Document number:	P2774R0
Date:	2023-05-08
Project:	Programming Language C++
Audience:	SG1
Reply-to:	Michael Florian Hava[1] <mfh.cpp@gmail.com>

Scoped thread-local storage

Abstract
This paper proposes a library facility for scoped thread-local storage, designed for parallel algorithms when there is no one-to-one mapping between input and output.

Tony Table

Before

```
span<Triangle> input = …;
double max_area = …;

mutex m;
unordered_map<thread::id, vector<Triangle>> tmp;

//process in parallel
for_each(execution::par, input.begin(), input.end(),
  [&](const auto & tria) {
    //get thread-specific storage
    auto & ref{[&] -> vector<Triangle> & {
      const auto tid{this_thread::get_id()};
      const lock_guard lock{m};
      const auto it{tmp.find(tid)};
      if(it != tmp.end()) return it->second;
      return *tmp.emplace(tid, {}).first;
    }()};

    //generating unbounded output
    for(const auto & t : split(tria, max_area))
      ref.emplace_back(t);
  }
);

//post-process results sequentially
for(const auto & tria : tmp | views::join)
  process(tria);

//clear content for future parallel processing
tmp.clear();
```

Proposed

```
span<Triangle> input = …;
double max_area = …;


tls<vector<Triangle>> tmp;

//process in parallel
for_each(execution::par, input.begin(), input.end(),
  [&](const auto & tria) {
    //get thread-specific storage
    auto [ref, _]{tmp.local()};

    //generating unbounded output
    for(const auto & t : split(tria, max_area))
      ref.emplace_back(t);
  }
);

//post-process results sequentially
for(const auto & tria : tmp | views::join)
  process(tria);

//clear content for future parallel processing
tmp.clear();
```

Revisions
R0: Initial version

Motivation
C++17 introduced parallel algorithms to the standard library. The design of said algorithms embodies the popular fork-join model of parallelization. Combining this structured parallelization style with the functional aspects of the "STL" was a perfect match for querying (e.g. `std::find`), in-place transformations (e.g. `std::sort`), and one-to-one transformations (e.g. `std::transform`).

One class of algorithms the standard library never supported (apart from "abusing" `std::for_each`) were one-to-many transformations. Applying the fork-join model to these

[1] RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at

algorithms proves to be difficult as their unbounded nature doesn't lend itself easily to aggregating the results in a singular target object without overt locking.

If no singular result object is needed, the issue of locking could be sidestepped by the usage of `thread_local` variables - but such an approach has extensive hidden costs for all threads and transforms a local issue into a global problem.

We propose an alternative approach providing lazily initialized, thread-specific storage bound to a local object. The proposed `std::tls` does not require expensive locking for concurrent write access, nor does it increase the static memory set of a thread.

Design Space

A `std::tls<T>` composes a concurrent storage for thread-specific instances of `T` and a concurrency-safe initialization function for storage entries. Conceptually it is similar to the following class:

```
template<typename T, typename Allocator = allocator<T>>
class tls {
  mutex m;
  unordered_map<thread::id, T, hash<thread::id>, key_equal<thread::id>, Allocator> storage;
  //NOTE: at the moment none of the standard polymorphic function wrappers has allocator support.
  unmovable_function<Allocator, T() const> init_func;
public:
  // (1) constructors
  tls(Allocator alloc = Allocator{}) noexcept requires is_default_constructible_v<T>;
  tls(T value, Allocator alloc = Allocator{}) requires is_copy_constructible_v<T>;
  tls(auto func, Allocator alloc = Allocator{}) requires is_convertible_v<T, invoke_result_t<decltype(func)>>;

  // (2) not copy- nor moveable
  tls(const tls &) =delete;
  auto operator=(const tls &) -> tls & =delete;
  ~tls() noexcept;

  // (3) modifiers
  [[nodiscard]]
  auto local() -> tuple<T &, bool>; //thread-safe!
  void clear() noexcept;

  // (4) iteration support
  class iterator { … };
  static_assert(forward_iterator<iterator>);

  auto begin() -> iterator;
  auto end() -> iterator;
};
```

Implementations should use more efficient synchronization mechanisms than locking - our reference implementation employs *atomic operations* to represent `storage` similarly to a concurrent hash map with separate chaining.

