Document number: P3346R0
Date: 2024-10-16
Author: Nat Goodspeed (nat@lindenlab.com)
Audience: LEWG, EWG

thread_local means fiber-specific

Abstract

P0876R18[1] does not specify any changes to the semantics of thread storage duration ([basic.stc.thread]). The implication is that if two fibers are running on the same thread, they will both share the same value of a given `thread_local` variable. Each fiber will see modifications made by the other fiber. They will not race ([intro.races]), since at every moment a specific thread is running exactly one fiber.

Nonetheless, this could be problematic for an existing library that relies on `thread_local` variables if multiple fibers on the same thread take turns making calls into that library. From the library's point of view, the value of its `thread_local` variables might change unexpectedly.

It is suggested that every `thread_local` variable should have a distinct value for each fiber that accesses it. This paper details changes to the Standard[2] to express that functionality.

When the Standard first introduced `thread_local`, library authors were strongly encouraged to migrate from `static` variables to `thread_local`. The C++ community has already paid that price. Now, instead of introducing another new keyword requiring another new migration, we should leverage the existing keyword to retain the desired behavior of a variable whose value persists between successive calls, independently of calls into the same library by other execution agents.

To address the less common use case of a variable local to a `std::thread` but shared by all fibers within that thread, we also introduce `thread_specific_ptr`, a design pioneered in the Boost library suite.[3]

This paper depends on P0876R18.

Revision History

Initial revision.

Acknowledgments

The author would like to thank ADAM Martin.

Wording

This wording is relative to N4981.[2]

Modify §6.7.5.3 [basic.stc.thread] as follows:

1 All variables declared with the `thread_local` keyword have *thread storage duration*. The storage for these entities lasts for the duration of the thread fiber in which they are created. There is a distinct object or reference per thread fiber, and use of the declared name refers to the entity associated with the current thread fiber.

2 [*Note 1:* A variable with thread storage duration is initialized as specified in [basic.start.static], [basic.start.dynamic], and [stmt.dcl] and, if constructed, is destroyed on thread fiber exit ([basic.start.term]) ([fiber.context.overview]). *— end note*]

Modify §6.9.3.4 [basic.start.term] paragraph 2 as follows:

2 Constructed objects with thread storage duration within a given thread fiber are destroyed as a result of returning from the initial function of that thread fiber and as a result of that thread fiber calling `std::exit`. The destruction of all constructed objects with thread storage duration within that thread fiber strongly happens before destroying any object with static storage duration.

Modify §11.4.9.3 [class.static.data] paragraph 1 as follows:

1 A static data member is not part of the subobjects of a class. If a static data member is declared `thread_local` there is one copy of the member per thread fiber. If a static data member is not declared `thread_local` there is one copy of the data member that is shared by all the objects of the class.

Modify §17.5 [support.start.term] paragraph 9.1 as follows:

(9.1) — First, objects with thread storage duration and associated with the current thread thread's default fiber are destroyed. Next, objects with static storage duration are destroyed and functions registered by calling `atexit` are called.[191]

See [basic.start.term] for the order of destructions and calls. (Objects with automatic storage duration are not destroyed as a result of calling `exit()`.)[192]

If a registered function invoked by `exit` exits via an exception, the function `std::terminate` is invoked([except.terminate]).

Modify §33.10.10.2 [futures.task.members] paragraph 23 as follows:

23 *Effects:* As if by *INVOKE*`<R>(f, t 1, t 2,..., t N)` ([func.require]), where `f` is the stored task and `t 1, t 2,..., t N` are the values in `args...`. If the task returns normally, the return value is stored as the asynchronous result in the shared state of `*this`, otherwise the exception thrown by the task is stored. In either case, this is done without making that state ready([futures.state]) immediately. Schedules the shared state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread thread's default fiber have been destroyed.

Insert new final subclause in clause 33 [thread] as indicated:

33.13 thread_specific_ptr [thread.ptr]

33.13.1 Overview [thread.ptr.overview]

1 Objects with thread storage duration now have a distinct instance for each fiber within a thread, and are destroyed when the fiber terminates. It is sometimes desirable to access storage that is shared between all fibers on a thread, but distinct for each referencing thread.

2 The `thread_specific_ptr` class manages pointers, one per referencing thread. This can be used to access a distinct object with dynamic storage duration for each thread, that is nonetheless shared between all fibers on that thread.

