Java Virtual Threads

Virtual threads (JEP-425) are **JVM-managed lightweight threads** that **help in writing high-throughput concurrent applications** (*throughput* means how many units of information a system can process in a given amount of time).

Platform Threads (and their problems)

Traditionally, **Java has treated the platform threads as thin wrappers around operating system (OS) threads**. Creating such platform threads has always been costly (due to a large stack and other resources that are maintained by the operating system), so Java has been using the thread pools to avoid the overhead in thread creation.

The number of platform threads also has to be limited because these resource-hungry threads can affect the performance of the whole machine. This is mainly because platform threads are mapped `1:1` to OS threads.

Scalability Issues

Platform threads have always been easy to model, program and debug because they use the platform's unit of concurrency to represent the application's unit of concurrency. It is called **thread-per-request** pattern.

However, this pattern limits the throughput of the server because the **number of concurrent requests *(that server can handle)* becomes directly proportional to the server's hardware performance**. So, the number of available threads has to be limited even in multicore processors.

Apart from the number of threads, **latency** is also a big concern. If you watch closely, in today's world of microservices, a request is served by fetching/updating data on multiple systems and servers. **While the application waits for the information from other servers, the current platform thread remains in an idle state.** This is a waste of computing resources and a major hurdle in achieving a high throughput application.

Reactive Programming Issues

Reactive style programming solved the problem of platform threads waiting for responses from other systems. The asynchronous APIs do not wait for the response, rather they work through the callbacks. Whenever a thread invokes an async API, the platform thread is returned to the pool until the response comes back from the remote system or database. Later, when the response arrives, the JVM will allocate

another thread from the pool that will handle the response and so on. This way, **multiple threads are involved in handling a single async request**.

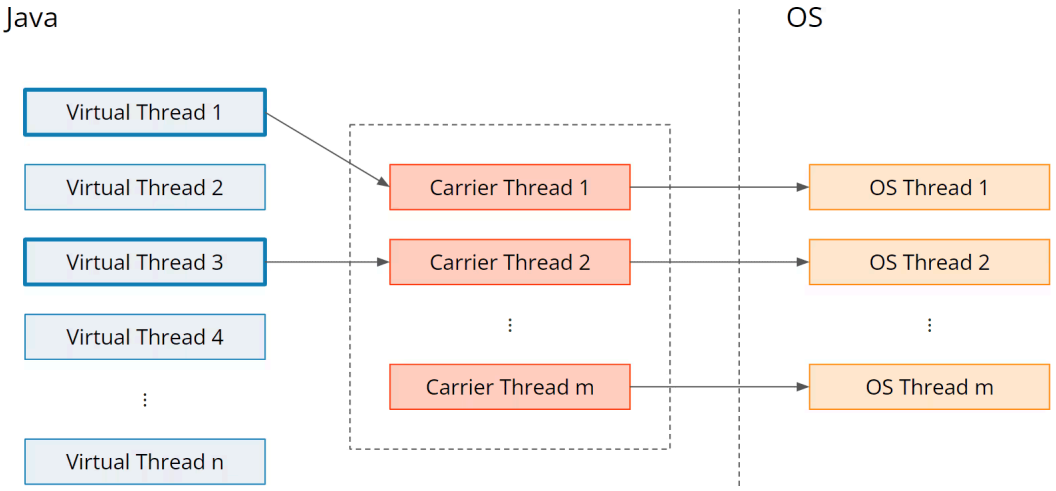
In async programming, the latency is removed but the number of platform threads are still limited due to hardware limitations, so we have a limit on scalability. Another big issue is that such **async programs are executed in different threads, so it is very hard to debug or profile them**.

Also, we have to adopt a new programming style away from typical loops and conditional statements. The new lambda-style syntax makes it hard to understand the existing code and write programs because we must now break our program into multiple smaller units that can be run independently and asynchronously.

Virtual Threads

We can say that **virtual threads improve the code quality** by adapting the traditional syntax while having the benefits of reactive programming.

