Roar11.com
coding and such

A Platform-Independent Thread Pool Using C++14

Introduction

One of the major benefits provided by the new generation of graphics APIs is much better support for multithreaded command list generation and submission. It's not uncommon for computers nowadays to contain 2, 4, 8, or even 16 core processors. The goal of the solution in this post is to ensure we can use the power our CPU provides, not just for generating graphics command lists, but for any task that can be easily parallelized.

At it's simplest, a thread pool is a collection of threads that run continuously waiting to take on a task to complete. If there's no task available, they yield or sleep for some amount of time, wake back up, and check again. When a task is available, one of the waiting threads claims it, runs it, and returns to the waiting state.

The reason we would want to use a thread pool instead of creating new threads over and over for each task we want to run on a separate thread is to save on the time it would otherwise take to construct a thread, submit work to it, and deconstruct it when it's done running. With a small collection of threads continuously running and waiting on tasks, we're only left with the middle step – work submission.

Implementation

The thread pool presented here is based off the implementation provided in [1]. It has been updated to include variadic arguments for added flexibility.

A THREAD-SAFE QUEUE

Before we build the pool itself, we need a means of submitting work in a thread-safe manner. Jobs should be picked up in the same order they are submitted to the pool, which means a queue is a good candidate. Jobs are pushed to the back of the queue, and popped from the front.

```
1  /**
2   * The ThreadSafeQueue class.
3   * Provides a wrapper around a basic queue to provide thread s
4   */
5  #pragma once
6
7  #ifndef THREADSAFEQUEUE_HPP
8  #define THREADSAFEQUEUE_HPP
9
10 #include <atomic>
11 #include <condition_variable>
12 #include <mutex>
13 #include <queue>
14 #include <utility>
15
16 namespace MyNamespace
17 {
18     template <typename T>
19     class ThreadSafeQueue
20     {
21     public:
22         /**
23          * Destructor.
24          */
25         ~ThreadSafeQueue(void)
26         {
27             invalidate();
28         }
29
30         /**
31          * Attempt to get the first value in the queue.
32          * Returns true if a value was successfully written to
33          */
34         bool tryPop(T& out)
35         {
36             std::lock_guard<std::mutex> lock{m_mutex};
37             if(m_queue.empty() || !m_valid)
38             {
39                 return false;
40             }
```

```
41              out = std::move(m_queue.front());
42              m_queue.pop();
43              return true;
44          }
45
46          /**
47           * Get the first value in the queue.
48           * Will block until a value is available unless clear
49           * Returns true if a value was successfully written to
50           */
51          bool waitPop(T& out)
52          {
53              std::unique_lock<std::mutex> lock{m_mutex};
54              m_condition.wait(lock, [this]()
55              {
56                  return !m_queue.empty() || !m_valid;
57              });
58              /*
59               * Using the condition in the predicate ensures th
60               * but empty queue will not proceed, so only need
61               */
62              if(!m_valid)
63              {
64                  return false;
65              }
66              out = std::move(m_queue.front());
67              m_queue.pop();
68              return true;
69          }
70
71          /**
72           * Push a new value onto the queue.
73           */
74          void push(T value)
75          {
76              std::lock_guard<std::mutex> lock{m_mutex};
77              m_queue.push(std::move(value));
78              m_condition.notify_one();
79          }
80
81          /**
82           * Check whether or not the queue is empty.
83           */
84          bool empty(void) const
85          {
86              std::lock_guard<std::mutex> lock{m_mutex};
87              return m_queue.empty();
88          }
89
90          /**
91           * Clear all items from the queue.
92           */
93          void clear(void)
94          {
95              std::lock_guard<std::mutex> lock{m_mutex};
96              while(!m_queue.empty())
97              {
```

```
98                  m_queue.pop();
99              }
100             m_condition.notify_all();
101         }
102
103         /**
104          * Invalidate the queue.
105          * Used to ensure no conditions are being waited on in
106          * a thread or the application is trying to exit.
107          * The queue is invalid after calling this method and
108          * to continue using a queue after this method has bee
109          */
110         void invalidate(void)
111         {
112             std::lock_guard<std::mutex> lock{m_mutex};
113             m_valid = false;
114             m_condition.notify_all();
115         }
116
117         /**
118          * Returns whether or not this queue is valid.
119          */
120         bool isValid(void) const
121         {
122             std::lock_guard<std::mutex> lock{m_mutex};
123             return m_valid;
124         }
125
126     private:
127         std::atomic_bool m_valid{true};
128         mutable std::mutex m_mutex;
129         std::queue<T> m_queue;
130         std::condition_variable m_condition;
131     };
132 }
133
134 #endif
```

Most of this is pretty standard fare for designing a thread-safe class. We lock a mutex anytime we need to read or write data and provide a simplified interface over a std::queue where writes are checked for validity before being performed. This is why tryPop and waitPop return bools for success and write to the provide parameter in successful cases.

