Here's my number; call me, maybe. Callbacks in a multithreaded world

Anthony Williams

Just Software Solutions Ltd
https://www.justsoftwaresolutions.co.uk

11th April 2019

Here’s my number; call me, maybe. Callbacks in a multithreaded world

- Task based programming
- Potential issues and solutions
- Guidelines

Task based programming

Task based programming

What is task-based programming?

- Work is divided into small chunks, “tasks”
- Tasks are submitted to a thread pool or other executor to run
- No explicit thread management

Thread pools

Thread pool properties vary widely:

- Multiple tasks per thread
- **May** be able to wait for task to finish
- **May** be able to obtain result from task
- **May** be able to cancel task via handle
- Tasks cannot (in general) wait for other tasks
- **May** be able to chain continuations

Executors

Thread pools are a special case of **Executors**.

Executor — An object that controls how, where and when a task is executed

The standardization proposal (**P0443**) allows properties to be queried.

P1244 specifies properties for retrieving a result (`execution::twoway`) and for chaining a continuation (`execution::then`).

Building blocks

A basic executor with a `submit` function with a `void` return is enough for anyone!

Building blocks

A basic executor with a `submit` function with a `void` return is enough for anyone!

We can build:

- Task waiting
- Task cancellation
- Task results
- Task chaining

Executors and callbacks

If you register callbacks with an external library you may not be in charge of the executor.

You can always wrap your callback to schedule a task on your chosen executor:

```
void foo(){
  x.register_callback([]{
    my_executor.submit(real_callback);
  });
}
```

Issues with asynchronous tasks

Issues with asynchronous tasks

- Race conditions
- Reentrancy
- Lifetimes
- Safe shutdown

Race conditions

Race conditions are ubiquitous in concurrent code.

Task-based code is no different.

Plus, submitted tasks can race to be executed, and order of task execution may have consequences.

Reentrancy

Callbacks often want to perform operations on the data structures that trigger them.

```
SomeClass my_object;
my_object.register_callback([&](){
  my_object.do_something();
});
```

Reentrancy II

Reentrancy is particularly a problem with concurrent code as you need to protect from other threads too.

```
void SomeClass::some_method(){
  std::lock_guard guard(m_mutex);
  // ...

}
void SomeClass::do_something(){
  std::lock_guard guard(m_mutex);
}
```

Reentrancy II

Reentrancy is particularly a problem with concurrent code as you need to protect from other threads too.

```
void SomeClass::some_method(){
  std::lock_guard guard(m_mutex);
  // ...
  run_callbacks();
}
void SomeClass::do_something(){
  std::lock_guard guard(m_mutex);
}
```

Reentrancy II

Reentrancy is particularly a problem with concurrent code as you need to protect from other threads too.

```
void SomeClass::some_method(){
  std::lock_guard guard(m_mutex);
  // ...
  run_callbacks();
}                       my_object.do_something();
void SomeClass::do_something(){
  std::lock_guard guard(m_mutex);
}
```

Guideline

Never run user-supplied code while holding a mutex

Reentrancy III

Simple solution:

```
void SomeClass::some_method(){
  std::vector<CallbackType> local_callbacks;
  std::unique_lock guard(m_mutex);
  do_stuff();
  local_callbacks=callbacks;
  guard.unlock();
  run_callbacks(local_callbacks);
}
```

Reentrancy IV

The simple solution has downsides:

- Multiple sets of callbacks could be invoked concurrently
- Callbacks for later updates may run before those for earlier ones
- Unregistering callbacks is race-prone

Queued Callbacks

Rather than running the callbacks in `some_method`, add them to a queue.

A separate thread then runs the callbacks one at a time, in order.

Queued Callbacks

Rather than running the callbacks in `some_method`, add them to a queue.

A separate thread then runs the callbacks one at a time, in order.

Unregistering callbacks is now easier too.

