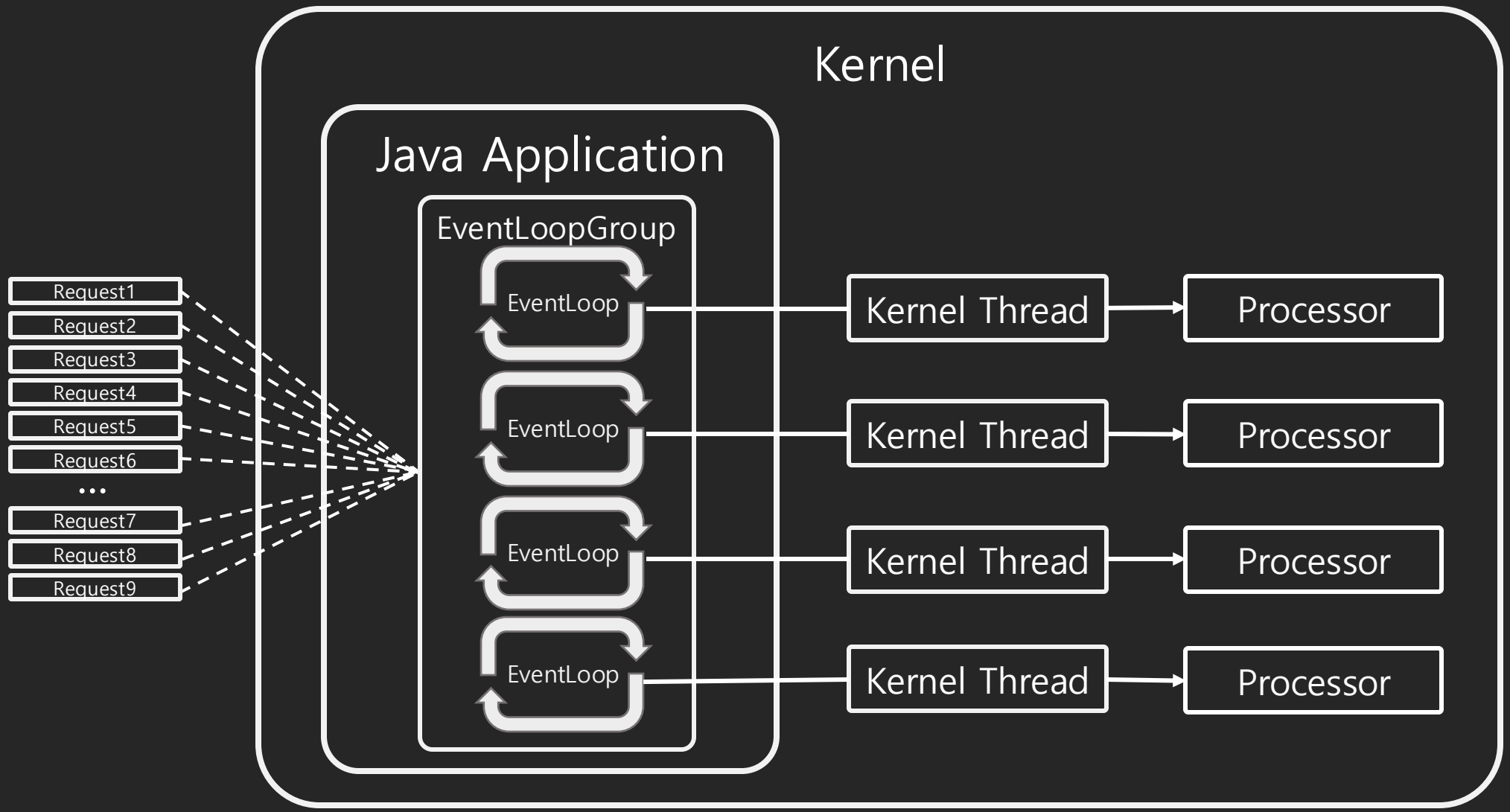
https://www.baeldung.com/spring-webflux-concurrency

Netty is an asynchronous event-driven network application framework
for rapid development of maintainable high performance protocol servers & clients.

Reactor is a reactive library, based on the Reactive Streams specification,
for building non-blocking applications on the JVM

Netty - Thread Model

```
int DEFAULT_IO_WORKER_COUNT = Integer.parseInt(System.getProperty(
        ReactorNetty.IO_WORKER_COUNT,
        "" + Math.max(Runtime.getRuntime().availableProcessors(), 4)));
```



FYI) Netty – Default Event Loop Count

netty

```
DEFAULT_EVENT_LOOP_THREADS = Math.max(1, SystemPropertyUtil.getInt(
        "io.netty.eventLoopThreads", NettyRuntime.availableProcessors() * 2));
```

reactor-netty

```
int DEFAULT_IO_WORKER_COUNT = Integer.parseInt(System.getProperty(
        ReactorNetty.IO_WORKER_COUNT,
        "" + Math.max(Runtime.getRuntime().availableProcessors(), 4)));
```

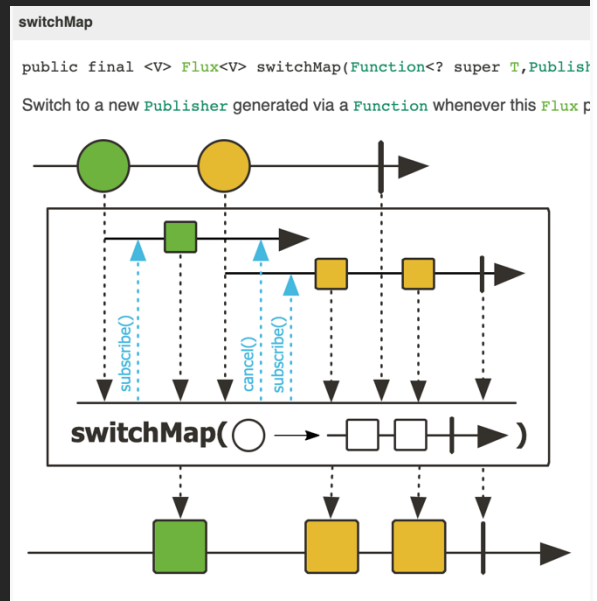

https://projectreactor.io/docs/netty/release/reference/index.html#server-tcp-level-configurations-event-loop-group

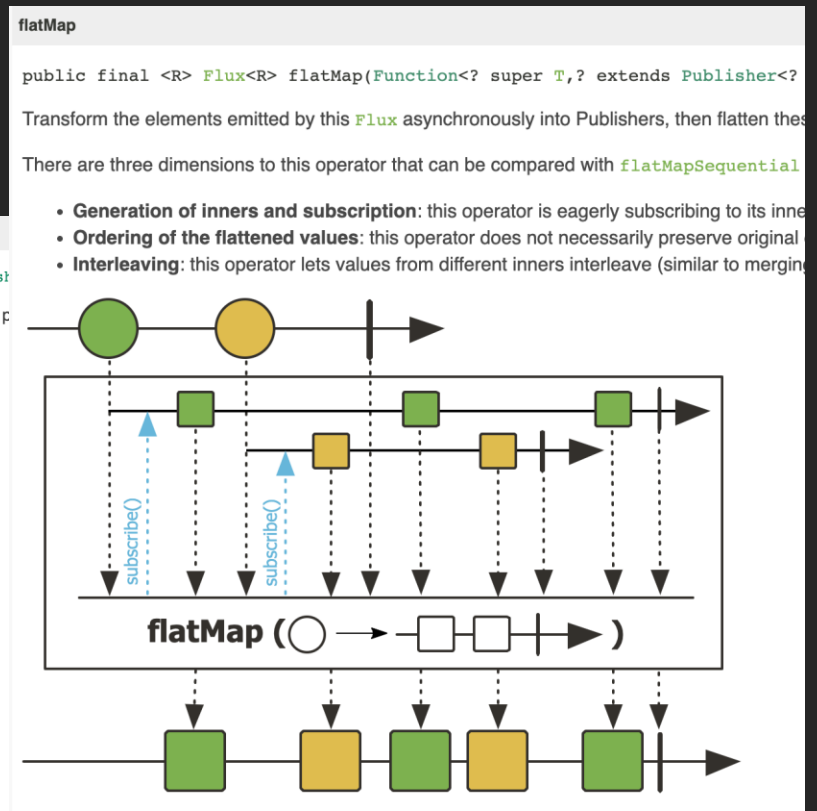
Reactive Programming Pros.

- **Support various convenience features for asynchronous events and data streams**
- **Good compatibility with functional programming**
- **Use much few threads**
  - **Achieve high concurrency even with few memory**
  - **Low-cost context-switching**

Reactive Programming Cons. – Learning Curve

- **Reactive Programming?**
- **Reactive Streams?**
- **ReactiveX? RxJava? Reactor?**
- **Mono? Flux?**
- **flatMap? switchMap?**





Reactive Programming Cons. – Complicated Code

```java
public List<Order> findUserOrdersByTelNo(String telNo) {
    User user = userRepo.findUserByTel(telNo);
    return orderRepo.findOrders(user);
}
```

Blocking + Sync

```java
public CompletableFuture<List<Order>> findUserOrdersByTelNoAsync(String telNo) {
    return userRepo
            .findUserByTelAsync(telNo)
            .thenCompose(user → orderRepo.findOrdersAsync(user));
}
```