Constructors

A `std::tls` provides several constructors to specify the initial value a storage entry should be initialized with. Internally, initialization is handled via an initialization function:

Constructor	Initialization function
`tls();`	`[] { return T{}; }`
`tls(T value);`	`[value{move(value)}] { return value; }`
`tls(auto func);`	`func`

Based on our usage experience with similar classes, we propose that `std::tls` should be neither copy- nor movable. Our design rationale for this is as follows: We consider copying a thread-local storage to be semantically ill-formed. Whilst moving the storage entries would be possible, it was never necessary for us. Furthermore, it introduces additional implementation overhead in the form of a *moved-from state*. Such a state would either have to be equivalent to the *empty state* or have to be checked against on every call to `local`. Depending on the selected implementation strategy for `storage` this introduces considerable complexity and may require allocating on move.

Additionally, the proposed `std::tls` internally wraps a *type-erased polymorphic function wrapper*. There are unresolved technical issues concerning allocator support in this category of classes, that led to its removal from `std::function` ([P0302]). Recent classes in this category (`std::move_only_function` and `std::copyable_function`) don't provide allocator support either.

Modifiers

`std::tls` offers two modifiers: `local` and `clear`. The former acts as a getter and retrieves the storage entry of the current thread. If no such element exists, it's constructed using the initialization function. In addition to a reference of the storage entry a `bool` flag is returned that indicates whether the element was constructed during this call. This diverges from the established practice[2] that provides said `bool` flag as an optional output parameter (employing overloading). Given the recent spotlight on C++ safety, we consider the `tuple` to be superior as it prevents the usage of potentially uninitialized parameters.

`clear` removes all previously created storage entries - a cleared `std::tls` is considered semantically equivalent to a newly constructed one.

Iteration support

Contrary to established practice we don't propose reduction operations to combine the thread-local values into a final result. Instead we propose to add iteration support, enabling users to post-process computation results with any STL-style algorithm of their choice.

Impact on the Standard

This proposal is a pure library addition.

Implementation Experience

The proposed design has been implemented at https://github.com/MFHava/P2774.

Proposed Wording

Wording is relative to [N4944]. Additions are presented like this, removals like ~~this~~ and drafting notes like **this**

[version.syn]

```
#define __cpp_lib_tls YYYYMML //also in <tls>
```

[DRAFTING NOTE: Adjust the placeholder value as needed to denote the proposal's date of adoption.]

[thread.general], extend Table [tab:thread.summary]

	Subclause	Header
…	…	…
[futures]	Futures	`<future>`
[thread.storage]	Thread-local storage	`<tls>`

[2] e.g. `combinable` (PPL/TBB)

[thread.storage]

[DRAFTING NOTE: Add a new section in *[thread]*]

??.?? Thread-local storage **[thread.storage]**

??.??.1 General **[thread.storage.general]**

1 ??.?? describes components that provide scoped thread-local storage. Every thread is lazily assigned an associated storage entry. The lifetime of storage entries is decoupled from the lifetime of the respective thread and instead bound to the containing object.
[*Note 1*: The storage can outlive the respective thread or be destroyed before the thread of execution has ended. — *end note*]

??.??. 2 Header <tls> synopsis **[thread.storage.syn]**

```
namespace std {
  // [thread.storage.tls.class], class template tls
  template<class T, class Allocator = allocator<T>> class tls;

  namespace pmr {
    template<class T>
      using tls = std::tls<T, polymorphic_allocator<T>>;
  }
}
```

??.??.3 Class template tls **[thread.storage.tls.class]**

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class tls {
  public:
    using iterator       = implementation-defined;
    using const_iterator = implementation-defined;

