Modern Java
Multithreading in Java
using
Virtual Threads

Dilip Sundarraj

About Me

- Dilip
- Building Software’s since 2008
- Teaching in **UDEMY** Since 2016

What’s Covered in this Course?

- Introduction to Platform Threads and Virtual Threads.
  - Limitations of Platform Threads.
- Explore Virtual Threads and how it works under the hood?
- Look into the older concurrency APIs and understand the limitations.
- Deep dive in to Structured Concurrency API.
  - Implement a realtime use-case using Structured Concurrency API.
- Build HTTP Client and use them with Virtual Threads.
- Using Future API alongside Virtual Threads.
- Use VirtualThreads in a SpringBoot Application.
  - Profile and Understand the benefits of using VirtualThreads in a SpringBoot app.

Targeted Audience

- Any developer who is curious to understand the latest concurrency advancements in Java.
- Any developer who is interested in learning about Threads (PlatformThreads) and Virtual Threads.
- Using Springboot along with VirtualThreads.

Source Code

Thank You!

Prerequisites

- Java 21
- Prior Java Experience is a must
  - Functional programming concepts such as Lambdas, Streams API.
- Experience working with JUnit5.
- **Intellij** or any other IDE
- Prior Spring Framework/SpringBoot is a nice to have.

What is a Java Thread, Why do we need them ?

- Threads in Java existed since Java 1.0

- Any program in Java is executed by a thread.

```
public class Main {
    public static void main(String[] args) {
      System.out.println("["+Thread.currentThread().getName()+"] "+"Hello and welcome!");
    }
}
```

```
[main] Hello and welcome!
```

What is a Java Thread, Why do we need them ?

- Starting Java 21, we have two types of Threads:
  - Platform Threads (aka Java Threads until 20)
  - Virtual Threads
- Platform Threads in Java are used to run tasks in the background.
  - This allows a program to execute multiple things at the same time without interrupting the main thread.
- How do we execute multiple tasks in the background ?

What is a Java Thread, Why do we need them ?

```
public class ExploreThreads {
    public  void doSomeWork(){
        try {
            sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Error Occurred");
        }
        log("doSomeWork");
    }

    public static void main(String[] args) {

        var thread = Thread.ofPlatform().name("t1");
        thread.start(() -> log("Run task1 in the background"));

        var thread1 = Thread.ofPlatform().name("t2");
        thread1.start(() -> new ExploreThreads().doSomeWork());

        System.out.println("Program Completed!");

    }
}
```

1

2

Starting Java 21, Java Threads are called as Platform Threads.

A Task is a piece of code.

Task 1

Task 2

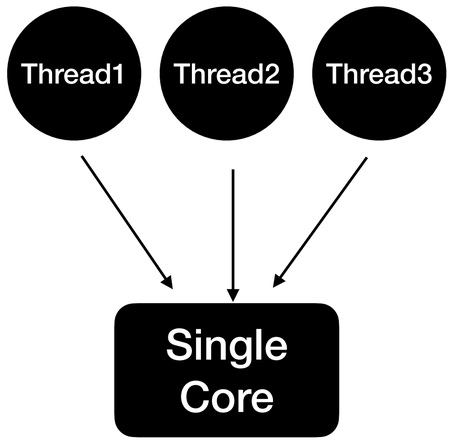
Benefits of Java Threads

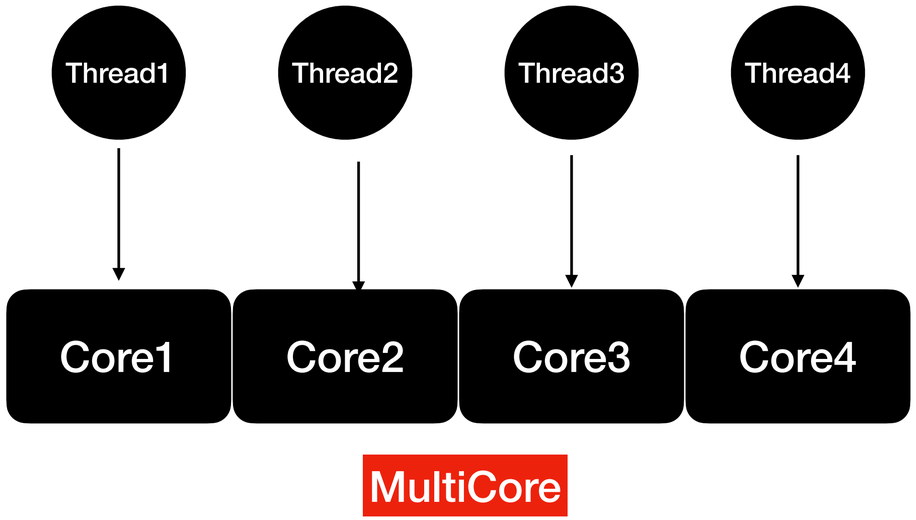
- Modern computers today, typically have multiple cores.
  - Even mobile phones have multiple cores.
- Threads basically help us to spin up multiple tasks in the background(Multithreading) which can access all these cores and eventually execute the code faster.

Concurrency
vs
Parallelism

Concurrency

- Concurrency is a concept where two or more independent tasks can run simultaneously
- In Java, Concurrency is achieved using **Threads**
  - Are the tasks running in interleaved fashion?
  - Are the tasks running simultaneously ?





Concurrency Example

- In a real application, Threads normally need to interact with one another
  - Shared Objects or Messaging Queues
- Issues:
  - Race Condition
  - DeadLock and more
- Tools to handle these issues:
  - Synchronized Statements/Methods
  - Reentrant Locks, Semaphores
  - Concurrent Collections
  - Conditional Objects and More

```
public class HelloWorldThreadExample {
    private static String result="";

    private static void hello(){
        delay(500);
        result = result.concat("Hello");
    }
    private static void world(){
        delay(600);
        result = result.concat(" World");
    }

    public static void main(String[] args) throws InterruptedException {

        Thread helloThread = new Thread(()-> hello());
        Thread worldThread = new Thread(()-> world());

        //Starting the thread
        helloThread.start();
        worldThread.start();

        //Joining the thread (Waiting for the threads to finish)
        helloThread.join();
        worldThread.join();

        System.out.println("Result is : " + result);

    }
}
```