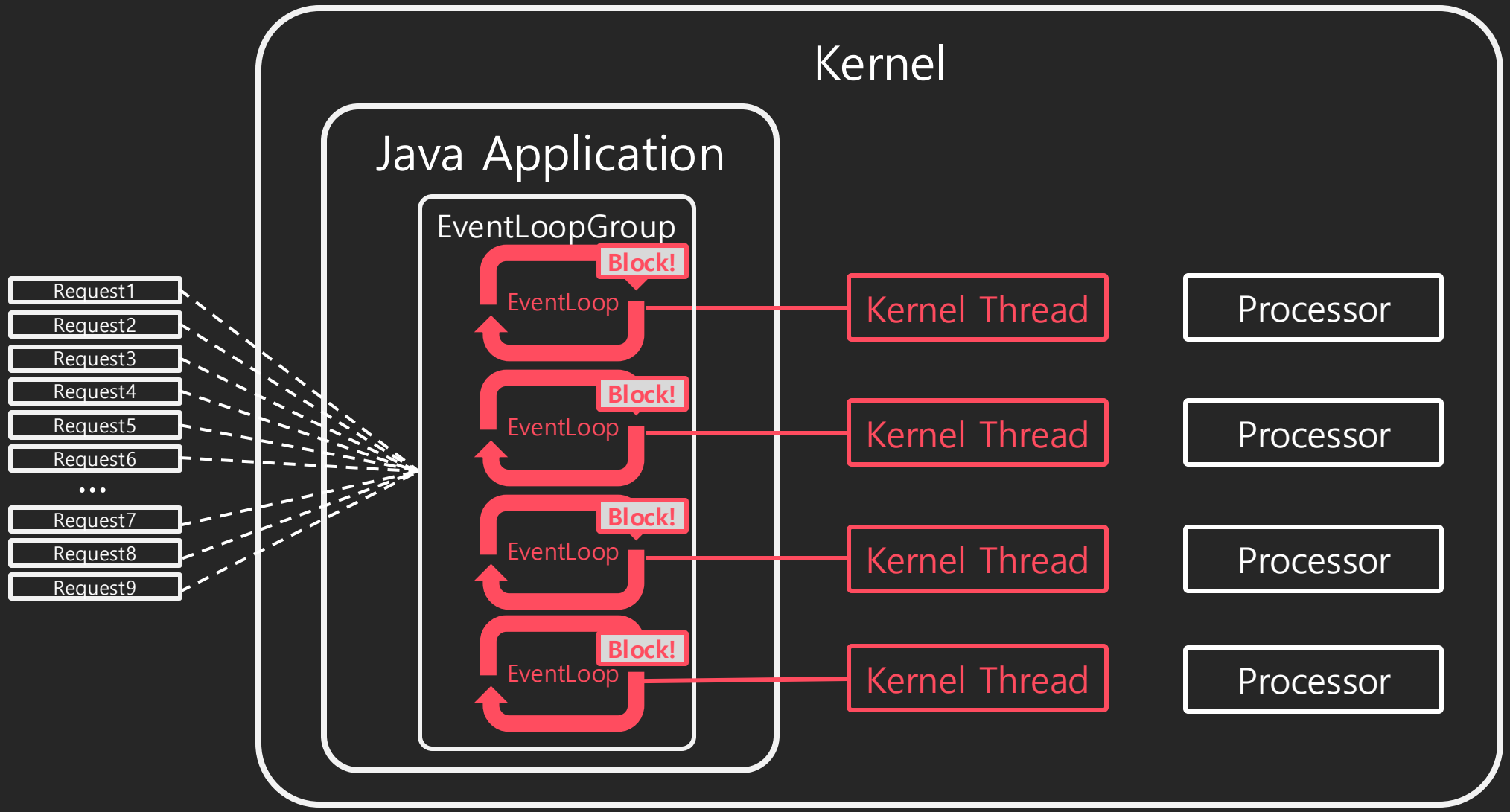
Non-Blocking + Async

http://gunsdevlog.blogspot.com/2020/09/reactive-streams-reactor-webflux.html

Reactive Programming Cons. – Hard to Debug & Test

http://gunsdevlog.blogspot.com/2020/09/reactive-streams-reactor-webflux.html

Reactive Programming Cons. – Be careful about Blocking



- **Basically, Reactive Programming MUST not include blocking**
- **Blocking causes critical performance degradation in Reactive Programming**

Reactive Programming Cons. – Library Development

- **Library Developer** should develop both of Non-Reactive and Reactive
  - e.g. Spring Data JPA vs Spring Data R2DBC
  - e.g. MySQL JDBC vs r2dbc-mysql
  - e.g. OpenFeign vs OpenFeign Reactive

- Sometimes **Library Users** have to wait for Reactive Library
  - e.g. r2dbc-mysql, openfeign-reactor

Thread-per-Request? Event Loop?

Thread-per-Request (with Native Thread)	Event Loop

• If we have enough resources (e.g. Memory), you can create as many as we want.
• Intuitive code
• Easy Debugging & Test
	• **Use very few threads**
	  • **Achieve high concurrency even with few memory**
	  • **Low-cose context-switching**

• Context-Switching happens on Kernel-level. So, expensive. • **If we use Thread Pool to limit memory usage of Threads, if there are a lot of requests, Threads will be exhausted and bottleneck will occur.**	• **Learning Curve** • **Complex Code** • **Hard to Debug & Test** • **Shout be careful about Blocking** • **Library Developer should develop both of Non-Reactive and Reactive**

Thread-per-Request? Event Loop?

Thread-per-Request (with Native Thread)	Event Loop
Simple	**Use Resource efficiently** **High-Concurrency**
Resource Problem	**Complicated**

Virtual Thread


Java Virtual Threads
Borislav Stoilov · Follow
Published in CodeX · 17 mi
un 11, 2022
Beyond Million Threads — All you need to know about Virtual Threads
Sankarganes
6 min read
JVM Language Summit 2023
46:53
The Challenges of Introducing Virtual Java Platform - Project Loom
조회수 9.6천회 • 1개월 전
Java
Presented by Alan Bateman - Architect (Java Platform Group - Orac
Intro | Terminology | Implementation | APIs | Progress |...
Coroutines and Loom behind the scenes by Roman Elizarov
조회수 1.6만회 • 4개월 전
Kotlin by JetBrains
OpenJDK Pro
41:21
for task executor convert it to quarantine dispatche...
IntelliJ IDEA Live Stream
Virtual threads and Structured concurrency in Java 21 with Loom
José Paumard
1:14:41
Virtual Threads and Structured Concurrency in Java 21 With Loom
조회수 1.8만회 • 스트리밍 시간: 4개월 전
IntelliJ IDEA by JetBrains
Loom brought 2 preview features in Java 19: virtual threads and structured concurrency. Virtual threads are a new model of thread ...
Java 21 new feature: Virtual Thr
조회수 2.6만회 • 1개월 전
Java
Learn about
자막
Thread virtualThread =
Thread.ofVirtual().unstarted(task);
virtualThread.start();
// do something else
virtualThread.join();
Java 21
Virtual Threads
33:35
45:22
Roman Elizarov
Virtual Threads
Jis Soares · Follow
min read · Aug 1
Thread in J
MEsfandiari · Follow
3 min read · Aug 11
Spring into the Future: Embracing Virtual Threads with Java's Project Loom
조회수 9.7천회 • 5개월 전
Dan Vega
In this tutorial you will lea
12:15
spring

Game Changer in Java Concurrency : Virtual Thread

Richard HC
Order Fulfillment

Project Loom

- Goal
  - support easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform.

- Sub-Tasks
  - Virtual Thread
  - Structured Concurrency
  - Scoped Value

Virtual Thread

Virtual threads are 'lightweight threads' that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications

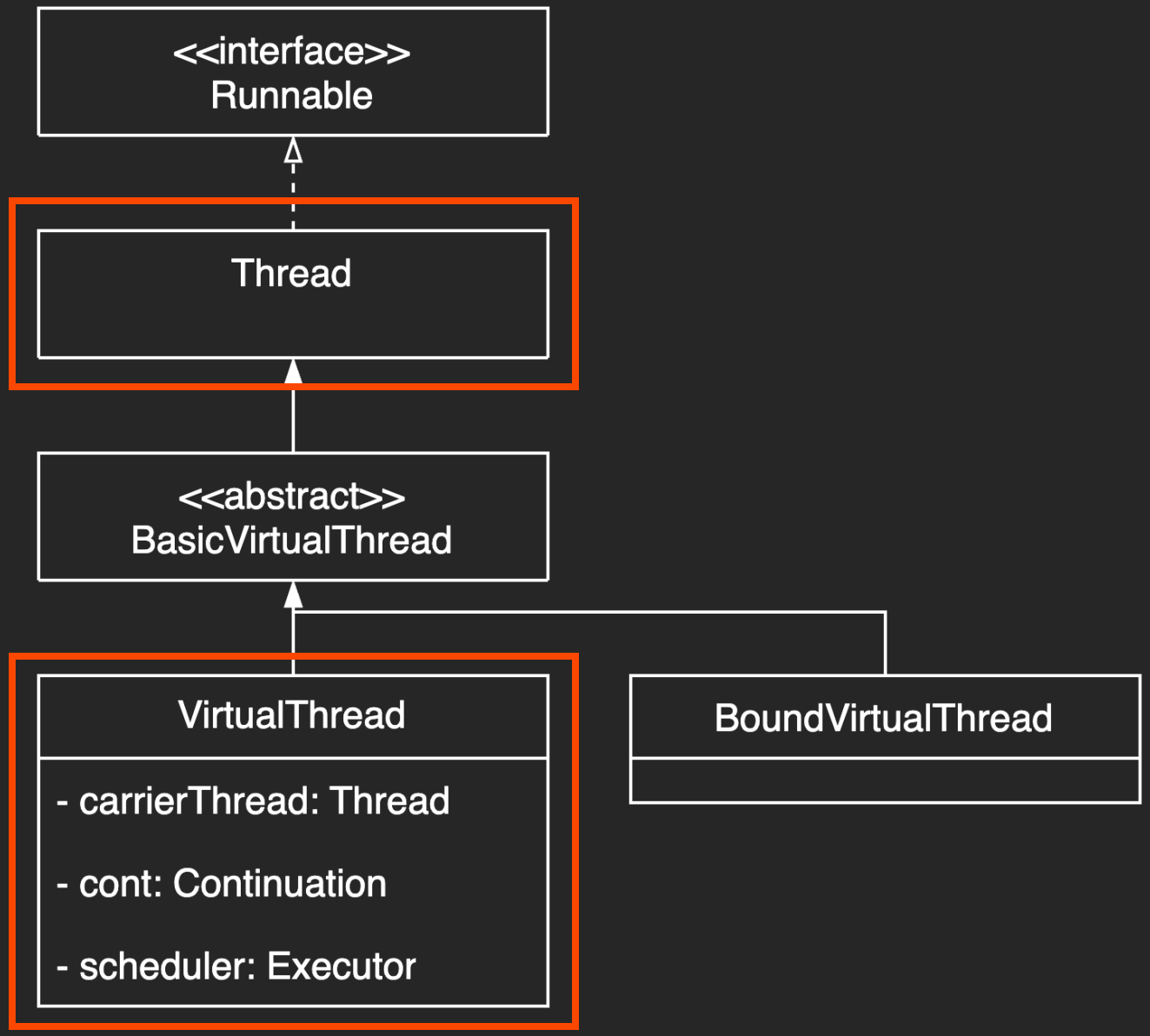
- JEP 425: Virtual Threads (Preview) (JDK 19)
- JEP 436: Virtual Threads (Second Preview) (JDK 20)
- JEP 444: Virtual Threads (JDK 21, LTS)

- Dependencies
  - JEP 353 (Reimplement the Legacy Socket API) (JDK 13)
  - JEP 373 (Reimplement the Legacy DatagramSocket API) (JDK 15)
  - JEP 416 (Reimplement Core Reflection with Method Handles) (JDK 18)
  - JEP 418 (Internet-Address Resolution SPI) (JDK 18)

Virtual Thread - Goal

- **Enable server applications written in the simple thread-per-request style to scale with near-optimal hardware utilization**
- **Enable existing code that uses the java.lang.Thread API to adopt virtual threads with minimal change**
- **Enable easy troubleshooting, debugging, and profiling of virtual threads with existing JDK tools**

Virtual Thread – Class Diagram



Virtual Thread – Getting Started

```
Runnable task = () -> {
    // Your task here
}

// 1)
Thread virtualThread = Thread.ofVirtual().unstarted(task);
virtualThread.start();

// 2)
Thread.ofVirtual().start(task);

// 3)
Thread.startVirtualThread(task);
```

