Not All Agents Have TLS

This paper: P1367R1.

Author: ogiroux@nvidia.com

Date: 16/06/2019.

The reality of `thread_local` is more complex than the Standard's terminology can be used to explain. This paper proposes to make the current state-of-the-art more comprehensible.

Revision history

R1: This version. Makes more clear what is proposed to be documented.

R0: First version. Already included some feedback from the reflector.

Introduction

About the word "thread"...

There are two concepts that we call a "thread":

1. One of a concrete kind of execution agent underneath `thread` and `main`.
2. The closure over all evaluations *sequenced after* a specific evaluation.

Today, the Standard continues to conflate these two concepts in many places, despite the wording improvements we made around execution agents. That conflation appears harmless in the Standard today, because the execution agents which the Standard itself has to account for are those created for `thread` and `main` only. Unfortunately, that makes it impossible to model real implementations' non-`thread`, non-`main` execution agents using the Standard's terminology without getting confused - including many which are tempting to use underneath parallel algorithm.

Step 0: a thread is a `thread`.

Before I propose any deeper changes, we need to agree (at least temporarily while you read this) to address this conflation:

1. The word *thread* (in prose font) and the identifier `thread` (in code font) should be clarified to mean the same intuitive thing: a concrete execution agent with all of the capabilities of an OS thread.
2. The extant term *thread of execution* can then be clarified to refer to the evaluations performed by an execution agent, that are *sequenced after* the initial one.

Current state-of-the-art

The Standard requires that `thread_local` objects be instantiated for every thread. Because of the conflation discussed above, it's unclear which of the two concepts is truly meant here. This said, implementations all take it to mean the same thing: it's *threads*, not *threads of execution*.

The problem with this (universal) interpretation is that no one actually provides a definition for what `thread_local` means in executions agents besides *threads*. The Standard neither provides this meaning, nor does it provide the words needed to provide this meaning. Let's address that.

Proposed direction: 5 steps to better clarity.

Step 1: `thread_local` relates to *threads of execution*, some of them.

Let's first consider that `thread_local` and TLS are also different concepts. The former qualifies an object instance declaration, to tie an instance of this type to a "thread", which we could de-conflate to either be a *thread* or a *thread of control*. The latter is a storage mechanism that *threads* tend to provide, is understood to be useful to be implement `thread_local`, and is clearly a source of inspiration for `thread_local`.

The assumption that `thread_local` is automatically related to *threads* causes these problems:

1. Names are introduced in C++ for all *threads of execution*, even those not on `thread` or `main`.
2. Accesses to `thread_local` objects have undocumented behavior, except on `thread` or `main`.
3. We have proposals to duplicate this pattern: add X`_local` for each execution agent type X.

My suggestion is to turn this assumption around: `thread_local` is about *threads of control*, but accesses to `thread_local` objects have *undefined behavior* if conditions on the execution agent are not met. There are only two settings here, not a plethora of options: either `thread_local` has the one meaning the Standard gives it now, or it has *undefined behavior* (or IFNDR, possibly).

While we're on this topic, I also suggest that we interpret the `std::this_thread` namespace to refer to the *thread of execution*. We may want to add functions to query execution agent properties in this namespace, which may be implemented with TLS (true) or underlying execution agent features under implementation control. For users of *threads*, nothing changes; for users of non-thread execution agents, now they get documented behavior.

Step 2: threads shall support `thread_local`.

Before we consider execution agents that do not support `thread_local`, and what that may mean, it's clear that *threads* must support `thread_local`. That this be unconditionally true is required for compatibility with C++11, we cannot revisit this.

Implementations are allowed to offer "threads" that are not equivalent to ISO C++ *threads* with different TLS semantics than `thread_local`, but these "threads" cannot be spelled `thread` or `main`. The only exception to this is for Freestanding implementation, see below.

Step 3: `main` has *implementation-defined* TLS.

There are only two valid cases:

- If `thread` is supported, then `main` has `thread_local` all the same.
- Else, `thread_local` objects may be simple global variables. This only applies to Freestanding implementations that choose not to support `thread`.

```
thread_local int x = 0;

int main(int,char*[]) {
  x = 1;              // OK, may be a static on Freestanding
  std::thread t([](){ // OK on Hosted, impl-def on Freestanding
    x = 2;            // OK, must be supported on std::thread
  }).join();
  return 0;
}
```

Step 4: when execution agents do not support `thread_local`.

Executors *should* (are encouraged to) create execution agents that support `thread_local`, the same as `thread` and `main`. Executors that do not, however, *must* indicate what they support to the user through queryable properties on the executor's interface and/or the `this_thread` namespace (if we extend it to do so). Note that execution agents created for parallel algorithms with standard policies may have *implementation-defined* support, as if the implementation used an executor to do so (as they are wont to do).

For example:

```
namespace std {
  namespace this_thread {
    static constexpr bool has_locals(); // example extension
  }
}

thread_local int x = 0;

void foo() {
  if(std::this_thread::has_locals())
    x = 1; // OK
  else
    x = 2; // Oops
}
```

If `thread_local` is not supported on an execution agent, then uses of such objects on that execution agent are either *undefined* or *ill-formed, no diagnostic required*.

Step 5: attributes may be used for selective support.

Execution agents that report support for `thread_local` shall allow access to all instances of `thread_local` objects in the program. Those which do not are not expected to support any such instances.

What of execution agents that could offer selective support? These would report no support in the Standard sense, but may be documented by the implementation to have some kind of support.

To clarify this support, implementations with execution agents "XYZ" that have *implementation-defined* support should:

1. Offer a custom queryable property, e.g. `has_xyz_locals()`.
2. Use a custom attribute on `thread_local` declarations, e.g. `[[xyz_local]] thread_local` to introduce a `thread_local` object in this implementation-

specific local storage.

Note that attributes may always be ingored and, in fact should be ignored in other contexts. Hence, continuing with our example:

1. The set of every object declared with `thread_local` regardless of attributes, is available for `thread`, `main` and execution agents where `std::this_thread::has_locals()` is `true`.
2. The subset of those objects declared with `[[xyz_local]] thread_local`, is available for execution agents where `std::this_thread::has_locals()` is `false`, but `has_xyz_locals()` is `true`.

This allow implementations to introduce specialized execution agents with selective TLS, in a portable way. That is, the semantics of a program that uses the attribute remains clear / well-defined on other implementations with other execution agents, especially those based on *threads*.

In closing

In the real world, not all execution agents have TLS, but the Standard leaves us without words to describe them. If we increase the separation between the concepts of *threads* and *threads of execution*, we can begin to define what `thread_local` means on different kinds of execution agents. We can then offer minor extensions that make it possible to write portable programs even when they make use of specialized execution agents.

See also

http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2012/n3487.pdf

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3556.pdf

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4439.pdf

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0097r0.html

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0108r1.html

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0772r1.pdf