OpenJDK
Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds
Mailing lists
Wiki · IRC
Bylaws · Census
Legal
Workshop
JEP Process
Source code
Mercurial
GitHub
Tools
Git
jtreg harness
Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web
Projects
(overview, archive)
Amber
Audio Engine
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Font Scaler
Galahad
Graal
Graphics Rasterizer
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 21, 22)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Mission Control
Modules
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
Tiered Attribution
Tsan
Type Annotations
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC
ORACLE

JEP 444: Virtual Threads

Author	Ron Pressler & Alan Bateman
Owner	Alan Bateman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	core-libs
Discussion	loom dash dev at openjdk dot org
Relates to	JEP 436: Virtual Threads (Second Preview)
Reviewed by	Alex Buckley
Endorsed by	Brian Goetz
Created	2023/03/06 18:00
Updated	2023/09/22 16:54
Issue	8303683

Summary

Introduce *virtual threads* to the Java Platform. Virtual threads are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.

History

Virtual threads were proposed as a preview feature by JEP 425 and delivered in JDK 19. To allow time for feedback and to get more experience, they were proposed as a preview feature again by JEP 436 and delivered in JDK 20. This JEP proposes to finalize virtual threads in JDK 21 with the following changes from JDK 20, informed by developer feedback:

- Virtual threads now always support thread-local variables. It is no longer possible, as it was in the preview releases, to create virtual threads that cannot have thread-local variables. Guaranteed support for thread-local variables ensures that many more existing libraries can be used unchanged with virtual threads, and helps with the migration of task-oriented code to use virtual threads.
- Virtual threads created directly with the `Thread.Builder` API (as opposed to those created through `Executors.newVirtualThreadPerTaskExecutor()`) are now also, by default, monitored throughout their lifetime and observable via the new thread dump described in the *Observing virtual threads* section.

Goals

- 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.

Non-Goals

- It is not a goal to remove the traditional implementation of threads, or to silently migrate existing applications to use virtual threads.
- It is not a goal to change the basic concurrency model of Java.
- It is not a goal to offer a new data parallelism construct in either the Java language or the Java libraries. The Stream API remains the preferred way to process large data sets in parallel.

Motivation

Java developers have relied on threads as the building block of concurrent server applications for nearly three decades. Every statement in every method is executed inside a thread and, since Java is multithreaded, multiple threads of execution happen at once. The thread is Java's *unit of concurrency:* a piece of sequential code that runs concurrently with — and largely independently of — other such units. Each thread provides a stack to store local variables and coordinate method calls, as well as context when things go wrong: Exceptions are thrown and caught by methods in the same thread, so developers can use a thread's stack trace to find out what happened. Threads are also a central concept for tools: Debuggers step through the statements in a thread's methods, and profilers visualize the behavior of multiple threads to help understand their performance.

The thread-per-request style

Server applications generally handle concurrent user requests that are independent of each other, so it makes sense for an application to handle a request by dedicating a thread to that request for its entire duration. This *thread-per-request style* is easy to understand, easy to program, and easy to debug and profile because it uses the platform's unit of concurrency to represent the application's unit of concurrency.

The scalability of server applications is governed by Little's Law, which relates latency, concurrency, and throughput: For a given request-processing duration (i.e., latency), the number of requests an application handles at the same time (i.e., concurrency) must grow in proportion to the rate of arrival (i.e., throughput). For example, suppose an application with an average latency of 50ms achieves a throughput of 200 requests per second by processing 10 requests concurrently. In order for that application to scale to a throughput of 2000 requests per second, it will need to process 100 requests concurrently. If each request is handled in a thread for the request's duration then, for the application to keep up, the number of threads must grow as throughput grows.

Unfortunately, the number of available threads is limited because the JDK implements threads as wrappers around operating system (OS) threads. OS threads are costly, so we cannot have too many of them, which makes the implementation ill-suited to the thread-per-request style. If each request consumes a thread, and thus an OS thread, for its duration, then the number of threads often becomes the limiting factor long before other resources, such as CPU or network connections, are exhausted. The JDK's current implementation of threads caps the application's throughput to a level well below what the hardware can support. This happens even when threads are pooled, since pooling helps avoid the high cost of starting a new thread but does not increase the total number of threads.

Improving scalability with the asynchronous style

Some developers wishing to utilize hardware to its fullest have given up the thread-per-request style in favor of a thread-sharing style. Instead of handling a request on one thread from start to finish, request-handling code returns its thread to a pool when it waits for another I/O operation to complete so that the thread can service other requests. This fine-grained sharing of threads — in which code holds on to a thread only while it performs calculations, not while it waits for I/O — allows a high number of concurrent operations without consuming a high number of threads. While it removes the limitation on throughput imposed by the scarcity of OS threads, it comes at a high price: It requires what is known as an *asynchronous* programming style, employing a separate set of I/O methods that do not wait for I/O operations to complete but rather, later on, signal their completion to a callback. Without a dedicated thread, developers must break down their request-handling logic into small stages, typically written as lambda expressions, and then compose them into a sequential pipeline with an API (see CompletableFuture, for example, or so-called "reactive" frameworks). They thus forsake the language's basic sequential composition operators, such as loops and `try/catch` blocks.

