Terms and definitions related to threads

Document number:	N4231
Date:	2014-10-10

Authors:	Torvald Riegel, Clark Nelson
Reply-to:	Torvald Riegel <triegel@redhat.com>

1 "Thread" seems to be used ambiguously

There seems to be quite a bit of disagreement—or just confusion—about what a "thread" refers to in the context of the standard. Similarly, there is further disagreement about how we should name and define related things such as execution mechanisms that are lighter-weight than `std::thread`.

The goal of this paper is to try to disambiguate the term "thread" and improve the terminology for related concepts such as execution agents. A secondary goal is to find definitions that can be shared with ISO C.

We will summarize the definitions in the current standard and the current version of the WG14 CPLEX TS (N1862)[1], and propose a terminology based on execution agents.

2 Definitions in the current standard

"Thread of execution" This is the most basic definition of a thread. It is defined as "a single flow of control within a program" (see §1.10p1). *sequenced-before* is defined exactly for a thread of execution (see §1.9p13). So, another way to find out whether something is a thread of execution would be to ask whether *sequenced-before* is defined for it.

`std::thread` This is specified as a component "that can be used to create and manage threads" (see §30.1p1 and §30.3p1), where "threads" explicitly refers to the definition of "threads of execution" (see §30.1p1). `std::thread` specifically "provides a mechanism to create a new thread of execution" (see §30.3.1p1).

A non-normative note states that `std::thread` is "intended to map one-to-one with operating system threads", but it is not mentioned whether it is also intended for the OS

[1] `http://open-std.org/JTC1/SC22/WG14/www/docs/n1862.pdf`

thread to have the same lifetime as the respective `std::thread` instance, which would matter for programs that want to use OS-thread features that are affected by this (e.g., OS-thread-provided thread-specific storage with destruction on OS thread exit).

"Thread" In the standard, "thread" is a short-hand for a thread of execution, *not* for `std::thread` (see §1.10p1: "thread of execution (also known as a thread)").

The standard is a little sloppy in two places, where it refers to `std::thread` or uses the term "thread" but cites §30.3 when specifying things that seem to apply to threads of execution in general; nonetheless, we believe this is simply due to `std::thread` currently being the only mechanism by which a thread of execution can be created.

"Execution agent" In §30.2.5.1p1, an execution agent is defined as "an entity such as a thread that may perform work in parallel with other execution agents". It is also noted that "implementations or users may introduce other kinds of agents such as processes or thread-pool tasks".

The term "thread" in this definition, in contrast with the rest of the standard, likely means `std::thread` because a thread of execution is a flow of control, not primarily a means of execution. Also noteworthy is that thread-pool *tasks* are given as an example of an execution agent, not whole thread pools; this aligns well with `std::thread` being a mechanism to create (and implicitly trigger execution of) *one* thread of execution.

3 WG14 CPLEX TS (N1862)

"Thread of execution" This is defined in roughly the same way as in C++ (at least that is the explicitly stated intent).

"OS thread" This is defined as a "service provided by an operating system for executing multiple threads of execution concurrently". It is not quite clear to me whether this intends to state multiple threads of execution can be executed using one OS thread, or simply whether the purpose of OS threads is to have several of them running concurrently.

"Thread" The TS allows this as a short-hand for either thread of execution or OS thread, and points out the ambiguity.

"Execution agent" The definition in the TS is very much the same as in the C++ standard, except that an OS thread is given as example of an execution agent, and that it explicitly mentions that several threads of execution work in parallel ("entity, such as an OS thread, that may execute a thread of execution in parallel with other execution agents").

"Task" This is defined as a "thread of execution within a program that can be correctly executed asynchronously with respect to (certain) other parts of the program".

4 A terminology proposal based on execution agents

The set of terms and definitions we will outline next use the existing notion of an execution agent as a placeholder for all the mechanisms that execute threads of execution. It does not require significant changes to the standard, just makes the notion of an execution agent more important and clarifies a few things.

"Thread of execution" Keep the standard's definition. In other words, it is a distinct flow of control that is part of the program, and a program can have many threads of execution.

"Execution agent" Every mechanism that executes a particular thread of execution is called an execution agent (EA). Note that this is the name of the conceptual entity that executes threads of execution, not an implementation artifact; this is similar to us speaking about threads of execution instead of the combination of an instruction pointer and a stack.

There is a one-to-one correspondence between an EA (i.e., an EA instance) and a thread of execution. This makes it easier to define properties such as progress for an EA because it doesn't lump together the execution of several threads of execution.

An EA is responsible for (or, determines under which circumstances) the thread of execution is allowed to make progress; informally, it does so by executing the flow of control on whatever underlying resource it may use. See N4156 for the different classes of progress guarantees (i.e., concurrent, parallel, weakly parallel).

Because of the one-to-one mapping between EA and thread of execution, one might consider the EA term to be an unnecessary indirection. However, it adds properties to the notion of a flow of control (i.e., a thread of execution), such as the progress guarantees or other semantics related to how the flow of control is executed. Also, it conveniently avoids the misunderstanding that the short-hand of "thread" for "thread of execution" would refer to `std::thread`. It should make it more obvious that also other things than `std::thread` or a simple OS thread could execute flows of control (e.g., SIMD lanes in vectorized execution).

We already have the notion of execution agents in the standard, albeit with a slightly different description and not as widely used.

`std::thread` This is a mechanism to create and manage an EA with execution properties that are equal to those offered by a typical OS thread. One could also roughly say that `std::thread` *is* an EA, but this would put in my opinion too much focus on whether the interface that `std::thread` offers is the defining property of an EA. On the other hand, `std::thread` is already close to a one-shot EA-execution mechanism (i.e., one thread of execution), so it would not do much harm to say that it is an EA.

"Thread" We believe this should remain to be the short-hand form for "thread of execution"; most of the statements about multi-threaded programs or multi-threaded

behavior in the standard apply to or arise for threads of execution—in other words, the mere presence of concurrency. These are independent of questions related to progress, which we can define for the EAs.

While keeping the short-hand form does not avoid all potential confusion in the standard, having to write "multi-thread-of-execution program" would probably be awkward. We think that teaching that "thread" refers to "threads of execution" in general should be doable. Also, we suppose that in many cases where "thread" could be misunderstood as `std::thread` due to the context, we could use the term "execution agent" to disambiguate.

"Task" We do not have a strong opinion on what a "task" created by a parallelism abstraction (e.g., a parallel loop) should be. We lean towards saying that a task is an execution agent launched by the parallelism abstraction, but we could also choose to let it be one of the threads of execution that the parallelism abstraction splits the work into.

Examples

- A `std::thread` is (roughly) an EA. (More precisely, it contains so little around an EA to justify saying that it is an EA).
- A SIMD loop launches EAs.
- A thread pool is not an EA, but can be used to launch EAs.
- An executor can launch EAs.
- An OS thread could or could not be the base on which an EA is implemented; for example, a thread of execution on a GPU is often not related to any particular thread on the host CPU.
- Because an EA is not necessarily implemented based on an OS thread (and thus, `std::thread` if we want to accept the non-normative note in §30.3 as a requirement), it depends on the kind of EA whether, for example, OS-specific thread-specific store is available.