1

2

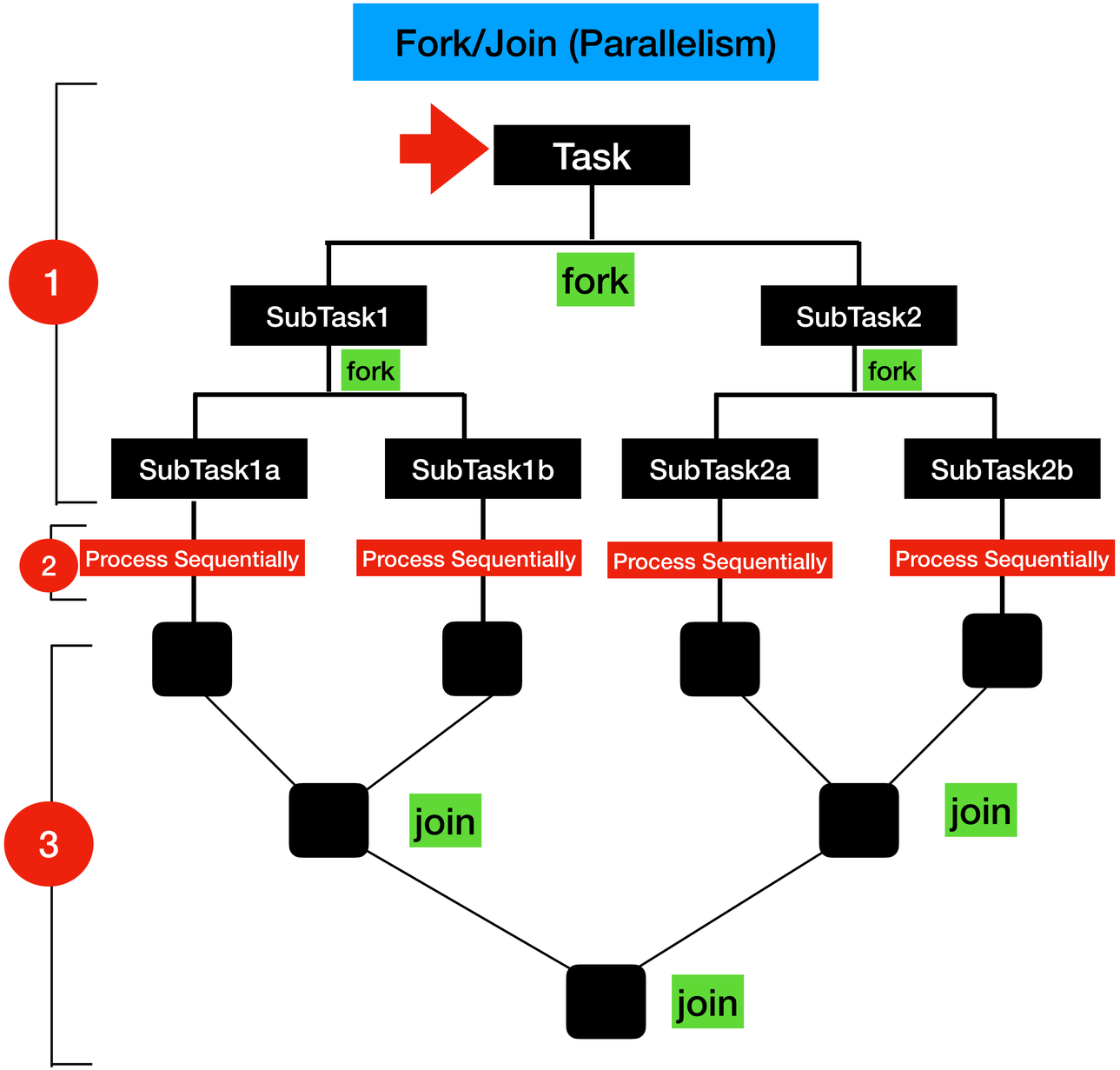
3

Threads

Hello World

Parallelism

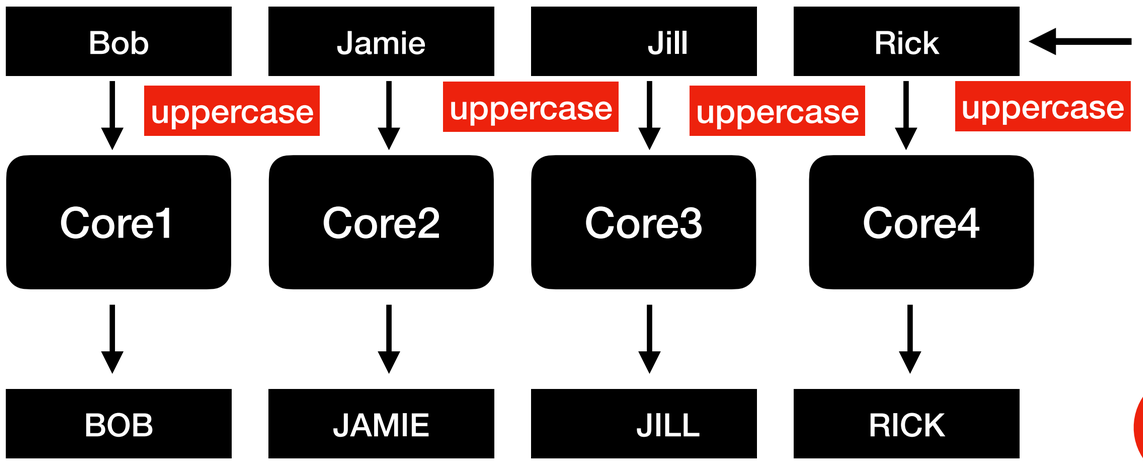
- Parallelism is a concept in which two or more related tasks are literally going to run in parallel.
- Parallelism involves these steps:
  - Decomposing the tasks in to SubTasks(Forking)
  - Execute the subtasks in sequential
  - Joining the results of the tasks(Join)
- Whole process is also called **Fork/ Join**

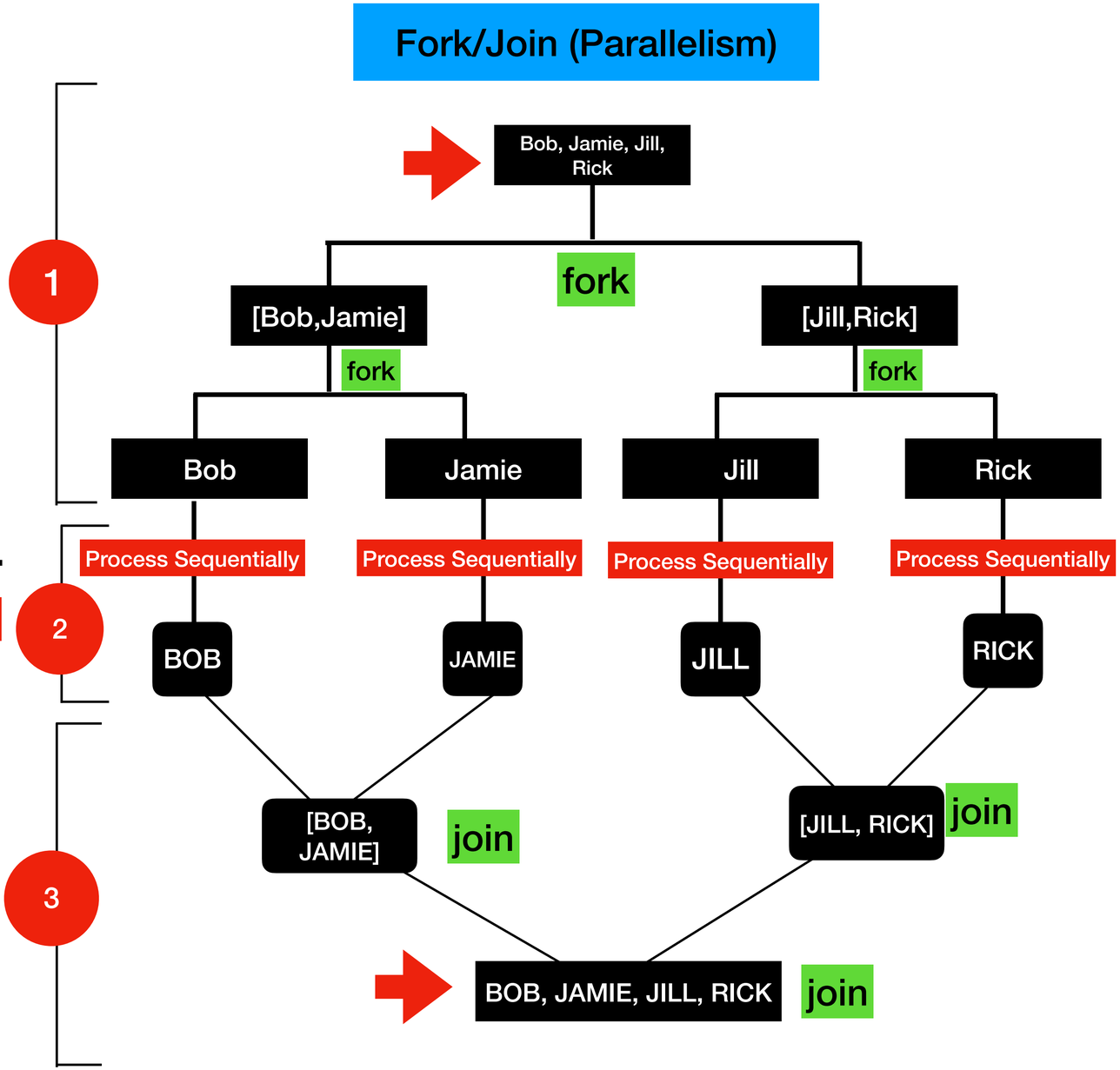


Parallelism Example

UseCase: Transform to UpperCase

[Bob, Jamie, Jill, Rick] -> [BOB, JAMIE, JILL, RICK]





Parallelism Example

```
public class ParallelismExample {

    public static void main(String[] args) {

        List<String> namesList = List.of("Bob", "Jamie", "Jill", "Rick");
        System.out.println("namesList : " + namesList);
        List<String> namesListUpperCase = namesList
                .parallelStream()
                .map(String::toUpperCase)
                .collect(Collectors.toList());

        System.out.println("namesListUpperCase : " + namesListUpperCase);

    }
}
```

Concurrency vs Parallelism

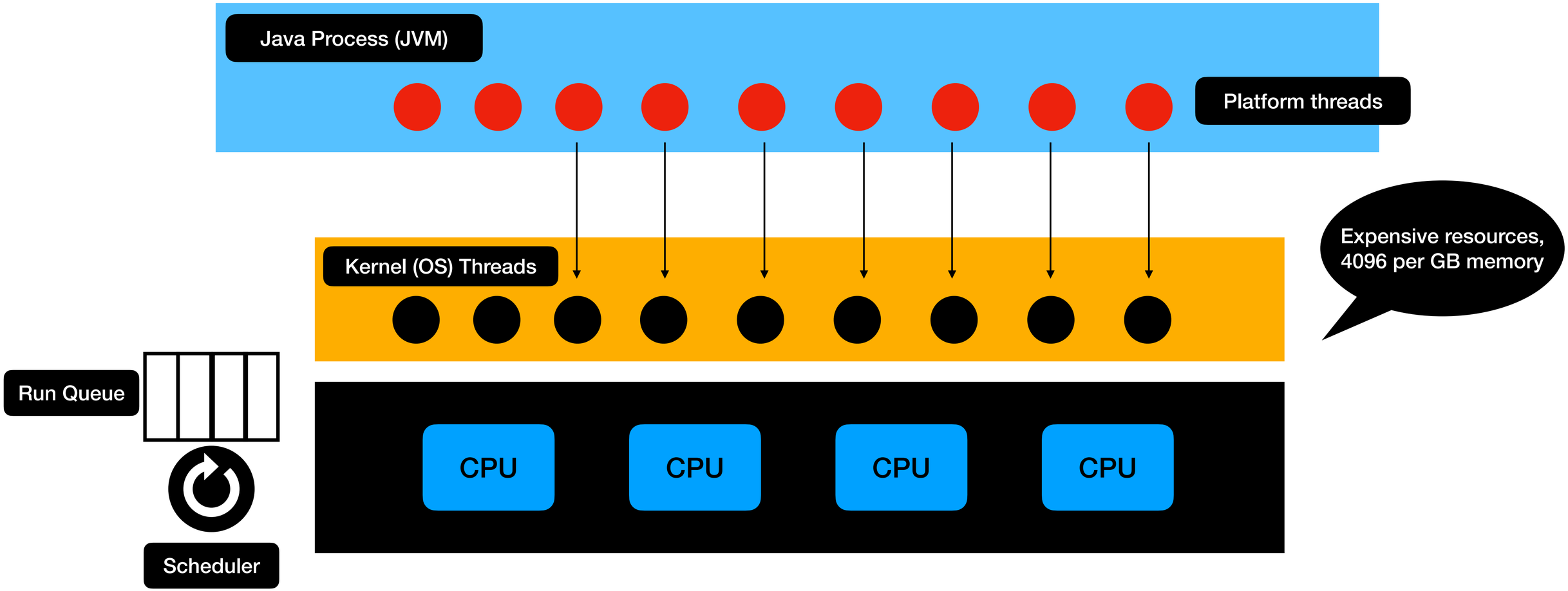
- Concurrency is a concept where two or more tasks independent can run in simultaneously.
- Concurrency can be implemented in single or multiple cores.
- Concurrency is about correctly and efficiently controlling access to shared resources .