In the asynchronous style, each stage of a request might execute on a different thread, and every thread runs stages belonging to different requests in an interleaved fashion. This has deep implications for understanding program behavior: Stack traces provide no usable context, debuggers cannot step through request-handling logic, and profilers cannot associate an operation's cost with its caller. Composing lambda expressions is manageable when using Java's stream API to process data in a short pipeline but problematic when all of the request-handling code in an application must be written in this way. This programming style is at odds with the Java Platform because the application's unit of concurrency — the asynchronous pipeline — is no longer the platform's unit of concurrency.

Preserving the thread-per-request style with virtual threads

To enable applications to scale while remaining harmonious with the platform, we should strive to preserve the thread-per-request style. We can do this by implementing threads more efficiently, so they can be more plentiful. Operating systems cannot implement OS threads more efficiently because different languages and runtimes use the thread stack in different ways. It is possible, however, for a Java runtime to implement Java threads in a way that severs their one-to-one correspondence to OS threads. Just as operating systems give the illusion of plentiful memory by mapping a large virtual address space to a limited amount of physical RAM, a Java runtime can give the illusion of plentiful threads by mapping a large number of *virtual* threads to a small number of OS threads.

A *virtual thread* is an instance of `java.lang.Thread` that is not tied to a particular OS thread. A *platform thread*, by contrast, is an instance of `java.lang.Thread` implemented in the traditional way, as a thin wrapper around an OS thread.

Application code in the thread-per-request style can run in a virtual thread for the entire duration of a request, but the virtual thread consumes an OS thread only while it performs calculations on the CPU. The result is the same scalability as the asynchronous style, except it is achieved transparently: When code running in a virtual thread calls a blocking I/O operation in the `java.*` API, the runtime performs a non-blocking OS call and automatically suspends the virtual thread until it can be resumed later. To Java developers, virtual threads are simply threads that are cheap to create and almost infinitely plentiful. Hardware utilization is close to optimal, allowing a high level of concurrency and, as a result, high throughput, while the application remains harmonious with the multithreaded design of the Java Platform and its tooling.

Implications of virtual threads

Virtual threads are cheap and plentiful, and thus should never be pooled: A new virtual thread should be created for every application task. Most virtual threads will thus be short-lived and have shallow call stacks, performing as little as a single HTTP client call or a single JDBC query. Platform threads, by contrast, are heavyweight and expensive, and thus often must be pooled. They tend to be long-lived, have deep call stacks, and be shared among many tasks.

In summary, virtual threads preserve the reliable thread-per-request style that is harmonious with the design of the Java Platform while utilizing the available hardware optimally. Using virtual threads does not require learning new concepts, though it may require unlearning habits developed to cope with today's high cost of threads. Virtual threads will not only help application developers — they will also help framework designers provide easy-to-use APIs that are compatible with the platform's design without compromising on scalability.

Description

Today, every instance of `java.lang.Thread` in the JDK is a *platform thread*. A platform thread runs Java code on an underlying OS thread and captures the OS thread for the code's entire lifetime. The number of platform threads is limited to the number of OS threads.

A *virtual thread* is an instance of `java.lang.Thread` that runs Java code on an underlying OS thread but does not capture the OS thread for the code's entire lifetime. This means that many virtual threads can run their Java code on the same OS thread, effectively sharing it. While a platform thread monopolizes a precious OS thread, a virtual thread does not. The number of virtual threads can be much larger than the number of OS threads.

Virtual threads are a lightweight implementation of threads that is provided by the JDK rather than the OS. They are a form of *user-mode threads*, which have been successful in other multithreaded languages (e.g., goroutines in Go and processes in Erlang). User-mode threads even featured as so-called "green threads" in early versions of Java, when OS threads were not yet mature and widespread. However, Java's green threads all shared one OS thread (M:1 scheduling) and were eventually outperformed by platform threads, implemented as wrappers for OS threads (1:1 scheduling). Virtual threads employ M:N scheduling, where a large number (M) of virtual threads is scheduled to run on a smaller number (N) of OS threads.