Virtual Thread – Getting Started

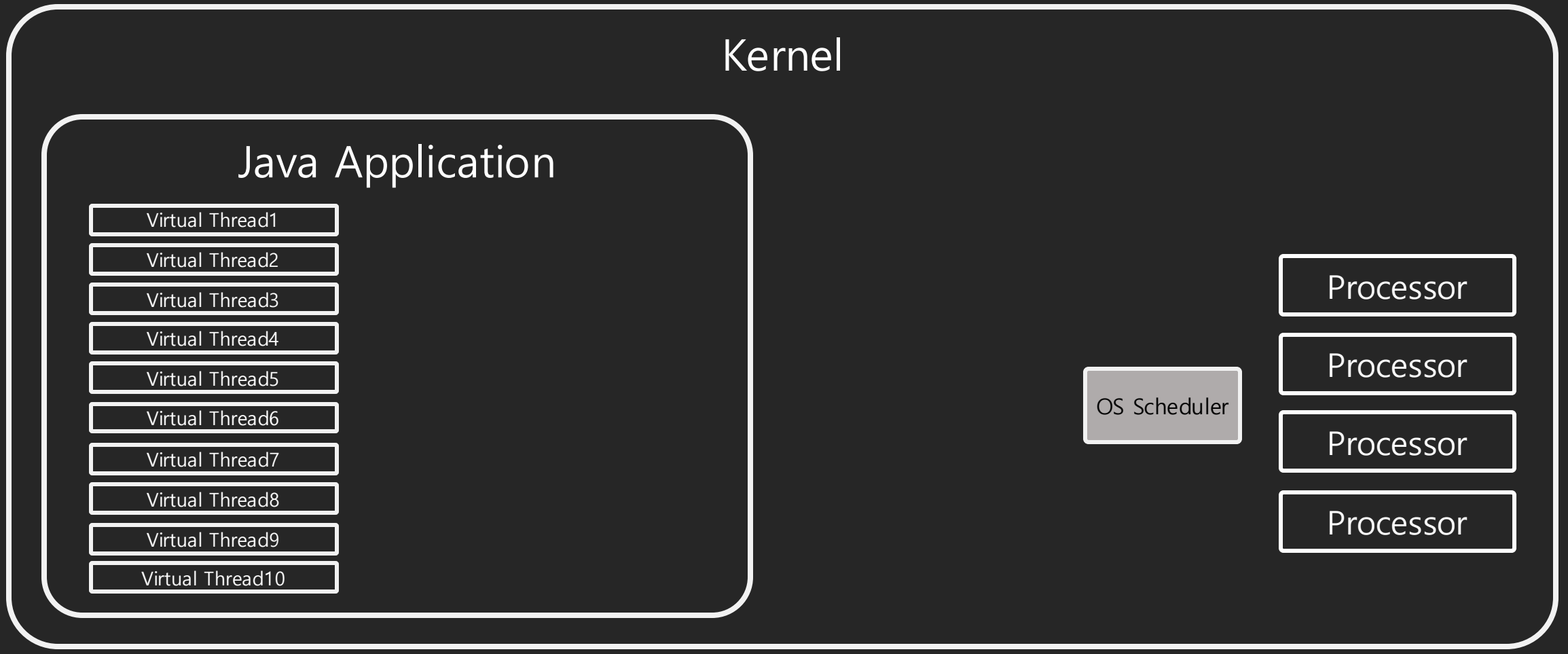
```
Runnable task = () -> {
    // Your task here
}

// 1)
Thread virtualThread = Thread.ofVirtual().unstarted(task);
virtualThread.start();

// 2)
Thread.ofVirtual().start(task);

// 3)
Thread.startVirtualThread(task);

// 4) Using ExecutorService
ExecutorService executorService = Executors.newVirtualThreadPerTaskExecutor();
executorService.execute(task);
```

Virtual Thread – Class Definition

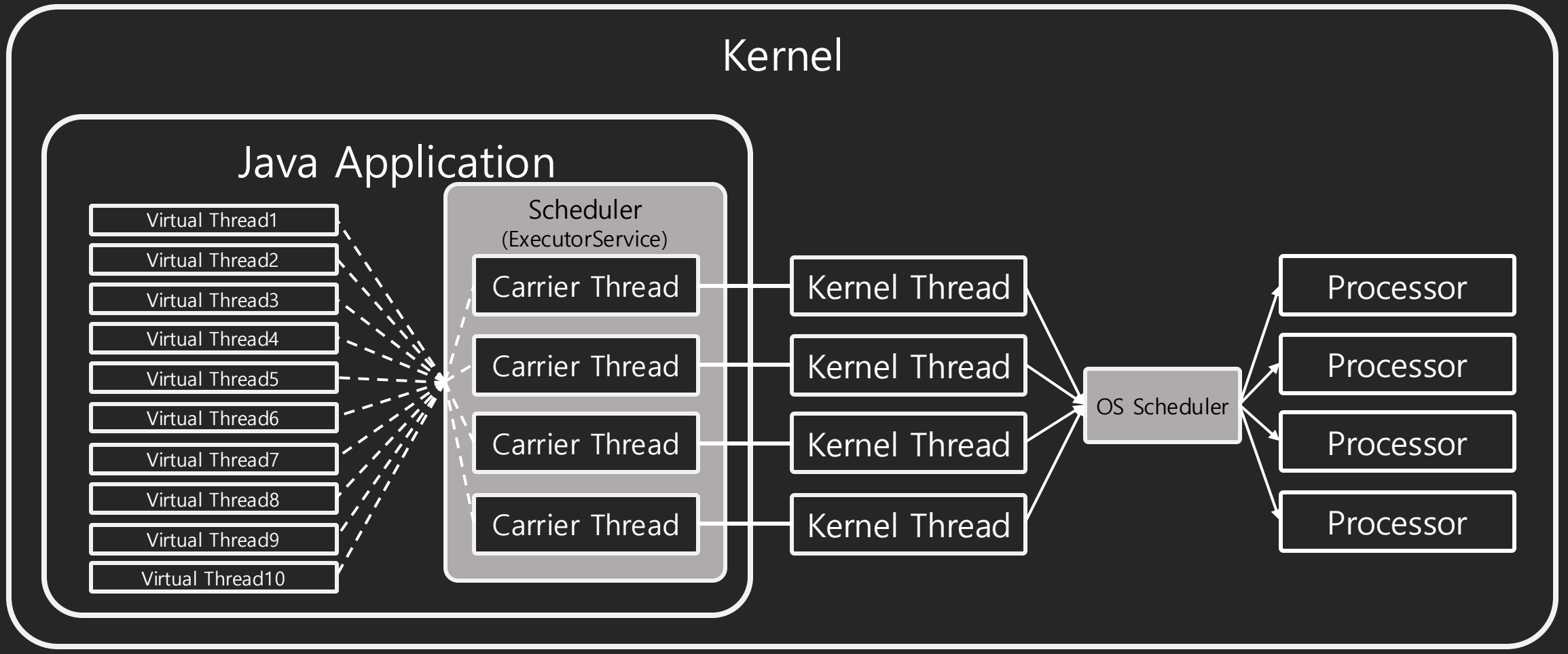
```java
/**
 * A thread that is scheduled by the Java virtual machine rather than the operating
 * system.
 */
final class VirtualThread extends BaseVirtualThread {

    // scheduler and continuation
    private final Executor scheduler;
    private final Continuation cont;
    private final Runnable runContinuation;

    // virtual thread state, accessed by VM
    private volatile int state;

    // carrier thread when mounted, accessed by VM
    private volatile Thread carrierThread;
```

?

