
18.3. Starting a Thread in Detail

Having introduced the high- and low-level interfaces to (possibly) start threads and deal with return values or exceptions, let's summarize the concepts and provide some details not mentioned yet.

Conceptually, we have the following layers to start threads and deal with their return values or exceptions (see Figure 18.1):

- With the low-level interface of class `thread`, we can start a thread. To return data, we need shared variables (global or static or passed as argument). To return exceptions, we could use the type `std::exception_ptr`, which is returned by `std::current_exception()` and can be processed by `std::rethrow_exception()` (see Section 4.3.3, page 52).
- The concept of a *shared state* allows us to deal with return values or exceptions in a more convenient way. With the low-level interface of a `promise`, we can create such a *shared state*, which we can process by using a `future`.
- At a higher level, with class `packaged_task` or `async()`, the *shared state* is automatically created and set with a return statement or an uncaught exception.
- With `packaged_task`, we can create an object with a *shared state* where we explicitly have to program when to start the thread.
- With `std::async()`, we don't have to care when the thread exactly gets started. The only thing we know is that we have to call `get()` when we need the outcome.

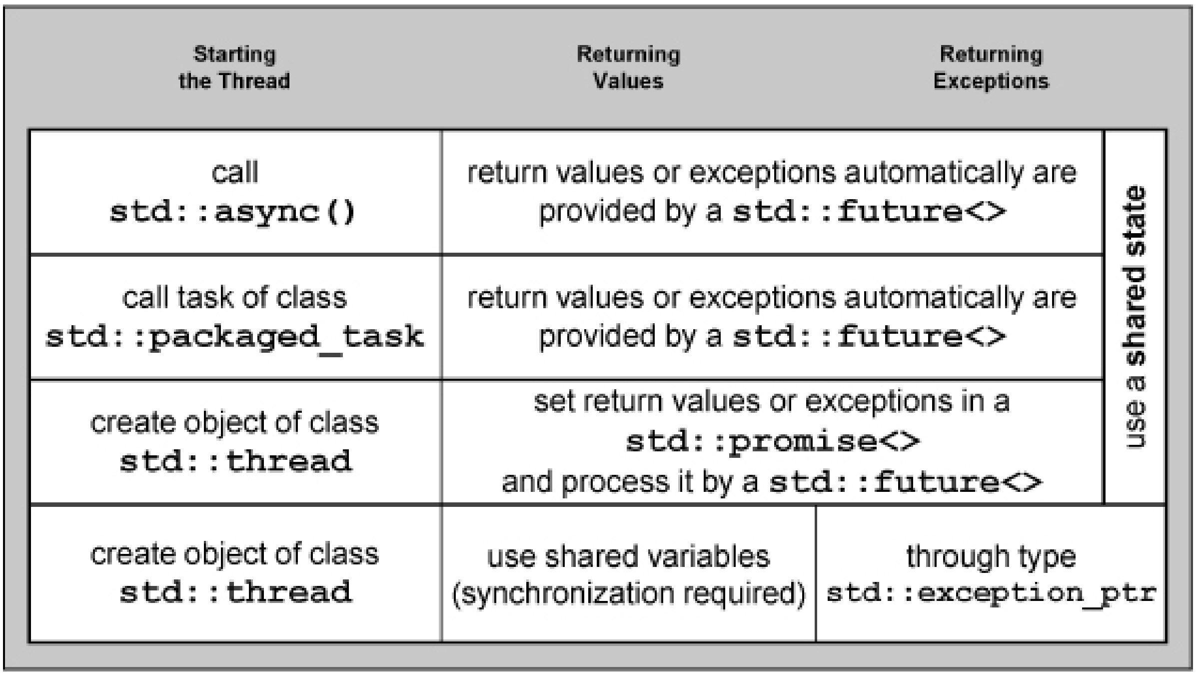

Figure 18.1. Layers of Thread Interfaces

Shared States

As you can see, a central concept used by almost all these features is a *shared state*. It allows the objects that start and control a background functionality (a promise, a packaged task, or `async()`) to communicate with the objects that process its outcome (a future or a shared future). Thus, a shared state is able to hold the functionality to start, some state information, and its outcome (a return value or an exception).