Using virtual threads vs. platform threads

Developers can choose whether to use virtual threads or platform threads. Here is an example program that creates a large number of virtual threads. The program first obtains an `ExecutorService` that will create a new virtual thread for each submitted task. It then submits 10,000 tasks and waits for all of them to complete:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
}  // executor.close() is called implicitly, and waits
```

The task in this example is simple code — sleep for one second — and modern hardware can easily support 10,000 virtual threads running such code concurrently. Behind the scenes, the JDK runs the code on a small number of OS threads, perhaps as few as one.

Things would be very different if this program used an `ExecutorService` that creates a new platform thread for each task, such as `Executors.newCachedThreadPool()`. The `ExecutorService` would attempt to create 10,000 platform threads, and thus 10,000 OS threads, and the program might crash, depending on the machine and operating system.

Things would be not much better if the program, instead, used an `ExecutorService` that obtains platform threads from a pool, such as `Executors.newFixedThreadPool(200)`. The `ExecutorService` would create 200 platform threads to be shared by all 10,000 tasks, so many of the tasks would run sequentially rather than concurrently and the program would take a long time to complete. For this program, a pool with 200 platform threads can only achieve a throughput of 200 tasks-per-second, whereas virtual threads achieve a throughput of about 10,000 tasks-per-second (after sufficient warmup). Moreover, if the `10_000` in the example program is changed to `1_000_000`, then the program would submit 1,000,000 tasks, create 1,000,000 virtual threads that run concurrently, and (after sufficient warmup) achieve a throughput of about 1,000,000 tasks-per-second.

If the tasks in this program performed a calculation for one second (e.g., sorting a huge array), rather than merely sleeping, then increasing the number of threads beyond the number of processor cores would not help, whether they are virtual threads or platform threads. Virtual threads are not faster threads — they do not run code any faster than platform threads. They exist to provide scale (higher throughput), not speed (lower latency). There can be many more of them than platform threads, so they enable the higher concurrency needed for higher throughput according to Little's Law.

To put it another way, virtual threads can significantly improve application throughput when

- The number of concurrent tasks is high (more than a few thousand), and
- The workload is not CPU-bound, since having many more threads than processor cores cannot improve throughput in that case.

Virtual threads help to improve the throughput of typical server applications precisely because such applications consist of a great number of concurrent tasks that spend much of their time waiting.

A virtual thread can run any code that a platform thread can run. In particular, virtual threads support thread-local variables and thread interruption, just like platform threads. This means that existing Java code that processes requests will easily run in a virtual thread. Many server frameworks will choose to do this automatically, starting a new virtual thread for every incoming request and running the application's business logic in it.

Here is an example of a server application that aggregates the results of two other services. A hypothetical server framework (not shown) creates a new virtual thread for each request and runs the application's `handle` code in that virtual thread. The application code, in turn, creates two new virtual threads to fetch resources concurrently via the same `ExecutorService` as the first example:

```
void handle(Request request, Response response) {
    var url1 = ...
    var url2 = ...

    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        var future1 = executor.submit(() -> fetchURL(url1));
        var future2 = executor.submit(() -> fetchURL(url2));
        response.send(future1.get() + future2.get());
    } catch (ExecutionException | InterruptedException e) {
        response.fail(e);
    }
}

String fetchURL(URL url) throws IOException {
    try (var in = url.openStream()) {
        return new String(in.readAllBytes(), StandardCharsets.UTF_8);
    }
}
```

A server application like this, with straightforward blocking code, scales well because it can employ a large number of virtual threads.

`Executor.newVirtualThreadPerTaskExecutor()` is not the only way to create virtual threads. The new `java.lang.Thread.Builder` API, discussed below, can create and start virtual threads. In addition, structured concurrency offers a more powerful API to create and manage virtual threads, particularly in code similar to this server example, whereby the relationships among threads are made known to the platform and its tools.

Do not pool virtual threads

Developers will typically migrate application code to the virtual-thread-per-task `ExecutorService` from a traditional thread-pool based `ExecutorService`. A thread pool, like any resource pool, is intended to share expensive resources, but virtual threads are not expensive so there is never a need to pool them.

Developers sometimes use thread pools to limit concurrent access to limited resources. For example, if a service cannot handle more than 20 concurrent requests then making all requests to the service via tasks submitted to a thread pool of size 20 will ensure that. This idiom has become ubiquitous because the high cost of platform threads has made thread pools ubiquitous, but do not be tempted to pool virtual threads in order to limit concurrency. Instead use constructs specifically designed for that purpose, such as semaphores.

In conjunction with thread pools, developers sometimes use thread-local variables to share expensive resources among multiple tasks that share the same thread. For example, if a database connection is expensive to create then you can open it once and store it in a thread-local variable for later use by other tasks in the same thread. If you migrate code from using a thread pool to using a virtual thread per task, be wary of usages of this idiom since creating an expensive resource for every virtual thread may degrade performance significantly. Change such code to use alternative caching strategies so that expensive resources can be shared efficiently among a very large number of virtual threads.

Observing virtual threads

Writing clear code is not the full story. A clear presentation of the state of a running program is also essential for troubleshooting, maintenance, and optimization, and the JDK has long offered mechanisms to debug, profile, and monitor threads. Such tools should do the same for virtual threads — perhaps with some accommodation to their large quantity — since they are, after all, instances of `java.lang.Thread`.

Java debuggers can step through virtual threads, show call stacks, and inspect variables in stack frames. JDK Flight Recorder (JFR), which is the JDK's low-overhead profiling and monitoring mechanism, can associate events from application code (such as object allocation and I/O operations) with the correct virtual thread. These tools cannot do these things for applications written in the asynchronous style. In that style tasks are not related to threads, so debuggers cannot display or manipulate the state of a task, and profilers cannot tell how much time a task spends waiting for I/O.

The thread dump is another popular tool for troubleshooting applications written in the thread-per-request style. Unfortunately the JDK's traditional thread dump, obtained with `jstack` or `jcmd`, presents a flat list of threads. This is suitable for dozens or hundreds of platform threads, but is unsuitable for thousands or millions of virtual threads. Accordingly, we will not extend traditional thread dumps to include virtual threads; we will, rather, introduce a new kind of thread dump in `jcmd` to present virtual threads alongside platform threads, all grouped in a meaningful way. Richer relationships among threads can be shown when programs use structured concurrency.

Because visualizing and analyzing a great many threads can benefit from tooling, `jcmd` can emit the new thread dump in JSON format in addition to plain text:

```
$ jcmd <pid> Thread.dump_to_file -format=json <file>
```

The new thread dump format does not include object addresses, locks, JNI statistics, heap statistics, and other information that appears in traditional thread dumps. Moreover, because it might need to list a great many threads, generating a new thread dump does not pause the application.

If the system property `jdk.trackAllThreads` is set to `false`, i.e. with the `-Djdk.trackAllThreads=false` command-line option, virtual threads created directly with the `Thread.Builder` API will not always be tracked by the runtime and may not appear in the new thread dump. In that case, the new thread dump would list virtual threads that are blocked in network I/O operations, and virtual threads that are created by the new-thread-per-task `ExecutorService` shown above.

Here is an example of such a thread dump, taken from an application similar to the second example above, rendered in a JSON viewer (click to enlarge):

Since virtual threads are implemented in the JDK and are not tied to any particular OS thread, they are invisible to the OS, which is unaware of their existence. OS-level monitoring will observe that a JDK process uses fewer OS threads than there are virtual threads.

Scheduling virtual threads

To do useful work a thread needs to be scheduled, that is, assigned for execution on a processor core. For platform threads, which are implemented as OS threads, the JDK relies on the scheduler in the OS. For virtual threads, by contrast, the JDK has its own scheduler. Rather than assigning virtual threads to processors directly, the JDK's scheduler assigns virtual threads to platform threads (this is the M:N scheduling of virtual threads mentioned earlier). The platform threads are then scheduled by the OS as usual.

The JDK's virtual thread scheduler is a work-stealing `ForkJoinPool` that operates in FIFO mode. The *parallelism* of the scheduler is the number of platform threads available for the purpose of scheduling virtual threads. By default it is equal to the number of available processors, but it can be tuned with the system property `jdk.virtualThreadScheduler.parallelism`. This `ForkJoinPool` is distinct from the common pool which is used, for example, in the implementation of parallel streams, and which operates in LIFO mode.

The platform thread to which the scheduler assigns a virtual thread is called the virtual thread's *carrier*. A virtual thread can be scheduled on different carriers over the course of its lifetime; in other words, the scheduler does not maintain *affinity* between a virtual thread and any particular platform thread. From the perspective of Java code, a running virtual thread is logically independent of its current carrier:

- The identity of the carrier is unavailable to the virtual thread. The value returned by `Thread.currentThread()` is always the virtual thread itself.
- The stack traces of the carrier and the virtual thread are separate. An exception thrown in the virtual thread will not include the carrier's stack frames. Thread dumps will not show the carrier's stack frames in the virtual thread's stack, and vice-versa.
- Thread-local variables of the carrier are unavailable to the virtual thread, and vice-versa.

In addition, from the perspective of Java code, the fact that a virtual thread and its carrier temporarily share an OS thread is invisible. From the perspective of native code, by contrast, both the virtual thread and its carrier run on the same native thread. Native code that is called multiple times on the same virtual thread may thus observe a different OS thread identifier at each invocation.

The scheduler does not currently implement *time sharing* for virtual threads. Time sharing is the forceful preemption of a thread that has consumed an allotted quantity of CPU time. While time sharing can be effective at reducing the latency of some tasks when there are a relatively small number of platform threads and CPU utilization is at 100%, it is not clear that time sharing would be as effective with a million virtual threads.

Executing virtual threads

To take advantage of virtual threads, it is not necessary to rewrite your program. Virtual threads do not require or expect application code to explicitly hand control back to the scheduler; in other words, virtual threads are not *cooperative*. User code must not make assumptions about how or when virtual threads are assigned to platform threads any more than it makes assumptions about how or when platform threads are assigned to processor cores.

To run code in a virtual thread, the JDK's virtual thread scheduler assigns the virtual thread for execution on a platform thread by *mounting* the virtual thread on a platform thread. This makes the platform thread become the carrier of the virtual thread. Later, after running some code, the virtual thread can *unmount* from its carrier. At that point the platform thread is free so the scheduler can mount a different virtual thread on it, thereby making it a carrier again.

Typically, a virtual thread will unmount when it blocks on I/O or some other blocking operation in the JDK, such as `BlockingQueue.take()`. When the blocking operation is ready to complete (e.g., bytes have been received on a socket), it submits the virtual thread back to the scheduler, which will mount the virtual thread on a carrier to resume execution.

The mounting and unmounting of virtual threads happens frequently and transparently, and without blocking any OS threads. For example, the server application shown earlier included the following line of code, which contains calls to blocking operations:

