Document number: N4024
Date: 2014-05-22
Project: Programming Language C++, SG1
Reply-to: Nat Goodspeed ( nat at lindenlab dot com)
Oliver Kowalke (oliver dot kowalke at gmail dot com)

# Distinguishing coroutines and fibers

## Overview

The purpose of this paper is to foreshadow a forthcoming proposal to introduce fibers to the C++ standard library; to briefly describe the features in the proposed fiber library; and to contrast it with the coroutine library proposed in N3985[6].

It is hoped that this comparison will help to clarify the feature set of the proposed coroutine library. Certain features properly belong in the coroutine library; other conceptually-related features more properly belong in the fiber library.

## Background

A coroutine library was initially proposed in N3708[4]; the proposal was subsequently revised in N3985[6]. Some of the ensuing discussion suggests that, for purposes of refining the coroutine proposal, it might be useful for the authors to disambiguate the conceptual space addressed by the coroutine library from the conceptual space addressed by the fiber library.

## Fibers

### A Quick Sketch of the Fiber Library

For purposes of this paper, we can regard the term 'fiber' to mean 'user-space thread.' A fiber is launched, and conceptually it can have a lifespan independent of the code that launched it. A fiber can be detached from the launching code; alternatively, one fiber can join another. A fiber can sleep until a specified time, or for a specified duration. Multiple conceptually-independent fibers can run on the same kernel thread. When a fiber blocks, for instance waiting for a result that's not yet available, other fibers on the same thread continue to run. 'Blocking' a fiber implicitly transfers control to a fiber scheduler to dispatch some other ready-to-run fiber.

Fibers conceptually resemble kernel threads. In fact, the forthcoming fiber library proposal intentionally emulates much of the std::thread API. It provides fiber-local storage. It provides several variants of fiber mutexes. It provides condition_variables and barriers. It provides bounded and unbounded queues. It provides future, shared_future, promise and packaged_task. These fiber-oriented synchronization mechanisms differ from their thread counterparts in that when (for instance) a mutex blocks its caller, it blocks only the calling fiber – not the whole thread on which that fiber is running.

When a fiber blocks, it cannot assume that the scheduler will awaken it the moment its wait condition has been satisfied. Satisfaction of that condition marks the waiting fiber ready-to-run; eventually the scheduler will select that ready fiber for dispatch.

The key difference between fibers and kernel threads is that fibers use cooperative context switching, instead of preemptive time-slicing. Two fibers on the same kernel thread will not run simultaneously on different processor cores. At most one of the fibers on a particular kernel thread can be running at any given moment.

This has several implications:

- Fiber context switching does not engage the kernel: it takes place entirely in user space. This permits a fiber implementation to switch context significantly faster than a thread context switch.
- Two fibers in the same thread cannot execute simultaneously. This can greatly simplify sharing data between such fibers: it is impossible for two fibers in the same thread to race each other. Therefore, within the domain of a particular thread, it is not necessary to lock shared data.
- The coder of a fiber must take care to sprinkle voluntary context switches into long CPU-bound operations. Since fiber context switching is entirely cooperative, a fiber library cannot guarantee progress for every fiber without such precautions.
- A fiber that calls a standard library or operating-system function that blocks the calling thread will in fact block the entire thread on which it is running, including all other fibers on that same thread. The coder of a fiber must take care to use asynchronous I/O operations, or operations that engage fiber blocking rather than thread blocking.

In effect, fibers extend the concurrency taxonomy:

- on a single computer, multiple processes can run
- within a single process, multiple threads can run
- within a single thread, multiple fibers can run.

### A Few Fiber Use Cases

A fiber is useful when you want to launch a (possibly complex) sequence of asynchronous I/O operations, especially when you must iterate or make decisions based on their results.

Distinct fibers can be used to perform concurrent asynchronous fetch operations, aggregating their results into a fiber-specific queue for another fiber to consume.