Virtual Thread - How it works



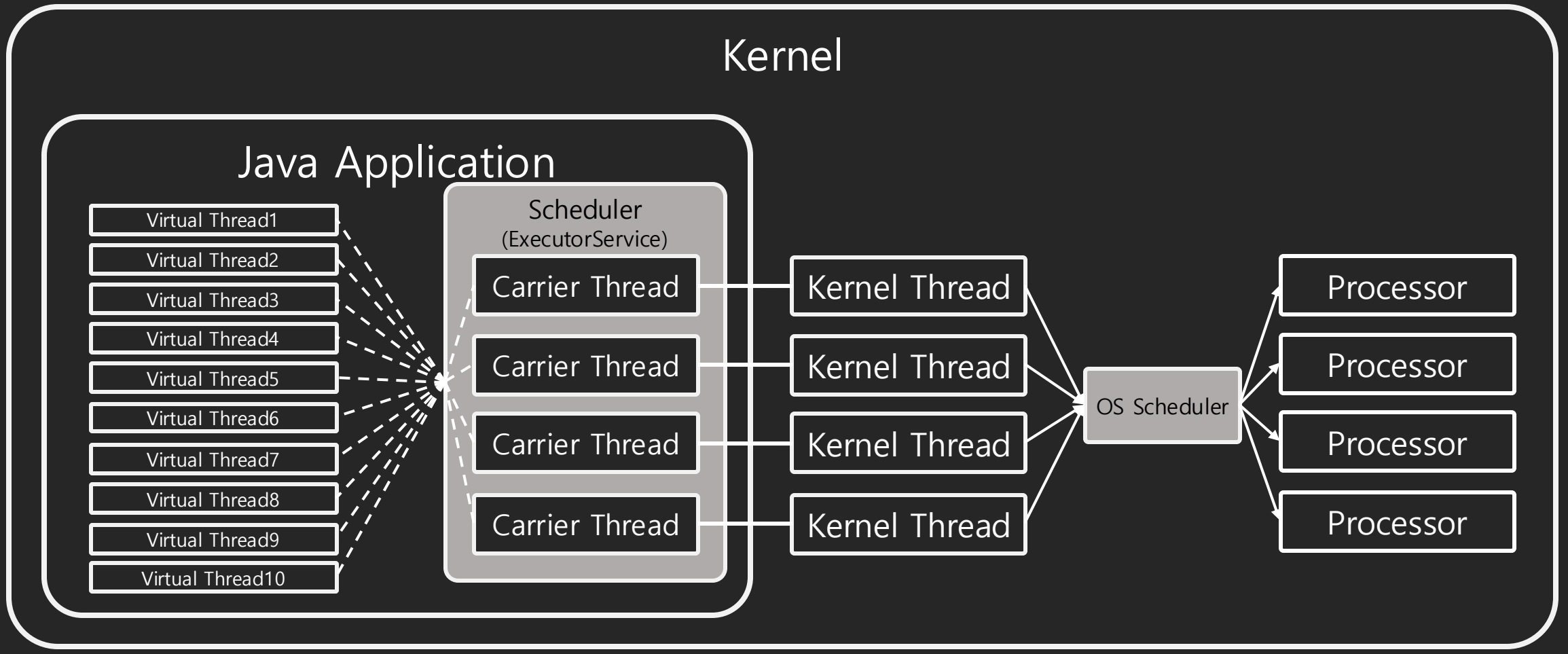
Virtual Thread - How it works



light-weight

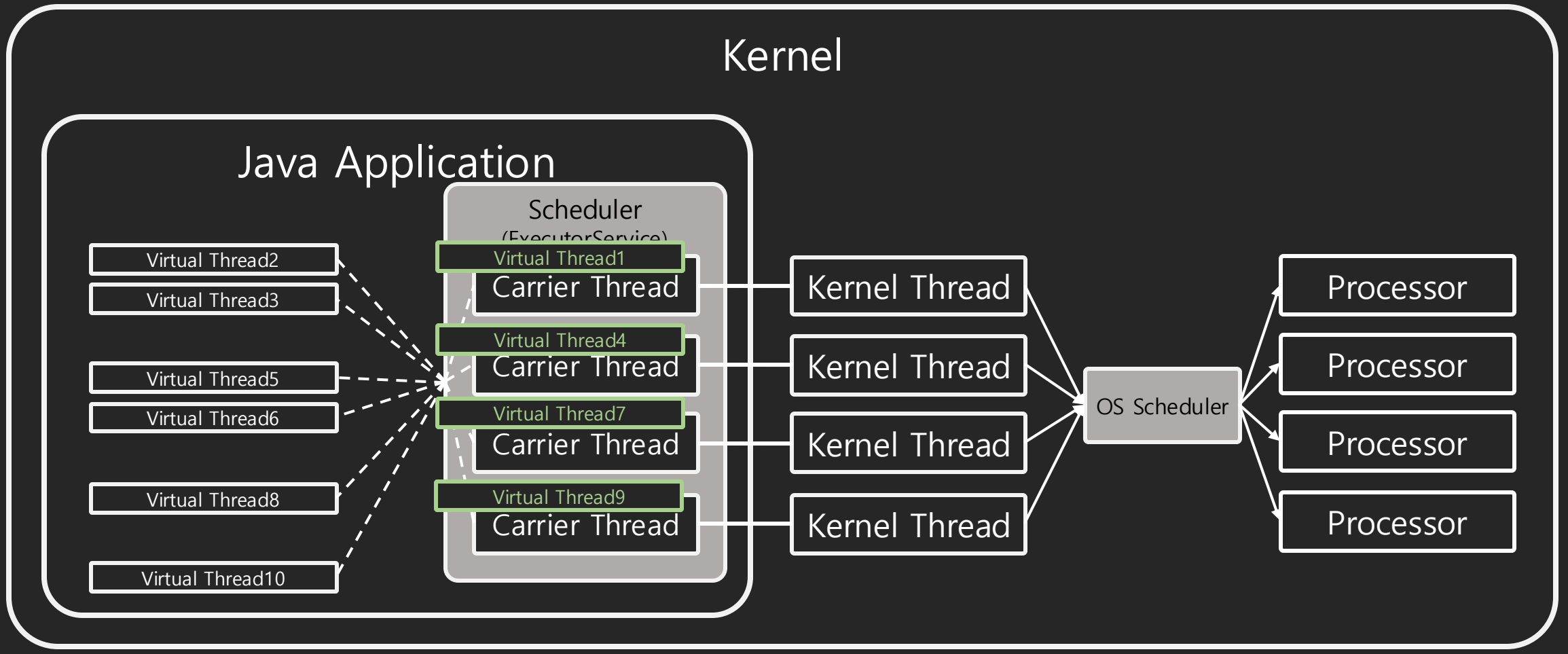
- Few Carrier Threads (default: Carrier Threads count == Processors count)
- Viritaul Thread cost == Java Object cost (hundreds Bytes ~ number of KB)
- Virtual Thread의 Scheduling과 Context Switching도 JVM단에서 이루어지므로 가벼움

Virtual Thread - How it works – create



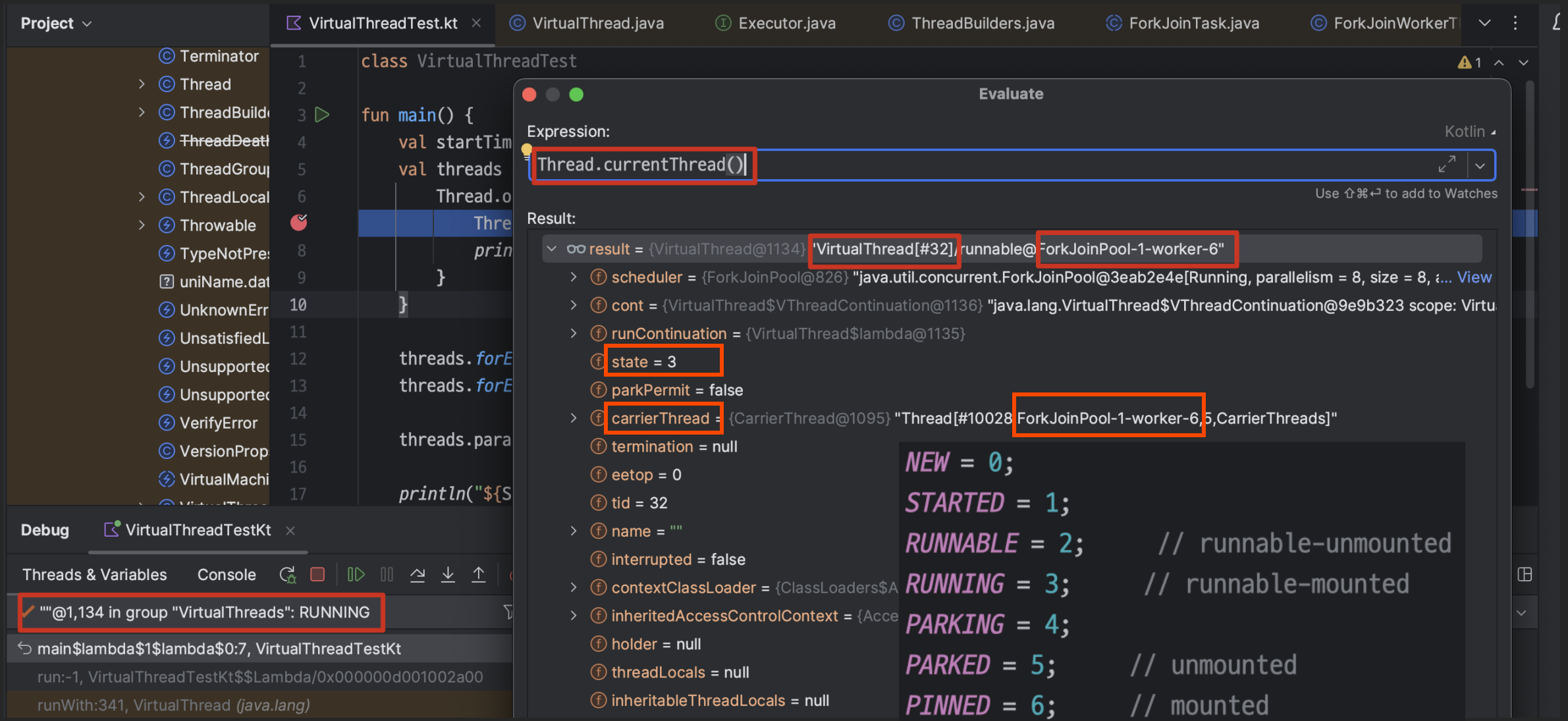

```
Runnable task = () -> { ... // your task }

Thread virtualThread  = Thread.ofVirtual().unstarted(task)
```

Virtual Thread - How it works – start() & mount()

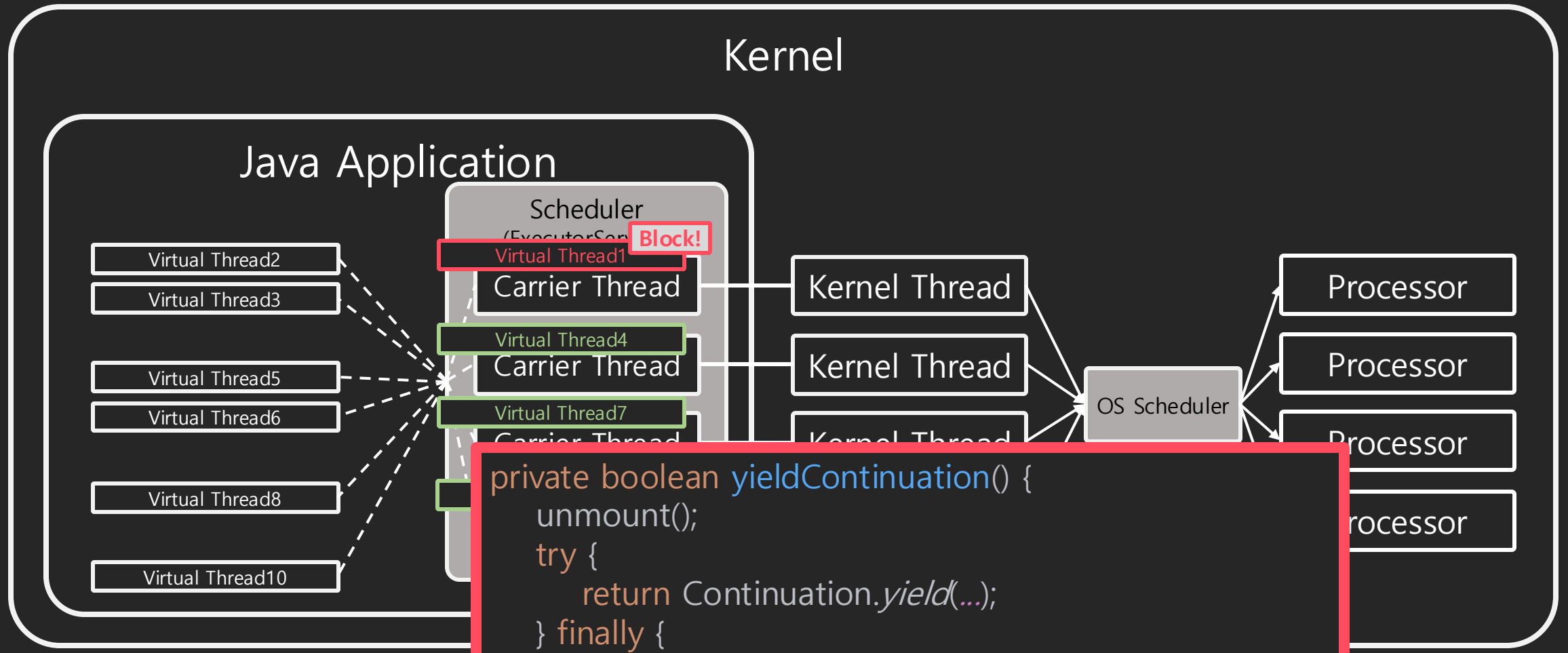


virtualThread.start() → virtualThread.mount()

