Project Loom: Fibers and Continuations for the Java Virtual Machine

Overview

Project Loom's mission is to make it easier to write, debug, profile and maintain concurrent applications meeting today's requirements. Threads, provided by Java from its first day, are a natural and convenient concurrency construct (putting aside the separate question of communication among threads) which is being supplanted by less convenient abstractions because their current implementation as OS kernel threads is insufficient for meeting modern demands, and wasteful in computing resources that are particularly valuable in the cloud. Project Loom will introduce fibers as lightweight, efficient threads managed by the Java Virtual Machine, that let developers use the same simple abstraction but with better performance and lower footprint. We want to make concurrency simple(r) again! A fiber is made of two components — a continuation and a scheduler. As Java already has an excellent scheduler in the form of `ForkJoinPool`, fibers will be implemented by adding continuations to the JVM.

Motivation

Many applications written for the Java Virtual Machine are concurrent — meaning, programs like servers and databases, that are required to serve many requests, occurring concurrently and competing for computational resources. Project Loom is intended to significantly reduce the difficulty of writing efficient concurrent applications, or, more precisely, to eliminate the tradeoff between simplicity and efficiency in writing concurrent programs.

One of Java's most important contributions when it was first released, over twenty years ago, was the easy access to threads and synchronization primitives. Java threads (either used directly, or indirectly through, for example, Java servlets processing HTTP requests) provided a relatively simple abstraction for writing concurrent applications. These days, however, one of the main difficulties in writing concurrent programs that meet today's requirements is that the software unit of concurrency offered by the runtime — the thread — cannot match the scale of the domain's unit of concurrency, be it a user, a transaction or even a single operation. Even if the unit of application concurrency is coarse — say, a session, represented by single socket connection — a server can handle upward of a million concurrent open sockets, yet the Java runtime, which uses the operating system's threads for its implementation of Java threads, cannot efficiently handle more than a few thousand. A mismatch in several orders of magnitude has a big impact.

Programmers are forced to choose between modeling a unit of domain concurrency directly as a thread and lose considerably in the scale of concurrency a single server can support, or use other constructs to implement concurrency on a finer-grained level than threads (tasks), and support concurrency by writing *asynchronous* code that does not block the thread running it.

Recent years have seen the introduction of many asynchronous APIs to the Java ecosystem, from asynchronous NIO in the JDK, asynchronous servlets, and many asynchronous third-party libraries. Those APIs were created not because they are easier to write and to understand, for they are actually harder; not because they are easier to debug or profile — they are harder (they don't even produce meaningful stacktraces); not because they compose better than synchronous APIs — they compose less elegantly; not because they fit better with the rest of the language or integrate well with existing code — they are a much worse fit, but just because the implementation of the software unit of concurrency in Java — the thread — is insufficient from a footprint and performance perspective. This is a sad case of a good and natural abstraction being abandoned in favor of a less

natural one, which is overall worse in many respects, merely because of the runtime performance characteristics of the abstraction.

While there are some advantages to using kernel threads as the implementation of Java threads — most notably because all native code is supported by kernel threads, and so Java code running in a thread can call native APIs — the disadvantages mentioned above are too great to ignore, and result either in hard-to-write, expensive-to-maintain code, or in a significant waste of computing resources, that is especially costly when code runs in the cloud. Indeed, some languages and language runtimes successfully provide a lightweight thread implementation, most famous are Erlang and Go, and the feature is both very useful and popular.

The main goal of this project is to add a lightweight thread construct, which we call fibers, managed by the Java runtime, which would be optionally used alongside the existing heavyweight, OS-provided, implementation of threads. Fibers are much more lightweight than kernel threads in terms of memory footprint, and the overhead of task-switching among them is close to zero. Millions of fibers can be spawned in a single JVM instance, and programmers need not hesitate to issue synchronous, blocking calls, as blocking will be virtually free. In addition to making concurrent applications simpler and/or more scalable, this will make life easier for library authors, as there will no longer be a need to provide both synchronous and asynchronous APIs for a different simplicity/performance tradeoff. Simplicity will come with no tradeoff.