Queued Callbacks II

```
void SomeClass::some_method(){
  std::lock_guard guard(m_mutex);
  do_stuff();
  for(auto& entry: callbacks){
    callback_queue.push_back(entry);
  }
  ensure_cb_task_scheduled();
}
```

Queued Callbacks III

```
void SomeClass::ensure_cb_task_scheduled(){
  if(!cb_task_scheduled){
    pool.submit([this]{
      run_cb_queue();
    });
    cb_task_scheduled=true;
  }
}
```

Queued Callbacks IV

```
void SomeClass::run_cb_queue(){
  std::unique_lock lock(m_mutex);
  while(true){
    if(callback_queue.empty()){
      cb_task_scheduled=false;
      return;
    }
    auto entry=callback_queue.front();
    callback_queue.pop();
    lock.unlock();
    entry();
    lock.lock();
  }
}
```

Queued Callbacks IV

```
void SomeClass::run_cb_queue(){
  std::unique_lock lock(m_mutex);
  while(true){
    if(callback_queue.empty()){
      cb_task_scheduled=false;
      return;
    }
    auto entry=callback_queue.front();
    callback_queue.pop();
    lock.unlock();
    entry(); // check if still registered
    lock.lock();
  }
}
```

Lifetimes

Dangling pointers and references cause undefined behaviour.

Easy to get with multithreaded code.

Lifetimes II

```
thread_pool tp;
void launch_tasks(){
  for(unsigned i=0;i<num_tasks;++i){
    tp.submit([&]{run_task(i);});
  }
}
```

Lifetimes II

```
thread_pool tp;
void launch_tasks(){
  for(unsigned i=0;i<num_tasks;++i){
    tp.submit([&]{run_task(i);});
  }
}
```

Lifetimes III

```
thread_pool tp;
void launch_tasks(){
  for(unsigned i=0;i<num_tasks;++i){
    tp.submit([=]{run_task(i);});
  }
}
```

Capture by value when passing data to tasks if possible to avoid accidental data races and dangling references or pointers

Lifetimes IV

Sometimes you **need** a reference or pointer.

```
class SomeClass{
  FileHandle file;

  void async_load_data(){
    if(file.at_eof()) return;
    file.async_read([this](auto block){
      process_chunk(block);
      async_load_data();
    });
  }
};
```

Lifetimes V

Consider

```
void foo(){
  SomeClass x;
  x.async_load_data();
}
```

Lifetimes V

Consider

```
void foo(){
  SomeClass x;
  x.async_load_data();
}
```

Unless the destructor does something, the async tasks will outlive `x`, and have dangling pointers

Lifetimes VI

If your tasks hold a pointer or reference, you need to keep the data alive

Lifetimes VI

If your tasks hold a pointer or reference, you need to keep the data alive

- Give the tasks ownership of the data

Lifetimes VI

If your tasks hold a pointer or reference, you need to keep the data alive

- Give the tasks ownership of the data
- Wait for all tasks

Lifetimes VI

If your tasks hold a pointer or reference, you need to keep the data alive

- Give the tasks ownership of the data
- Wait for all tasks
- Set a “shutting down” flag to stop new tasks

Sharing ownership with tasks

If the tasks own the data they refer to then there are no dangling pointers or references

Use `std::shared_ptr<T>` to manage the data

Sharing ownership with tasks II

```
class SomeClass{
  struct Data:
    std::enable_shared_from_this<Data> {
    FileHandle file;
    void async_load_data();
  };
  std::shared_ptr<Data> impl;

  void async_load_data(){
    impl->async_load_data();
  }
};
```

Sharing ownership with tasks III

```
void SomeClass::Data::async_load_data(){
  if(file.at_eof()) return;
  auto self=shared_from_this();
  file.async_read([self](auto block){
    self->process_chunk(block);
    self->async_load_data();
  });
}
```

Sharing ownership with tasks IV

This avoids dangling pointers, but we can still get **dangling tasks**.

Sharing ownership with tasks IV

This avoids dangling pointers, but we can still get **dangling tasks**.

$\Rightarrow$ We still need to wait for our tasks and do an orderly shutdown.

Avoid std::weak_ptr in tasks

```
std::shared_ptr<Data> data;
void foo(){
  std::weak_ptr<Data> data_ref=data;
  pool.submit([data_ref]{
    if(auto p=data_ref.lock()){
      do_stuff(p);
    }
  });
}
```

Avoid `std::weak_ptr` in tasks II

Thread 1	**Thread 2**
Running task	
`p=data_ref.lock()`	
(Returns non-null)	
	`data.reset()`
`do_stuff(p)`	
Destroys `p`	
⇒ destroys `Data` object	

Safe shutdown

To shutdown safely we must signal the tasks to stop, and prevent new tasks being started.

C++20 will give us `std::stop_source` and `std::stop_token` for this purpose.