- Parallelism is a concept where two or more related tasks are literally running in parallel.
- Parallelism can only be implemented in a multi-core machine.
- Parallelism is about using more resources to access the result faster.

Thread Internals - How it works behind the scenes?

```
var thread1 = Thread.ofPlatform().name("t2");

thread1.start(() -> new ExploreThreads().doSomeWork());
```

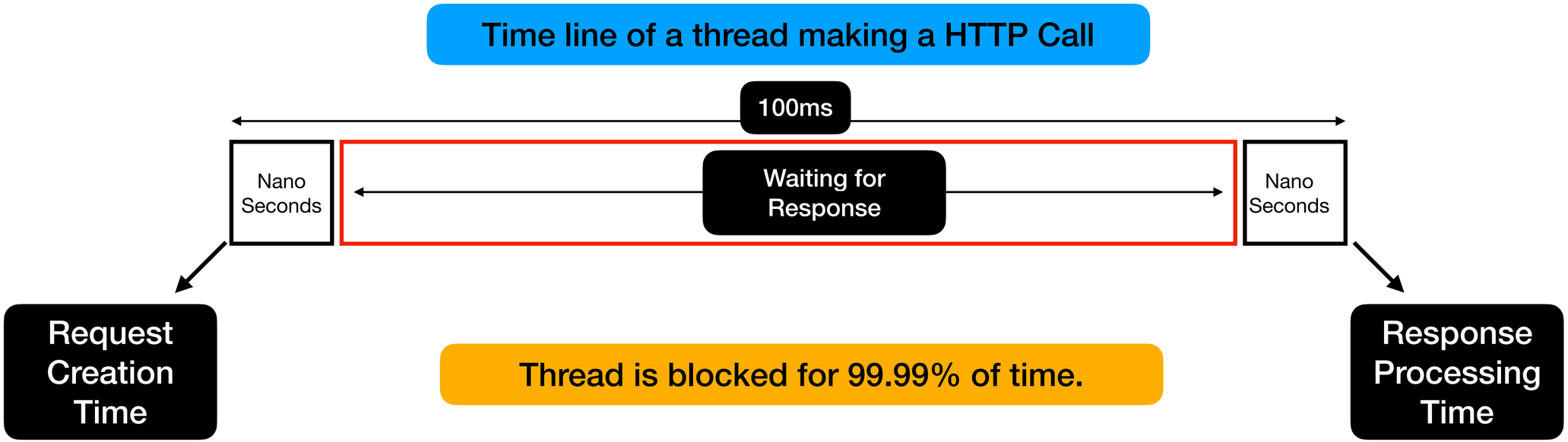


Thread Scalability and Drawbacks

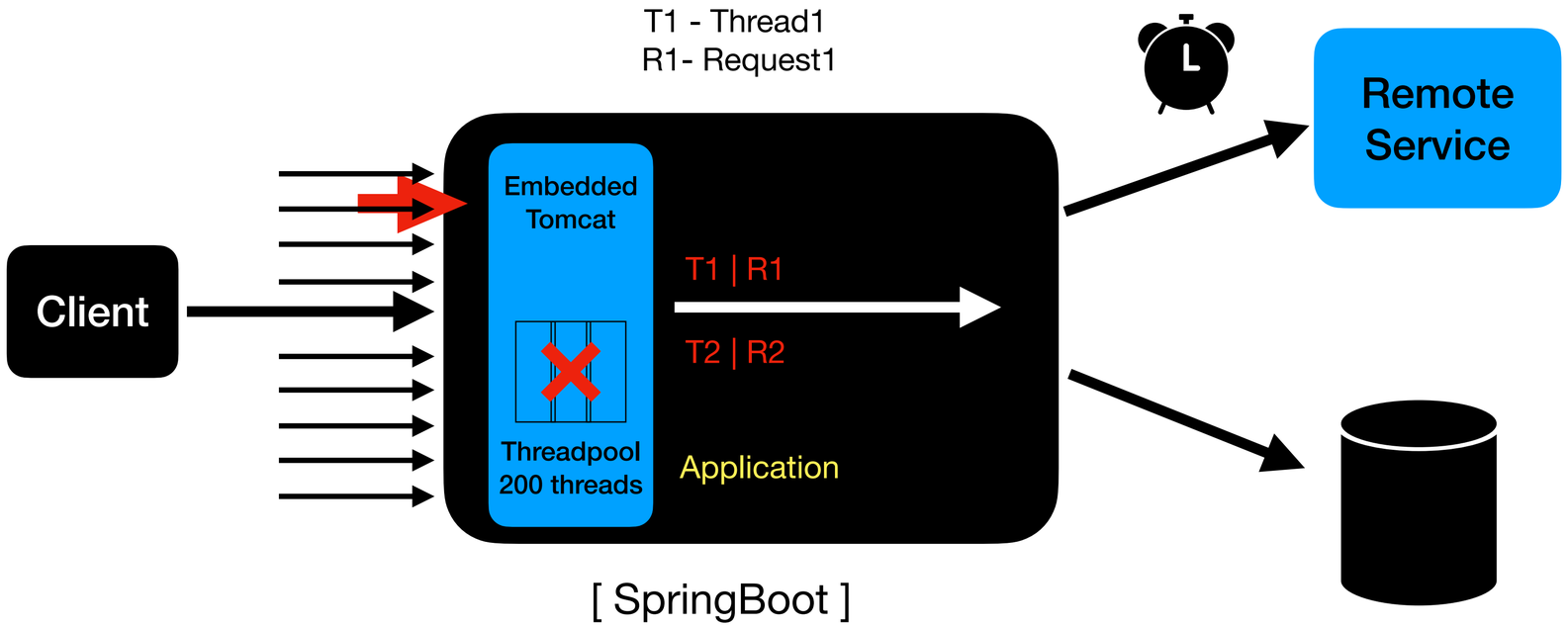
- Thread is an expensive resource:
  - Threads can take up to ~1ms to start up
  - It can take up to 1MB to 2MB memory for stack
  - Thread Context Switching also eats up some time(100μS).
  - Threads live in the heap memory
- What are the drawbacks?
  - We can only create so many threads.
  - If we need to support million transactions, then we cannot million threads to handle it.

Blocking Nature of Java Threads

- Threads are blocked and tied with the task until it completes.
- Lets say we make a HTTP IO call.



Typical Backend Application Architecture



- Threads in this architecture is not efficiently used, because it spends more waiting than efficiently handling client requests.
- We cannot create a new thread per request , if we do then we will run into the OutOfMemoryError.

Solutions for Handling High Load (Throughput) ?

- Reactive Programming is an option which is very popular in the recent years.
  - Requires a big learning curve.
  - Code is completely written in Functional Programming Style.

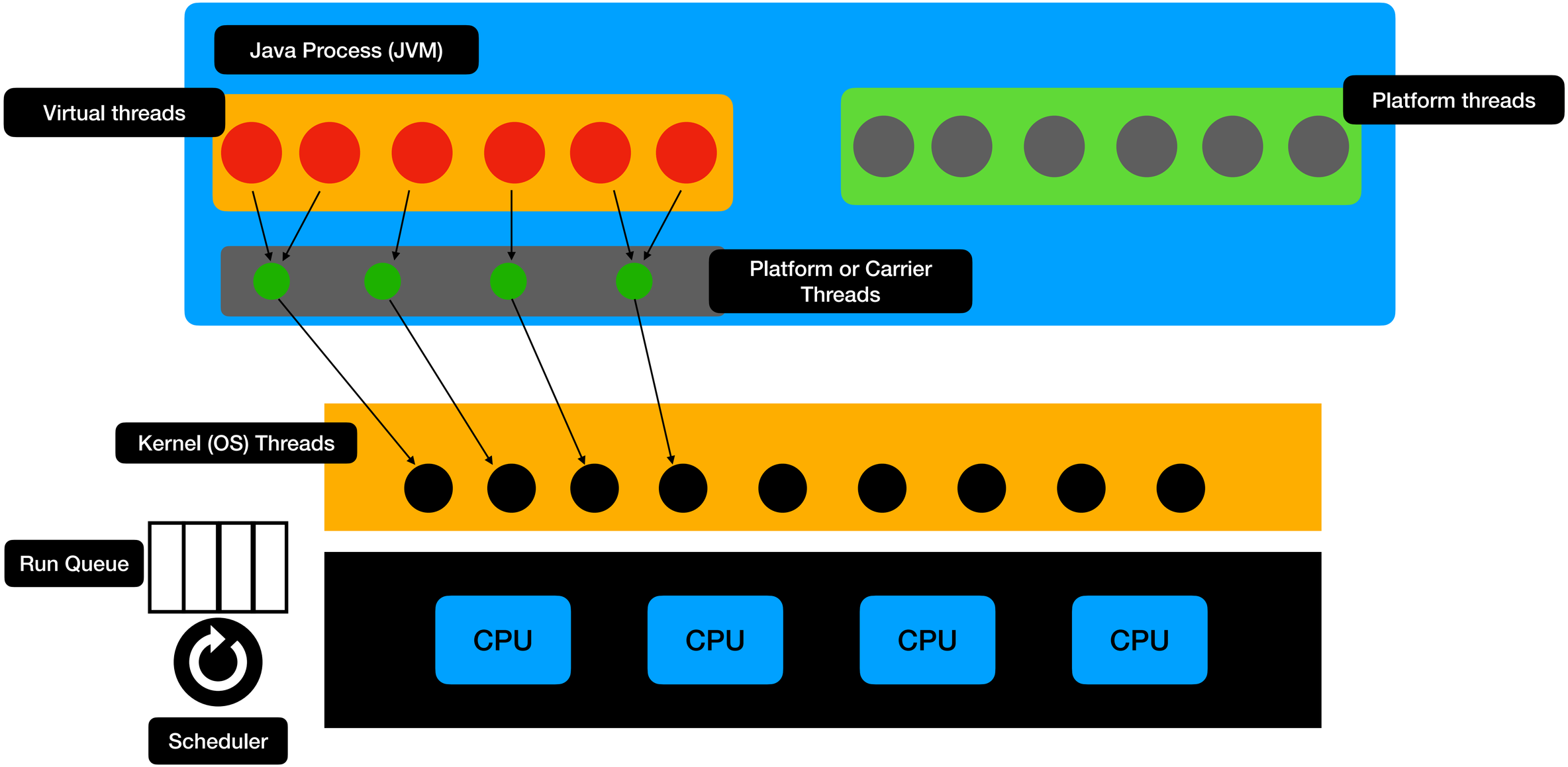
Solutions for Handling High Load (Throughput) ?