    // [thread.storage.tls.ctor], constructors, and destructor
    explicit tls(const Allocator & allocator = Allocator());
    explicit tls(const Type & value, const Allocator & allocator = Allocator());
    explicit tls(Type && value, const Allocator & allocator = Allocator());
    template<class Func>
      explicit tls(Func init, const Allocator & allocator = Allocator());

    tls(const tls&) =delete;
    tls& operator=(const tls &) =delete;
    ~tls();

    // [thread.storage.tls.mod], modifiers
    [[nodiscard]] tuple<T&, bool> local();
    void clear() noexcept;

    // [thread.storage.tls.iter], iteration
    const_iterator begin() const noexcept;
    iterator begin() noexcept;
    const_iterator cbegin() const noexcept;

    const_iterator end() const noexcept;
    iterator end() noexcept;
    const_iterator cend() const noexcept;
  };
}
```

1 The `tls` class template provides thread-specific storage of type T with a custom initialization function.
[*Note 1*: A thread's storage entry may be identified by its `thread::id`. — *end note*]

2 T shall be a cv-unqualified type that meets the *Cpp17Destructible* requirements (***[tab:cpp17.destructible]***).

3 `Allocator` shall be a cv-unqualified type that meets the *Cpp17Allocator* requirements (***[allocator.requirements.general]***) and can be safely used concurrently.

3 `tls::iterator` meets the forward iterator requirements (***[forward.iterators]***) with value type T.

4 `tls::const_iterator` meets the requirements of a constant iterator and those of a forward iterator with value type T.

5 *Recommended practice*: Implementations should avoid high synchronization overhead for concurrent access to storage.

??.??.3.1 Constructors, and destructor **[thread.storage.tls.ctor]**

`explicit tls(const Allocator & allocator = Allocator());`

1 *Constraints*: `is_default_constructible_v<T>` is `true`.

2 *Effects*: As if by:
```
  tls([] { return T(); }, allocator);
```

`explicit tls(const T & value, const Allocator & allocator = Allocator());`

3 *Constraints*: is_copy_constructible_v<T> is true.

4 Effects: As if by:
```
  tls([=] { return value; }, allocator);
```

```
explicit tls(T && value, const Allocator & allocator = Allocator());
```

5 *Constraints*: is_copy_constructible_v<T> is true.

6 Effects: As if by:

```
    tls([value = std::move(value)] { return value; }, allocator);
```

```
template<typename Func>
  explicit tls(Func func, const Allocator & allocator = Allocator());
```

7 *Constraints*: `is_convertible_v<Type, invoke_result_t<Func>>` is `true`.

8 *Preconditions*:

(8.1) — It is safe to call `func` concurrently from multiple threads.

(8.2) — If `func` is a function pointer, `f == nullptr` is `false`.

9 *Effects*: Constructs empty storage using `allocator` and the initialization function with `func`.

```
~tls();
```

10 *Effects*: Destroys all storage entries and the initialization function; any memory obtained is deallocated.

??.??.3.2 Modifiers [thread.storage.tls.mod]

```
[[nodiscard]] tuple<T&, bool> local();
```

1 Let f be `true` if there is no storage entry for the current thread, otherwise `false`.

2 *Synchronization*: Modifications to the storage are synchronized.
[*Note 1*: If a new storage entry is added, iterators previously obtained by `*this` are invalidated. — *end note*]

3 *Postconditions*: There is a storage entry e for the current thread.

4 *Returns*: `e, f`

5 *Throws*: Any exception thrown by initialization of the new storage entry. May throw `bad_alloc` if allocation of the new storage entry fails.

```
void clear() noexcept;
```

6 *Postconditions*: Storage contains no entries.

??.??.3.3 Iteration [thread.storage.tls.iter]

```
const_iterator begin() const noexcept;
iterator begin() noexcept;
const_iterator cbegin() const noexcept;
```

1 *Returns*: An iterator referring to the start of storage.

```
const_iterator end() const noexcept;
iterator end() noexcept;
const_iterator cend() const noexcept;
```

2 *Returns*: An iterator representing the past-the-end of storage.

Acknowledgements

Thanks to RISC Software GmbH for supporting this work. Thanks to Peter Kulczycki for proof reading.