Tasks

Edit New Page

Jump to bottom

aribibek edited this page on 10 Jan 2020 · 7 revisions

Task objects

Task objects are the central component of Async++. The `task<T>` class represents a value of type `T` which may or may not have been produced yet. `T` can be void, in which case the task simply acts as a signal to indicate whether an event has occured.

The most common way to create a task is to use the `spawn()` function, which will run a given function asynchronously on a thread pool and return a task which is linked to the result of the function.

The result of a task can be obtained using the `get()` member function, which will wait for the task to complete if the value is not yet ready. Note that a task object can only be read once: After `get()` is called, the task object is cleared. If you want to call `get()` multiple times, use a `shared_task<T>` instead.

You can wait for a task to complete without retrieving its value (and without clearing the task object) by calling the `wait()` member function. You can also poll whether the task has completed by calling the `ready()` member function, which doesn't block.

It is also possible to create a task already containing a predefined value, using the `make_task()` function. In that case the task is already considered to have completed and no waiting is necessary. Similarly, you can use `make_exception_task<T>()` to create a task initialized with an exception.

Example:

```
// Create a task which runs asynchronously
auto my_task = async::spawn([] {
    return 42;
});

// Do other stuff while the task is not finished
while (!my_task.ready()) {
    // Do stuff...
}

// Wait for the task to complete without getting the result
my_task.wait();

// Wait for the task to complete and get the result
int answer = my_task.get();

// Create a task with a preset value
auto my_task2 = async::make_task(42);

// Print the value stored in the task
std::cout << my_task2.get() << std::endl;
```

Continuations

The issue with blocking is that it is hard to reliably predict when a task will complete. It is possible to schedule a function to run immediately after a task finishes by using the `then()` member function.

The function passed can take one of the following as parameters:

- The result type of the parent task, if the parent task has a non-void result type.
- No parameters, in which case any result from the parent task is discarded.
- A reference to the parent task type, which allows handling exception results from the parent task.

Because `then()` returns a new task object linked to the continuation function, it is possible to chain continuations by calling `then()` on the returned task object.

As with `get()`, the task object is cleared once `then()` is called. Use a `shared_task<T>` to add multiple continuations to a task.

Example:

```
// Spawn a task
auto t1 = async::spawn([] {
    return 42;
});

// Chain a value-based continuation
auto t2 = t1.then([](int result) {
    return result;
});

// Chain a task-based continuation
t2.then([](task<int> parent) {
    std::cout << parent.get() << std::endl;
});

// Equivalent code with direct chaining
async::spawn([] {
    return 42;
}).then([](int result) {
    return result;
}).then([](task<int> parent) {
    std::cout << parent.get() << std::endl;
});
```

Composition

The `when_all()` function returns a task which finishes when all of the tasks in a set have completed. There are 2 forms of this function:

- If the set of tasks is given directly as parameters, the returned task is of type `async::task<std::tuple<T...>>` where `T` is the task type that was given to the function (`task<U>` or `shared_task<U>`).
- If the set of tasks is given as a range or pair of iterators, the returned task is of type `async::task<std::vector<T>>` where `T` is the type of the tasks in the range (`task<U>` or `shared_task<U>`). This type is identical for all tasks (by definition, since a range can only contain a single type).

The `when_any()` function returns a task which finishes when at least one of the tasks in a set have completed. As with `when_all()` , there are 2 forms of this function:

- If the set of tasks is given directly as parameters, the returned task is of type `async::task<async::when_any_result<std::tuple<T...>>>` where `T` is the task type that was given to the function (`task<U>` or `shared_task<U>`).
- If the set of tasks is given as a range or pair of iterators, the returned task is of type `async::task<async::when_any_result<std::vector<T>>>` where `T` is the type of the tasks in

the range (`task<U>` or `shared_task<U>`). This type is identical for all tasks (by definition, since a range can only contain a single type).

The `when_any_result` struct contains two elements:

- `index` which is the index of one of the tasks that has completed. It may not necessarily be the first task that has completed.
- `tasks` which is a vector or tuple of all the task objects that were given as input.

Example:

```
// Using when_any to find task which finishes first
async::task<char> tasks[] = {
    async::spawn([] {return 'A';}),
    async::spawn([] {return 'B';}),
    async::spawn([] {return 'C';})
};
async::when_any(tasks).then(
[](async::when_any_result<std::vector<async::task<char>>> result) {
    std::cout << "Task " << result.index << " finished with value "
              << result.tasks[result.index].get() << std::endl;
});

// Using when_all to combine results of multiple tasks
auto a = async::spawn([] {return std::string("Hello ");});
auto b = async::spawn([] {return std::string("World!");});
async::when_all(a, b).then(
[](std::tuple<async::task<std::string>, async::task<std::string>> result) {
    std::cout << std::get<0>(result).get() << std::get<1>(result).get() << std::endl;
});

// Output:
// Task 0 finished with value A
// Hello World!
```

Exceptions

Exceptions thrown in a task function are rethrown when that task's `get()` function is called. If a task has continuations, exceptions are propagated to the continuations in different ways depending on the type of continuations:

- Value-based continuations, which take the result type of the previous task as parameter, are not executed if the parent task throws. Instead the parent's exception is directly copied to the result of the continuation.
- Task-based continuations, which take a reference to the parent task as parameter, are executed even if the parent task throws. The parent's exception can be handled by calling `get()` on the parent task.

