Get off my thread: Techniques for moving work to background threads

Anthony Williams

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

September 2020

Get off my thread: Techniques for moving work to background threads

- **Why do we need to** move work off the current thread?
- **How do we** move work off the current thread?
- Final Guidelines and Questions

Why do we need to move work off the current thread?

Why do we need to move work off the current thread?

Many environments have a dedicated thread for processing events:

- GUIs
- Client-Server applications

Performing extensive processing on the event thread prevents other events from being handled.

Why do we need to move work off the current thread?

Delaying response to external events can have undesirable consequences:

Why do we need to move work off the current thread?

Delaying response to external events can have undesirable consequences:

- Microsoft Windows will grey-out the entire application window if it doesn't respond to events

Why do we need to move work off the current thread?

Delaying response to external events can have undesirable consequences:

- Microsoft Windows will grey-out the entire application window if it doesn't respond to events
- Web browsers will time out if the web server doesn't respond within a reasonable time

Why do we need to move work off the current thread?

Delaying response to external events can have undesirable consequences:

- Microsoft Windows will grey-out the entire application window if it doesn't respond to events
- Web browsers will time out if the web server doesn't respond within a reasonable time
- Other network applications will assume an operation failed if no response is received within a reasonable time

Why do we need to move work off the current thread?

We don't just need to move the **work**, we need to prevent **blocking** on our event-handling threads.

```
void event_handler(){
  auto handle=spawn_background_task();
  handle.wait(); // no benefit
}
```

Aside: Non-Blocking vs Lock-free

In lots of cases, short-term blocking (e.g. with short-lived `std::mutex` locks) is OK.

$\Rightarrow$ for those cases, **Non-Blocking** means **Not waiting for a lengthy task to run**.

Aside: Non-Blocking vs Lock-free

In lots of cases, short-term blocking (e.g. with short-lived `std::mutex` locks) is OK.

$\Rightarrow$ for those cases, **Non-Blocking** means **Not waiting for a lengthy task to run**.

In other cases, **Non-Blocking** means **Obstruction Free — If you suspend all but one thread at any point, that one thread will complete its task**.

$\Rightarrow$ for those cases, you need **Lock-free** allocators, message queues, etc.

How do we move work off the current thread?

How do we move work off the current thread?

Possible ways to move the work off the current thread:

How do we move work off the current thread?

Possible ways to move the work off the current thread:

- Spawn a new thread for each event handler

How do we move work off the current thread?

Possible ways to move the work off the current thread:

- Spawn a new thread for each event handler
- Pass data to a dedicated background thread

How do we move work off the current thread?

Possible ways to move the work off the current thread:

- Spawn a new thread for each event handler
- Pass data to a dedicated background thread
- Submit tasks to a generic thread pool

How do we move work off the current thread?

Possible ways to move the work off the current thread:

- Spawn a new thread for each event handler
- Pass data to a dedicated background thread
- Submit tasks to a generic thread pool
- Submit tasks to a special purpose executor

Spawning new threads

There are lots of ways to spawn new threads:

- `std::thread`
- `std::jthread`
- `std::async(std::launch::async,...)`
- Platform-specific APIs

Spawning new threads

There are lots of ways to spawn new threads:

- `std::thread`
- `std::jthread`
- `std::async(std::launch::async,...)`
- Platform-specific APIs

They all have the same problem: **detaching** the thread leaves it running, but **joining** the thread means you have to keep the handle around.

Managing thread handles

`std::thread` and `std::jthread` are similar:

```
std::vector<std::jthread> pending_threads;

void handle_event(Event details){
  auto handle=std::jthread(
    [=]{process_event(details);});
  pending_threads.push_back(
    std::move(handle));
}
```

Managing thread handles

`std::thread` and `std::jthread` are similar:

```
std::vector<std::jthread> pending_threads;

void handle_event(Event details){
  auto handle=std::jthread(
    [=]{process_event(details);});
  pending_threads.push_back(
    std::move(handle));
}
```

There is no reasonable way to test if any of the entries in `pending_threads` can be joined.

