API Reference

Jump to bottom

Amanieu d'Antras edited this page on Feb 12, 2015 · 16 revisions

task

```
class task<Result> {
    // Result type of the task
    typedef Result result_type;

    // Create an empty task object (operator bool returns false)
    task();

    // Movable but not copyable
    task(const task&) = delete;
    task(task&&);
    task& operator=(const task&) = delete;
    task& operator=(task&&);

    // Destroy the task object. This detaches the task and does not wait for
    // it to finish.
    ~task();

    // Returns true if this task object contains a reference to a valid
    // task. If this is not the case then all other functions throw a
    // std::invalid_argument exception.
    explicit operator bool() const;

    // Waits for the task to complete and then retrieves the result of the
    // task. If the task contains an exception then that exception is
    // rethrown. The result is moved out of the task, and then the task is
    // cleared (operator bool returns false).
    Result get();

    // Waits for the task to have finished executing
```

```
    void wait() const;

    // Returns whether the task has finished executing
    bool ready() const;

    // Returns whether the task was canceled due to an exception
    bool canceled() const;

    // Returns the std::exception_ptr associated with this task if it
    // was canceled by an exception. This calls wait() internally.
    std::exception_ptr get_exception() const;

    // Register a continuation function to be run after this task and
    // invalidate the task object. The function can take a parameter
    // of Result or task<Result>, and will be scheduled with the
    // given scheduler.
    task<T> then(Func f); // T = result type of f
    task<T> then(Sched& sched, Func f); // T = result type of f

    // Turn this task object into a shared_task<T>. This will
    // invalidate the current task object.
    shared_task<Result> share();
};

// shared_task<T> is almost identical to task<T>, except:
// - shared_task<T> is copyable, copies all refer to the same task
// - shared_task<T> doesn't get invalidated on get() and then()
// - get() returns a const reference to the result
class shared_task<Result> {
    typedef Result result_type;
    shared_task();
    shared_task(const shared_task&);
    shared_task(shared_task&&);
    shared_task& operator=(const shared_task&);
    shared_task& operator=(shared_task&&);
    ~shared_task();
    explicit operator bool() const;
    void get() const; // If Result is void
    Result& get() const; // If Result is a reference type
    const Result& get() const; // Otherwise
    void wait() const;
    bool ready() const;
    bool canceled() const;
    std::exception_ptr get_exception() const;
    task<T> then(Func f) const; // T = result type of f
```

```
    task<T> then(Sched& sched, Func f) const; // T = result type of f
};

// Create a completed task containing a value
task<T> make_task(T value);
task<T&> make_task(std::reference_wrapper<T> value);
task<void> make_task();

// Create a canceled task containing an exception
task<T> make_exception_task<T>(std::exception_ptr except);

// Spawn a task to run the given function, optionally using the given
// scheduler instead of the default.
task<T> spawn(Func f); // T = result type of f
task<T> spawn(Sched& sched, Func f); // T = result type of f
```

local_task

```
class local_task<Func> {
    typedef Result result_type;

    // Local tasks can only be created using local_spawn()
    local_task() = delete;

    // Local tasks are non-movable and non-copyable
    local_task(const local_task&) = delete;
    local_task(local_task&&) = delete;
    local_task& operator=(const local_task&) = delete;
    local_task& operator=(local_task&&) = delete;

    // The destructor implicitly waits for the task to finish
    ~local_task();

    // Waits for the task to complete and then retrieves the result of the
    // task. If the task contains an exception then that exception is
    // rethrown. The result is moved out of the task.
    void get(); // If Result is void
    Result& get(); // If Result is a reference type
    Result get(); // Otherwise

    // Waits for the task to have finished executing
    void wait() const;
```

```
    // Returns whether the task has finished executing
    bool ready() const;
};

// Spawn a local task to run the given function, optionally using the given
// scheduler instead of the default. Note that because local_task is
// non-movable, the result of this function must be captured using auto&&.
local_task<Func> local_spawn(Func f);
local_task<Func> local_spawn(Sched& sched, Func f);
```

event_task

```
struct abandoned_event_task {};

class event_task<Result> {
    // Creates an event_task initialized with a new event
    event_task();

    // Movable but not copyable
    event_task(const event_task&) = delete;
    event_task(event_task&&);
    event_task& operator=(const event_task&) = delete;
    event_task& operator=(event_task&&);

    // If a result has not been set, the event is canceled with an
    // abandoned_event_task exception.
    ~event_task();

    // Get a task associated with this event. This can only be called once.
    task<Result> get_task();

    // Set the value of the event. The event can only be set once.
    bool set() const; // If Result is void
    bool set(Result& r) const; // If Result is a reference type
    bool set(Result&& r) const; // Otherwise
    bool set(const Result& r) const; // Otherwise

    // Cancel the event with an exception. The event can only be set once.
    bool set_exception(std::exception_ptr except) const;
};
```