As we will see, a thread is not an atomic construct, but a composition of two concerns — a scheduler and a *continuation*. It is our current intention to separate the two concerns, and implement Java fibers on top of those two building blocks, and, although fibers are the main motivation for this project, to also add continuations as a user facing abstraction, as continuations have other uses, too (e.g. Python's generators).

Goals and Scope

Fibers can provide a low-level primitive upon which interesting programming paradigms can be implemented, like channels, actors and dataflow, but while those uses will be taken into account, it is *not* the goal of this project to design any of those higher level constructs, nor to suggest new programming styles or recommended patterns for the exchange information among fibers (e.g. shared memory vs. message passing). As the issue of limiting memory access for threads is the subject of other OpenJDK projects, and as this issue applies to any implementation of the thread abstraction, be it heavyweight or lightweight, this project will probably intersect with others.

It *is* the goal of this project to add a lightweight thread construct — fibers — to the Java platform. What user-facing form this construct may take will be discussed below. The goal is to allow *most* Java code (meaning, code in Java class files, not necessarily written in the Java programming language) to run inside fibers unmodified, or with minimal modifications. It is *not* a requirement of this project to allow native code called from Java code to run in fibers, although this *may* be possible in some circumstances. It is also *not* the goal of this project to ensure that *every* piece of code would enjoy performance benefits when run in fibers; in fact, some code that is less appropriate for lightweight threads may suffer in performance when run in fibers.

It *is* the goal of this project to add a public *delimited continuation* (or *coroutine*) construct to the Java platform. However, this goal is secondary to fibers (which require continuations, as explained later, but those continuations need not necessarily be exposed as a public API).

It *is* the goal of this project to experiment with various *schedulers* for fibers, but it is *not* the intention of this project to conduct any serious research in scheduler design, largely because we think that `ForkJoinPool` can serve as a very good fiber scheduler.

As adding the ability to manipulate call stacks to the JVM will undoubtedly be required, it is *also* the goal of this project to add an even lighter-weight construct that will allow unwinding the stack to some point and then invoke a method with given arguments (basically, a generalization of efficient tail-calls). We will call that feature *unwind-and-invoke*, or UAI. It is *not* the goal of this project to add an automatic tail-call optimization to the JVM.

This project will likely involve different components of the Java platform, with features believed to be divided thus:

- Continuations and UAI will be done in the JVM and exposed as very thin Java APIs.
- Fibers will be mostly implemented in Java in the JDK libraries, but may require some support in the JVM.
- JDK libraries making use of native code that blocks threads would need to be adapted to be able to run in fibers. In particular this implies changing the `java.io` classes.
- JDK libraries that make use of low-level thread synchronization (and in particular the `LockSupport` class), such as `java.util.concurrent` will need to be adapted to support fibers, but the amount of work required will depend on the fiber API, and in any event, expected to be small (as fibers expose a very similar API to threads).
- Debuggers, profilers and other serviceability services would need to be aware of fibers to provide a good user experience. This means that JFR and JVMTI would need to accommodate fibers, and relevant platform MBeans may be added.
- At this point we do not foresee a need for a change in the Java language.

It is early days for this project, and so everything — including its scope — is subject to change.

Terminology

As kernel threads and lightweight threads are just different implementations of the same abstraction, some confusion over terminology is bound to ensue. This document will adopt the following convention, and every correspondence in the project should follow suit:

- The word *thread* will refer to the abstraction only (which will be explored shortly) and never to a particular implementation, so *thread* may refer either to any implementation of the abstraction, whether done by the OS or by the runtime.
- When a particular implementation is referred, the terms *heavyweight thread*, *kernel threads* and *OS thread* can be used interchangeable to mean the implementation of thread provided by the operating system kernel. The terms *lightweight thread*, *user-mode thread*, and *fiber* can be used interchangeably to mean an implementation of threads provided by the language runtime — the JVM and JDK libraries in the case of the Java platform. Those words *do not* (at least in these early stages, when the API design is unclear) refer to specific Java classes.
- The capitalized words `Thread` and `Fiber` would refer to particular Java classes, and will be used mostly when discussing the design of the API rather than of the implementation.