```
response.send(future1.get() + future2.get());
```

These operations will cause the virtual thread to mount and unmount multiple times, typically once for each call to `get()` and possibly multiple times in the course of performing I/O in `send(...)`.

The vast majority of blocking operations in the JDK will unmount the virtual thread, freeing its carrier and the underlying OS thread to take on new work. However, some blocking operations in the JDK do not unmount the virtual thread, and thus block both its carrier and the underlying OS thread. This is because of limitations at either the OS level (e.g., many filesystem operations) or the JDK level (e.g., `Object.wait()`). The implementations of these blocking operations compensate for the capture of the OS thread by temporarily expanding the parallelism of the scheduler. Consequently, the number of platform threads in the scheduler's `ForkJoinPool` may temporarily exceed the number of available processors. The maximum number of platform threads available to the scheduler can be tuned with the system property `jdk.virtualThreadScheduler.maxPoolSize`.

There are two scenarios in which a virtual thread cannot be unmounted during blocking operations because it is *pinned* to its carrier:

1. When it executes code inside a `synchronized` block or method, or
2. When it executes a `native` method or a foreign function.

Pinning does not make an application incorrect, but it might hinder its scalability. If a virtual thread performs a blocking operation such as I/O or `BlockingQueue.take()` while it is pinned, then its carrier and the underlying OS thread are blocked for the duration of the operation. Frequent pinning for long durations can harm the scalability of an application by capturing carriers.

The scheduler does not compensate for pinning by expanding its parallelism. Instead, avoid frequent and long-lived pinning by revising `synchronized` blocks or methods that run frequently and guard potentially long I/O operations to use `java.util.concurrent.locks.ReentrantLock` instead. 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. As always, strive to keep locking policies simple and clear.

New diagnostics assist in migrating code to virtual threads and in assessing whether you should replace a particular use of `synchronized` with a `java.util.concurrent` lock:

- A JDK Flight Recorder (JFR) event is emitted when a thread blocks while pinned (see JDK Flight Recorder).
- The system property `jdk.tracePinnedThreads` triggers a stack trace when a thread blocks while pinned. Running with `-Djdk.tracePinnedThreads=full` prints a complete stack trace when a thread blocks while pinned, highlighting native frames and frames holding monitors. Running with `-Djdk.tracePinnedThreads=short` limits the output to just the problematic frames.

In a future release we may be able to remove the first limitation above, namely pinning inside `synchronized`. The second limitation is required for proper interaction with native code.

Memory use and interaction with garbage collection

The stacks of virtual threads are stored in Java's garbage-collected heap as *stack chunk* objects. The stacks grow and shrink as the application runs, both to be memory-efficient and to accommodate stacks of depth up to the JVM's configured platform thread stack size. This efficiency is what enables a large number of virtual threads and thus the continued viability of the thread-per-request style in server applications.

In the second example above, recall that a hypothetical framework processes each request by creating a new virtual thread and calling the `handle` method. Even if it calls `handle` at the end of a deep call stack (after authentication, transactions, etc.), `handle` itself spawns multiple virtual threads that only perform short-lived tasks. Therefore, for each virtual thread with a deep call stack, there will be multiple virtual threads with shallow call stacks consuming little memory.

The amount of heap space and garbage collector activity that virtual threads require is difficult, in general, to compare to that of asynchronous code. A million virtual threads require at least a million objects, but so do a million tasks sharing a pool of platform threads. In addition, application code that processes requests typically maintains data across I/O operations. Thread-per-request code can keep that data in local variables, which are stored on virtual thread stacks in the heap, while asynchronous code must keep that same data in heap objects that are passed from one stage of the pipeline to the next. On the one hand, the stack frame layout needed by virtual threads is more wasteful than that of a compact object; on the other hand, virtual threads can mutate and reuse their stacks in many situations (depending on low-level GC interactions) while asynchronous pipelines always need to allocate new objects, and so virtual threads might require fewer allocations. Overall, the heap consumption and garbage collector activity of thread-per-request versus asynchronous code should be roughly similar. Over time, we expect to make the internal representation of virtual thread stacks significantly more compact.

Unlike platform thread stacks, virtual thread stacks are not GC roots. Thus the references they contain are not traversed in a stop-the-world pause by garbage collectors, such as G1, that perform concurrent heap scanning. This also means that if a virtual thread is blocked on, e.g., `BlockingQueue.take()`, and no other thread can obtain a reference to either the virtual thread or the queue, then the thread can be garbage collected — which is fine, since the virtual thread can never be interrupted or unblocked. Of course, the virtual thread will not be garbage collected if it is running or if it is blocked and could ever be unblocked.

A current limitation of virtual threads is that the G1 GC does not support *humongous* stack chunk objects. If a virtual thread's stack reaches half the region size, which could be as small as 512KB, then a `StackOverflowError` might be thrown.

Detailed changes

The remaining subsections describe, in detail, the changes we propose across the Java Platform and its implementation:

- `java.lang.Thread`
- Thread-local variables
- `java.util.concurrent`
- Networking
- `java.io`
- Java Native Interface (JNI)
- Debugging (JVM TI, JDWP, and JDI)
- JDK Flight Recorder (JFR)
- Java Management Extensions (JMX)

`java.lang.Thread`

We update the `java.lang.Thread` API as follows:

- `Thread.Builder`, `Thread.ofVirtual()`, and `Thread.ofPlatform()` are new APIs to create virtual and platform threads. For example,