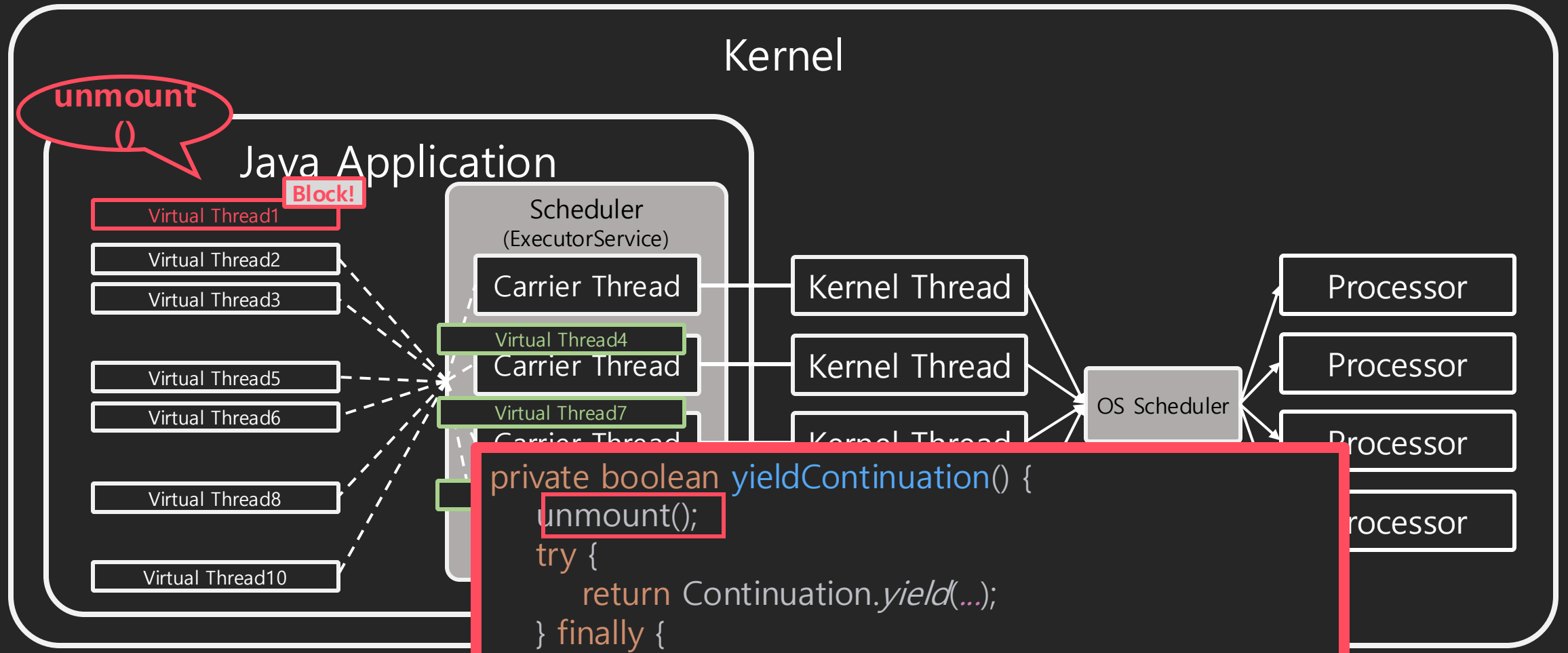
Virtual Thread - How it works – mount()



Virtual Thread - How it works – unmount() & yield()

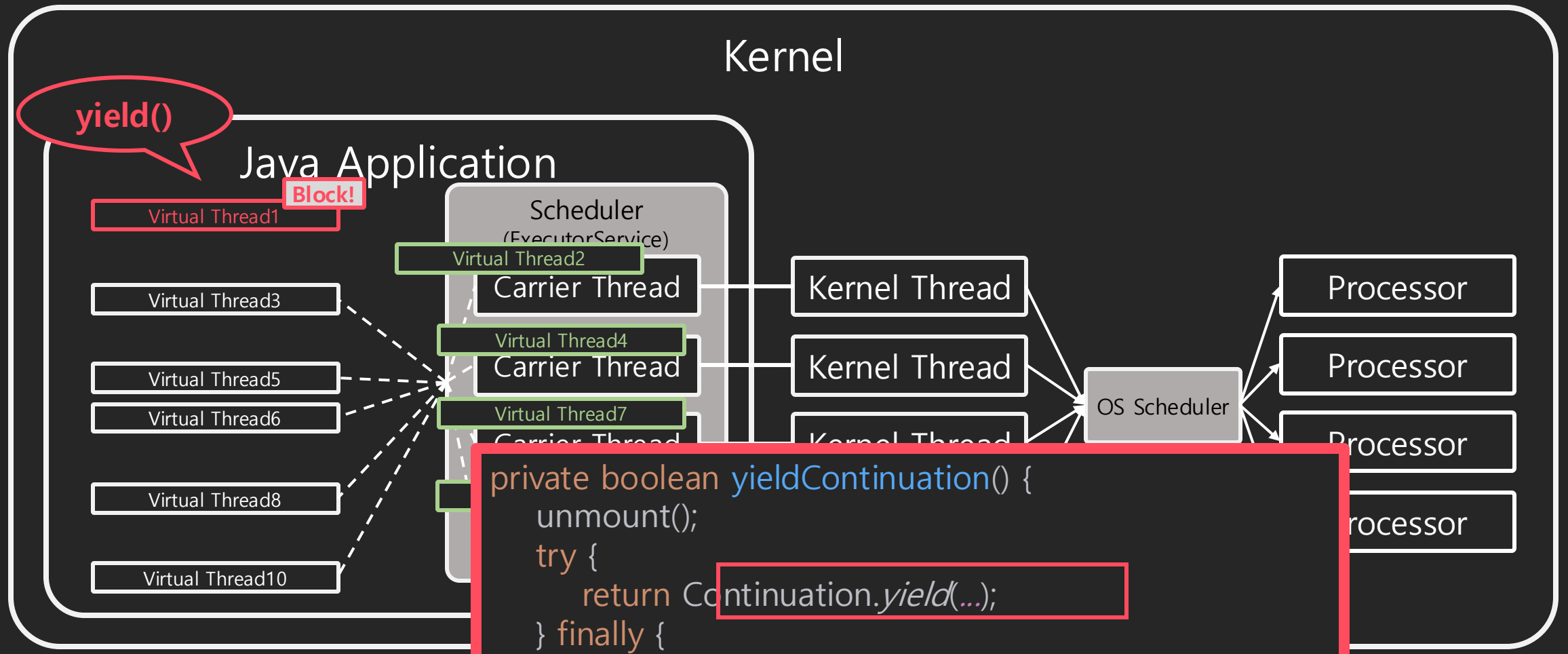


Virtual Thread - How it works – unmount() & yield()

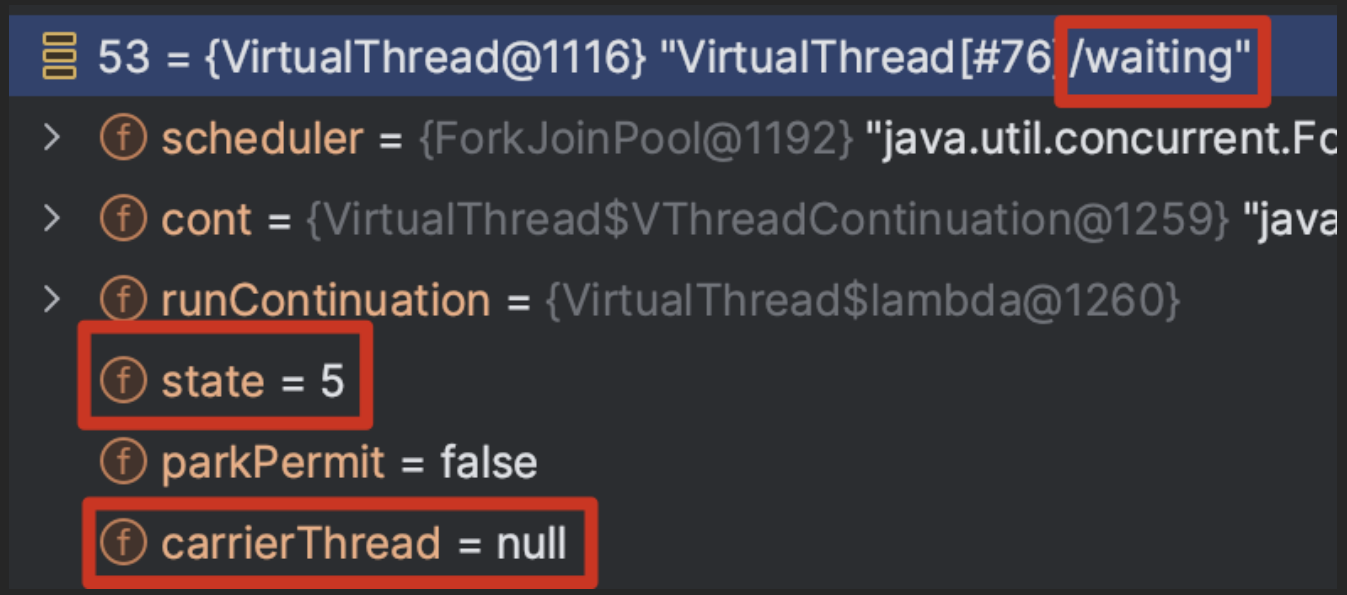



```
private boolean yieldContinuation() {
    unmount();
    try {
        return Continuation.yield(...);
    } finally {
        // re-mount
        mount();
    }
}
```

Virtual Thread - How it works – unmount() & yield()