What Threads Are

A *thread* is a sequence of computer instructions executed sequentially. As we are dealing with operations that may involve not just calculations but also IO, timed pauses, and synchronization — in general, instructions that cause the stream of computation to wait for some event external to it — a thread, then, has the ability to *suspend* itself, and to *automatically resume* when the event it waits for occurs. While a thread waits, it should vacate the CPU core, and allow another to run.

These capabilities are provided by two different concerns. A *continuation* is a sequence of instructions that execute sequentially, and may suspend itself (a more thorough treatment of continuations is given later, in the section Continuations). A *scheduler* assigns continuations to CPU cores, replacing a paused one with another that's ready to run, and ensuring that a continuation that is ready to resume will eventually be assigned to a CPU core. A thread, then, requires two constructs: a continuation and a scheduler, although the two may not necessarily be separately exposed as APIs.

Again, threads — at least in this context — are a fundamental abstraction, and do not imply any programming paradigm. In particular, they refer only to the abstraction allowing programmers to write sequences of code that can run and pause, and not to any mechanism of sharing information among threads, such as shared memory or passing messages.

As there are two separate concerns, we can pick different implementations for each. Currently, the thread construct offered by the Java platform is the `Thread` class, which is implemented by a kernel thread; it relies on the OS for the implementation of both the continuation and the scheduler.

A continuation construct exposed by the Java platform can be combined with existing Java schedulers — such as `ForkJoinPool`, `ThreadPoolExecutor` or third-party ones — or with ones especially optimized for this purpose, to implement fibers.

It is also possible to split the implementation of these two building-blocks of threads between the runtime and the OS. For example, modifications to the Linux kernel done at Google (video, slides), allow user-mode code to take over scheduling kernel threads, thus essentially relying on the OS just for the implementation of continuations, while having libraries handle the scheduling. This has the benefits offered by user-mode scheduling while still allowing native code to run on this thread implementation, but it still suffers from the drawbacks of relatively high footprint and not resizable stacks, and isn't available yet. Splitting the implementation the other way — scheduling by the OS and continuations by the runtime — seems to have no benefit at all, as it combines the worst of both worlds.

But why would user-mode threads be in any way better than kernel threads, and why do they deserve the appealing designation of *lightweight*? It is, again, convenient to separately consider both components, the continuation and the scheduler.

In order to suspend a computation, a continuation is required to store an entire call-stack context, or simply put, store the stack. To support native languages, the memory storing the stack must be contiguous and remain at the same memory address. While virtual memory does offer some flexibility, there are still limitations on just how lightweight and flexible such kernel continuations (i.e. stacks) can be. Ideally, we would like stacks to grow and shrink depending on usage. As a language runtime implementation of threads is not required to support arbitrary native code, we can gain more flexibility over how to store continuations, which allows us to reduce footprint.

The much bigger problem with the OS implementation of threads is the scheduler. For one, the OS scheduler runs in kernel mode, and so every time a thread blocks and control returned to the scheduler, a non-cheap user/kernel switch must occur. For another, OS schedulers are designed to be general-purpose and schedule many different kinds of program threads. But a thread running a video encoder behaves very differently from one serving requests coming over the network, and the same scheduling algorithm will not be optimal for both. Threads handling transactions on servers tend to present certain behavior patterns that present a challenge to a general-purpose OS scheduler. For example, it is a common pattern for a transaction-serving thread `A` to perform some action on the request, and then pass data on to another thread, `B`, for further processing. This requires some synchronization of a handoff between the two threads that can involve either a lock or a message queue, but the pattern is the same: `A` operates on some data `x`, hands it over to `B`, wakes `B` up and then blocks until it is handed another request from the network or another thread. This pattern is so common that we