Safe shutdown II

```
struct SomeClass::Data{
  std::stop_source stop_flag;
};
void SomeClass::Data::async_load_data(){
  if(stop_flag.stop_requested()) return;
  //...
}
SomeClass::~SomeClass(){
  impl->stop_flag.request_stop();
}
```

Safe shutdown III

`std::stop_token` also allows for callbacks to interrupt tasks

```
struct SomeClass::Data{
  std::optional<std::stop_callback> stop_cb;
};
void SomeClass::Data::async_load_data(){
  stop_cb.emplace(stop_flag.get_token(),
  [this]{
    file.stop_async_task();
  });
  //...
}
```

Safe shutdown III

`std::stop_token` also allows for callbacks to interrupt tasks

```
struct SomeClass::Data{
  std::optional<std::stop_callback> stop_cb;
};
void SomeClass::Data::async_load_data(){
  stop_cb.emplace(stop_flag.get_token(),
  [this]{ // Outer callback keeps alive
    file.stop_async_task();
  });
  //...
}
```

`std::stop_source` and `std::stop_token`

You can read the proposal online at `https://wg21.link/p0660`

There is a sample implementation is on github: `https://github.com/josuttis/jthread`

If you can't use that, then a simple wrapper around `std::atomic<bool>` works in most cases.

Waiting for tasks to finish

Orderly shutdown without dangling tasks requires waiting for tasks to finish.

Waiting for tasks to finish II

Store `future`s for each task and wait for each in turn.

- Serial waiting is inefficient
- Requires synchronization of container

Waiting for tasks to finish III

Count tasks and wait for the count to reach zero.

- A counter needs synchronization
- Waiting requires something else like a `std::condition_variable`
- Less overhead than futures

Waiting for tasks to finish IV

```
class counting_executor {
  thread_pool &pool;
  std::mutex mutex;
  std::condition_variable cond;
  unsigned active_tasks;
  bool stop_requested;

public:
  counting_executor(thread_pool &pool_);
  ~counting_executor();
  template <typename Task>
  bool submit(Task task_);
};
```

Waiting for tasks to finish V

```
counting_executor::~counting_executor() {
  std::unique_lock guard(mutex);
  stop_requested= true;
  cond.wait(guard, [&] {
    return active_tasks == 0;
  });
}
```

Waiting for tasks to finish VI

```
template <typename Task>
bool counting_executor::submit(Task t) {
  std::lock_guard guard(mutex);
  if(stop_requested) return false;
  ++active_tasks;
  pool.submit([task= std::move(t), this] {
    task();
    std::lock_guard guard(mutex);
    if(!--active_tasks && stop_requested)
      cond.notify_all();
  });
  return true;
}
```

Waiting for tasks to finish VII

If you wait for all tasks to finish, you may not need shared ownership.

Guidelines

Guidelines

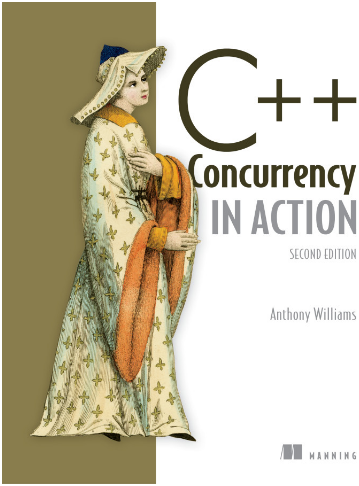
- Do not call user-provided code while holding a lock
- Watch out for dangling pointers and references
- Prefer copying values where possible
- Use `std::shared_ptr` to manage lifetimes
- Use `std::stop_token` and `std::stop_source` to avoid dangling tasks
- Wait for tasks to finish to avoid dangling tasks and pointers

Alternatives

Alternatives to task-based systems

- Explicit threads
- Actors and message-passing
- Parallel algorithms
- Coroutines with a multithreaded scheduler

My Book



C++ Concurrency in Action
Second Edition

Covers C++17 and the
Concurrency TS

Finally in print!

cplusplusconcurrencyinaction.com

Just::Thread Pro

`just::thread` Pro provides an actor framework, a concurrent hash map, a concurrent queue, synchronized values and a complete implementation of the C++ Concurrency TS, including a lock-free implementation of `atomic_shared_ptr` and RCU.

`http://stdthread.co.uk`

Questions?