- Virtual Threads
  - Using virtual threads we can create a thread per request.
  - Virtual Threads are light weight.

Virtual Threads

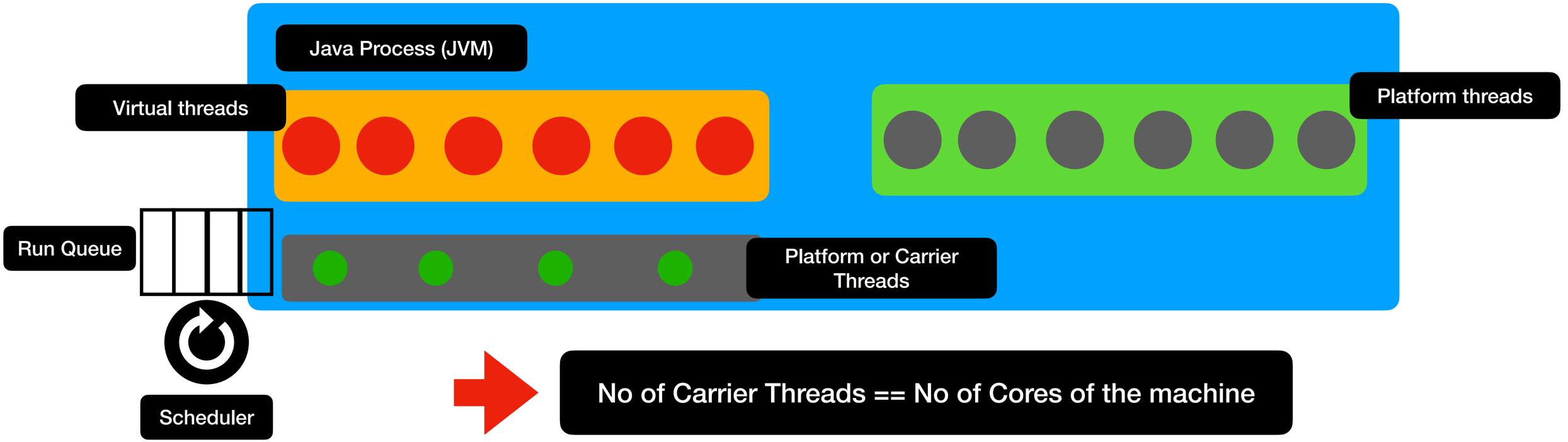
- Virtual Threads are officially part of Java 21.
- Virtual threads are called lightweight threads.
  - They are very cheap to create and destrory.
  - They don’t take up a lot of memory.
  - They have a shallow call stack.

Virtual Thread in the JVM



How Virtual Threads works behind the scenes?

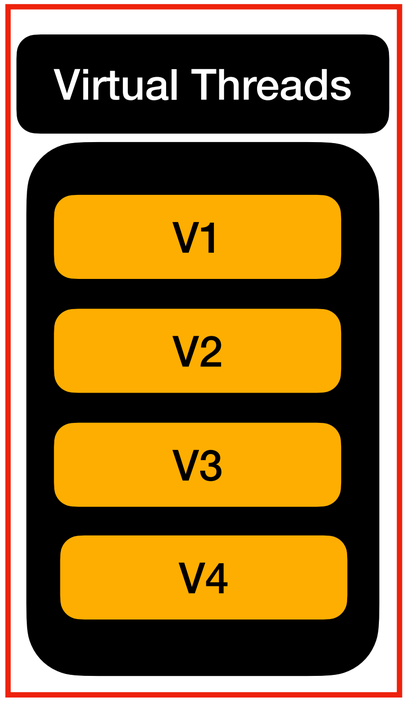
- Virtual Threads are managed by the JVM not the OS.
- JVM has its own **scheduler** to scheudle the virtual threads on the carrier threads.



Virtual Threads Scheduler

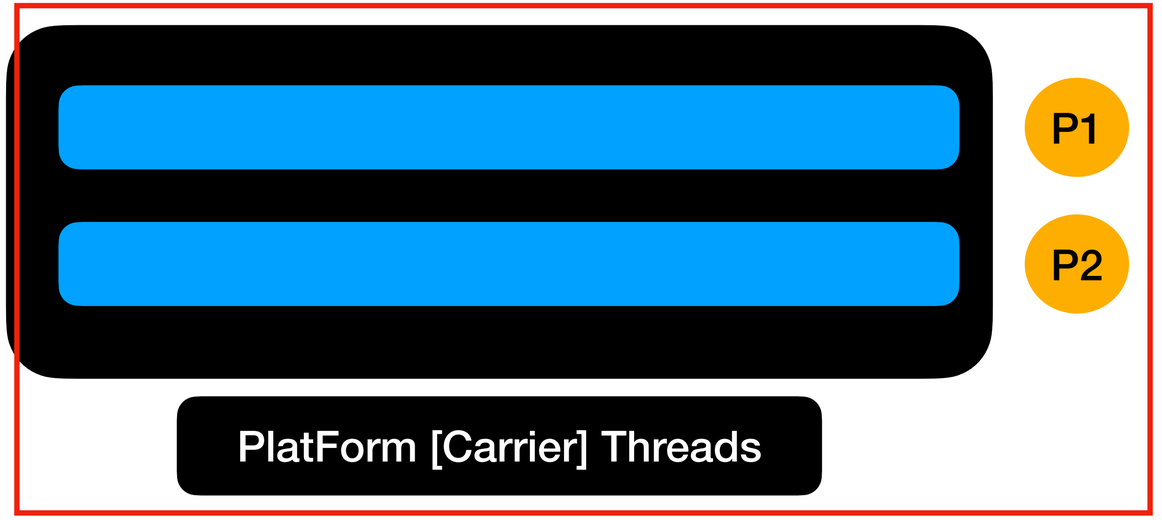
- Virtual Threads Scheduler uses the ForkJoinPool which has FIFO task queue.
  - So anytime we create a task it gets submitted to this ForkjoinPool.
- Scheduler maps these tasks to a Carrier(Platform) Thread.
  - Virtual Thread is mapped to a Carrier(Platform) Thread.

Mounting and Unmounting Virtual Threads




```
public static void doSomeWork(int index) {
    log("started doSomeWork : " + index);
    //In this case, we are just blocking the thread by calling sleep.
    //It could be any IO call such as HTTP or File IO call.
    Thread.sleep(5000); // blocking task
    log("finished doSomeWork : " + index);
}
```

```
IntStream.rangeClosed(1, 4)
        .forEach((i) -> {
            var threads = Thread.ofVirtual().start(() -> MaxVirtualThreads.doSomeWork(i));
        });
```