Any time push is called with a new task, it calls notify_one() on the condition variable which will wake one thread blocked on the condition. The mutex is locked, the predicate is checked, and if all conditions are met (the queue is not empty and the queue is still valid), a task is popped and returned from the queue.

Because this queue provides a blocking method, waitPop, that depends on a condition variable being set to continue, it also needs a way to signal to anything waiting on the condition in the case that the queue needs to be deconstructed while there are threads still blocked on the condition. This is accomplished through the invalidate() method that first sets the m_valid member to false and then calls notify_all() on the condition variable. This will wake up every thread blocked on the condition and waitPop will return with a value of false, indicating to the call site that no work is being returned.

Another nicety the condition variable gives us is protection from spurious wakeups [3]. If a spurious wakeup does occur and the entire predicate isn't met, the thread goes back to waiting.

THE THREAD POOL

The implementation of the thread pool is shown below.

```
1  /**
2   * The ThreadPool class.
3   * Keeps a set of threads constantly waiting to execute incomi
4   */
5  #pragma once
6
7  #ifndef THREADPOOL_HPP
8  #define THREADPOOL_HPP
9
10 #include "ThreadSafeQueue.hpp"
11
12 #include <algorithm>
13 #include <atomic>
14 #include <cstdint>
15 #include <functional>
16 #include <future>
17 #include <memory>
18 #include <thread>
19 #include <type_traits>
20 #include <utility>
21 #include <vector>
22
23 namespace MyNamespace
24 {
25     class ThreadPool
26     {
27     private:
28         class IThreadTask
```

```
29          {
30          public:
31              IThreadTask(void) = default;
32              virtual ~IThreadTask(void) = default;
33              IThreadTask(const IThreadTask& rhs) = delete;
34              IThreadTask& operator=(const IThreadTask& rhs) = d
35              IThreadTask(IThreadTask&& other) = default;
36              IThreadTask& operator=(IThreadTask&& other) = defa
37
38              /**
39               * Run the task.
40               */
41              virtual void execute() = 0;
42          };
43
44          template <typename Func>
45          class ThreadTask: public IThreadTask
46          {
47          public:
48              ThreadTask(Func&& func)
49                  :m_func{std::move(func)}
50              {
51              }
52
53              ~ThreadTask(void) override = default;
54              ThreadTask(const ThreadTask& rhs) = delete;
55              ThreadTask& operator=(const ThreadTask& rhs) = del
56              ThreadTask(ThreadTask&& other) = default;
57              ThreadTask& operator=(ThreadTask&& other) = defaul
58
59              /**
60               * Run the task.
61               */
62              void execute() override
63              {
64                  m_func();
65              }
66
67          private:
68              Func m_func;
69          };
70
71      public:
72          /**
73           * A wrapper around a std::future that adds the behavi
74           * Specifically, this object will block and wait for e
75           */
76          template <typename T>
77          class TaskFuture
78          {
79          public:
80              TaskFuture(std::future<T>&& future)
81                  :m_future{std::move(future)}
82              {
83              }
84
85              TaskFuture(const TaskFuture& rhs) = delete;
```

```
86              TaskFuture& operator=(const TaskFuture& rhs) = del
87              TaskFuture(TaskFuture&& other) = default;
88              TaskFuture& operator=(TaskFuture&& other) = defaul
89              ~TaskFuture(void)
90              {
91                  if(m_future.valid())
92                  {
93                      m_future.get();
94                  }
95              }
96
97              auto get(void)
98              {
99                  return m_future.get();
100             }
101
102
103         private:
104             std::future<T> m_future;
105         };
106
107     public:
108         /**
109          * Constructor.
110          */
111         ThreadPool(void)
112             :ThreadPool{std::max(std::thread::hardware_concurr
113         {
114             /*
115              * Always create at least one thread.  If hardware
116              * subtracting one would turn it to UINT_MAX, so g
117              * hardware_concurrency() and 2 before subtracting
118              */
119         }
120
121         /**
122          * Constructor.
123          */
124         explicit ThreadPool(const std::uint32_t numThreads)
125             :m_done{false},
126             m_workQueue{},
127             m_threads{}
128         {
129             try
130             {
131                 for(std::uint32_t i = 0u; i < numThreads; ++i)
132                 {
133                     m_threads.emplace_back(&ThreadPool::worker
134                 }
135             }
136             catch(...)
137             {
138                 destroy();
139                 throw;
140             }
141         }
142
```

```
143         /**
144          * Non-copyable.
145          */
146         ThreadPool(const ThreadPool& rhs) = delete;
147 
148         /**
149          * Non-assignable.
150          */
151         ThreadPool& operator=(const ThreadPool& rhs) = delete;
152 
153         /**
154          * Destructor.
155          */
156         ~ThreadPool(void)
157         {
158             destroy();
159         }
160 
161         /**
162          * Submit a job to be run by the thread pool.
163          */
164         template <typename Func, typename... Args>
165         auto submit(Func&& func, Args&&... args)
166         {
167             auto boundTask = std::bind(std::forward<Func>(func
168             using ResultType = std::result_of_t<decltype(bound
169             using PackagedTask = std::packaged_task<ResultType
170             using TaskType = ThreadTask<PackagedTask>;
171 
172             PackagedTask task{std::move(boundTask)};
173             TaskFuture<ResultType> result{task.get_future()};
174             m_workQueue.push(std::make_unique<TaskType>(std::m
175             return result;
176         }
177 
178     private:
179         /**
180          * Constantly running function each thread uses to acq
181          */
182         void worker(void)
183         {
184             while(!m_done)
185             {
186                 std::unique_ptr<IThreadTask> pTask{nullptr};
187                 if(m_workQueue.waitPop(pTask))
188                 {
189                     pTask->execute();
190                 }
191             }
192         }
193 
194         /**
195          * Invalidates the queue and joins all running threads
196          */
197         void destroy(void)
198         {
199             m_done = true;
```

```
200                 m_workQueue.invalidate();
201                 for(auto& thread : m_threads)
202                 {
203                     if(thread.joinable())
204                     {
205                         thread.join();
206                     }
207                 }
208             }
209
210         private:
211             std::atomic_bool m_done;
212             ThreadSafeQueue<std::unique_ptr<IThreadTask>> m_workQu
213             std::vector<std::thread> m_threads;
214         };
215
216         namespace DefaultThreadPool
217         {
218             /**
219              * Submit a job to the default thread pool.
220              */
221             template <typename Func, typename... Args>
222             inline auto submitJob(Func&& func, Args&&... args)
223             {
224                 return getThreadPool().submit(std::forward<Func>(f
225             }
226
227             /**
228              * Get the default thread pool for the application.
229              * This pool is created with std::thread::hardware_con
230              */
231             inline ThreadPool& getThreadPool(void)
232             {
233                 static ThreadPool defaultPool;
234                 return defaultPool;
235             }
236         }
237     }
238
239     #endif
```

