Schedulers

Jump to bottom

Amanieu d'Antras edited this page on Feb 12, 2015 · 5 revisions

Task scheduling

By default Async++ uses a work-stealing scheduler with a thread pool of `LIBASYNC_NUM_THREADS` (environment variable) threads (or the number of CPUs in the system if not specified). The scheduler is initialized on first use, and is destroyed at program exit while ensuring that all currently running tasks finish executing before the program exits.

While this scheduler is sufficient for the majority of workloads, sometimes more control is needed on the scheduling. Async++ allows you to provide a scheduler parameter to `spawn()`, `local_spawn()` and `then()` before the task function. Several built-in schedulers are provided:

- `default_threadpool_scheduler()` : This scheduler will run tasks in a work stealing thread pool, and is the scheduler used by default when one isn't explicitly specified.
- `inline_scheduler()` : This scheduler will run tasks immediately in the current thread.
- `thread_scheduler()` : This scheduler will spawn a new thread to run tasks in.

Additionally, two scheduler classes are available for you to make your own schedulers:

- `fifo_scheduler` : This scheduler holds tasks in a FIFO queue until a thread executes a task from the queue. A thread can execute one task by calling `try_run_one_task()` or run all tasks in the queue by calling `run_all_tasks()`,
- `threadpool_scheduler` : This scheduler runs tasks in a work-stealing thread pool. The number of threads in the pool is given as an argument to the constructor. The threads are destroyed when the thread pool is destroyed, but tasks submitted to the pool after the destructor started are not guaranteed to be executed.

Example:

```
// Spawn a task in a new thread
auto t = async::spawn(async::thread_scheduler(), [] {
    std::cout << "Running in a separate thread!" << std::endl;
});

// Create a new thread pool with 10 threads
async::threadpool_scheduler custom_pool(10);

// Spawn a continuation in the thread pool
t.then(custom_pool, [] {
    std::cout << "Running a continuation in inline scheduler!" << std::endl;
});

// Create a FIFO scheduler
async::fifo_scheduler fifo;

// Queue a local_task in the FIFO
auto&& t2 = async::local_spawn(fifo, [] {
    std::cout << "Running a local task from the queue!" << std::endl;
});

// Execute all tasks queued in the FIFO in the current thread
fifo.run_all_tasks();
```

Overriding the default scheduler

It is possible to override the default scheduler used when none is specified by setting the `LIBASYNC_CUSTOM_DEFAULT_SCHEDULER` macro and re-defining the `async::default_scheduler()` function before including `<async++.h>` .

Example:

```
// Forward declare custom scheduler
class my_scheduler;

// Declare async::default_scheduler with my_scheduler
namespace async {
my_scheduler& default_scheduler();
}
#define LIBASYNC_CUSTOM_DEFAULT_SCHEDULER

// Include async++.h
#include <async++.h>

// This will use the custom scheduler
```

```
auto t = async::spawn([] {
    std::cout << "Running in custom scheduler!" << std::endl;
});

// Implementation of default_scheduler
my_scheduler& async::default_scheduler()
{
    ...
}
```

Custom schedulers

If the predefined schedulers are insufficient, it is possible to create your own custom scheduler types. A scheduler is any type that defines the following member function:

```
void schedule(async::task_run_handle t);
```

The `task_run_handle` type wraps a handle that refers to a task object that should be run. The handle is movable and has a `run()` member function to run the task in the current thread. All a scheduler has to do is call the `run()` function on the handle at some point in the future.

To allow passing handles through C-based APIs, `task_run_handle` provides two utility functions to convert the handle to and from `void*` : `to_void_ptr()` and `from_void_ptr()` . Because these functions bypass C++'s type system, you should be very careful when using them: forgetting to convert a void pointer back into a `task_run_handle` will result in a memory leak!

The GTK scheduler example included in the examples directory shows a custom scheduler that allows worker threads to run code in the GTK main loop:

```
class gtk_scheduler_impl {
        // Get the task from the void* and execute it in the UI thread
        static gboolean callback(void* p)
        {
                async::task_run_handle::from_void_ptr(p).run();
                return FALSE;
        }

public:
        // Convert a task to void* and send it to the gtk main loop
        void schedule(async::task_run_handle t)
        {
                g_idle_add(callback, t.to_void_ptr());
        }
};

// Scheduler to run a task in the gtk main loop
gtk_scheduler_impl gtk_scheduler;
```

Wait handlers

By default, waiting for a task using `get()` or `wait()` will block the calling thread until the designated task has completed. This behavior is not always desirable, for example in a thread pool it would be useful to take advantage of this time to run a queued task. Another use is for debugging: it is against some UI guidelines to perform any blocking operations in the UI thread. A custom wait handler that throws an exception when called could be used to catch any blocking waits during development.

Async++ allows the behavior of waits to be customized using *wait handlers*. A wait handler is a function that accepts a `task_wait_handle` argument, which wraps a handle to the task that should be waited for. The state of the task can be checked by calling the `ready()` function of the handle, and a function can be registered to executed when the task has completed by calling the `on_ready()` function. The wait handler is only called if the task has not completed yet, so you do not need to check the task state again at the start of the handler.

There is a single active wait handler per thread, which can be set using the `set_wait_handler()` function. Additionally it is possible to set a wait handler only for the duration of a single task by using the `run_with_wait_handler()` function on `task_run_handle`.

Example:

```
// Wait handler that disallows blocking on the UI thread. This means that
// you must check that a task has completed before calling get().
void ui_wait_handler(async::task_wait_handle)
{
        std::cerr << "Error: Blocking wait in UI thread" << std::endl;
        std::abort();
}

// Generic wait handler that sleeps until the task has completed
void sleep_wait_handler(async::task_wait_handler t)
{
        std::condition_variable cvar;
        std::mutex lock;
        bool done = false;

        t.on_finish([&] {
                std::lock_guard<std::mutex> locked(lock);
                done = true;
                cvar.notify_one();
        });

        std::unique_lock<std::mutex> locked(lock);
        while (!done)
                cvar.wait(locked);
}
```

Pages 6

Find a page…

- Home
- API Reference
- Building and installing
- Parallel algorithms
- Schedulers
  - Task scheduling
  - Overriding the default scheduler
  - Custom schedulers
  - Wait handlers
- Tasks