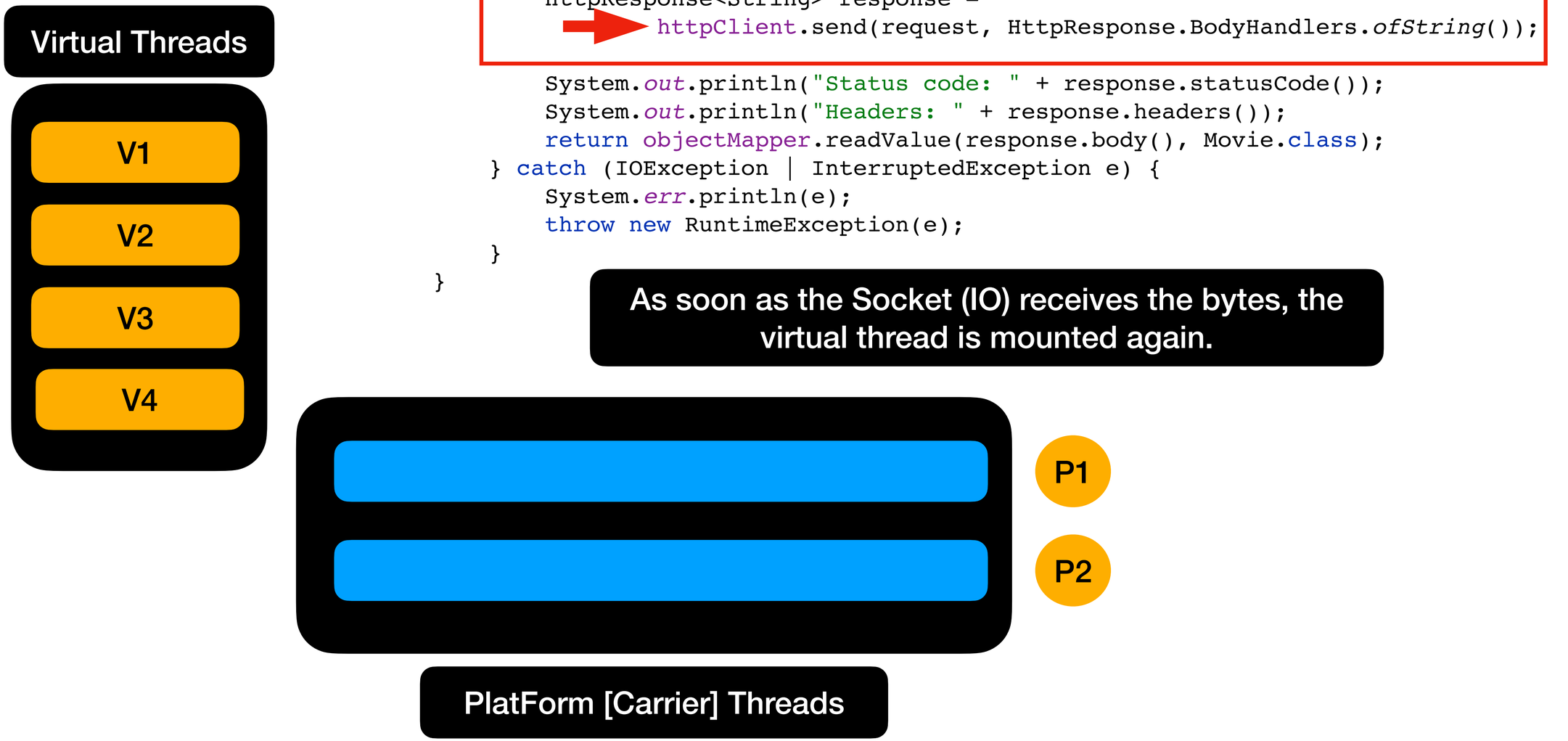


- **Mouting**
  - **Mapping a virtual thread to a carrier thread.**
- **UnMounting**
  - **Removing the virtual thread from the carrier thread.**

Mounting and Unmounting in a HTTP call

```java
public Movie getMovieById() {
    try {
        var request = requestBuilder(MOVIE_BY_ID_URL);
        HttpResponse<String> response =
                httpClient.send(request, HttpResponse.BodyHandlers.ofString());

        System.out.println("Status code: " + response.statusCode());
        System.out.println("Headers: " + response.headers());
        return objectMapper.readValue(response.body(), Movie.class);
    } catch (IOException | InterruptedException e) {
        System.err.println(e);
        throw new RuntimeException(e);
    }
}
```

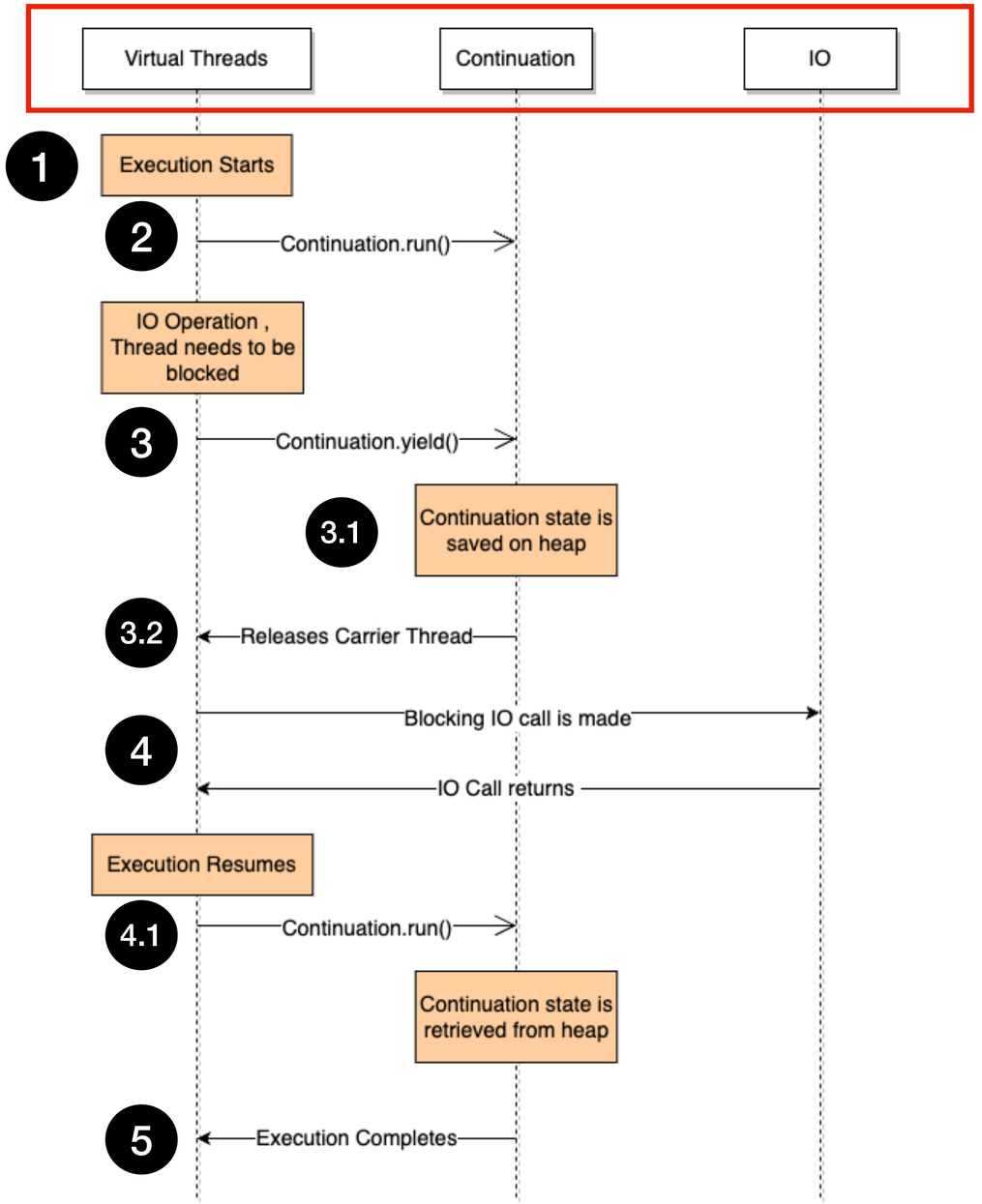


Virtual Threads - yield() and run() using Continuation API

- Virtual Threads are implemented using Continuation API.
- Continuation is a programming technique that allows the program to pause its execution at a specific point and resume from where its left off.
- Continuation API provides a way for a program to capture the following:
  - Current state, including its call stackand local variables in the heap.
  - It later restore that state to resume execution.

Virtual Threads and Continuation

1. Continuation Object is created when the virtual thread is started.
    1. It captures its current state.
2. Continuation.run() is invoked
3. Continuation.yield() is invoked anytime a blocking operation is performed.
    1. Examples : Thread.sleep() or HTTP IO or File IO or any other IO.
    2. Virtual Threads suspends and saves the Continuation state in the heap memory.
    3. Releases the carrier thread.
    4. Now the Carrier thread is free to execute other virtual threads
4. Once the blocking call returns:
    1. This means sleep is exhausted or the socket recieved the bytes from the IO call.
    2. Continuation.run() is invoked, the state is retrieved from heap and continue the execution from where it left off.
5. Virtual Thread completes the execution.



Pinned Threads

- This basically mean the virtual threads are not unmounted from the carrier thread even when performing a blocking operation.

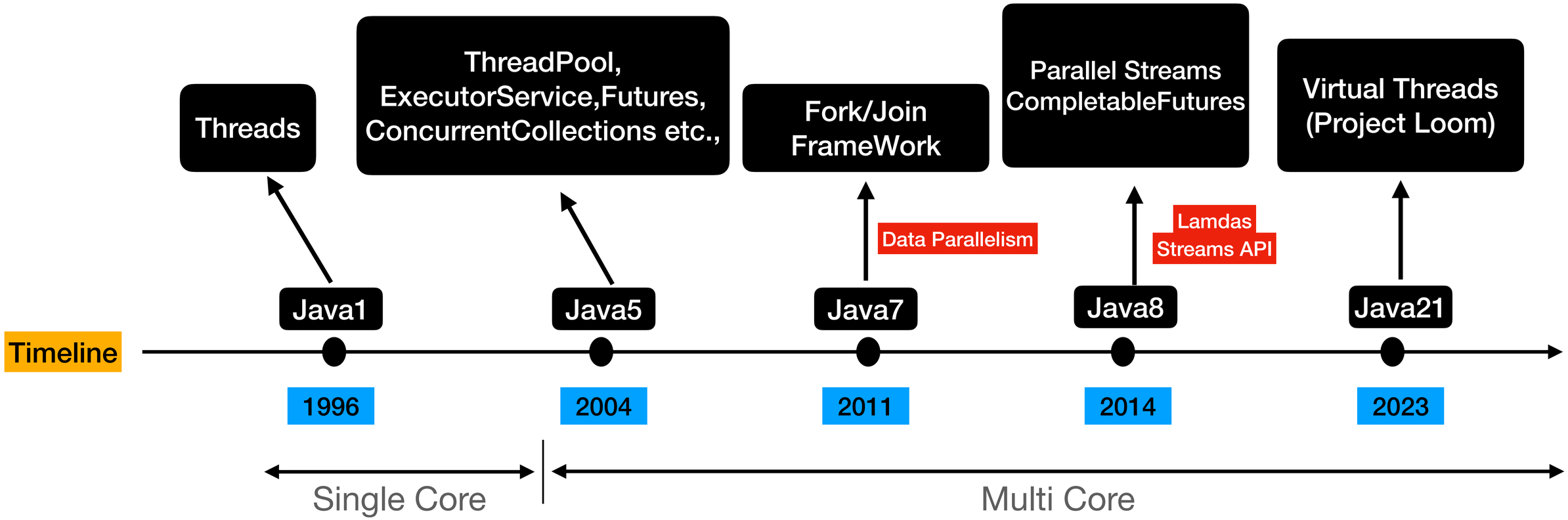
```
private final Object lock = new Object();
public int getAndIncrement(int index ) {
    synchronized (lock) {
        log("started doSomeWork : " + index);
        CommonUtil.sleep(1000);
        log("finished doSomeWork : " + index);
        return counter++;
    }
}
```

- Using synchronized blocks or synchronized functions will cause this to occur.