There are a few pieces to touch on here. First, we have an IThreadTask interface that defines an execute() pure virtual function. The reason for this inteface is simply so we can maintain a collection of them in one container type (the ThreadSafeQueue<T>). ThreadTask<T> implements IThreadTask and takes a callable type T for its template parameter.

When constructing the thread pool, we attempt to read the number of hardware threads available to the system by using std::thread::hardware_concurrency(). We always ensure the pool is started with at least one thread running, and ideally

started with hardware_concurrency – 1 threads running. The reason for the minus one will be discussed later. For each thread available, we construct a std::thread object that runs the private member function worker().

The worker function's only job is to endlessly check the queue to see if there is work to be done and execute the task if there is. Since we've taken care to design the queue in a thread-safe manner, we don't need to do any additional synchronization here. The thread will enter the loop, get to waitPop, and either pop and execute a queued task, or wait on a task to become available via the submit function. If waitPop returns true, we know pTask has been written to and can immediately execute it. If it returns false, it most likely means that the queue has been invalidated.

The submit function is the public facing interface of the thread pool. It starts by creating a few handy type definitions that make the actual implementation easier to follow. First, the provided function and its arguments are bound to a callable object with no parameters using std::bind. We need this for our ThreadTask<T> class to be able to call execute on its functor without having to know the arguments that came with the original function. We then create a std::packaged_task with the bound task and extract the std::future from it before pushing it onto the queue. Here again, we do not need to do any additional synchronization due to the thread-safe implementation of the queue. You'll notice the std::future returned from the std::packaged_task is wrapped in a class called TaskFuture<T>. This was a design decision because of the way I intend to use the pool in my specific application. I wanted the futures to mimic the way std::async futures work, specifically that they will block until their work is complete when they are going out of scope and being destructed. std::packaged_task futures don't do this out of the box, so we give them a simple wrapper to emulate the behavior [2]. Like std::future, TaskFuture is movable-only, so the synchronization does not have to occur in the same method as the call site as long as it's passed along from the method.