when_all/when_any

```
// Result type returned by when_any
struct when_any_result<Result> {
    // Index of the task that completed first
    std::size_t index;

    // Vector or tuple of results
    Result result;
};

// Return a task which is completed when any one of the given tasks is
// completed. The result will contain the index a task that completed
// and a list of all the tasks that were passed in. For the variadic
// form, the tasks are allowed to have different types.
task<when_any_result<std::tuple<task<T>/shared_task<T>...>>>
when_any(task<T>/shared_task<T>... tasks);
task<when_any_result<std::vector<task<T>/shared_task<T>>>>
when_any(Iter begin, Iter end);
task<when_any_result<std::vector<task<T>/shared_task<T>>>>
when_any(Range tasks);

// Return a task which is completed when all of the given tasks are
// completed. The result will contain a list of all the tasks that
// were passed in. For the variadic form, the tasks are allowed to
// have different types.
task<std::tuple<task<T>/shared_task<T>...>>
when_all(task<T>/shared_task<T>... tasks);
task<std::vector<task<T>/shared_task<T>>> when_all(Iter begin, Iter end);
task<std::vector<task<T>/shared_task<T>>> when_all(Range tasks);
```

Cancellation

```
// Exception thrown by cancel_current_task
struct task_canceled {};

// A cancellation token is just a boolean flag that indicates whether to
// cancel a set of task. It must be explicitly checked by tasks.
class cancellation_token {
    // A token is initialized to the 'not canceled' state
    cancellation_token();
```

```
    // Tokens are non-movable and non-copyable
    cancellation_token(const cancellation_token&) = delete;
    cancellation_token(cancellation_token&&) = delete;
    cancellation_token& operator=(const cancellation_token&) = delete;
    cancellation_token& operator=(cancellation_token&&) = delete;

    // Returns whether the token has been canceled
    bool is_canceled() const;

    // Sets the token to the canceled state
    void cancel();

    // Reset the token to a non-canceled state
    void reset();
};

// Throws a task_canceled exception if the token is canceled
void interruption_point(const cancellation_token& token);
```

Ranges and partitioners

```
// Range object representing 2 iterators
class range<Iter> {
    Iter begin() const;
    Iter end() const;
};

// Create a range from 2 iterators
range<Iter> make_range(Iter begin, Iter end);

// Integer range between 2 integers
class int_range<T> {
    class iterator;
    iterator begin() const;
    iterator end() const;
};

// Create an integer range
int_range<T> irange(T begin, T end);

// Partitioners which wrap a range and split it between threads. A
```

```
// partitioner is just a range with an additional split() function.

// A simple partitioner which splits until a grain size is reached. If a
// grain size is not specified, one is chosen automatically.
<detail> static_partitioner(Range&& range);
<detail> static_partitioner(Range&& range, size_t grain);

// A more advanced partitioner which initially divides the range into one
// chunk for each available thread. The range is split further if a chunk
// gets stolen by a different thread.
<detail> auto_partitioner(Range&& range);

// Convert a range to a partitioner. This is a utility function for
// implementing new parallel algorithms. If the argument is already a
//partitioner then it is simply passed on.
decltype(auto_partitioner(range)) to_partitioner(Range&& range);
Partitioner&& to_partitioner(Partitioner&& partitioner);
```

Parallel algorithms

```
// Run a set of functions in parallel, optionally using a scheduler
void parallel_invoke(Func&&... funcs);
void parallel_invoke(scheduler& sched, Func&&... funcs);

// Run a function over a range in parallel. The range parameter can also be
// a partitioner to explicitly control how jobs are distributed to threads.
void parallel_for(Range&& range, const Func& func);
void parallel_for(scheduler& sched, Range&& range, const Func& func);

// Reduce a range in parallel. The range parameter can also be a partitioner
// to explicitly control how jobs are distributed to threads.
Result parallel_reduce(Range&& range, const Result& initial,
                       const ReduceFunc& reduce);
Result parallel_reduce(scheduler& sched, Range&& range,
                       const Result& initial, const ReduceFunc& reduce);

// Apply a function to a range and reduce it in parallel. The range
// parameter can also be a partitioner to explicitly control how jobs are
// distributed to threads.
Result parallel_map_reduce(Range&& range, const Result& initial,
                           const MapFunc& map, const ReduceFunc& reduce);
Result parallel_map_reduce(scheduler& sched, Range&& range,
```

```
                             const Result& initial, const MapFunc& map,
                             const ReduceFunc& reduce);
```