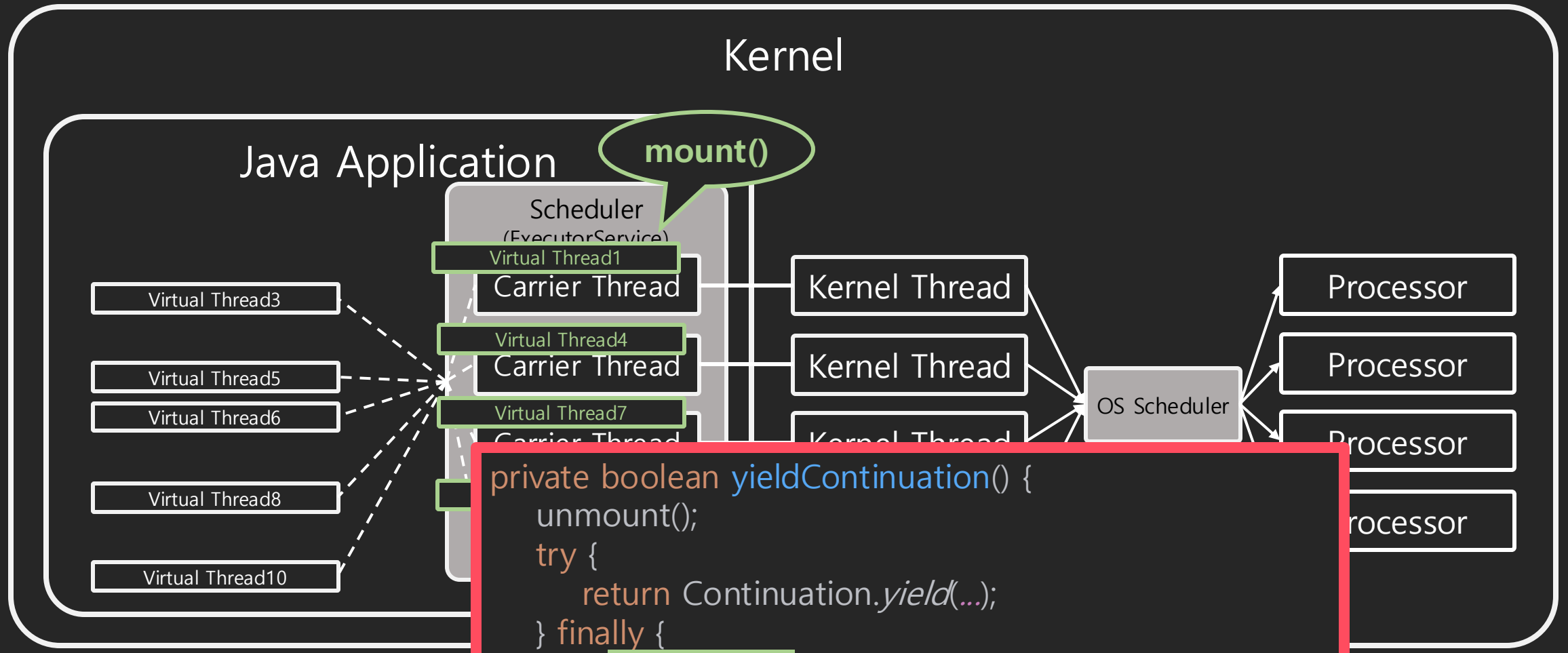


Virtual Thread - How it works – unmount() & yield()




```
NEW = 0;
STARTED = 1;
RUNNABLE = 2;      // runnable-unmounted
RUNNING = 3;       // runnable-mounted
PARKING = 4;
PARKED = 5;        // unmounted
PINNED = 6;        // mounted
YIELDING = 7;      // Thread.yield
TERMINATED = 99;   // final state
```

Virtual Thread - How it works – mount() again



Virtual Thread – Dive deep into Scheduler

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

  VirtualThread(Executor scheduler, String name, int characteristics, Runnable task) {
    super(name, characteristics, /*bound*/ false);
    Objects.requireNonNull(task);

    // choose scheduler if not specified
    if (scheduler == null) {
      Thread parent = Thread.currentThread();
      if (parent instanceof VirtualThread vparent) {
        scheduler = vparent.scheduler;
      } else {
        scheduler = DEFAULT_SCHEDULER;
      }
    }

    this.scheduler = scheduler;
    this.cont = new VThreadContinuation(this, task);
    this.runContinuation = this::runContinuation;
  }

  ...
}
```

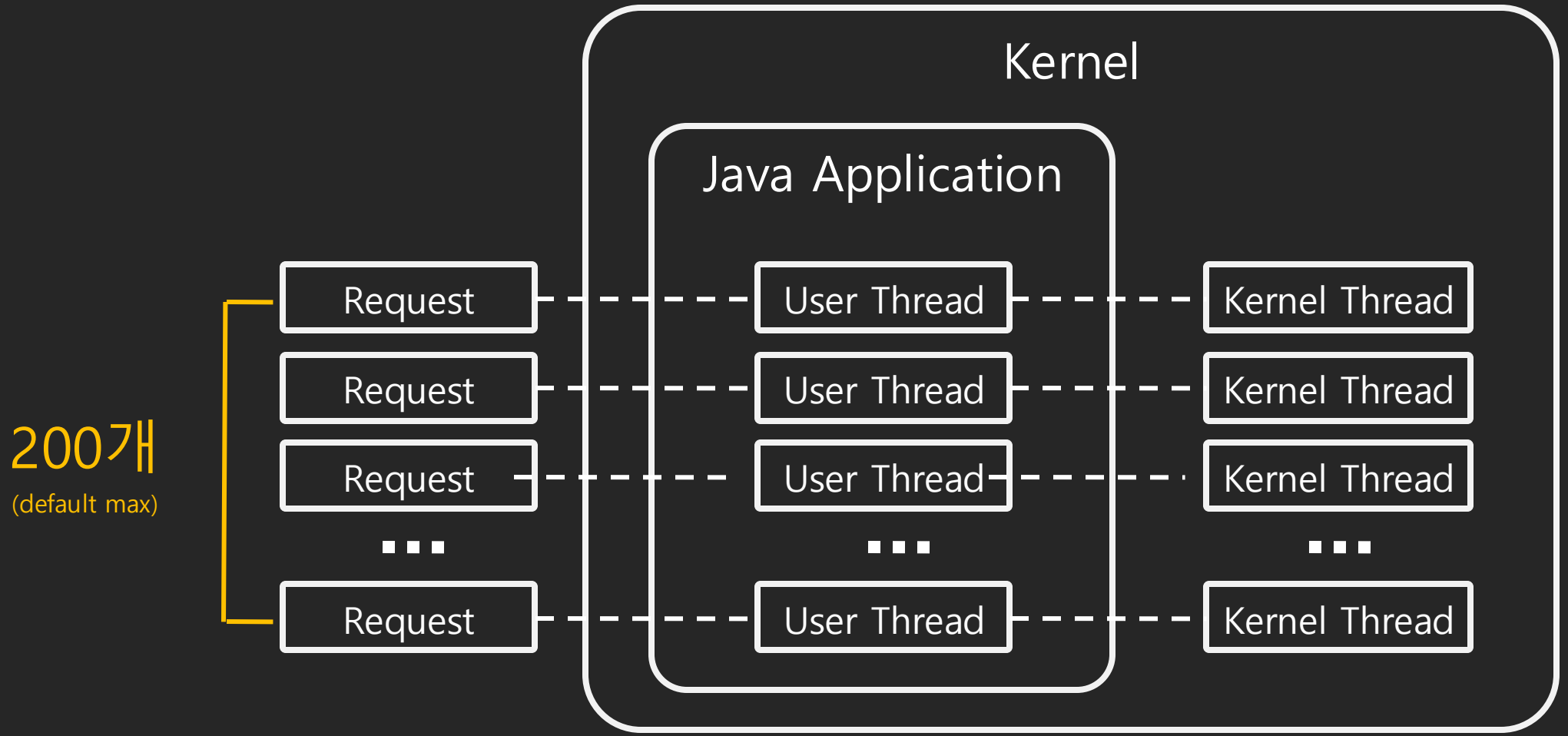
```
private static ForkJoinPool createDefaultScheduler() {
  ForkJoinWorkerThreadFactory factory = pool -> {
    PrivilegedAction<ForkJoinWorkerThread> pa = () -> new CarrierThread(pool);
    return AccessController.doPrivileged(pa);
  };
  PrivilegedAction<ForkJoinPool> pa = () -> {
    int parallelism, maxPoolSize, minRunnable;
    String parallelismValue = System.getProperty("jdk.virtualThreadScheduler.parallelism");
    String maxPoolSizeValue = System.getProperty("jdk.virtualThreadScheduler.maxPoolSize");
    String minRunnableValue = System.getProperty("jdk.virtualThreadScheduler.minRunnable");
        if (parallelismValue != null) {
        parallelism = Integer.parseInt(parallelismValue);
    } else {
        parallelism = Runtime.getRuntime().availableProcessors();
      }
    ...
    return new ForkJoinPool(parallelism, factory, handler, asyncMode,
    0, maxPoolSize, minRunnable, pool -> true, 30, SECONDS);
  };
  return AccessController.doPrivileged(pa);
}
```

Spring Web MVC

(Traditional; Platform-Thread Pool; non-reactive)

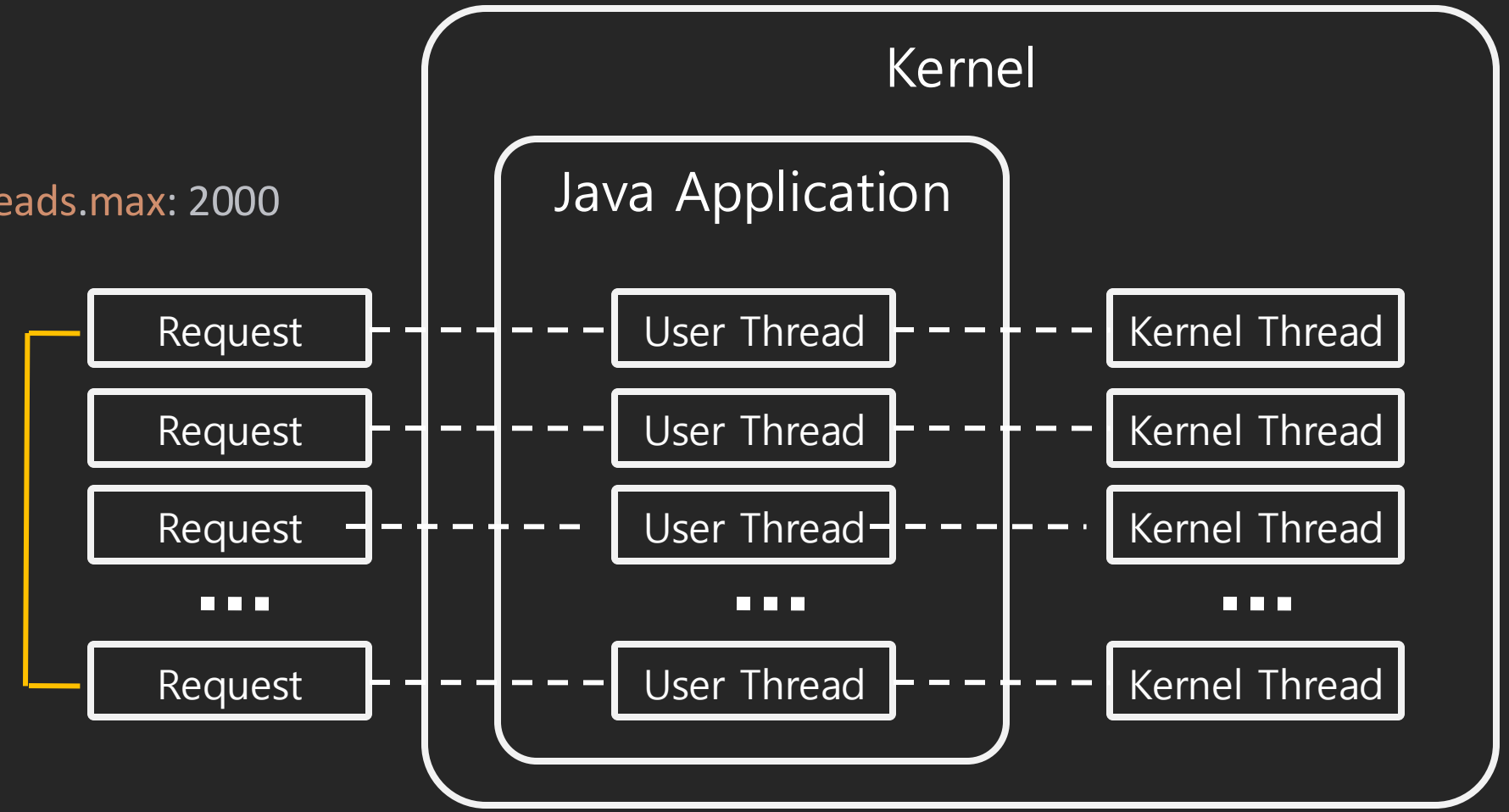
Spring Framework + Apache Tomcat

Spring Web MVC

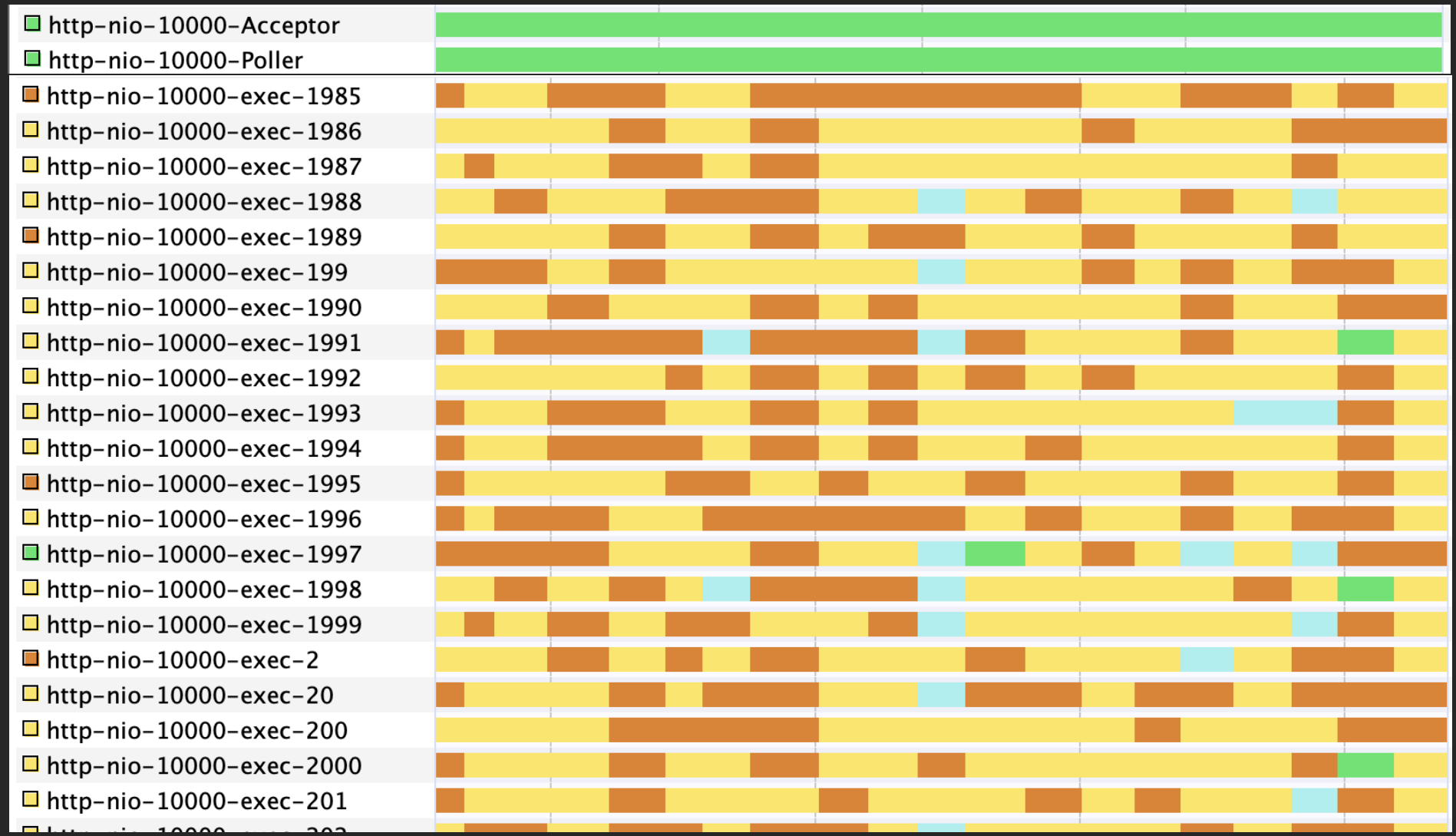


Spring Web MVC

server.tomcat.threads.max: 2000



Spring Web MVC (Platform Thread) - Threads



2GB Stack Memory

Spring MVC + Platform Thread – Load Test