Similar to traditional threads, **a virtual thread is also an instance of `_java.lang.Thread_`** that runs its code on an underlying OS thread, but it **does not block the OS thread for the code's entire lifetime**. Keeping the OS threads free means that many virtual threads can run their Java code on the same OS thread, effectively sharing it.

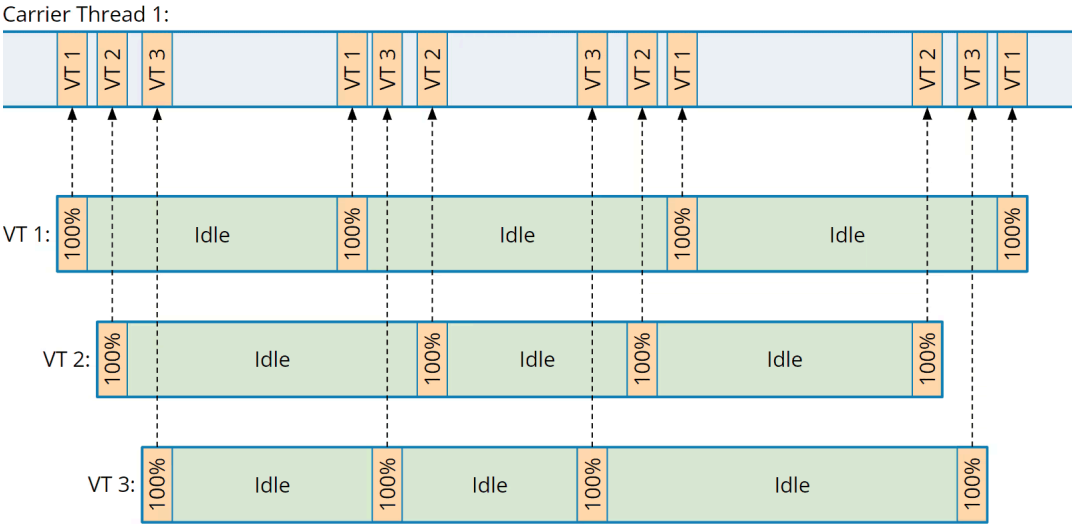


It is worth mentioning that we can create a very high number of virtual threads (*millions*) in an application without depending on the number of platform threads. These **virtual threads are managed by JVM**, so they do not add extra context-switching overhead as well because they are stored in RAM as normal Java objects.

Similar to traditional threads, the application's code runs in a virtual thread for the entire duration of a request (in *thread-per-request* style) but the **virtual thread consumes an OS thread only when it performs the calculations on the CPU**. They do not block the OS thread while they are waiting or sleeping.

Virtual threads help in achieving the same high scalability and throughput as the asynchronous APIs with the same hardware configuration, without adding the syntax complexity.

Virtual threads are best suited to executing code that spends most of its time blocked, waiting for data to arrive at a network socket or waiting for an element in queue for example.



Difference between Platform Threads and Virtual Threads

Virtual threads are always daemon threads. The `Thread.setDaemon(false)` method cannot change a virtual thread to be a non-daemon thread. Note that JVM terminates when all started non-daemon threads have terminated. This means JVM will not wait for virtual threads to complete before exiting.

```
Thread virtualThread = ...; //Create virtual thread

//virtualThread.setDaemon(true);  //It has no effect
```

Virtual threads always have the normal priority and the priority cannot be changed, even with `setPriority(n)` method. Calling this method on a virtual thread has no effect.

```
Thread virtualThread = ...; //Create virtual thread

//virtualThread.setPriority(Thread.MAX_PRIORITY);  //It has no effe
```

Virtual threads are not active members of thread groups. When invoked on a virtual thread, `Thread.getThreadGroup()` returns a placeholder thread group with the name "*VirtualThreads*".

Virtual threads do not support the *stop()*, *suspend()*, or *resume()* methods. These methods throw an *UnsupportedOperationException* when invoked on a virtual thread.

Performance Comparison

Let us understand the difference between both kinds of threads when they are submitted with the same executable code.