A *shared state* is *ready* when it holds the outcome of its functionality (when a value or an exception is ready for retrieval). A shared state is usually implemented as a reference-counted object that gets destroyed when the last object referring to it *releases* it.

18.3.1. `async()` in Detail

In general, as introduced in Section 18.1, page 946, `std::async()` is a convenience function to start some functionality in its own thread *if possible*. As a result, you can parallelize functionality if the underlying platform supports it but not lose any functionality if it doesn't.

However, the exact behavior of `async()` is complex and highly depends on the launch policy, which can be passed as the first optional argument. For this reason, each of the three standardized forms of how `async()` can be called as described here from an application programmer's point of view:

future **async** `(std::launch::async,` *F func*, *args*...)

- Tries to start *func* with *args* as an asynchronous task (parallel thread).
- If this is not possible, it throws an exception of type `std::system_error` with the error code `std::errc::resource_unavailable_try_again` (see Section 4.3.1, page 43).
- Unless the program aborts, the started thread is guaranteed to finish before the program ends.
- The thread will finish:
  - If `get()` or `wait()` is called for the returned future
  - If the last object that refers to the *shared state* represented by the returned future gets destructed
- This implies that the call of `async()` will block until *func* has finished if the return value of `async()` is not used.

future **async** `(std::launch::deferred,` *F func*, *args*...)

- Passes *func* with *args* as a "deferred" task, which gets synchronously called when `wait()` or `get()` for the returned future gets called.
- If neither `wait()` nor `get()` is called, the task will never start.

future **async** `(` *F func* `,` *args*... `)`

- Is a combination of calling `async()` with launch policies `std::launch:async` and `std::launch::deferred` . According to the current situation, one of the two forms gets chosen. Thus, `async()` will *defer* the call of *func* if an immediate call in `async` launch policy is not possible.
- Thus, if `async()` can start a new thread for *func*, it gets started. Otherwise, *func* is deferred until `get()` or `wait()` gets called for the returned future.
- The only guarantee this call gives is that after calling `get()` or `wait()` for the returned future, *func* will have been called and finished.
- Without calling `get()` or `wait()` for the returned future, *func* might never get called.
- Note that this form of `async()` will not throw a `system_error` exception if it can't call *func* asynchronously (it might throw a system error for other reasons, though).

For all these forms of `async()` , *func* might be a *callable object* (function, member function, function object, lambda; see Section 4.4, page 54). See Section 18.1.2, page 958, for some examples.

Passing a launch policy of `std::launch::async|std::launch::deferred` to `async()` results in the same behavior as passing no launching policy. Passing `0` as launch policy results in undefined behavior (this case is not covered by the C++ standard library, and different implementations behave differently).

18.3.2. Futures in Detail

Class `future<>` ,[10] introduced in Section 18.1, page 946, represents the *outcome* of an operation. It can be a return value or an exception but not both. The outcome is managed in a *shared state*, which in general can be created by `std::async()` , a `std::packaged_task` , or a promise. The outcome might not exist yet; thus, the future might also hold everything necessary to generate the outcome.

[10] Originally, the class was named `unique_future` in the standardization process.

If the future was returned by `async()` (see Section 18.3.1, page 974) and the associated task was *deferred*, `get()` or `wait()` will start it synchronously. Note that `wait_for()` and `wait_until()` do *not* start a deferred task.

The outcome can be retrieved only once. For this reason, a future might have a valid or invalid state: *valid* means that there is an associated operation for which the result or exception was not retrieved yet.

Table 18.1 lists the operations available for class `future<>` .

Table 18.1. Operations of Class *`future<>`*

Operation	Effect
future f	Default constructor; creates a future with an invalid state
future f(rv)	Move constructor; creates a new future, which gets the state of *rv*, and invalidates the state of *rv*
f.~*future*()	Destroys the state and destroys `*this`
f = *rv*	Move assignment; destroys the old state of *f*, gets the state of *rv*, and invalidates the state of *rv*
f.valid()	Yields `true` if *f* has a valid state, so you can call the following member functions
f.get()	Blocks until the background operation is done (forcing a *deferred* associated functionality to start synchronously), yields the result (if any) or raises any exception that occurred, and invalidates its state
f.wait()	Blocks until the background operation is done (forcing a *deferred* associated functionality to start synchronously)
f.wait_for(*dur*)	Blocks for duration *dur* or until the background operation is done (a *deferred* thread is *not* forced to start)
f.wait_until(*tp*)	Blocks until timepoint *tp* or until the background operation is done (a *deferred* thread is *not* forced to start)
f.share()	Yields a `shared_future` with the current state and invalidates the state of *f*