can assume that `A` will block shortly after unblocking `B`, and so scheduling `B` on the same core as `A` will be beneficial, as `x` is already in the core's cache; in addition, adding `B` to a core-local queue doesn't require any costly contended synchronization. Indeed, a work-stealing scheduler like `ForkJoinPool` makes this precise assumption, as it adds tasks scheduled by running task into a local queue. The OS kernel, however, cannot make such an assumption. As far as it knows, thread `A` may want to continue running for a long while after waking up `B`, and so it would schedule the recently unblocked `B` to a different core, thus both requiring some synchronization, and causing a cache-fault as soon as `B` accesses `x`.

Fibers

Fibers are, then, what we call Java's planned user-mode threads. This section will list the requirements of fibers and explore some design questions and options. It is not meant to be exhaustive, but merely present an outline of the design space and provide a sense of the challenges involved.

In terms of basic capabilities, fibers must run an arbitrary piece of Java code, concurrently with other threads (lightweight or heavyweight), and allow the user to await their termination, namely, join them. Obviously, there must be mechanisms for suspending and resuming fibers, similar to `LockSupport`'s `park`/`unpark`. We would also want to obtain a fiber's stack trace for monitoring/debugging as well as its state (suspended/running) etc.. In short, because a fiber is a thread, it will have a very similar API to that of heavyweight threads, represented by the `Thread` class. With respect to the Java memory model, fibers will behave exactly like the current implementation of `Thread`. While fibers will be implemented using JVM-managed continuations, we may also want to make them compatible with OS continuations, like Google's user-scheduled kernel threads.

There are a few capabilities unique to fibers: we want a fiber to be scheduled by a pluggable scheduler (either fixed at the fiber's construction, or changeable when it is paused, e.g. with an `unpark` method that takes a scheduler as a parameter), and we'd like fibers to be serializable (discussed in a separate section).

In general, the fiber API will be nearly identical to that of `Thread` as the abstraction is the same, and we'd also like to run code that so far has run in kernel threads to run in fibers with little or no modification. This immediately suggests two design options:

1. Represent fibers as a `Fiber` class, and factor out the common API for `Fiber` and `Thread` into a common super-type, provisionally called `Strand`. Thread-implementation-agnostic code would be programmed against `Strand`, so that `Strand.currentStrand` would return a fiber if the code is running in a fiber, and `Strand.sleep` would suspend the fiber if the code is running in a fiber.
2. Use the same `Thread` class for both kinds of threads — user-mode and kernel-mode — and choose an implementation as a dynamic property set in a constructor or a setter called prior to invoking `start`.

A separate `Fiber` class might allow us more flexibility to deviate from `Thread`, but would also present some challenges. Because a user-mode scheduler does not have direct access to CPU cores, assigning a fiber to a core is done by running it in some worker kernel thread, and so every fiber has an underlying kernel thread, at least while it is scheduled to a CPU core, although the identity of underlying kernel thread is not fixed, and may change if the scheduler decides to schedule the same fiber to a different worker kernel thread. If the scheduler is written in Java — as we want — every fiber even has an underlying `Thread` instance. If fibers are represented by the `Fiber` class, the underlying `Thread` instance would be accessible to code running in a fiber (e.g. with `Thread.currentThread` or `Thread.sleep`), which seems inadvisable.

If fibers are represented by the same `Thread` class, a fiber's underlying kernel thread would be inaccessible to user code, which seems reasonable but has a number of implications. For one, it would require more work in the JVM, which makes heavy use of the `Thread` class, and would need to be aware of a possible fiber implementation. For another, it would limit our design flexibility. It also creates some circularity when writing

schedulers, that need to *implement* threads (fibers) by assigning them to threads (kernel threads). This means that we would need to expose the fiber's (represented by `Thread`) continuation for use by the scheduler.