To show it, we define a simple task that waits for one second before printing a message in the console.

```
final AtomicInteger atomicInteger = new AtomicInteger();

Runnable runnable = () -> {
  try {
    Thread.sleep(Duration.ofSeconds(1));
  } catch(Exception e) {
      System.out.println(e);
  }
  System.out.println("Work Done - " + atomicInteger.incrementAndGet
};
```

Now we will create 10,000 threads from this *Runnable* and execute them with virtual threads and platform threads to compare the performance of both. We will use the *Duration.between()* api to measure the elapsed time in executing all the tasks.

First, we are using a pool of 100 platform threads. In this way, the *Executor* will be able to run 100 tasks at a time and other tasks will need to wait. As we have 10,000 tasks so the total time to finish the execution will be approximately 100 seconds.

```
Instant start = Instant.now();

try (var executor = Executors.newFixedThreadPool(100)) {
  for(int i = 0; i < 10_000; i++) {
    executor.submit(runnable);
  }
}

Instant finish = Instant.now();
long timeElapsed = Duration.between(start, finish).toMillis();
System.out.println("Total elapsed time : " + timeElapsed);
```

Output:

```
Total elapsed time : 101152 // Approx 101 seconds
```

Next, we will replace the *Executors.newFixedThreadPool(100)* with *Executors.newVirtualThreadPerTaskExecutor()*. This will **execute all the tasks in virtual threads** instead of platform threads.

```
Instant start = Instant.now();

try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
  for(int i = 0; i < 10_000; i++) {
    executor.submit(runnable);
  }
}

Instant finish = Instant.now();
long timeElapsed = Duration.between(start, finish).toMillis();
System.out.println("Total elapsed time : " + timeElapsed);
```

Output:

```
Total elapsed time : 1589 // Approx 1.5 seconds
```

Notice the blazing fast performance of virtual threads that brought down the execution time from 100 seconds to 1.5 seconds with no change in the *Runnable* code.

How to Create Virtual Threads

Using *Thread.startVirtualThread()*

This method creates a new virtual thread to execute a given *Runnable* task and schedules it to execute.

```
Runnable runnable = () -> System.out.println("Inside Runnable");
Thread.startVirtualThread(runnable);

//or

Thread.startVirtualThread(() -> {
    //Code to execute in virtual thread
    System.out.println("Inside Runnable");
});
```

Using *Thread.Builder*

If we want to explicitly start the thread after creating it, we can use `Thread.ofVirtual()` that returns a *VirtualThreadBuilder* instance. Its `start()` method starts a virtual thread.

It is worth noting that *Thread.ofVirtual().start(runnable)* is equivalent to *Thread.startVirtualThread(runnable)*.

```
Runnable runnable = () -> System.out.println("Inside Runnable");
Thread virtualThread = Thread.ofVirtual().start(runnable);
```

We can use the *Thread.Builder* reference to create and start multiple threads.

```
Runnable runnable = () -> System.out.println("Inside Runnable");

Thread.Builder builder = Thread.ofVirtual().name("JVM-Thread");

Thread t1 = builder.start(runnable);
Thread t2 = builder.start(runnable);
```

A similar API `Thread.ofPlatform()` exists for creating platform threads as well.

```
Thread.Builder builder = Thread.ofPlatform().name("Platform-Thread"

Thread t1 = builder.start(() -> {/*...*/});
Thread t2 = builder.start(() -> {/*...*/});
```

Using *Executors.newVirtualThreadPerTaskExecutor()*

This method **creates one new virtual thread per task**. The number of threads created by the *Executor* is unbounded.

In the following example, we are submitting 10,000 tasks and waiting for all of them to complete. The code will create 10,000 virtual threads to complete these 10,000 tasks.

Note that the following syntax is part of structured concurrency, a new feature proposed in ***Project Loom***.

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
}
```

Best Practices

DO NOT Pool the Virtual Threads

Java thread pool was designed to avoid the overhead of creating new OS threads because creating them was a costly operation. But creating virtual threads is not expensive, so, there is never a need to pool them. It is advised to create a new virtual thread everytime we need one.

Note that after using the virtual threads, our application may be able to handle millions of threads, but other systems or platforms handle only a few requests at a time. For example, we can have only a few database connections or network connections to other servers.

In these cases also, do not use the thread pool. Instead, use semaphores to make sure only a specified number of threads are accessing that resource.