Note that the return value of `get()` depends on the type `future<>` is specialized with:

- If it is `void`, `get()` also has type `void` and returns nothing.
- If the future is parametrized with a reference type, `get()` returns a reference to the return value.
- Otherwise, `get()` returns a copy or move assigns the return value, depending on whether the return type supports move assignment semantics.

Note that you can call `get()` only once, because `get()` invalidates the future's state.

For a future that has an invalid state, calling anything else but the destructor, the move assignment operator, or `valid()` results in undefined behavior. For this case, the standard recommends throwing an exception of type `future_error` (see Section 4.3.1, page 43) with the code `std::future_errc::no_state`, but this is not required.

Note that neither a copy constructor nor a copy assignment operator is provided, ensuring that no two objects can share the state of a background operation. You can move the state to another future object only by calling the move constructor or the move assignment operator. However, the state of background tasks can be shared in multiple objects by using a `shared_future` object, which `share()` yields.

If the destructor is called for a future that is the last owner of a shared state and the associated task has started but not finished yet, the destructor blocks until the end of the task.

18.3.3. Shared Futures in Detail

Class `shared_future<>` (introduced in Section 18.1.3, page 960) provides the same semantics and interface as class `future` (see Section 18.3.2, page 975) with the following differences:

- Multiple calls of `get()` are allowed. Thus, `get()` does not invalidate its state.
- Copy semantics (copy constructor, copy assignment operator) are supported.
- `get()` is a *constant* member function returning a `const` reference to the value stored in the *shared state* (which means that you have to ensure that the lifetime of the returned reference is shorter than the *shared state*). For class `std::future`, `get()` is a *nonconstant* member function returning a move-assigned copy (or a copy if that's not supported), unless the class is specialized by a reference type.
- Member `share()` is not provided.

The fact that the return value of `get()` is not copied creates some risks. Besides lifetime issues, data races are possible. Data races occur with unclear order of conflicting actions on the same data, such as nonsynchronized reads and writes from multiple threads, and result in undefined behavior (see Section 18.4.1, page 982).

The same problem applies to exceptions. One example discussed during the standardization was when an exception was caught by reference and then modified:

```
try {
    shared_future<void> sp = async(f);
    sp.get();
}
catch (E& e) {
    e.modify();      // risk of undefined behavior due to a data race
}
```

This code introduces a data race if another thread processes the exception. To solve this issue, it was proposed to require that `current_exception()` and `rethrow_exception()`, which are used internally to pass exceptions between threads, create copies of the exceptions. However, the costs for this change were considered too high. As a result, programmers have to know what they're doing when dealing with nonconstant references used in different threads.

18.3.4. Class `std::promise` in Detail

An object of class `std::promise`, introduced in Section 18.2.2, page 969, is provided to temporarily hold a (return) value or an exception. Or, in general, a promise can hold a *shared state* (see Section 18.3, page 973). The *shared state* is said to be *ready* if it holds a value or an exception. Table 18.2 lists the operations available for class `promise`.

Table 18.2. Operations of Objects of Class `promise`

Operation	Effect
promise *p*	Default constructor; creates a promise with shared state
promise *p*(allocator_arg,*alloc*)	Creates a promise with *shared state*, which uses *alloc* as allocator
promise *p*(*rv*)	Move constructor; creates a new promise object, which gets the state of *rv* and removes the *shared state* from *rv*
p.~*promise*()	Releases the *shared state* and if it is not ready (no value or exception), stores a std::future_error exception with condition broken_promise
p = *rv*	Move assignment; move assigns the state of *rv* to *p* and if *p* was not ready, stores a std::future_error exception with condition broken_promise there
swap(*p1*,*p2*)	Swaps states of *p1* and *p2*
p1.swap(*p2*)	Swaps states of *p1* and *p2*
p.get_future()	Yields a future object to retrieve the *shared state* (outcome of a thread)
p.set_value(*val*)	Sets *val* as (return) value and makes the state *ready* (or throws std::future_error)
p.set_value_at_thread_exit(*val*)	Sets *val* as (return) value and makes the state *ready* at the end of the current thread (or throws std::future_error)
p.set_exception(*e*)	Sets *e* as exception and makes the state *ready* (or throws std::future_error)
p.set_exception_at_thread_exit(*e*)	Sets *e* as exception and makes the state *ready* at the end of the current thread (or throws std::future_error)