Important Facts about Virtual Threads

- VirtualThreads aren't faster threads that's going to execute the code faster.
  - It's not going to execute more instructions in a given time compared to the platform thread.
- VirtualThreads makes sure it's going to use the resources efficiently.
  - It helps us to achieve better throughput not latency.
- VirtualThreads are lightweight and its perfect for I/O bound operations.
  - They don't require an OS thread, potentially millions of Virtual Threads can wait for FileSystem, DB or HTTP call.
  - In today's software development IO is pretty common with the applications being built using Microservices architecture.
- Do not pool VirtualThreads.

Evolution of Concurrency APIs



https://en.wikipedia.org/wiki/Java_version_history

Future and ExecutorService

- Future API is part of Java since Java5.
- Future API made multithreaded programming easier.
  - Thread is a low level API and it normally does not return a value.
  - Its hard to model business logic using the Thread API.

Future and ExecutorService

- We define a ExecutorService with fixed number of threads in the threadPool.
  - Threads were expensive to create and manage before virtual threads.
- ExecutorService enabled task based concurrency.

```
static ExecutorService executorService = Executors.newFixedThreadPool(6);


public Product retrieveProductDetails(String productId) throws ExecutionException, InterruptedException, TimeoutException {
1   Future<ProductInfo> productInfoFuture = executorService.submit(() -> productInfoService.retrieveProductInfo(productId));
2   Future<Reviews> reviewFuture = executorService.submit(() -> reviewService.retrieveReviews(productId));
    ProductInfo productInfo = productInfoFuture.get(); 3 // This is a blocking call
    Reviews reviews = reviewFuture.get(); 4 // This is a blocking call

   return new Product(productInfo, reviews); 5
}
```

Limitations of ExecutorService and Future API

- Designed to Block the Thread

```
ProductInfo productInfo = productInfoFuture.get();
Review review = reviewFuture.get();
```

- No better way to combine futures

```
ProductInfo productInfo = productInfoFuture.get();
Review review = reviewFuture.get();
return new Product(productId, productInfo, review);
```

CompletableFuture

- CompletableFuture API got released as part of Java 8.
- This API is implemented using Functional programming.
  - The implementation looks very similar to using Streams API.
- This is a callback based API and its very intuitive to compose and combine tasks using this API.
  - This API overcomes limitations of the Future API

CompletableFuture Summary

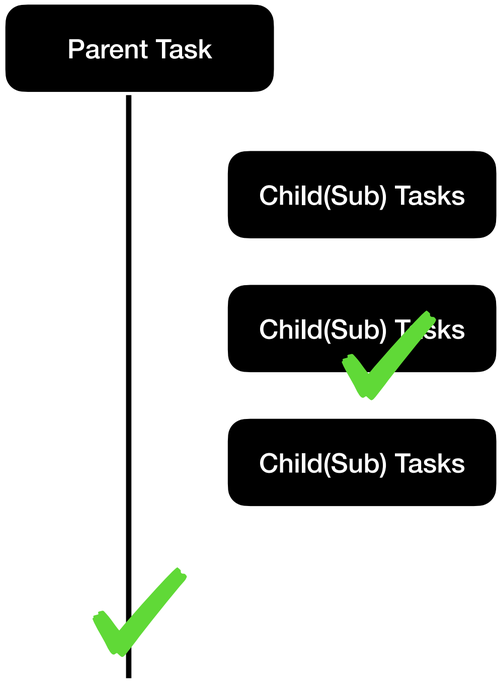
- Blocking IO tasks still going to block the thread in the ForkJoin pool.
- For someone new, you may have a steep learning curve in understanding the API.
- Exception and Error handling is different compared to the traditional way to dealing with exceptions using try/catch block.

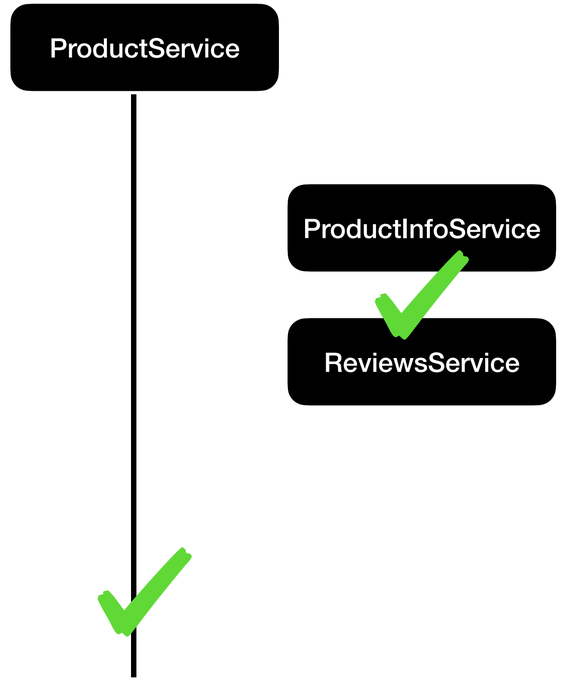
Structured Concurrency

- Structured Concurrency is a concept that helps developers to structure the business logic using VirtualThreads.
- This a new **API** and goal is to focus on the **business logic** instead of worrying about managing the virtual-threads in the thread pool or any non business logic related tasks.

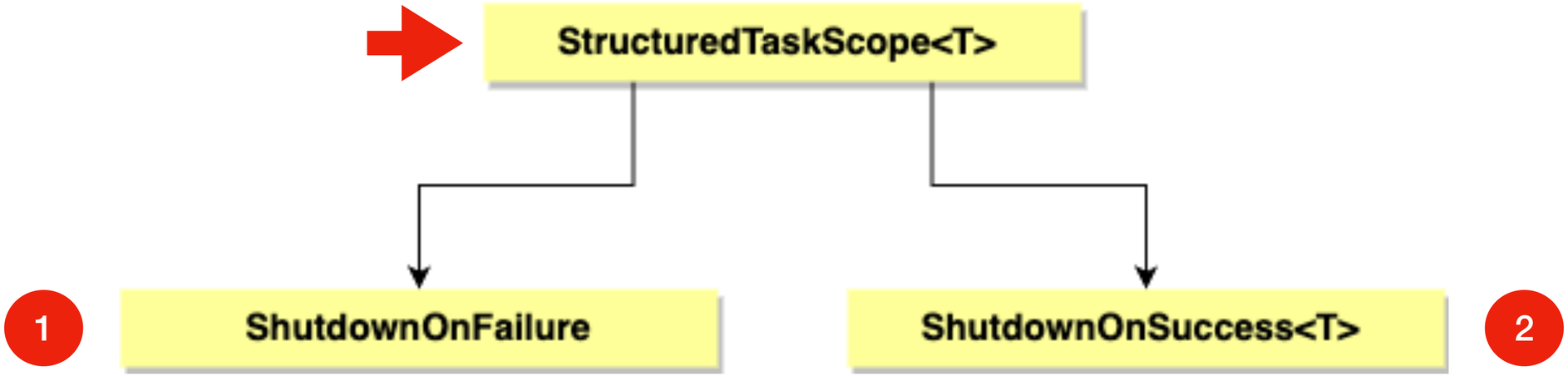
Structured Concurrency

- Structured Concurrency treats group of related tasks running in different threads as a single unit of work by enabling a Parent-Child relationship.





Structured Concurrency API



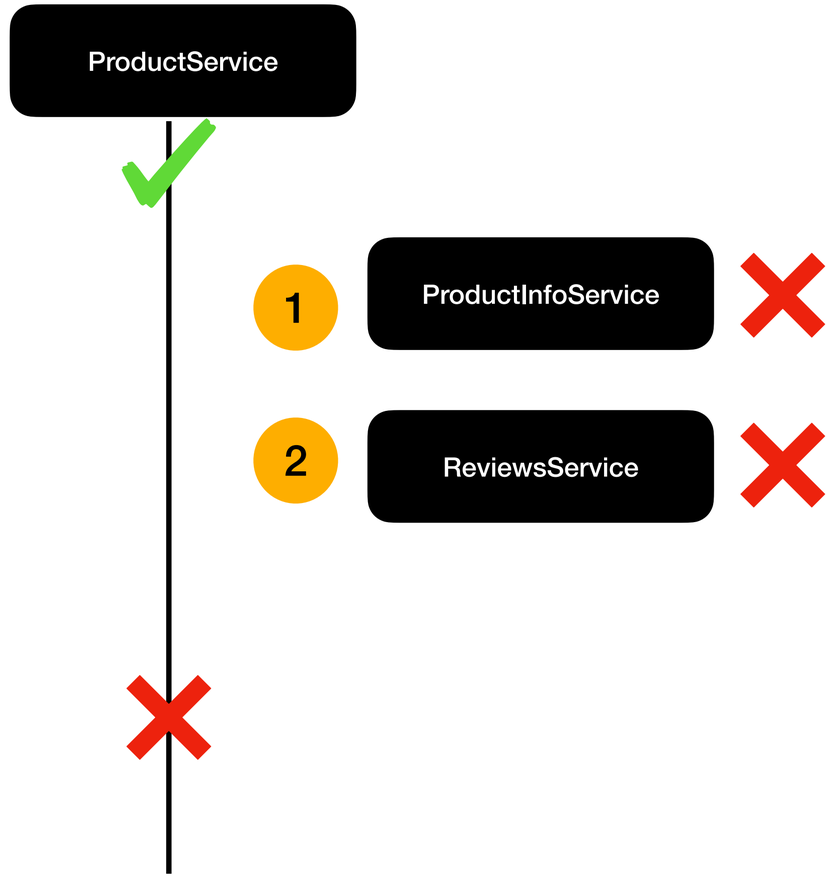
These are also called Shutdown Policies.