Managing thread handles II

`std::async` is a bit better: we can check the `std::future` to see if it is **ready**.

```
std::vector<std::future<void>> pending_threads;

void handle_event(Event details){
  auto handle=std::async(
    std::launch::async,
    [=]{process_event(details);});
  pending_threads.push_back(
    std::move(handle));
}
```

Managing thread handles III

Can remove completed tasks by periodically checking:

```
void check_for_done_threads(){
  for(auto it=pending_threads.begin();
    it!=pending_threads.end();){
    if(it->wait_for(0s)==
      std::future_status_ready)
      it=pending_threads.erase(it);
    else ++it;
  }
}
```

Managing thread handles III

Can remove completed tasks by periodically checking:

```
void check_for_done_threads(){
  for(auto it=pending_threads.begin();
    it!=pending_threads.end();){
    if(it->wait_for(0s)==
      std::future_status_ready)
      it=pending_threads.erase(it);
    else ++it;
  }
}
```

This is nasty: if you have a lot of events, you spawn a lot of threads, so the list will get large.

Guideline

Spawning a new thread for every task is a **bad idea**.

Dedicated threads

A dedicated thread can be a reasonable idea.

Dedicated threads

A dedicated thread can be a reasonable idea.

1. The dedicated thread is created

Dedicated threads

A dedicated thread can be a reasonable idea.

1. The dedicated thread is created
2. The event handler receives an event

Dedicated threads

A dedicated thread can be a reasonable idea.

1. The dedicated thread is created
2. The event handler receives an event
3. The event handler reposts the event as a message to the dedicated thread

Dedicated threads

A dedicated thread can be a reasonable idea.

1. The dedicated thread is created
2. The event handler receives an event
3. The event handler reposts the event as a message to the dedicated thread
4. The event handler returns ready to receive a new event

Dedicated threads

A dedicated thread can be a reasonable idea.

1. The dedicated thread is created
2. The event handler receives an event
3. The event handler reposts the event as a message to the dedicated thread
4. The event handler returns ready to receive a new event
5. (sometime later) The dedicated thread receives the message and processes it

Dedicated threads

A dedicated thread can be a reasonable idea.

1. The dedicated thread is created
2. The event handler receives an event
3. The event handler reposts the event as a message to the dedicated thread
4. The event handler returns ready to receive a new event
5. (sometime later) The dedicated thread receives the message and processes it
6. The dedicated thread waits for more messages to process

Dedicated threads

A dedicated thread can be a reasonable idea.

1. The dedicated thread is created
2. The event handler receives an event
3. The event handler reposts the event as a message to the dedicated thread
4. The event handler returns ready to receive a new event
5. (sometime later) The dedicated thread receives the message and processes it
6. The dedicated thread waits for more messages to process
7. (eventually) The dedicated thread is destroyed

Dedicated threads: downsides

Downsides to dedicated threads:

- The thread is long-lived and consumes resources

Dedicated threads: downsides

Downsides to dedicated threads:

- The thread is long-lived and consumes resources
- The handling of the event is disconnected from the event

Dedicated threads: downsides

Downsides to dedicated threads:

- The thread is long-lived and consumes resources
- The handling of the event is disconnected from the event
- No parallel processing: it's a single thread!

Dedicated threads: upsides

Upsides to dedicated threads:

- The events are handled in the same order as they arrive at the initial event handler

Dedicated threads: upsides

Upsides to dedicated threads:

- The events are handled in the same order as they arrive at the initial event handler
- The dedicated thread only communicates with the outside world via messages, so is easy to design and test

Dedicated threads: upsides

Upsides to dedicated threads:

- The events are handled in the same order as they arrive at the initial event handler
- The dedicated thread only communicates with the outside world via messages, so is easy to design and test
- Cancelling all outstanding tasks is easy: just shut down the thread

Thread pools

If a dedicated processing thread isn’t the ideal fit, an alternative is a **pool of threads**

Thread pools

If a dedicated processing thread isn't the ideal fit, an alternative is a **pool of threads**

1. The thread pool is created

Thread pools

If a dedicated processing thread isn't the ideal fit, an alternative is a **pool of threads**

1. The thread pool is created
2. The event handler receives an event

Thread pools

If a dedicated processing thread isn't the ideal fit, an alternative is a **pool of threads**