Fibers are useful for organizing response code in an event-driven program. Typically, an event handler in such a program cannot block its calling thread: that would stall handlers for all other events, such as mouse movement. Handlers must use asynchronous I/O instead of blocking I/O. A fiber allows a handler to resume upon completion of an asynchronous I/O operation, rather than breaking out subsequent logic as a completely distinct handler.

Fibers can be used to implement a task handling framework to address the C10K-problem.[1]
For instance *strict fork-join task parallelism*[5] with its two flavours - *fully-strict computation*[5] (no task can proceed until it joins all of its child-tasks) and *terminally-strict computations*[5] (child-tasks are joined only at the end of processing) - can be supported.
Furthermore, different scheduling strategies are possible: work-stealing and continuation-stealing.
For work-stealing, the scheduler creates a child-task (child-fiber) and immediately returns to the caller. Each child-task (child-fiber) is executed or stolen by the scheduler based on the available resources (CPU etc.).
For continuation-stealing, the scheduler immediately executes the spawned child-task (child-fiber); the rest of the function (continuation) is stolen by the scheduler as resources are available.

## Coroutines

### A Quick Recap of the Coroutine Library

A coroutine is instantiated and called. When the invoker calls a coroutine, control immediately transfers into that coroutine; when the coroutine yields, control immediately returns to its caller (or, in the case of symmetric coroutines, to the designated next coroutine).

A coroutine does not have a conceptual lifespan independent of its invoker. Calling code instantiates a coroutine, passes control back and forth with it for some time, and then destroys it. It makes no sense to speak of 'detaching' a coroutine. It makes no sense to speak of 'blocking' a coroutine: the coroutine library provides no scheduler. The coroutine library provides no facilities for synchronizing coroutines: coroutines are already synchronous.

Coroutines do not resemble threads. A coroutine much more closely resembles an ordinary function, with a semantic extension: passing control to its caller with the expectation of being resumed later at exactly the same point. When the invoker resumes a coroutine, the control transfer is immediate. There is no intermediary, no agent deciding which coroutine to resume next.

### A Few Coroutine Use Cases

Normally, when consumer code calls a producer function to obtain a value, the producer must return that value to the consumer, discarding all its local state in so doing. A coroutine allows you to write producer code that ‘pushes’ values (via function call) to a consumer that ‘pulls’ them with a function call.

For instance, a coroutine can adapt callbacks, as from a SAX parser, to values explicitly requested by the consumer.

Moreover, the proposed coroutine library provides iterators over a producer coroutine so that a sequence of values from the producer can be fed directly into an STL algorithm. This can be used, for example, to flatten a tree structure.

Coroutines can be chained: a source coroutine can feed values through one or more filter coroutines before those values are ultimately delivered to consumer code.

In all the above examples, as in every coroutine usage, the handshake between producer and consumer is direct and immediate.

## Relationship

The authors have a reference implementation of the forthcoming fiber library (boost.fiber[3]). The reference implementation is entirely coded in portable C++; in fact its original implementation was entirely in C++03.

This is possible because the reference implementation of the fiber library is built on boost.coroutine[2], which provides context management. The fiber library extends the coroutine library by adding a scheduler and the aforementioned synchronization mechanisms.

Of course it would be possible to implement coroutines on top of fibers instead. But the concepts map more neatly to implementing fibers in terms of coroutines. The corresponding operations are:

- a coroutine yields;
- a fiber blocks.

When a coroutine yields, it passes control directly to its caller (or, in the case of symmetric coroutines, a designated other coroutine).

When a fiber blocks, it implicitly passes control to the fiber scheduler. Coroutines have no scheduler because they need no scheduler.

## Summary

It is hoped that this paper will help to illuminate some perceived omissions in the proposed coroutine library, so that the reader can properly associate desired functionality with the coroutine library proposal versus the forthcoming fiber library proposal

## References

[1] The C10K problem, Dan Kegel

[2] boost.coroutine

[3] boost.fiber

[4] N3708: A proposal to add coroutines to the C++ standard library

[5] N3832: Task Region

[6] N3985: A proposal to add coroutines to the C++ standard library (Revision 1)