Because fibers are scheduled by Java schedulers, they need not be GC roots, as at any given time a fiber is either runnable, in which case a reference to it is held by its scheduler, or blocked, in which case a reference to it is held by the object on which it is blocked (e.g. a lock or an IO queue), so that it can be unblocked.

Another relatively major design decision concerns thread locals. Currently, thread-local data is represented by the (`Inheritable`) `ThreadLocal` class(es). How do we treat thread-locals in fibers? Crucially, `ThreadLocal`s have two very different uses. One is associating data with a thread context. Fibers will probably need this capability, too. Another is to reduce contention in concurrent data structures with striping. That use abuses `ThreadLocal` as an approximation of a processor-local (more precisely, a CPU-core-local) construct. With fibers, the two different uses would need to be clearly separated, as now a thread-local over possibly millions of threads (fibers) is not a good approximation of processor-local data at all. This requirement for a more explicit treatment of thread-as-context vs. thread-as-an-approximation-of-processor is not limited to the actual `ThreadLocal` class, but to any class that maps `Thread` instances to data for the purpose of striping. If fibers are represented by `Thread`s, then some changes would need to be made to such striped data structures. In any event, it is expected that the addition of fibers would necessitate adding an explicit API for accessing processor identity, whether precisely or approximately.

An important feature of kernel threads is timeslice-based preemption (which will be called forceful, or forced preemption here, for brevity). A kernel thread that computes for a while without blocking on IO or synchronization will be forcefully-preempted after some time. While at first glance this seems to be an important design and implementation issue for fibers — and, indeed, we may decide to support it; JVM safepoints should make it easy — not only is it not important, but having this feature doesn't make much of a difference at all (so it is best to forgo it). The reason is as follows: unlike kernel threads, the number of fibers may be very large (hundreds of thousands or even millions). If *many* fibers require so much CPU time that they need to *often* be forcefully preempted then as the number of threads exceeds the number of cores by several orders of magnitude, the application is under-provisioned by orders of magnitude, and no scheduling policy will help. If *many* fibers need to run long computations *infrequently*, then a good scheduler will work around this by assigning fibers to available cores (i.e. worker kernel threads). If a *few* fibers need to run long computations *frequently*, then it is better to run that code in heavyweight threads; while different thread implementations provide the same abstraction, there are times where one implementation is better than the other, and it is not necessary for our fibers to be preferable to kernel threads in every circumstance.

A real implementation challenge, however, may be how to reconcile fibers with internal JVM code that blocks kernel threads. Examples include hidden code, like loading classes from disk to user-facing functionality, such as `synchronized` and `Object.wait`. As the fiber scheduler multiplexes many fibers onto a small set of worker kernel threads, blocking a kernel thread may take out of commission a significant portion of the scheduler's available resources, and should therefore be avoided.

On one extreme, each of these cases will need to be made fiber-friendly, i.e., block only the fiber rather than the underlying kernel thread if triggered by a fiber; on the other extreme, all cases may continue to block the underlying kernel thread. In between, we may make some constructs fiber-blocking while leaving others kernel-thread-blocking. There is good reason to believe that many of these cases can be left unchanged, i.e. kernel-thread-blocking. For example, class loading occurs frequently only during startup and only very infrequently afterwards, and, as explained above, the fiber scheduler can easily schedule around such blocking. Many uses of *synchronized* only protect memory access and block for extremely short durations — so short that the issue can be ignored altogether. We may even decide to leave *synchronized* unchanged, and encourage those who surround IO access with *synchronized* and block frequently in this way, to change their code to make use of the

`j.u.c` constructs (which will be fiber-friendly) if they want to run the code in fibers. Similarly, for the use of `Object.wait`, which isn't common in modern code, anyway (or so we believe at this point), which uses `j.u.c`.