```
@GetMapping("/hello")
fun hello() {
    Thread.sleep(100)
}
```

Tool: JMeter

Requests: 2,000 Thread * 100 Loop

```
summary + 112062 in 00:00:11 = 10598.9/s Avg:   167 Min:   100 Max:   438
summary + 287938 in 00:00:22 = 13067.9/s Avg:   142 Min:   100 Max:   341
summary = 400000 in 00:00:33 = 12266.9/s Avg:   149 Min:   100 Max:   438

summary + 209384 in 00:00:18 = 11578.4/s Avg:   159 Min:   100 Max:   538
summary + 190616 in 00:00:14 = 13907.5/s Avg:   127 Min:   100 Max:   306
summary = 400000 in 00:00:32 = 12582.6/s Avg:   144 Min:   100 Max:   538

summary + 131153 in 00:00:12 = 10557.3/s Avg:   167 Min:   100 Max:   684
summary + 268847 in 00:00:20 = 13162.6/s Avg:   141 Min:   100 Max:   534
summary = 400000 in 00:00:33 = 12176.6/s Avg:   150 Min:   100 Max:   684
```

Average: 12,342 requests / s

https://github.com/LOG-INFO/java-concurrency-practice

Spring Webflux

(Reactive Programming)
(feat. Reactor Netty)

+

Spring Webflux Netty

Spring Webflux - Threads

Model Name:	MacBook Air
Model Identifier:	MacBookAir10,1
Chip:	Apple M1
Total Number of Cores:	8 (4 performance and 4 efficiency)
Memory:	16 GB

9MB Stack Memory

Spring Webflux – Load Test

```
@GetMapping("/hello")
fun hello(): Mono<Long> {
    return Mono.delay(ofMillis(100))
}
```

Tool: JMeter

Requests: 2,000 Thread * 100 Loop

```
summary +      1 in 00:00:00 =    4.1/s Avg:   153 Min:   153 Max:   153 E
summary + 399999 in 00:00:23 = 17208.0/s Avg:   106 Min:     1 Max:   433
summary = 400000 in 00:00:23 = 17030.7/s Avg:   106 Min:     1 Max:   433

summary + 116969 in 00:00:07 = 15677.4/s Avg:   108 Min:   100 Max:   277
summary + 283031 in 00:00:16 = 18012.5/s Avg:   102 Min:   100 Max:   143
summary = 400000 in 00:00:23 = 17260.0/s Avg:   104 Min:   100 Max:   277

summary + 233261 in 00:00:13 = 17313.2/s Avg:   105 Min:     1 Max:   473 |
summary + 166739 in 00:00:10 = 17507.2/s Avg:   102 Min:   100 Max:   145 |
summary = 400000 in 00:00:23 = 17392.8/s Avg:   104 Min:     1 Max:   473 |
```

Average: 17,228 requests / s (42% Faster)

https://github.com/LOG-INFO/java-concurrency-practice

Spring Webflux

(Reactive Programming)

(feat. Reactor Netty + Kotlin Coroutines)

Spring Webflux Netty Kotlin Coroutines

Spring Webflux + Kotlin Coroutines – Load Test

```kotlin
@GetMapping("/hello")
suspend fun hello(): Long {
    return Mono.delay(ofMillis(100))
        .awaitSingle()
}
```

Tool: JMeter

Requests: 2,000 Thread * 100 Loop

```
summary + 224925 in 00:00:13 = 16855.9/s Avg:   108 Min:     1 Max:   558
summary + 175075 in 00:00:10 = 17832.0/s Avg:   102 Min:   100 Max:   139
summary = 400000 in 00:00:23 = 17268.9/s Avg:   106 Min:     1 Max:   558

summary + 168288 in 00:00:10 = 16653.9/s Avg:   106 Min:   100 Max:   296
summary + 231712 in 00:00:13 = 17773.4/s Avg:   103 Min:   100 Max:   151
summary = 400000 in 00:00:23 = 17284.6/s Avg:   104 Min:   100 Max:   296

summary + 306164 in 00:00:17 = 17712.7/s Avg:   105 Min:   100 Max:   205
summary +  93836 in 00:00:06 = 15600.3/s Avg:   103 Min:   100 Max:   131
summary = 400000 in 00:00:23 = 17166.6/s Avg:   104 Min:   100 Max:   205
```

Average: 17,240 requests / s (42% Faster)

https://github.com/LOG-INFO/java-concurrency-practice

Spring Web MVC

(Non-Pooling Virtual-Thread; non-reactive)

Spring Framework + Apache Tomcat + Virtual Thread

Spring Web MVC + Virtual Thread

Before Spring Boot 3.2