  ```
  Thread thread = Thread.ofVirtual().name("duke").unstarted(runnable);
  ```

  creates a new unstarted virtual thread named "duke".

- `Thread.startVirtualThread(Runnable)` is a convenient way to create and then start a virtual thread.

- A `Thread.Builder` can create either a thread or a `ThreadFactory`, which can then create multiple threads with identical properties.

- `Thread.isVirtual()` tests whether a thread is a virtual thread.

- `Thread.getAllStackTraces()` now returns a map of all platform threads rather than all threads.

The `java.lang.Thread` API is otherwise unchanged by this JEP. The constructors defined by the `Thread` class create platform threads, as before. There are no new public constructors.

(Three methods in `Thread` which throw `UnsupportedOperationException` for virtual threads — `stop()`, `suspend()`, and `resume()` — were changed in JDK 20 to throw `UnsupportedOperationException` for platform threads too.)

The main API differences between virtual and platforms threads are:

- The public `Thread` constructors cannot create virtual threads.

- Virtual threads are always daemon threads. The `Thread.setDaemon(boolean)` method cannot change a virtual thread to be a non-daemon thread.

- Virtual threads have a fixed priority of `Thread.NORM_PRIORITY`. The `Thread.setPriority(int)` method has no effect on virtual threads. This limitation may be revisited in a future release.

- 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"`. The `Thread.Builder` API does not define a method to set the thread group of a virtual thread.

- Virtual threads have no permissions when running with a `SecurityManager` set.

Thread-local variables

Virtual threads support thread-local variables (`ThreadLocal`) and inheritable thread-local variables (`InheritableThreadLocal`), just like platform threads, so they can run existing code that uses thread locals. However, because virtual threads can be very numerous, use thread locals only after careful consideration. In particular, do not use thread locals to pool costly resources among multiple tasks sharing the same thread in a thread pool. Virtual threads should never be pooled since each is intended to run only a single task over its lifetime. We have removed many uses of thread locals from the JDK's `java.base` module in preparation for virtual threads in order to reduce memory footprint when running with millions of threads.

The system property `jdk.traceVirtualThreadLocals` can be used to trigger a stack trace when a virtual thread sets the value of any thread-local variable. This diagnostic output may assist with removing thread locals when migrating code to use virtual threads. Set the system property to `true` to trigger stack traces; the default value is `false`.

Scoped values (JEP 429) may prove to be a better alternative to thread locals for some use cases.

`java.util.concurrent`

The primitive API to support locking, `java.util.concurrent.LockSupport`, now supports virtual threads: Parking a virtual thread releases the underlying platform thread to do other work, and unparking a virtual thread schedules it to continue. This change to `LockSupport` enables all APIs that use it (`Lock`s, `Semaphore`s, blocking queues, etc.) to park gracefully when invoked in virtual threads.

Additionally, `Executors.newThreadPerTaskExecutor(ThreadFactory)` and `Executors.newVirtualThreadPerTaskExecutor()` create an `ExecutorService` that creates a new thread for each task. These methods enable migration and interoperability with existing code that uses thread pools and `ExecutorService`.

Networking

The implementations of the networking APIs in the `java.net` and `java.nio.channels` packages now work with virtual threads: An operation on a virtual thread that blocks, e.g., to establish a network connection or read from a socket, releases the underlying platform thread to do other work.

To allow for interruption and cancellation, the blocking I/O methods defined by `java.net.Socket`, `ServerSocket`, and `DatagramSocket` are now specified to be interruptible when invoked in a virtual thread: Interrupting a virtual thread blocked on a socket will unpark the thread and close the socket. Blocking I/O operations on these types of sockets when obtained from an `InterruptibleChannel` have always been interruptible, so this change aligns the behavior of these APIs when created with their constructors with their behavior when obtained from a channel.

`java.io`

The `java.io` package provides APIs for streams of bytes and characters. The implementations of these APIs are heavily synchronized and require changes to avoid pinning when they are used in virtual threads.

As background, the byte-oriented input/output streams are not specified to be thread-safe and do not specify the expected behavior when `close()` is invoked while a thread is blocked in a read or write method. In most scenarios it does not make sense to use a particular input or output stream from multiple concurrent threads. The character-oriented reader/writers are also not specified to be thread-safe, but they do expose a lock object for sub-classes. Aside from pinning, the synchronization in these classes is problematic and inconsistent; e.g., the stream decoders and encoders used by `InputStreamReader` and `OutputStreamWriter` synchronize on the stream object rather than the lock object.

To prevent pinning, the implementations now work as follows:

- `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, `BufferedWriter`, `PrintStream`, and `PrintWriter` now use an explicit lock rather than a monitor when used directly. These classes synchronize as before when they are sub-classed.
- The stream decoders and encoders used by `InputStreamReader` and `OutputStreamWriter` now use the same lock as the enclosing `InputStreamReader` or `OutputStreamWriter`.

Going further and eliminating all this often-needless locking is beyond the scope of this JEP.