Note that you can call `get_future()` only once. A second call throws a `std::future_error` with the error code `std::future_errc::future_already_retrieved`. In general, if no *shared state* is associated, a `std::future_error` with the error code `std::future_errc::no_state` might be thrown.

All member functions that set the value or exception are thread safe. That is, they behave as if a mutex ensures that only one of them can update the *shared state* at a time.

18.3.5. Class `std::packaged_task` in Detail

Class `std::packaged_task<>` is provided to hold both some functionality to perform and its outcome (the so-called *shared state* of the functionality, see Section 18.3, page 973), which might be a return value or an exception raised by the functionality. You can initialize the packaged task with the associated functionality. Then, you can call this functionality by calling operator `()` for the packaged task. Finally, you can process the outcome by getting a future for the packaged task. Table 18.3 lists the operations available for class `packaged_task`.

Table 18.3. Operations of Class `packaged_task<>`

Operation	Effect
packaged_task `pt`	Default constructor; creates a packaged task with no *shared state* and no stored task
packaged_task `pt(`*f*`)`	Creates an object for the task *f*
packaged_task `pt(`*alloc*`,`*f*`)`	Creates an object for the task *f* using allocator *alloc*
packaged_task `pt(`*rv*`)`	Move constructor; moves the packaged task *rv* (task and state) to `pt` (*rv* has no *shared state* afterward)
`pt.~`*packaged_task*`()`	Destroys `*this` (might make *shared state* ready)
`pt = `*rv*	Move assignment; move assigns the packaged task `rv` (task and state) to `pt` (*rv* has no *shared state* afterward)
`swap(pt1,pt2)`	Swaps packaged tasks
`pt1.swap(pt2)`	Swaps packaged tasks
`pt.valid()`	Yields `true` if `pt` has a *shared state*
`pt.get_future()`	Yields a future object to retrieve the *shared state* (outcome of the task)
`pt(`*args*`)`	Calls the task (with optional arguments) and makes the *shared state* ready
`pt.make_ready_at_thread_exit(`*args*`)`	Calls the task (with optional arguments) and at thread exit makes the *shared state* ready
`pt.reset()`	Creates a new *shared state* for `pt` (might make the old *shared state* ready)

Any exception caused by the constructor taking the task, such as if no memory is available, is also stored in its *shared state*.

Trying to call the task or `get_future()` if no state is available throws a `std::future_error` (see Section 4.3.1, page 43) with the error code `std::future_errc::no_state`. Calling `get_future()` a second time throws an exception of type `std::future_error` with the error code `std::future_errc::future_already_retrieved`. Calling the task a second time throws a `std::future_error` with the error code `std::future_errc::promise_already_satisfied`.

The destructor and `reset()` *abandon* the *shared state*, which means that the packaged task releases the *shared state* and, if the *shared state* was not ready yet, makes the state ready with a `std::future_error` with error code `std::future_errc::broken_promise` stored as outcome.

As usual, the `make_ready_at_thread_exit()` function is provided to ensure the cleanup of local objects and other stuff of a thread ending the task before the result gets processed.

18.3.6. Class `std::thread` in Detail

An object of class `std::thread`, introduced in Section 18.2.1, page 964, is provided to start and represent a thread. These objects are intended to map one-to-one with threads provided by the operating system. Table 18.4 lists the operations available for class `thread`.