1. The thread pool is created
2. The event handler receives an event
3. The event handler reposts the event as a **task** to the thread pool

Thread pools

If a dedicated processing thread isn't the ideal fit, an alternative is a **pool of threads**

1. The thread pool is created
2. The event handler receives an event
3. The event handler reposts the event as a **task** to the thread pool
4. The event handler returns ready to receive a new event

Thread pools

If a dedicated processing thread isn't the ideal fit, an alternative is a **pool of threads**

1. The thread pool is created
2. The event handler receives an event
3. The event handler reposts the event as a **task** to the thread pool
4. The event handler returns ready to receive a new event
5. (sometime later) A thread from the pool runs the **task**

Thread pools

If a dedicated processing thread isn't the ideal fit, an alternative is a **pool of threads**

1. The thread pool is created
2. The event handler receives an event
3. The event handler reposts the event as a **task** to the thread pool
4. The event handler returns ready to receive a new event
5. (sometime later) A thread from the pool runs the **task**
6. The thread then waits for a new **task**

Thread pools

If a dedicated processing thread isn't the ideal fit, an alternative is a **pool of threads**

1. The thread pool is created
2. The event handler receives an event
3. The event handler reposts the event as a **task** to the thread pool
4. The event handler returns ready to receive a new event
5. (sometime later) A thread from the pool runs the **task**
6. The thread then waits for a new **task**
7. (eventually) The thread pool is destroyed

Thread pools: upsides

Upsides to thread pools:

- The tasks are self-contained so easy to design and test

Thread pools: upsides

Upsides to thread pools:

- The tasks are self-contained so easy to design and test
- The number of threads can be scaled with the available hardware

Thread pools: upsides

Upsides to thread pools:

- The tasks are self-contained so easy to design and test
- The number of threads can be scaled with the available hardware
- The threads can be shared with the rest of the application

Thread pools: downsides

Downsides to thread pools:

- The tasks are run in an arbitrary order as they are picked up by the pool

Thread pools: downsides

Downsides to thread pools:

- The tasks are run in an arbitrary order as they are picked up by the pool
- The handling of the event is disconnected from the event

Thread pools: downsides

Downsides to thread pools:

- The tasks are run in an arbitrary order as they are picked up by the pool
- The handling of the event is disconnected from the event
- The tasks may be run concurrently, so their interactions need to be verified

Thread pools: downsides

Downsides to thread pools:

- The tasks are run in an arbitrary order as they are picked up by the pool
- The handling of the event is disconnected from the event
- The tasks may be run concurrently, so their interactions need to be verified
- Tasks may be delayed due to tasks submitted from elsewhere in the application

Thread pools: downsides

Downsides to thread pools:

- The tasks are run in an arbitrary order as they are picked up by the pool
- The handling of the event is disconnected from the event
- The tasks may be run concurrently, so their interactions need to be verified
- Tasks may be delayed due to tasks submitted from elsewhere in the application
- Cancelling all the tasks from one source without affecting others is harder

Addressing thread pool downsides

The biggest downside from thread pools is the unpredictable ordering.

Addressing thread pool downsides

The biggest downside from thread pools is the unpredictable ordering.

We can impose an ordering with **continuations**.

Addressing thread pool downsides

The biggest downside from thread pools is the unpredictable ordering.

We can impose an ordering with **continuations**.

If task B is a **continuation** of task A, it cannot run concurrently.

Continuations

```
thread_pool pool;
std::optional<task_handle> last_task;

void handle_event(Event details){
  auto func=[=]{process_event(details);};

  if(last_task){
    last_task=last_task->then(func);
  } else {
    last_task=pool.submit(func);
  }
}
```

Continuations: upsides

Upsides of using continuations like this:

Continuations: upsides

Upsides of using continuations like this:

- Events are handled in the order they arrive

Continuations: upsides

Upsides of using continuations like this:

- Events are handled in the order they arrive
- Events are not handled concurrently

Continuations: upsides

Upsides of using continuations like this:

- Events are handled in the order they arrive
- Events are not handled concurrently

Continuations: upsides

Upsides of using continuations like this:

- Events are handled in the order they arrive
- Events are not handled concurrently

No concurrency: lengthy tasks delay subsequent event processing

Continuations: downsides

Downsides of using continuations like this:

Continuations: downsides

Downsides of using continuations like this:

- No concurrency: lengthy tasks delay subsequent event processing

Continuations: downsides

Downsides of using continuations like this:

- No concurrency: lengthy tasks delay subsequent event processing
- Still no fix for other problems: cancellation, disconnect from event and handling, etc.