Additionally, the initial sizes of the buffers used by `BufferedOutputStream`, `BufferedWriter`, and the stream encoder for `OutputStreamWriter` are now smaller so as to reduce memory usage when there are many streams or writers in the heap — as might arise if there are a million virtual threads, each with a buffered stream on a socket connection.

Java Native Interface (JNI)

JNI defines one new function, `IsVirtualThread`, to test if an object is a virtual thread.

The JNI specification is otherwise unchanged.

Debugging

The debugging architecture consists of three interfaces: the JVM Tool Interface (JVM TI), the Java Debug Wire Protocol (JDWP), and the Java Debug Interface (JDI). All three interfaces now support virtual threads.

The updates to JVM TI are:

- Most functions that are called with a `jthread` (i.e., a JNI reference to a `Thread` object) can be called with a reference to a virtual thread. A small number of functions, namely `AgentStartFunction`, `PopFrame`, `ForceEarlyReturn*`, `StopThread`, and `GetThreadCpuTime`, are not supported or are optionally supported on virtual threads. The `SetLocal*` functions are limited to setting local variables in the top-most frames of virtual threads that are suspended at a breakpoint or single step event.
- The `GetAllThreads` and `GetAllStackTraces` functions are now specified to return all platform threads rather than all threads.
- All events, with the exception of those posted during early VM startup or during heap iteration, can have event callbacks invoked in the context of a virtual thread.
- The suspend/resume implementation allows virtual threads to be suspended and resumed by debuggers, and it allows platform threads to be suspended when a virtual thread is mounted.
- A new capability, `can_support_virtual_threads`, gives agents finer control over thread start and end events for virtual threads.
- New functions support the bulk suspension and resumption of virtual threads; these require the `can_support_virtual_threads` capability.

Existing JVM TI agents will mostly work as before, but may encounter errors if they invoke functions that are not supported on virtual threads. These will arise when an agent that is unaware of virtual threads is used with an application that uses virtual threads. The change to `GetAllThreads` to return an array containing only the platform threads may be an issue for some agents. Existing agents that enable the `ThreadStart` and `ThreadEnd` events may encounter performance issues since they lack the ability to limit these events to platform threads.

The updates to JDWP are:

- A new command allows debuggers to test if a thread is a virtual thread.
- A new modifier on the `EventRequest` command allows debuggers to restrict thread start and end events to platform threads.

The updates to JDI are:

- A new method in `com.sun.jdi.ThreadReference` tests if a thread is a virtual thread.
- New methods in `com.sun.jdi.request.ThreadStartRequest` and `com.sun.jdi.request.ThreadDeathRequest` limit the events generated for the request to platform threads.

As noted above, virtual threads are not considered to be active threads in a thread group. Consequently the thread lists returned by the JVM TI function `GetThreadGroupChildren`, the JDWP command `ThreadGroupReference/Children`, and the JDI method `com.sun.jdi.ThreadGroupReference.threads()` include only platform threads.

JDK Flight Recorder (JFR)

JFR supports virtual threads with several new events:

- `jdk.VirtualThreadStart` and `jdk.VirtualThreadEnd` indicate virtual thread start and end. These events are disabled by default.
- `jdk.VirtualThreadPinned` indicates that a virtual thread was parked while pinned, i.e., without releasing its platform thread (see above). This event is enabled by default, with a threshold of 20ms.
- `jdk.VirtualThreadSubmitFailed` indicates that starting or unparking a virtual thread failed, probably due to a resource issue. This event is enabled by default.

Java Management Extensions (JMX)

`java.lang.management.ThreadMXBean` only supports the monitoring and management of platform threads. The `findDeadlockedThreads()` method finds cycles of platform threads that are in deadlock; it does not find cycles of virtual threads that are in deadlock.

A new method in `com.sun.management.HotSpotDiagnosticsMXBean` generates the new-style thread dump described above. This method can also be invoked indirectly via the platform `MBeanServer` from a local or remote JMX tool.

Alternatives

- Continue to rely on asynchronous APIs. Asynchronous APIs are difficult to integrate with synchronous APIs, create a split world of two representations of the same I/O operations, and provide no unified concept of a sequence of operations that can be used by the platform as context for troubleshooting, monitoring, debugging, and profiling.
- Add *syntactic stackless coroutines* (i.e., async/await) to the Java language. These are easier to implement than user-mode threads and would provide a unifying construct representing the context of a sequence of operations.

  That construct would be new, however, and separate from threads, similar to them in many respects yet different in some nuanced ways. It would split the world between APIs designed for threads and APIs designed for coroutines, and would require the new thread-like construct to be introduced into all layers of the platform and its tooling. This would take longer for the ecosystem to adopt, and would not be as elegant and harmonious with the platform as user-mode threads.

Most languages that have adopted syntactic coroutines have done so due to an inability to implement user-mode threads (e.g., Kotlin), legacy semantic guarantees (e.g., the inherently single-threaded JavaScript), or language-specific technical constraints (e.g., C++). These limitations do not apply to Java.