Table 18.4. Operations of Objects of Class `thread`

Operation	Effect
thread t	Default constructor; creates a *nonjoinable* thread object
thread $t(f,...)$	Creates a thread object, representing *f* started as thread (with additional args), or throws `std::system_error`
thread $t(rv)$	Move constructor; creates a new thread object, which gets the state of *rv*, and makes *rv* *nonjoinable*
t.~*thread*()	Destroys `*this`; calls `std::terminate()` if the object is *joinable*
t = *rv*	Move assignment; move assigns the state of *rv* to *t* or calls `std::terminate()` if *t* is *joinable*
t.`joinable()`	Yields `true` if *t* has an associated thread (is *joinable*)
t.`join()`	Waits for the associated thread to finish (throws `std::system_error` if the thread is not *joinable*) and makes the object *nonjoinable*
t.`detach()`	Releases the association of *t* to its thread while the thread continues (throws `std::system_error` if the thread is not *joinable*) and makes the object *nonjoinable*
t.`get_id()`	Returns a unique `std::thread::id` if *joinable* or `std::thread::id()` if not
t.`native_handle()`	Returns a platform-specific type `native_handle_type` for nonportable extensions
t1.`swap(`*t2*`)`	Swaps the state of *t1* and *t2*
`swap(`*t1*, *t2*`)`	Swaps the state of *t1* and *t2*

The association between a thread object and a thread starts by initializing (or move copy/assign) a *callable object* (see Section 4.4, page 54) to it with optional additional arguments. The association ends either with `join()` (waiting for the outcome of the thread) or with `detach()` (explicitly losing the association to the thread). One or the other must be called before the lifetime of a thread object ends or a new thread gets move assigned. Otherwise, the program aborts with `std::terminate()` (see Section 5.8.2, page 162).

If the thread object has an associated thread, it is said to be *joinable*. In that case, `joinable()` yields `true`, and `get_id()` yields a thread ID that differs from `std::thread::id()`.

Thread IDs have their own type `std::thread::id`. Its default constructor yields a unique ID representing “no thread.” `thread::get_id()` yields this value if no thread is associated or another unique ID if the thread object is associated with a thread (is *joinable*). The only supported operations for thread IDs are to compare them or to write them to an output stream. In addition, a hash function is provided to manage thread IDs in unordered containers (see Section 7.9, page 356). A thread ID of a terminated thread might be reused again. Don't make any other assumptions about thread IDs other than that, especially regarding their values. See Section 18.2.1, page 968, for details.

Note that detached threads should not access objects whose lifetimes have ended. This implies the problem that when ending the program, you have to ensure that detached threads don't access global/static objects (see Section 18.2.1, page 967).

Number of Available Threads

In addition, class `std::thread` provides a static member function to query a hint for the possible number of parallel threads:

`unsigned int` **std::thread::hardware_concurrency** `()`

- Returns the number of possible threads.
- This value is just a hint and does not guarantee to be exact.
- Returns `0` if the number is not computable or well defined.

18.3.7. Namespace `this_thread`

For any thread, including the main thread, `<thread>` declares namespace `std::this_thread`, which provides the thread-specific global functions listed in Table 18.5.

Table 18.5. Thread-Specific Operations of Namespace *`std::this_thread`*

Operation	Effect
`this_thread::get_id()`	Yields the ID of the current thread
`this_thread::sleep_for(`*dur*`)`	Blocks the thread for duration *dur*
`this_thread::sleep_until(`*tp*`)`	Blocks the thread until timepoint *tp*
`this_thread::yield()`	Hint to reschedule to the next thread

Note that `sleep_for()` and `sleep_until()` usually will differ when dealing with system-time adjustments (see Section 5.7.5, page 160, for details).

The operation `this_thread::yield()` is provided to give a hint to the system that it is useful to give up the remainder of the current thread's time slice so that the runtime environment can reschedule to allow other threads to run. One typical example is to give up control when you wait or "poll" for another thread (see Section 18.1.1, page 955) or an atomic flag to be set by another thread (see Section 18.4.3, page 986):[11]

[11] Thanks to Bartosz Milewski for this example.

```
while (!readyFlag) {    // loop until data is ready
    std::this_thread::yield();
}
```

As another example, when you fail to get a lock or a mutex while locking multiple locks/mutexes at a time, you can make the application faster by using `yield()` prior to trying the locks/mutexes in a different order.[12]

[12] Thanks to Howard Hinnant for this example.