Schedulers

```
// Default scheduler used when a scheduler isn't explicitly specified.
// This can be overriden by setting the LIBASYNC_CUSTOM_DEFAULT_SCHEDULER
// preprocessor macro. By default this calls default_threadpool_scheduler().
<detail>& default_scheduler();

// Built-in thread pool scheduler with a size that is configurable from the
// LIBASYNC_NUM_THREADS environment variable. If that variable does not
// exist then the number of CPUs in the system is used instead.
threadpool_scheduler& default_threadpool_scheduler();

// Scheduler that runs tasks inline in the calling thread
<detail>& inline_scheduler();

// Scheduler that runs tasks in a new thread. Note that this does not wait
// for threads to finish on shutdown.
<detail>& thread_scheduler();

// Scheduler that holds a list of tasks which can then be explicitly
// executed by a thread. Both adding and running tasks are thread-safe
// operations.
class fifo_scheduler {
    fifo_scheduler();

    // Movable but not copyable
    fifo_scheduler(const fifo_scheduler&) = delete;
    fifo_scheduler(fifo_scheduler&&);
    fifo_scheduler& operator=(const fifo_scheduler&) = delete;
    fifo_scheduler& operator=(fifo_scheduler&&);

    // Note that any remaining tasks are not executed
    ~fifo_scheduler();

    // Add a task to the queue
    void schedule(task_run_handle t);

    // Try running one task from the queue. Returns false if the queue was
    // empty.
```

```
    bool try_run_one_task();

    // Run all tasks in the queue
    void run_all_tasks();
};

// Scheduler that runs tasks in a work-stealing thread pool of the given
// size. Note that destroying the thread pool before all tasks have
// completed may result in some tasks not being executed.
class threadpool_scheduler {
    // Create a new thread pool with the given number of threads.
    threadpool_scheduler(size_t num_threads);

    // Movable but not copyable
    threadpool_scheduler(const threadpool_scheduler&) = delete;
    threadpool_scheduler(threadpool_scheduler&&);
    threadpool_scheduler& operator=(const threadpool_scheduler&) = delete;
    threadpool_scheduler& operator=(threadpool_scheduler&&);

    // Any tasks that are currently executing are finished, but any tasks
    // added after destruction has begun are not executed.
    ~threadpool_scheduler();

    // Run a task on the thread pool
    void scheduler(task_run_handle t);
};

// Improved version of std::hardware_concurrency:
// - It never returns 0, 1 is returned instead.
// - It is guaranteed to remain constant for the duration of the program.
std::size_t hardware_concurrency();
```

Custom schedulers

```
// Handle representing a task that needs to be run
class task_run_handle {
    // Create an invalid handle
    task_run_handle();

    // Movable but not copyable
    task_run_handle(const task_run_handle&) = delete;
    task_run_handle(task_run_handle&&);
```

```
    task_run_handle& operator=(const task_run_handle&) = delete;
    task_run_handle& operator=(task_run_handle&&);

    // Check if a handle is valid
    explicit operator bool() const;

    // Execute the task and invalidate the handle
    void run();

    // Same as before, but uses the given wait handler
    void run_with_wait_handler(wait_handler handler)

    // Convert to a void pointer and invalidate the handle
    void* to_void_ptr();

    // Convert a void pointer back to a task_run_handle
    static task_run_handle from_void_ptr(void*);
};

// Handle to represent a task that needs to be waited for
class task_wait_handle {
    // Create an invalid handle
    task_wait_handle();

    // Movable and copyable
    task_wait_handle(const task_wait_handle&);
    task_wait_handle(task_wait_handle&&);
    task_wait_handle& operator=(const task_wait_handle&);
    task_wait_handle& operator=(task_wait_handle&&);

    // Check if a handle is valid
    explicit operator bool() const;

    // Check if the task has finished
    bool ready() const;

    // Queue a function to be executed when the task has finished executing.
    void on_finish(Func&& func)
};

// Set the wait handler for the current thread. This is used to allow a
// thread to do useful work while waiting for a task to complete. This also
// returns the previously defined wait handler.
typedef void (*wait_handler)(task_wait_handle t);
wait_handler set_thread_wait_handler(wait_handler w);
```

```
// Exception thrown when a task_run_handle is destroyed without having
// run its task.
struct task_not_executed {};
```


Clone this wiki locally

```
https://github.com/Amanieu/asyncplusplus.wiki.git
```