In any event, a fiber that blocks its underlying kernel thread will trigger some system event that can be monitored with JFR/MBeans.

While fibers encourage the use of ordinary, simple and natural synchronous blocking code, it is very easy to adapt existing asynchronous APIs, turning them into fiber-blocking ones. Suppose that a library exposes this asynchronous API for some long-running operation, `foo`, which returns a `String`:

```
interface AsyncFoo {
   public void asyncFoo(FooCompletion callback);
}
```

where the callback, or completion handler `FooCompletion` is defined like so:

```
interface FooCompletion {
  void success(String result);
  void failure(FooException exception);
}
```

We will provide an async-to-fiber-blocking construct that may look something like this:

```
abstract class _AsyncToBlocking<T, E extends Throwable> {
    private _Fiber f;
    private T result;
    private E exception;

    protected void _complete(T result) {
        this.result = result;
        unpark f
    }

    protected void _fail(E exception) {
        this.exception = exception;
        unpark f
    }

    public T run() throws E {
        this.f = current fiber
        register();
        park
        if (exception != null)
           throw exception;
        return result;
    }

    public T run(_timeout) throws E, TimeoutException { ... }

    abstract void register();
}
```

We can then create a blocking version of the API by first defining the following class:

```
abstract class AsyncFooToBlocking extends _AsyncToBlocking<String, FooException>
     implements FooCompletion {
  @Override
  public void success(String result) {
    _complete(result);
  }
  @Override
  public void failure(FooException exception) {
    _fail(exception);
  }
}
```

which we then use to wrap the asynchronous API with as synchronous version:

```
class SyncFoo {
    AsyncFoo foo = get instance;

    String syncFoo() throws FooException {
        new AsyncFooToBlocking() {
          @Override protected void register() { foo.asyncFoo(this); }
        }.run();
    }
}
```

We can include such ready integrations for common asynchronous classes, such as `CompletableFuture`.

Continuations

The motivation for adding continuations to the Java platform is for the implementation of fibers, but continuations have some other interesting uses, and so it is a secondary goal of this project to provide continuations as a public API. The utility of those other uses is, however, expected to be much lower than that of fibers. In fact, continuations don't add expressivity on top of that of fibers (i.e., continuations can be implemented on top of fibers).

In this document and everywhere in Project Loom, the word *continuation* will mean a *delimited continuation* (also sometimes called a *coroutine*[1]). Here we will think of delimited continuations as sequential code that may suspend (itself) and resume (be resumed by a caller). Some may be more familiar with the point of view that sees continuations as objects (usually subroutines) representing "the rest" or "the future" of a computation. The two describe the very same thing: a suspended continuation, is an object that, when resumed or "invoked", carries out the rest of some computation.

A delimited continuation is a sequential sub-program with an entry point (like a thread), which we'll call simply the *entry point* (in Scheme, this is the *reset point*), which may suspend or yield execution at some point, which we'll call the *suspension point* or the *yield point* (the *shift* point in Scheme). When a delimited continuation suspends, control is passed outside of the continuation, and when it is resumed, control returns to the last yield point, with the execution context up to the entry point intact. There are many ways to present delimited continuations, but to Java programmers, the following rough pseudocode would explain it best:

```
foo() { // (2)
  ...
  bar()
  ...
}

bar() {
  ...
  suspend // (3)
  ... // (5)
}

main() {
  c = continuation(foo) // (0)
  c.continue() // (1)
  c.continue() // (4)
}
```

A continuation is created (0), whose entry point is `foo`; it is then invoked (1) which passes control to the entry point of the continuation (2), which then executes until the next suspension point (3) inside the `bar` subroutine, at which point the invocation (1) returns. When the continuation is invoked again (4), control returns to the line following the yield point (5).