Structured Concurrency

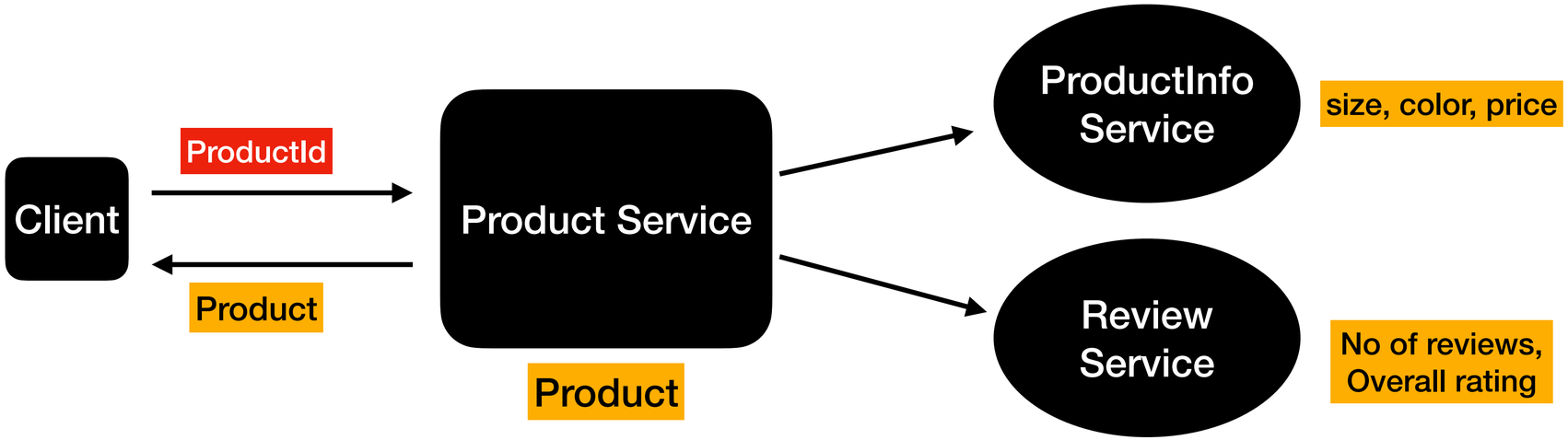
- Benefits:
  - Better Errorhandling using short-circut techniques.
    - If one of the task fails then the API can cancel other tasks automatically for us.
  - Cancellation
    - Parent tasks cancellation propogates the cancellation of child tasks.
  - Enhanced clarity and observation.

ShutDownOnFailure Policy

- This shutdown policy is useful when there are multiple independent but related tasks are involved.
- This policy captures the first exception thrown by one of its subtasks, then invoke the shutdown method.
  - This prevents any new subtasks from starting, interrupts all unfinished threads running other subtasks, and enables the application to continue running.

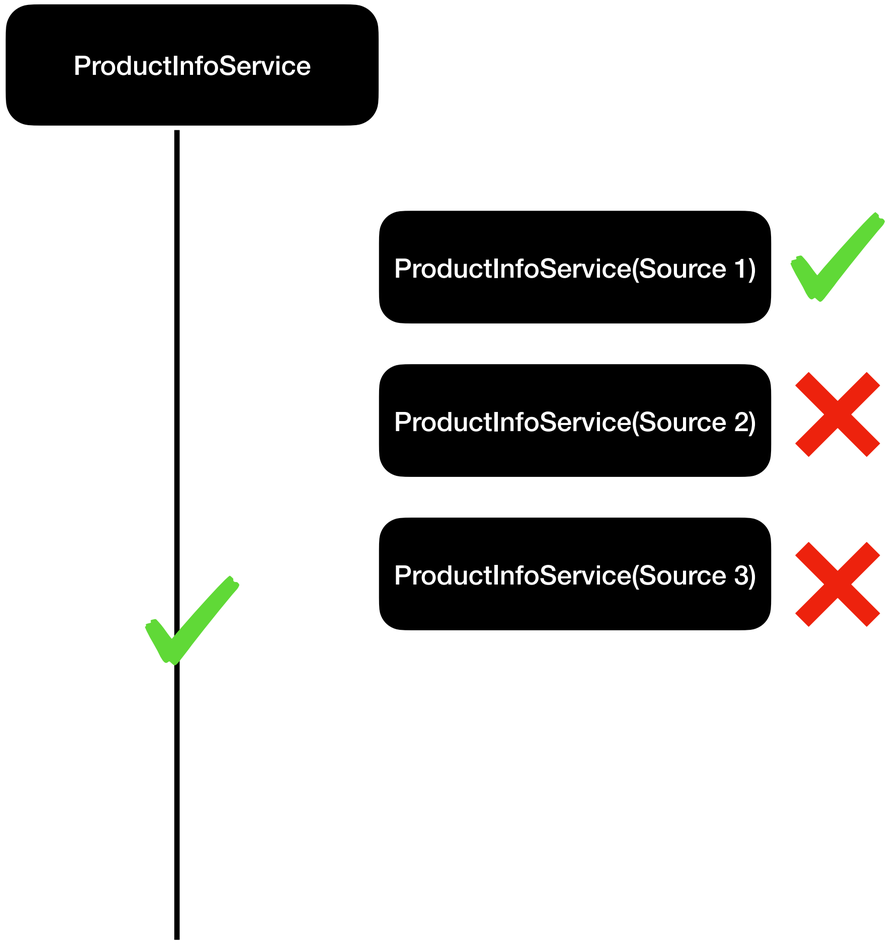


Product Service

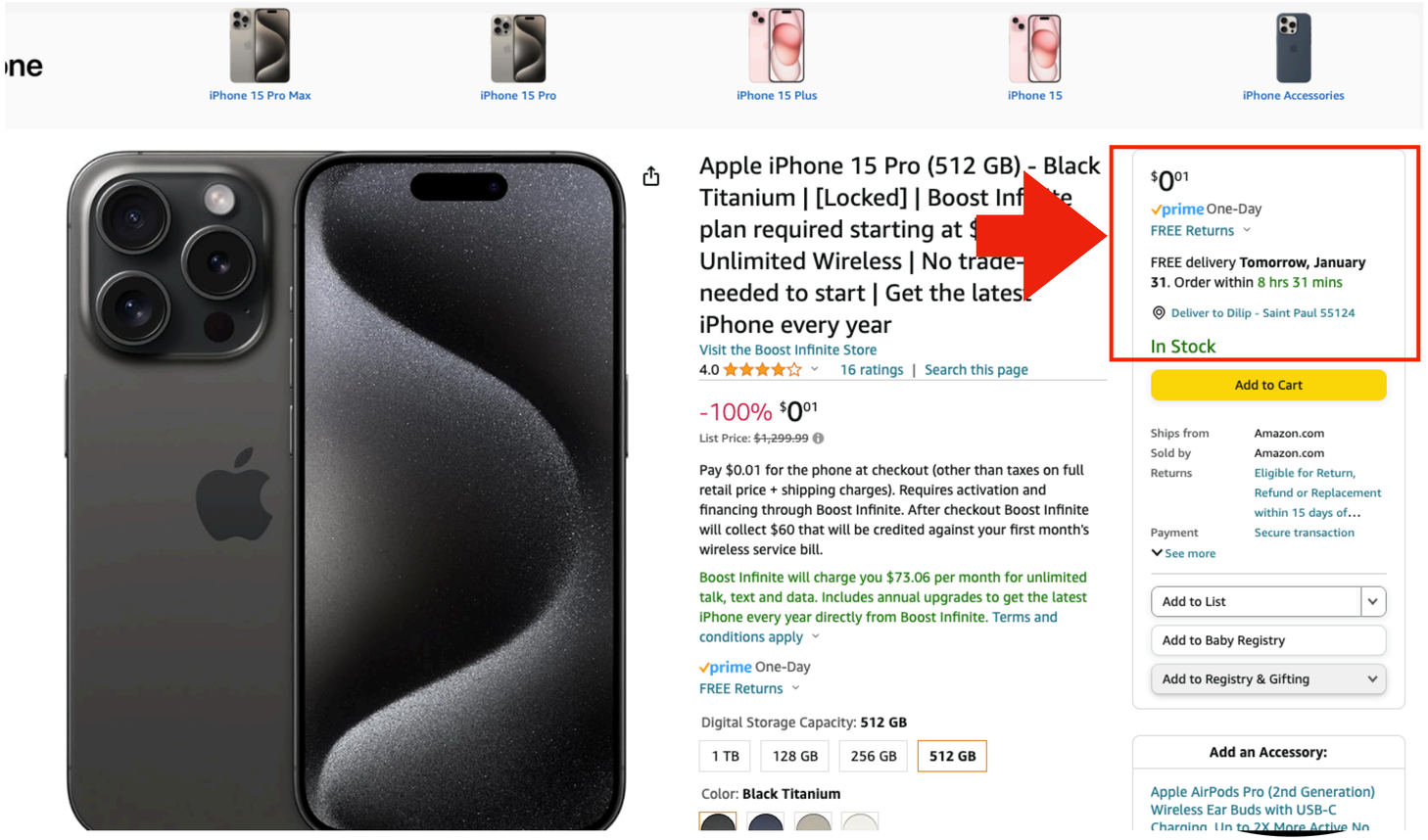


ShutdownOnSuccess Policy

- This policy is useful when you are dealing with the same data being pulled from different sources.
- This policy captures the result of the first subtask to be completed and then invokes the shutDown to cancel the other tasks.