```
@Bean
public TomcatProtocolHandlerCustomizer<?> protocolHandlerVirtualThreadExecutorCustomizer() {
    return protocolHandler -> {
        protocolHandler.setExecutor(Executors.newVirtualThreadPerTaskExecutor());
    };
}
```

Spring Boot 3.2+

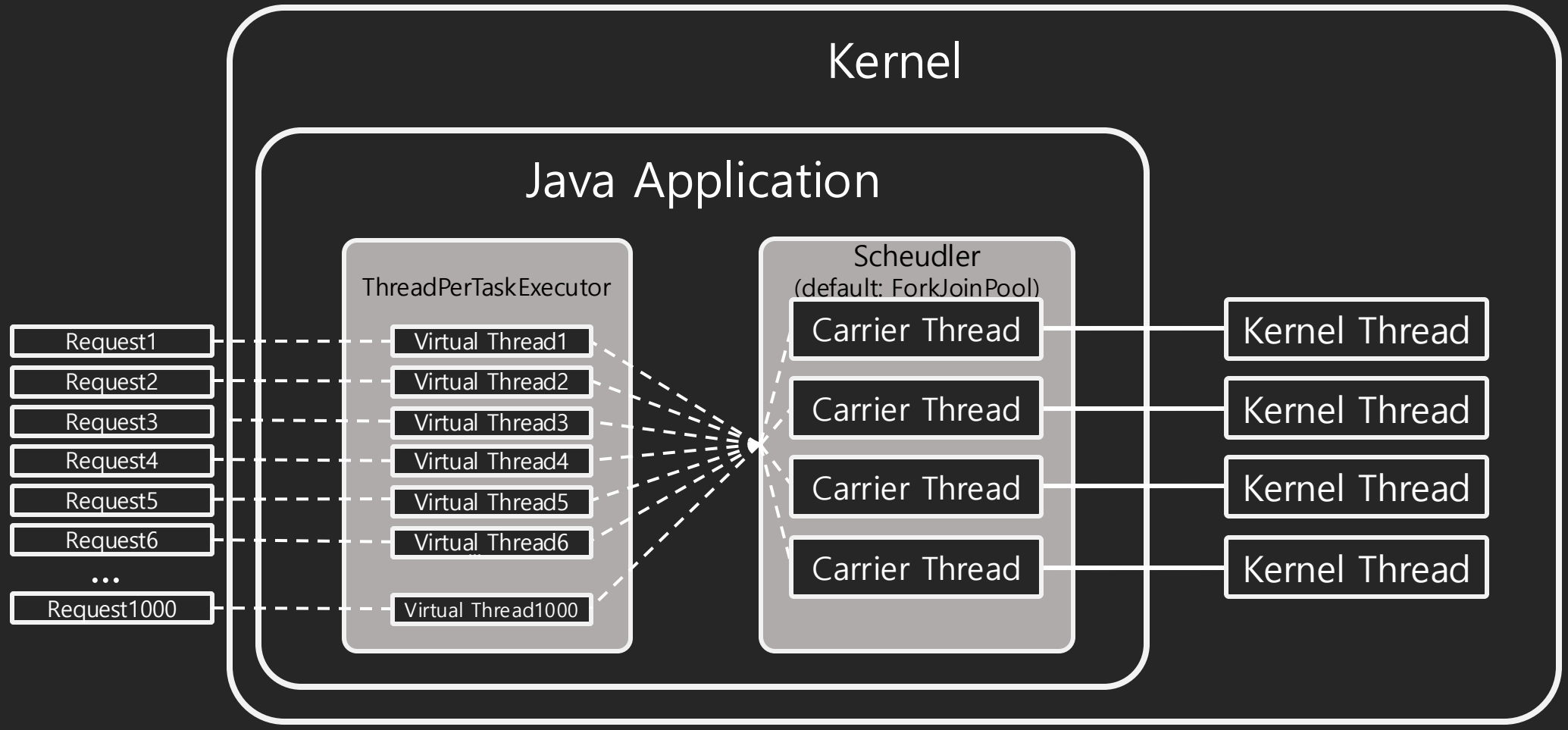
```
spring:
  threads:
    virtual:
      enabled: true
```

Spring Web MVC + Virtual Thread

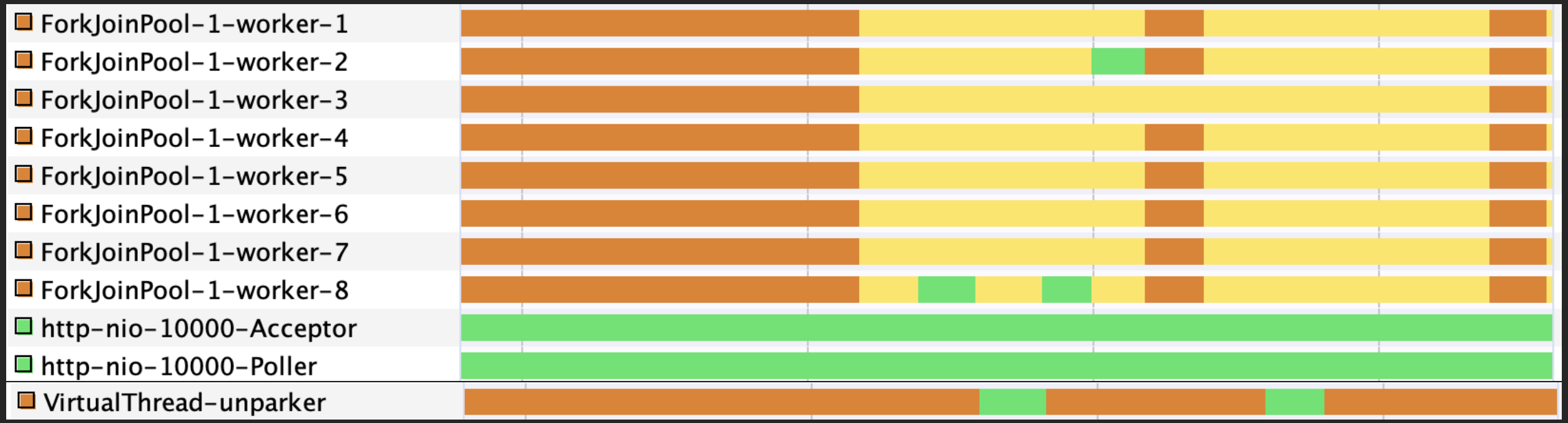
```
@Bean(TaskExecutionAutoConfiguration.APPLICATION_TASK_EXECUTOR_BEAN_NAME)
public AsyncTaskExecutor asyncTaskExecutor() {
    return new TaskExecutorAdapter(Executors.newVirtualThreadPerTaskExecutor());
}
```

```
@Async
public void asyncMethod() {
    // do something asynchronously
}
```

Spring Web MVC + Virtual Thread



Spring MVC (Virtual Thread) - Threads



11MB Stack Memory

Spring MVC + Virtual Thread – Load Test

```
@GetMapping("/hello")
fun hello() {
   Thread.sleep(100)
}
```

Tool: JMeter

Requests: 2,000 Thread * 100 Loop

```
summary + 337793 in 00:00:20 = 16691.9/s Avg:   112 Min:   100 Max:   304
summary +  62207 in 00:00:04 = 14520.8/s Avg:   105 Min:   100 Max:   147
summary = 400000 in 00:00:25 = 16311.2/s Avg:   111 Min:   100 Max:   304

summary +      1 in 00:00:00 =    3.9/s Avg:   156 Min:   156 Max:   156 E
summary + 399999 in 00:00:24 = 16505.7/s Avg:   110 Min:   100 Max:   259
summary = 400000 in 00:00:24 = 16332.5/s Avg:   110 Min:   100 Max:   259

summary + 361509 in 00:00:22 = 16801.1/s Avg:   112 Min:   100 Max:   268
summary +  38491 in 00:00:03 = 12372.5/s Avg:   107 Min:   100 Max:   161
summary = 400000 in 00:00:25 = 16241.0/s Avg:   111 Min:   100 Max:   268
```

Average: 16,294 requests / s (33% Faster)

https://github.com/LOG-INFO/java-concurrency-practice

Conclusion

- Increased memory efficiency by creating only few Carrier Threads
- Low-cost context-switching as context switching occurs in JVM rather than Kernel
- Taking both of memory efficiency & simplicity of development
- It may still be too early
  - Not yet sufficiently reliable (bugs, etc.)
  - Not yet completely fix Carrier Thread Pinning issue

Conclusion

Nevertheless, it seems clear that it is a Game-Changer for Concurrency in JVM ecosystem

Learn more…

- Continuation
- Project Loom
  - Structured Concurrency
  - Scoped Value
- Goroutines in Go & Processes in Earlang
- Dive deep into raw VirtualThread code

Referrences

- https://www.javatpoint.com/why-must-user-threads-be-mapped-to-kernel-thread
- https://www.baeldung.com/kotlin/spring-boot-kotlin-coroutines
- https://www.ibm.com/docs/en/ztpf/2023?topic=threads-understanding-java-native-thread-details
- https://www.baeldung.com/java-threading-models
- https://www.baeldung.com/java-virtual-thread-vs-thread
- https://web.archive.org/web/20130116024929/http://www.codestyle.org/java/faq-Threads.shtml#greenthread
- https://www.sco.com/developers/java/j2sdk122-001/ReleaseNotes.html#THREADS
- https://en.wikipedia.org/wiki/Green_thread
- https://www.geeksforgeeks.org/green-vs-native-threads-and-deprecated-methods-in-java/
- https://perfectacle.github.io/2019/03/10/green-thread-vs-native-thread/
- https://www.infoq.com/articles/java-virtual-threads

Reactor and Kotlin Coroutines become unnecessary?

They have different interests

Virtual Thread only deals with Low-Level Concurrency

In order to handle concurrency at a high-level,
libraries that provide rich features for concurrency(e.g. Kotlin Coroutines)
will continue to be necessary

e.g.) Asynchronously request multiple APIs and join within a single flow

Will Platform Thread -> Virtual Thread Switching improve performance?

성능이 좋아진다고 하기보단 아래 두 가지를 개선하는 것이 더 맞을 것 가틈

- Thread 생성에 소요되는 메모리 효율
  - Platform Thread: 1MB (Stack memory)
  - Virtual Thread: 수백 Byte ~ (Heap memory)

- Thread Context Switching 효율

Platform Thread -> Virtual Thread 전환하면 성능이 개선될까요?

프로그램의 '성능'을 이야기하기 위해선 굉장히 많은 것들을 고려해야 함

프로그램의 자원의 종류가 여러가지인 만큼, 성능의 병목지점은 여러가지가 있음

(CPU, Memory, Thread 개수, Network, Disk Usage, ...)

Platform Thread -> Virtual Thread 전환하면 성능이 개선될까요?

성능이 향상되는 케이스

- Thread에 의한 Memory Usage가 병목인 경우
- Thread Pooling에 의해 제한된 Thread 개수와 Blocking Call이 병목인 경우
- (드문 경우) Thread의 Context Switching이 너무 빈번한 경우

성능에 별 영향이 없는 케이스

- CPU 사용량이 병목인 경우
  - e.g.) Blocking보다 CPU 연산이 많아 CPU가 부족한 상황
- Network가 병목인 경우

Virtual Thread 한계점은 없나요?

Carrier Thread에 PINNED되는 두 가지 시나리오. (Carrier Thread도 같이 Block됨)

- synchronized block에서 Blocking 발생하는 경우
- native method 또는 foreign function에서 Blocking 발생하는 경우

High-Level에서 비동기로직을 나이스하게 처리하긴 힘듦

결국 기존의 Reactor, Kotlin Coroutines, Completable Future 등과 공존할 것임

Release된지 얼마 되지 않음(Java 21은 2023년 09월에 GA Release)

아무래도 안정성에 대한 고려에 의해 몇년간 Critical한 시스템에서 사용하기는 힘들

Virtual Thread 한계점은 없나요?

Spring Boot 3.x에선 synchronized를 많이 없애서 Virtual Thread와 호환이 좋음

ref) Embracing Virtual Threads

현재 기준 MySQL JDBC Driver 내에선 synchronized block이 굉장히 많기 때문에 Virtual Thread와 호환성이 좋다고 보기는 어려움

PostgreSQL JDBC Driver는 synchronized block이 적어서 VT와 호환성이 좋음

Virtual Thread 한계점은 없나요?

Virtual Thread는 Thread Dump에 조회되지 않음