The continuations discussed here are "stackful", as the continuation may block at any nested depth of the call stack (in our example, inside the function `bar` which is called by `foo`, which is the entry point). In contrast, stackless continuations may only suspend in the same subroutine as the entry point. Also, the continuations discussed here are non-reentrant, meaning that any invocation of the continuation may change the "current" suspension point. In other words, the continuation object is stateful.

The main technical mission in implementing continuations — and indeed, of this entire project — is adding to HotSpot the ability to capture, store and resume callstacks not as part of kernel threads. JNI stack frames will likely *not* be supported.

As continuations serve as the basis for fibers, if continuations are exposed as a public API, we will need to support nested continuations, meaning code running inside a continuation must be able to suspend not only the continuation itself, but an enclosing one (e.g., suspend the enclosing fiber). For example, a common use for continuations is in the implementation of generators. A generator exposes an iterator, and the code running inside the generator produces another value for the iterator every time it yields. It should therefore be possible to write code like this:

```
new _Fiber(() -> {
  for (Object x : new _Generator(() -> {
      produce 1
      fiber sleep 100ms
      produce 2
      fiber sleep 100ms
      produce 3
  })) {
      System.out.println("Next: " + x);
  }
})
```

In the literature, nested continuations that allow such behavior are sometimes call "delimited continuations with multiple named prompts", but we'll call them *scoped continuations*. See this blog post for a discussion of the theoretical expressivity of scoped continuations (to those interested, continuations are a "general effect", and can be used to implement any effect — e.g. assignment — even in a pure language that has no other side-effect; this is why, in some sense, continuations are the fundamental abstraction of imperative programming).

Code running inside a continuation is not expected to have a reference to the continuation, and the scopes normally have some fixed names (so suspending scope `A` would suspend the innermost enclosing continuation of scope `A`). However, the yield point provides a mechanism to pass information from the code to the continuation instance and back. When a continuation suspends, no `try/finally` blocks enclosing the yield point are triggered (i.e., code running in a continuation cannot detect that it is in the process of suspending).

As one of the reasons for implementing continuations as an independent construct of fibers (whether or not they are exposed as a public API) is a clear separation of concerns. Continuations, therefore, are not thread-safe and none of their operations creates cross-thread happens-before relations. Establishing the memory visibility guarantees necessary for migrating continuations from one kernel thread to another is the responsibility of the fiber implementation.

A rough outline of a possible API is presented below. Continuations are a very low-level primitive that will only be used by library authors to build higher-level constructs (just as `java.util.Stream` implementations leverage `Spliterator`). It is expected that classes making use of contiuations will have a private instance of the continuation class, or even, more likely, of a subclass of it, and that the continuation instance will not be directly exposed to consumers of the construct.

```
class _Continuation {
    public _Continuation(_Scope scope, Runnable target)
    public boolean run()
    public static _Continuation suspend(_Scope scope, Consumer<_Continuation> ccc)

    public ? getStackTrace()
}
```

The `run` method returns `true` when the continuation terminates, and false if it suspends. The `suspend` method allows passing information from the yield point to the continuation (using the `ccc` callback that can inject information into the continuation instance it is given), and back from the continuation to the suspension point (using the return value, which is the continuation instance itself, from which information can be queried).