Cancellation

Cancellation

Later events may require you to cancel tasks submitted to handle earlier events.

This is particularly the case for long-running tasks.

We need to avoid **dangling tasks** where the task outlives the desire for it to do anything.

Cancellation: Stop tokens

Pass a `std::stop_token` to each task that may need to be cancelled.

Keep the corresponding `std::stop_source` in the event handler.

Call `source.request_stop()` when the tasks need to be stopped.

Cancellation: Waiting for cancelled tasks

Cancelled tasks may continue running for a short time.

$\Rightarrow$ You need to ensure they are stopped before cleaning up the data

The simplest solution is a count of remaining tasks

Cancellation: Counting outstanding tasks

```
std::atomic<unsigned> pending_tasks;
std::stop_source source;

void handle_event(Event details){
  ++pending_tasks;
  pool.submit(
    [=,&pending_tasks,
    stopper=source.get_token()]{
      process_event(details,stopper);
      if(!--pending_tasks)
        pending_tasks.notify_all();
    });
}
```

Cancellation: Counting outstanding tasks

Cancel the tasks in some event handler:

```
void cancel_tasks(){
  source.request_stop();
}
```

Cancellation: Counting outstanding tasks

Cancel the tasks in some event handler:

```
void cancel_tasks(){
  source.request_stop();
}
```

Then wait for them when cleaning up (**not in an event handler!**):

```
void wait_for_tasks(){
  while(auto count=pending_tasks.load()){
    pending_tasks.wait(count);
  }
}
```

Progress Updates

Progress Updates

Background threads might need to update the event thread with information.

Progress Updates

Background threads might need to update the event thread with information.

- Progress updates in a GUI

Progress Updates

Background threads might need to update the event thread with information.

- Progress updates in a GUI
- Updating other UI information

Progress Updates

Background threads might need to update the event thread with information.

- Progress updates in a GUI
- Updating other UI information
- Initiating IO on an IO loop

Progress Updates

Typically you need to trigger an event on the event thread.

- Custom Windows messages
- Using `eventfd` to create a file handle for events
- Similar mechanism for other platforms

Progress Updates

Background thread:

```
void foo(unsigned progress){
  post_progress_event(progress);
}
```

Event loop:

```
void handle_progress(ProgressEvent ev){
  update_ui(ev.progress);
}
```

Coroutines

Coroutines

Coroutines are **not** a magical solution.

Coroutines

Coroutines are **not** a magical solution.

... but they can provide simpler-looking code

Coroutines: example

```
void handle_event(Event details){
  schedule_on(thread_pool,process(details));
}

task<void> process(Event details){
  co_await step_1(details)
  co_await schedule_on_gui_thread(
    update_progress(1));
  co_await step_2(details)
  co_await schedule_on_gui_thread(
    update_progress(2));
  // ...
}
```

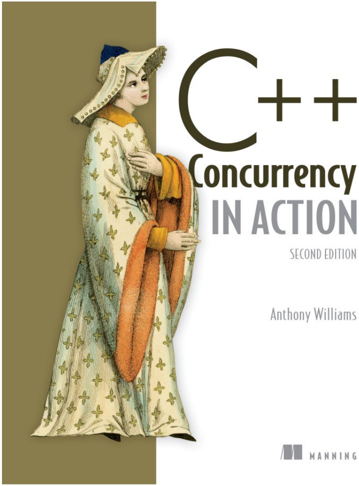
Guidelines

Guidelines

- Do not run time consuming tasks on threads that must be **responsive**
- Use a dedicated thread for long running tasks
- Use a thread pool for other tasks
- Use `std::stop_token` or similar for cancellation
- Ensure tasks are finished before destroying data they reference
- Check whether you need **lock-free** code or just short-lived locks

Questions?

My Book



C++ Concurrency in Action
Second Edition

Covers C++17 and the Concurrency TS

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`