33.13.2 Header <thread_specific_ptr> synopsis [thread.ptr.synopsis]

```
namespace std {

    // [thread.ptr], class thread_specific_ptr
    template <class T, class Deleter=default_delete<T>>
    class thread_specific_ptr;

}
```

33.13.3 Class thread_specific_ptr [thread.ptr.class]

```
namespace std {

template <class T, class Deleter=default_delete<T>>
class thread_specific_ptr {
public:
    // [thread.ptr.cons], constructors, move and assignment
    thread_specific_ptr();
    explicit thread_specific_ptr(const Deleter& deleter);

    ~thread_specific_ptr();

    // [thread.ptr.mem], members
    T* get() const noexcept;
    T* operator->() const noexcept;
    T& operator*() const noexcept;

    [[nodiscard]] T* release() noexcept;
    void reset() noexcept;
    void reset(T* new_value);

    explicit operator bool() const noexcept;

    // disable copy from lvalue
    thread_specific_ptr(const thread_specific_ptr&) = delete;
    thread_specific_ptr& operator=(const thread_specific_ptr&) = delete;
    // disable move
    thread_specific_ptr(thread_specific_ptr&&) = delete;
    thread_specific_ptr& operator=(thread_specific_ptr&&) = delete;

private:
    Deleter del;                    // exposition only
};

} // namespace std
```

33.13.3.1 Constructors, move and assignment [thread.ptr.cons]

thread_specific_ptr() ;

1 *Constraints:* `is_pointer_v<Deleter> == false` and `is_default_constructible_v<Deleter> == true`

2 *Mandates:* `is_nothrow_invocable_v<Deleter, T*> == true`

3 *Preconditions:* `Deleter` meets the *Cpp17DefaultConstructible* requirements and such construction does not exit via an exception.

4 *Effects:*

— Constructs a `thread_specific_ptr` object for storing a pointer to an object of type `T` specific to each thread. Value-initializes `del`.

5 *Postconditions:* `bool(*this) == false`

6 *Throws:*

— `system_error` if an error occurs.

7 *Error conditions:*`resource_unavailable_try_again` – the system lacked the necessary resources to instantiate a `thread_specific_ptr`.

explicit thread_specific_ptr(const Deleter& deleter) ;

1 *Constraints:*`is_constructible_v<Deleter, decltype(deleter)> == true`

2 *Mandates:*`is_nothrow_invocable_v<Deleter, T*> == true`

3 *Preconditions:*`Deleter` meets the *Cpp17CopyConstructible* requirements and such construction does not exit via an exception.

4 *Effects:*

— Constructs a `thread_specific_ptr` object for storing a pointer to an object of type `T` specific to each thread. Initializes `del` from `std::forward<decltype(deleter)>(deleter)`.

5 *Postconditions:*`bool(*this) == false`

6 *Throws:*

— `system_error` if an error occurs.

7 *Error conditions:*`resource_unavailable_try_again` – the system lacked the necessary resources to instantiate a `thread_specific_ptr`.

~thread_specific_ptr() ;

1 *Preconditions:*

— All thread specific instances associated with this `thread_specific_ptr` (except the one associated with the calling thread) must be `nullptr`.

2 *Effects:*

— Calls `this->reset()` to clean up the associated value for the calling thread, and destroys `*this`.

[*Note:* The Precondition avoids the necessity for an implementation to track all thread specific instances associated with each `thread_specific_ptr`. — *end note*]

33.13.3.2 Members [thread.ptr.mem]

T* get() const noexcept ;

1 *Returns:*

— The pointer associated with the calling thread.

T* operator->() const noexcept ;

1 *Returns:*

— `this->get()`

T& operator*() const noexcept ;

1 *Preconditions:*

— `bool(*this) == true`

2 *Returns:*

— `*(this->get())`

[[nodiscard]] T* release() noexcept ;

1 *Effects:*

— Let `ptr` be the current value of `this->get()`.

— Stores `nullptr` as the pointer associated with the calling thread.

2 *Returns:*

— `ptr`

3 *Postconditions:*

— `bool(*this) == false`

void reset() noexcept ;

1 *Effects:* Equivalent to: `reset(nullptr);`

2 *Postconditions:*

— `bool(*this) == false`

void reset(T* new_value) ;

1 *Effects:*

— If `this->get() != new_value && bool(*this)`, let `tempdel` be a temporary copy of `del`. Invokes `tempdel(this->get())`. [*Note:* Calling a copy of `del` avoids potential data races from concurrent executions of `del`. —*end note*]

— Stores `new_value` as the pointer associated with the calling thread.

2 *Postconditions:*

— `this->get() == new_value`

3 *Throws:*

— `system_error` if an error occurs.

4 *Error conditions:*`resource_unavailable_try_again` – the system lacked the necessary resources to store a new thread specific pointer value.

explicit operator bool() const noexcept ;

1 *Effects:* Equivalent to: `return (this->get() != nullptr);`

Header File

Add a new header file to Table 24 in §16.4.2.3 [headers]:

```
<thread_specific_ptr>
```

Feature-test Macro

Add a new feature-test macro to §17.3.2 [version.syn] as indicated:

```
#define __cpp_lib_thread_specific_ptr 202XXXL // also in <thread_specific_ptr>
```

References

[1] P0876R18: fibers without scheduler

[2] N4981: Working Draft, Programming Languages – C++

[3] Boost C++ Libraries: Thread Local Storage