```
private static final Semaphore SEMAPHORE = new Semaphore(50);

SEMAPHORE.acquire();

try {
  // semaphore limits to 50 concurrent access requests
  //Access the database or resource
} finally {
  SEMAPHORE.release();
}
```

Avoid using Thread-local Variables

Virtual threads support thread-local behavior the same way as platform threads, but because the virtual threads can be created in millions, thread-local variables should be used only after careful consideration.

For example, if we scale a million virtual threads in the application, there will be a million *ThreadLocal* instances along with the data they refer to. Such a large number of instances can put enough burden on the physical memory and should be avoided.

Extent-Local variables [JEP-429] are a better alternative. Note that in Java 21 [JEP-444], virtual threads now support thread-local variables all the time. It is no longer possible, as it was in the preview releases, to create virtual threads that cannot have thread-local variables.

Use *ReentrantLock* instead of *Synchronized* Blocks

There are two specific scenarios in which a virtual thread can block the platform thread (called **pinning of OS threads**):

- When it executes code inside a *synchronized* block or method
- When it executes a *native method* or a *foreign function*

Such `synchronized` block does not make the application incorrect, but it limits the scalability of the application similar to platform threads.

As a best practice, if a method is used very frequently and uses a *synchronized* block then consider replacing it with the *ReentrantLock* mechanism.

So instead of using `synchronized` block like this:

```
public synchronized void method() {
    try {
        // ... access resource
    } finally {
        //
    }
}
```

use `ReentrantLock` like this:

```
private final ReentrantLock lock = new ReentrantLock();

public void method() {
    lock.lock();  // block until condition holds
    try {
        // ... access resource
    } finally {
        lock.unlock();
    }
}
```

It is suggested that **there is no need to replace *synchronized* blocks and methods that are used infrequently** (e.g., only performed at startup) or that guard in-memory operations.

Annex: ReentrantLock, CountDownLatch, CompletableFuture

ReentrantLock

The `ReentrantLock` class in Java is a synchronization mechanism that provides the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements but with extended capabilities.

`ReentrantLock` is part of the `java.util.concurrent.locks` package and offers some advanced features compared to the standard synchronized block, such as:

- Reentrancy: This feature allows the lock to be acquired multiple times by the same thread without causing a deadlock, provided the thread holds the lock, it can re-enter any block of code under that lock.
- Lock Interruptibility: Threads waiting for a lock can be interrupted and asked to stop waiting, enabling the handling of thread interruptions.
- Try Lock: This feature allows a thread to attempt to acquire the lock without waiting, providing an immediate return of success or failure, which is useful for avoiding deadlock scenarios.
- Fairness Parameter: Optionally, the lock can be fair, meaning it can grant access in a FIFO manner, preventing thread starvation but potentially reducing throughput.

When integrating `ReentrantLock` with virtual threads, the lightweight nature of virtual threads and their efficient handling of blocking operations bring several benefits:

- Reduced Overhead in Lock Management: Virtual threads are lightweight and can be blocked and unblocked with minimal resource overhead. This characteristic makes the use of locks, like `ReentrantLock`, less costly in terms of system resources compared to using platform threads.
- Improved Scalability: With virtual threads, applications can handle a significantly higher number of concurrent tasks. This scalability extends to synchronization primitives like `ReentrantLock`, allowing more threads to wait for or hold a lock without the heavy resource footprint associated with platform threads.
- Simplified Concurrent Code: The ability to use virtual threads encourages a programming model where developers can write simple, straightforward synchronous code. Synchronization mechanisms like `ReentrantLock` fit naturally into this model, as they provide a clear and well-understood mechanism for managing concurrent access to shared resources.

- Integration with Java Concurrency Utilities: `ReentrantLock` and other concurrency utilities are designed to work seamlessly with virtual threads. This means that the same patterns and practices used with platform threads for locking and synchronization can be applied to virtual threads, with the added benefits of the lightweight nature of virtual threads.
- Fairness and Throughput Considerations: While the fairness parameter in `ReentrantLock` can prevent thread starvation, its usage with a large number of virtual threads should be considered carefully. Fair locks typically have lower throughput than non-fair locks, and with the high scalability of virtual threads, the impact on throughput might be more pronounced.