- Introduce a new public class to represent user-mode threads, unrelated to `java.lang.Thread`. This would be an opportunity to jettison the unwanted baggage that the `Thread` class has accumulated over 25 years. We explored and prototyped several variants of this approach, but in every case grappled with the issue of how to run existing code.

  The main problem is that `Thread.currentThread()` is used, directly or indirectly, pervasively in existing code (e.g., in determining lock ownership, or for thread-local variables). This method must return an object that represents the current thread of execution. If we introduced a new class to represent user-mode threads then `currentThread()` would have to return some sort of wrapper object that looks like a `Thread` but delegates to the user-mode thread object.

  It would be confusing to have two objects represent the current thread of execution, so we eventually concluded that preserving the old `Thread` API is not a significant hurdle. With the exception of a few methods such as `currentThread()`, developers rarely use the `Thread` API directly; they mostly interact use higher-level APIs such as `ExecutorService`. Over time we will jettison unwanted baggage from the `Thread` class, and associated classes such as `ThreadGroup`, by deprecating and removing obsolete methods.

Testing

- Existing tests will ensure that the changes we propose here do not cause unexpected regressions in the multitude of configurations and execution modes in which they are run.
- We will extend the `jtreg` test harness to allow existing tests to be run in the context of a virtual thread. This will avoid needing two versions of many tests.
- New tests will exercise all new and revised APIs, and all areas changed to support virtual threads.
- New stress tests will target areas that are critical to reliability and performance.
- New microbenchmarks will target performance-critical areas.
- We will use a number of existing servers, including Helidon and Jetty, for larger-scale testing.

Risks and Assumptions

The primary risks of this proposal are ones of compatibility due to changes in existing APIs and their implementations:

- The revisions to the internal (and undocumented) locking protocol used in the `java.io.BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, `BufferedWriter`, `PrintStream`, and `PrintWriter` classes may impact code that assumes that I/O methods synchronize on the stream upon which they are invoked. These changes do not impact code that extends these classes and assumes locking by the superclass, nor do they impact code that extends `java.io.Reader` or `java.io.Writer` and uses the lock object exposed by those APIs.

A few source and binary incompatible changes may impact code that extends `java.lang.Thread`:

- `Thread` defines several new methods. If code in an existing source file extends `Thread` and a method in the subclass conflicts with any of the new `Thread` methods then the file will not compile without change.
- `Thread.Builder` is a new nested interface. If code in an existing source file extends `Thread`, imports a class named `Builder`, and code in the subclass references `Builder` as a simple name, then the file will not compile without change.
- `Thread.isVirtual()` is a new final method. If there is existing compiled code that extends `Thread` and the subclass declares a method with the same name and return type then an `IncompatibleClassChangeError` will be thrown at run-time if the subclass is loaded.

A few behavioral differences between platform threads and virtual threads may be observed when mixing existing code with newer code that takes advantage of virtual threads or the new APIs:

- The `Thread.setPriority(int)` method has no effect on virtual threads, which always have a priority of `Thread.NORM_PRIORITY`.
- The `Thread.setDaemon(boolean)` method has no effect on virtual threads, which are always daemon threads.

- `Thread.getAllStackTraces()` now returns a map of all platform threads rather than a map of all threads.
- The blocking I/O methods defined by `java.net.Socket`, `ServerSocket`, and `DatagramSocket` are now interruptible when invoked in the context a virtual thread. Existing code could break when a thread blocked on a socket operation is interrupted, which will wake the thread and close the socket.
- Virtual threads are not active members of a `ThreadGroup`. Invoking `Thread.getThreadGroup()` on a virtual thread returns a dummy `"VirtualThreads"` group that is empty.
- Virtual threads have no permissions when running with a security manager set. See JEP 411 (Deprecate the Security Manager for Removal) for information about running with a security manager on Java 17 and later.
- In JVM TI, the `GetAllThreads` and `GetAllStackTraces` functions do not return virtual threads. Existing agents that enable the `ThreadStart` and `ThreadEnd` events may encounter performance issues since they lack the ability to limit the events to platform threads.
- The `java.lang.management.ThreadMXBean` API supports the monitoring and management of platform threads, but not virtual threads.
- The `-XX:+PreserveFramePointer` flag has a drastic negative impact on virtual thread performance.

Dependencies

- JEP 416 (Reimplement Core Reflection with Method Handles) in JDK 18 removed the VM-native reflection implementation. This allows virtual threads to park gracefully when methods are invoked reflectively.
- JEP 353 (Reimplement the Legacy Socket API) in JDK 13, and JEP 373 (Reimplement the Legacy DatagramSocket API) in JDK 15, replaced the implementations of `java.net.Socket`, `ServerSocket`, and `DatagramSocket` with new implementations designed for use with virtual threads.
- JEP 418 (Internet-Address Resolution SPI) in JDK 18 defined a service-provider interface for host name and address lookup. This will allow third-party libraries to implement alternative `java.net.InetAddress` resolvers that do not pin threads during host lookup.