Product Service - New Integration



size, color, price

No of reviews, Overall rating

Same Day, Next Day Two Day

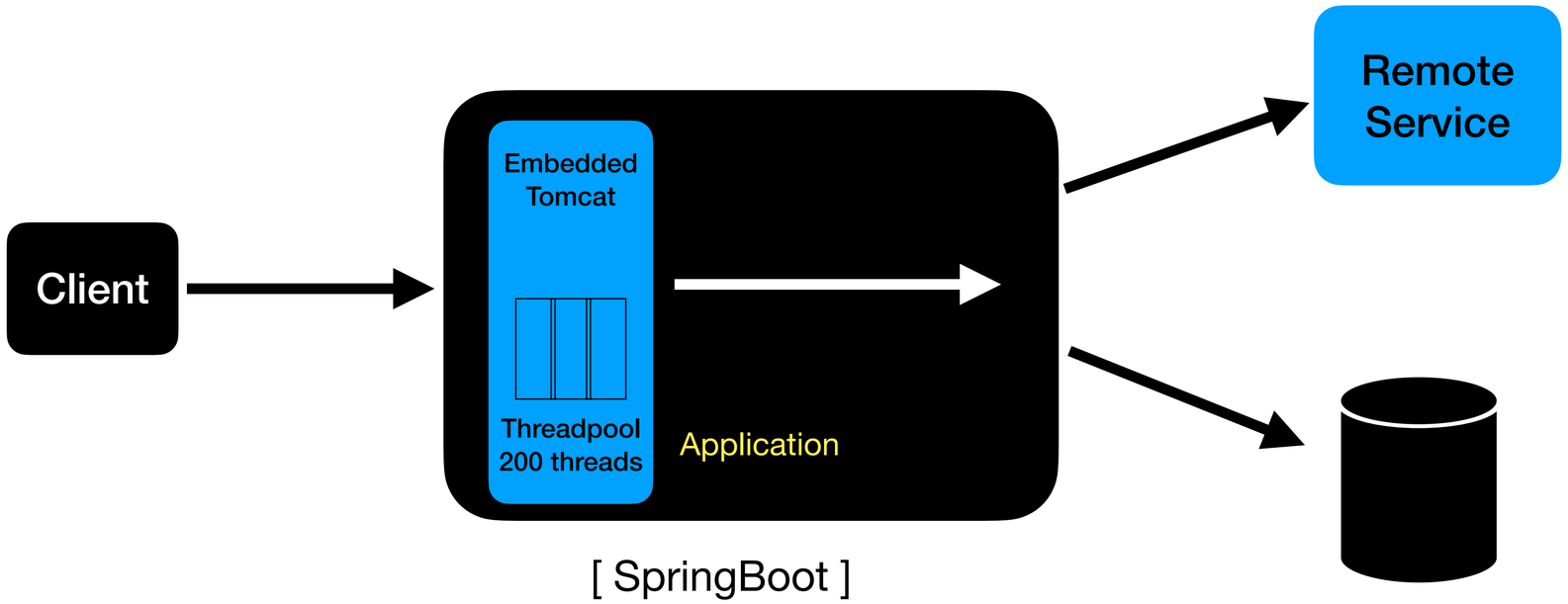
Simple Web Server

- Java18 released a **Simple Web Server.**
  - It’s part of the Java Distribution that’s installed in our machine.
  - This **webserver** servers files and folders from your machine.
- This can be primarily used for prototyping, testing and debugging.
- We can launch the webserver by running the **jwebserver** in the terminal.

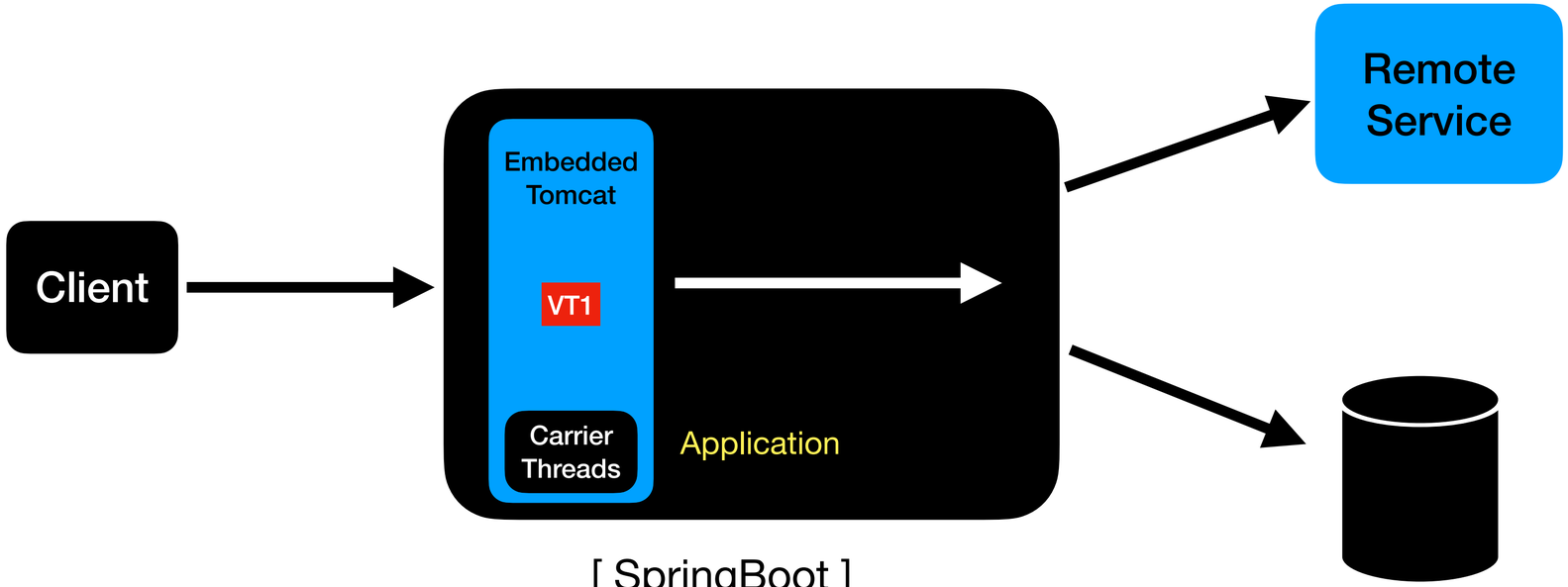
Future & ExecutorService + Virtual Threads

- Use VirtualThreads alongside ExecutorService if the use case is simple.
  - If the business logic involves just one IO call.

Spring MVC Thread Per Request Architecture



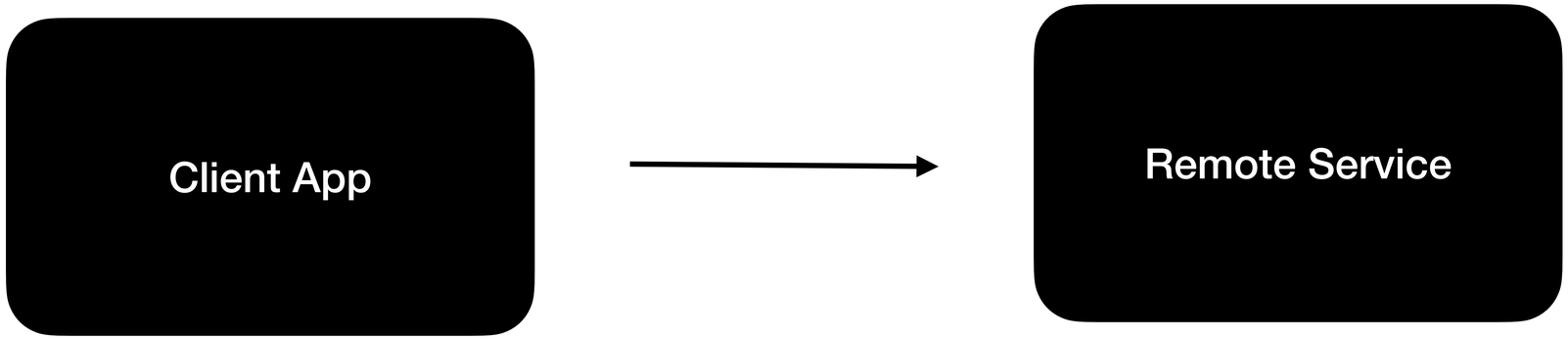
Spring MVC Virtual Thread Per Request Architecture



[SpringBoot]

- New Virtual Thread is created for every request.
  - It gets exeuted by the Carrier Thread.
- Once the request is fulfilled then the virtual thread gets destroyed.
- No threadpools needed because VirtualThreads are cheap to create and destroy.

Overview of the application



Virtual Threads to handle the Http Request.

Virtual Threads to make Http calls.