```
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockExample {

    public static void main(String[] args) {
        ReentrantLock lock = new ReentrantLock();

        Runnable task = () -> {
            lock.lock();
            try {
                System.out.println(Thread.currentThread().getName()
                // Simulate some work
                Thread.sleep(1000);
                System.out.println(Thread.currentThread().getName()
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        };

        Thread virtualThread1 = Thread.ofVirtual().start(task);
        Thread virtualThread2 = Thread.ofVirtual().start(task);

        virtualThread1.join();
        virtualThread2.join();
    }
}
```

In this example, we create a `ReentrantLock` and have two Virtual Threads compete for the lock. Each thread acquires the lock, performs some work (simulated by `Thread.sleep`), and then releases the lock. The lock ensures that only one thread can execute the critical section at a time.

CountDownLatch

The `CountDownLatch` class in Java is a concurrency utility that allows one or more threads to wait until a set of operations being performed in other threads completes. It's part of the `java.util.concurrent` package and is a versatile tool for synchronizing the start or completion of tasks in a concurrent environment.

`CountDownLatch` works on a count-down mechanism. You initialize it with a count, and `await()` methods block until the count reaches zero. Other threads, usually executing different parts of the program, decrement this count using the `countDown()` method after completing their tasks. Key features include:

- Synchronization Point: `CountDownLatch` serves as a synchronization point where one or more threads wait for other threads to complete their tasks.
- One-Time Use: The latch is a single-use barrier; once the count reaches zero, it cannot be reset.
- Versatility: It can be used for starting a set of tasks after certain preparatory work is complete or for waiting for a set of tasks to complete before proceeding.

With the introduction of virtual threads, `CountDownLatch` becomes an even more powerful tool due to the lightweight nature and scalability of virtual threads. Here's how they interact and the benefits they offer:

- Efficient Waiting Mechanism: Virtual threads can be blocked without consuming significant resources. When a virtual thread calls `await()` on a `CountDownLatch`, it can be unscheduled and parked efficiently until the count reaches zero, freeing up the underlying carrier thread to execute other tasks. This capability makes `CountDownLatch` a more resource-efficient synchronization tool in a virtual threads environment.
- High Scalability in Concurrent Operations: The scalability of virtual threads allows a large number of operations to be coordinated using a `CountDownLatch`. You can initiate a vast number of tasks, each running on a virtual thread, and synchronize their completion or initiation point using a latch, without worrying about the overhead that traditionally comes with managing a large number of threads.
- Simplified Task Synchronization: The `CountDownLatch` allows you to synchronize tasks in a straightforward manner, which complements the simplified programming model encouraged by virtual threads. Developers can structure their concurrent code around clear synchronization points without getting entangled in the complexities of thread management.
- Enhanced Responsiveness and Throughput: The ability of virtual threads to efficiently block and unblock, combined with the count-down mechanism of `CountDownLatch`, means that applications can be more responsive. Tasks don't need to poll or busy-wait for other tasks to complete; they can wait on a latch and be reactivated precisely when needed, leading to better CPU utilization and throughput.

- Facilitating Complex Workflow Coordination: In scenarios where complex workflows involve multiple stages or phases of execution, `CountDownLatch` can effectively manage dependencies between different stages. With virtual threads, even complex, multi-stage workflows can scale efficiently, with each stage waiting on or triggering subsequent stages through latches.