To demonstrate how easily fibers can be implemented in terms of continuations, here is a partial, simplistic implementation of a `_Fiber` class representing a fiber. As you'll note, most of the code maintains the fiber's state, to ensure it doesn't get scheduled more than once concurrently:

```
class _Fiber {
    private final _Continuation cont;
    private final Executor scheduler;
    private volatile State state;
    private final Runnable task;

    private enum State { NEW, LEASED, RUNNABLE, PAUSED, DONE; }

    public _Fiber(Runnable target, Executor scheduler) {
        this.scheduler = scheduler;
        this.cont = new _Continuation(_FIBER_SCOPE, target);

        this.state = State.NEW;
        this.task = () -> {
              while (!cont.run()) {
                  if (park0())
                     return; // parking; otherwise, had lease -- continue
              }
              state = State.DONE;
        };
    }

    public void start() {
        if (!casState(State.NEW, State.RUNNABLE))
            throw new IllegalStateException();
        scheduler.execute(task);
    }

    public static void park() {
        _Continuation.suspend(_FIBER_SCOPE, null);
    }

    private boolean park0() {
        State st, nst;
        do {
            st = state;
            switch (st) {
              case LEASED:   nst = State.RUNNABLE; break;
              case RUNNABLE: nst = State.PAUSED;   break;
              default:       throw new IllegalStateException();
            }
        } while (!casState(st, nst));
        return nst == State.PAUSED;
    }

    public void unpark() {
        State st, nst;
        do {
            State st = state;
            switch (st) {
              case LEASED:
              case RUNNABLE: nst = State.LEASED;   break;

              case PAUSED:   nst = State.RUNNABLE; break;
```

```
                default:       throw new IllegalStateException();
            }
        } while (!casState(st, nst));
        if (nst == State.RUNNABLE)
            scheduler.execute(task);
    }

    private boolean casState(State oldState, State newState) { ... }
}
```

Schedulers

As mentioned above, work-stealing schedulers like `ForkJoinPools` are particularly well-suited to scheduling threads that tend to block often and communicate over IO or with other threads. Fibers, however, will have pluggable schedulers, and users will be able to write their own ones (the SPI for a scheduler can be as simple as that of `Executor`). Based on prior experience, it is expected that `ForkJoinPool` in asynchronous mode can serve as an excellent default fiber scheduler for most uses, but we may want to explore one or two simpler designs, as well, such as a pinned-scheduler, that always schedules a given fiber to a specific kernel thread (which is assumed to be pinned to a processor).

Unwind-and-Invoke

Unlike continuations, the contents of the unwound stack frames is not preserved, and there is no need in any object reifying this construct.

TBD

Additional Challenges

While the main motivation for this goal is to make concurrency easier/more scalable, a thread implemented by the Java runtime and over which the runtime has more control, has other benefits. For example, such a thread could be paused and serialized on one machine and then deserialized and resumed on another. This is useful in distributed systems where code could benefit from being relocated closer to the data it accesses, or in a cloud platform offering function-as-a-service, where the machine instance running user code could be terminated while that code awaits some external event, and later resumed on another instance, possibly on a different physical machine, thus making better use of available resources and reducing costs for both host and client. A fiber would then have methods like `parkAndSerialize`, and `deserializeAndUnpark`.

As we want fibers to be serializable, continuations should be serializable as well. If they are serializable, we might as well make them cloneable, as the ability to clone continuations actually adds expressivity (as it allows going back to a previous suspension point). It is, however, a very serious challenge to make continuation cloning useful enough for such uses, as Java code stores a lot of information off-stack, and to be useful, cloning would need to be "deep" in some customizable way.

Other Approaches

An alternative solution to that of fibers to concurrency's simplicity vs. performance issue is known as async/await, and has been adopted by C# and Node.js, and will likely be adopted by standard JavaScript. Continuations and fibers dominate async/await in the sense that async/await is easily implemented with continuations (in fact, it can be implemented with a weak form of delimited continuations known as stackless continuations, that don't capture an entire call-stack but only the local context of a single subroutine), but not vice-versa.

While implementing async/await is easier than full-blown continuations and fibers, that solution falls far too short of addressing the problem. While async/await makes code simpler and gives it the appearance of normal, sequential code, like asynchronous code it still requires significant changes to existing code, explicit support in libraries, and does not interoperate well with synchronous code. In other words, it does not solve what's known as the "colored function" problem.

---

1. Whether we'll call it continuation or coroutine going forward is TBD — there is a difference in meaning, but the nomenclature does not seem to be fully standardized, and continuation seems to be used as the more general term.↩