`when_any()` will return an exception if the first task to finish throws an exception. If later tasks throw exceptions, they are ignored.

`when_all()` will return an exception if any task throws. Note that this may cause the returned task to appear finished when other tasks have not yet finished.

```
async::spawn([] {
    throw std::runtime_error("Some error");
}).then([](int result) {
    // This is not executed because it is a value-based continuation
}).then([](task<void> t) {
    // The exception from the parent task is propagated through value-based
    // continuations and caught in task-based continuations.
    try {
        t.get();
    } catch (const std::runtime_error& e) {
        std::cout << e.what() << std::endl;
    }
});
```

Cancellation

Async++ does not provide built-in cancellation support, instead it provides tools that allow you to specify *interruption points* where your task can be safely canceled. The `cancellation_token` class defines a simple boolean flag which indicates whether work should be canceled. You can then use the `interruption_point()` function with a cancellation token to throw a `task_canceled` exception if the token has been set.

Example:

```
// Create a token
async::cancellation_token c;

auto t = async::spawn([&c] { // Capture a reference to the token
    // Throw an exception if the task has been canceled
    async::interruption_point(c);

    // This is equivalent to the following:
    // if (c.is_canceled())
    //     async::cancel_current_task(); // throws async::task_canceled
});

// Set the token to cancel work
c.cancel();

// Because the task and c.cancel() are executing concurrently, the token
// may or may not be canceled by the time the task reaches the interruption
// point. Depending on which comes first, this may throw a
```

```
// async::task_canceled exception.
t.get();
```

Shared tasks

Normal `task<T>` objects are single-use: once `get()` or `then()` are called, they become empty and any further operation on them is an error except assigning a new task to them (from `spawn()` or from another task object). In order to use a task multiple times, it is possible to convert it to a shared task by using the `share()` member function. This causes the original task to become empty, but returns a `shared_task<T>` which can have its result retrieved multiple times and multiple continuations added to it. It also becomes copyable and assignable, as opposed to the basic task class which is move-only. The downside of shared tasks is that using them involves extra overhead due to reference counting, and task results are copied instead of moved when using `get()` or `then()`.

Example:

```
// Parent task, note the use of .shared() to get a shared task
auto t = async::spawn([] {
    std::cout << "Parent task" << std::endl;
}).share();

// First child, using value-based continuation
t.then([] {
    std::cout << "Child task A" << std::endl;
});

// Second child, using task-based continuation
t.then([](async::shared_task<void>) {
    std::cout << "Child task B" << std::endl;
});
```

Task unwraping

Sometimes it is necessary for a task to wait for another task to complete before returning a value. For example, a task might be waiting for file I/O (wrapped in a task) to complete before it can indicate it has completed. While it would be possible to call `get()` or `wait()` on the innner task, this would cause the thread to block while waiting for the task to complete.

This problem can be solved using task unwraping: when a task function returns a task object, instead of setting its result to the task object, the inner task will "replace" the outer task. This means that the outer task will complete when the inner task finishes, and will acquire the result of the inner task.

Example:

```
// The outer task is task<int>, and its result is set when the inner task
// completes
async::spawn([] {
    std::cout << "Outer task" << std::endl;
    // Return a task<int>
    return async::spawn([] {
        std::cout << "Inner task" << std::endl;
        return 42;
    });
}).then([](int result) {
    std::cout << "Continuation task" << std::endl;
    std::cout << result << std::endl;
});

// Output:
// Outer task
// Inner task
// Continuation task
// 42
```

Event tasks

Sometimes it is necessary to wait for an external event to happen, or some combination of tasks to complete. Async++ allows you to define custom task objects, which can be set to any value or exception arbitrarily, by using the `event_task<T>` class. You can retrieve a task object associated with the event by calling the `get_task()` member function. The task can be set using the `set()` member function, or it can be set to an exception using the `cancel()` or `set_exception()` functions. All set functions return a bool to indicate whether the value was sucessfully set: they will return false if a value has already been set since an event can only be set once.

Example:

```
// Create an event
async::event_task<int> e;

// Get a task associated with the event
auto t = e.get_task();

// Add a continuation to the task
t.then([](int result) {
    std::cout << result << std::endl;
});

// Set the event value, which will cause the continuation to run
e.set(42);

// To set an exception:
// e.cancel();
```

```
// e.set_exception(std::make_exception_ptr(async::task_canceled));
// These are equivalent but cancel is slightly more efficient
```

Local tasks

Sometimes maximum performance is necessary and not all of the features provided by Async++ are required. The `local_task<F>` class provides a non-copyable, non-movable task type which resides entirely on the stack. It requires no memory allocations but only support a restricted set of operations: it doesn't support continuations and composition, the only operations allowed on it are `get()`, `wait()` and `ready()`, and it has an implicit `wait()` in its destructor. Because it is non-movable, it must be captured directly from the return value of `local_spawn()` by rvalue-reference, as shown in the example.

Example:

```
auto&& t = async::local_spawn([] {
    std::cout << "Local task" << std::endl;
});
```