```
import java.util.concurrent.CountDownLatch;

public class CountDownLatchExample {

    public static void main(String[] args) throws InterruptedExcept
        int numberOfThreads = 3;
        CountDownLatch latch = new CountDownLatch(numberOfThreads);

        Runnable task = () -> {
            System.out.println(Thread.currentThread().getName() + "
            // Simulate some work
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "
            latch.countDown();
        };

        for (int i = 0; i < numberOfThreads; i++) {
            Thread.ofVirtual().start(task);
        }

        latch.await();
        System.out.println("All threads have completed their work."
    }
}
```

In this example, we use a `CountDownLatch` to coordinate multiple Virtual Threads. Each thread performs some work and then counts down the latch. The main thread waits until all Virtual Threads have completed their work, as indicated by the latch.

CompletableFuture Basics

The `CompletableFuture` class in Java, a key component of the `java.util.concurrent` package, represents a future result of an asynchronous computation—a result that will eventually appear in the future. It provides a rich API that facilitates chaining asynchronous tasks, combining multiple tasks, and applying transformations and actions upon the completion of the asynchronous operations. Notable features include:

- Asynchronous Computation: `CompletableFuture` allows you to specify what should happen after a particular asynchronous operation completes, without blocking the thread waiting for the result.
- Chaining and Combining: You can chain multiple stages sequentially or execute them in a fork-join fashion where the completion of all or any one of the previous stages triggers the execution of a subsequent stage.
- Error Handling: It provides methods for gracefully handling exceptions that occur during the asynchronous computation.

The integration of `CompletableFuture` with virtual threads brings forth a synergy that leverages the lightweight, scalable nature of virtual threads along with the non-blocking, asynchronous capabilities of `CompletableFuture`. Here's how they complement each other:

- Efficient Resource Utilization: Virtual threads are extremely lightweight and can be blocked or parked with minimal resource overhead. When using `CompletableFuture` with virtual threads, the asynchronous tasks running in virtual threads can block without consuming significant system resources, leading to more efficient resource utilization.
- Enhanced Scalability for Asynchronous Tasks: Virtual threads allow you to spawn a large number of concurrent tasks without the heavy memory footprint associated with traditional threads. This scalability is particularly beneficial for `CompletableFuture` operations, as it means you can have a significantly higher number of asynchronous operations being processed concurrently, improving the overall throughput of your application.
- Simplified Error Handling in Asynchronous Flows: `CompletableFuture` provides robust mechanisms for handling exceptions in asynchronous code. When combined with virtual threads, error handling becomes more straightforward and manageable, even in a highly concurrent environment. The lightweight nature of virtual threads means that each asynchronous operation can be handled in its isolated thread without impacting the performance or stability of the overall system.
- Blocking APIs in Asynchronous Operations: One of the challenges with asynchronous programming is dealing with APIs that are inherently blocking. Virtual threads make it feasible to use blocking APIs within `CompletableFuture` chains without incurring significant performance penalties. The virtual thread can be blocked, but the underlying carrier thread is released to execute other tasks, thereby maintaining high system throughput.
- Seamless Integration with Existing Code: Both `CompletableFuture` and virtual threads are designed to integrate seamlessly with existing Java code. This means you can enhance the concurrency and asynchronicity of your existing applications with minimal refactoring, bringing the benefits of non-blocking asynchronous programming and scalable concurrency to legacy systems.

```
import java.util.concurrent.CompletableFuture;

public class CompletableFutureExample {

    public static void main(String[] args) throws InterruptedExcept
        CompletableFuture<Integer> future1 = CompletableFuture.supp
            System.out.println(Thread.currentThread().getName() + "
            // Simulate some work
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return 42;
        });

        CompletableFuture<Integer> future2 = CompletableFuture.supp
            System.out.println(Thread.currentThread().getName() + "
            // Simulate some work
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return 10;
        });

        CompletableFuture<Integer> combinedFuture = future1.thenComb

        combinedFuture.thenAccept(result -> {
            System.out.println("Combined result: " + result);
        });

        // Wait for the combined future to complete
        combinedFuture.join();
    }
}
```

In this example, we use `CompletableFuture` to perform asynchronous computations with Virtual Threads. Two asynchronous tasks (`future1` and `future2`) are created, each simulating some work. We then combine the results of these tasks and print the combined result when they are both completed.

Resources

- Virtual Threads: New Foundations for High-Scale Java Applications
- Guide to java.util.concurrent.Locks