You will see where the queue's invalidate method is called in the thread pool's destroy() method, which is called from the destructor or if an exception is thrown while creating the threads in the constructor, **before** joining the threads, and **after**

setting the thread pool's done marker to true. The order is important to ensure that the threads know to exit their worker functions instead of re-attempting to obtain more work from the invalidated queue. Due to the way the predicate is set up on the queue's condition variable, it is not an error to re-enter waitPop on an invalidated queue since it will just return false, but it is a waste of time.

An optional nicety I decided to throw in is the DefaultThreadPool namespace. This creates a thread pool with the maximum number of threads as discussed previously and is accessible from anywhere in the application that includes the thread pool header. I prefer using this as opposed to having each subsystem owning its own thread pool, but there's nothing wrong with creating thread pool instances through the constructors, either.

SUBMITTING WORK TO THE THREAD POOL

With the above in place. Submitting work is as simple as including the thread pools header file and calling its submit function with a callable object and optionally arguments to be provided to it.

```
auto taskFuture = DefaultThreadPool::submitJob([]()
{
    lengthyProcess();
});

auto taskFuture2 = DefaultThreadPool::submitJob([](int a, float
{
    lengthyProcessWithArguments(a, b);
}, 5, 10.0f);
```

If submitting a reference for an argument, it is important to remember to wrap it with std::ref or std::cref.

```
MyObject obj;
auto taskFuture = DefaultThreadPool::submitJob([](const MyObject
{
    lengthyProcessThatNeedsToReadObject(object);
}, std::cref(obj));
```

Does It Work?

To ensure the thread pool and backing queue work not only in ideal cases, but also in the case where work is being submitted faster than the threads can take it on, we can write a little program that submits a bunch of jobs that sleep for a while and then synchronizes on them. My machine reports eight as the result of std::thread::hardware_concurrency(), so I create a thread pool with seven threads. The task I'm running is just to sleep whatever thread is executing for one second and finish. I'll submit twenty-one of these jobs to the pool. We know that this would take about twenty-one seconds if executed serially, and since we're running a thread pool with seven threads, we know that if everything is working well the jobs should all complete in about three seconds.

```
Timer saturationTimer;
const auto startTime = saturationTimer.tick();
std::vector<ThreadPool::TaskFuture<void>> v;
for(std::uint32_t i = 0u; i < 21u; ++i)
{
    v.push_back(DefaultThreadPool::submitJob([]()
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }));
}
for(auto& item: v)
{
    item.get();
}
const auto dt = saturationTimer.tick() - startTime;
```

Running the above code on my machine, the result is just about what would be expected, averaging around 3.005 seconds over a dozen runs.

About the Number of Pooled Threads

Earlier I mentioned that I start the thread pool with std::thread::hardware_concurrency() – 1 threads. The reason for this is simple. The thread that's calling the thread pool is a perfectly valid thread to do work on while you're waiting for the results of submitted tasks to become available. Despite the example from the Does It Work? section, submitting a bunch of jobs

and then just waiting on them to complete is hardly optimal, so it makes sense to have the thread pool executing up to NumThreads – 1 jobs and the main thread doing whatever work it can accomplish in the meantime. Splitting the workload up evenly across all available threads is usually the best approach with a task-based setup like this.

Conclusion

This post has discussed what a thread pool is, why they're useful, and how to get started implementing one. There are very likely ways to make the provided thread pool more performant by specializing it more to avoid memory allocations on job submissions, but for my use cases I typically ensure the jobs being submitted are large enough that they make up for the time lost to allocating and deallocating memory with the time gained by running them in parallel with other large tasks. Your mileage may vary, but at the very least you should have a solid start to customizing a thread pool to fit your exact needs.

Thank You

A big thank you to the members of /r/cpp who helped with code review and provided excellent feedback!

References

[1] William, Anthony. *C++ Concurrency in Action: Practical Multithreading*. ISBN: 9781933988771

[2] http://scottmeyers.blogspot.com/2013/03/stdfutures-from-stdasync-arent-special.html

[3] http://en.cppreference.com/w/cpp/thread/condition_variable/wait


WillP / January 3, 2016 / Development

13 thoughts on “A Platform-Independent Thread Pool Using C++14”


brenton

May 20, 2016 at 10:43 PM

Nicely done

slight error – probably a typo – coz it wont work without it – you have

m_threads.emplace_back(&ThreadPool::worker, this);

when I think you mean

m_threads.emplace_back(std::thread(&ThreadPool::worker), this);

it runs on only one thread as written

Also another typo you are missing a > after #include <utility in the #include list of the thread pool.


WillP

May 20, 2016 at 10:56 PM

Thanks for bringing the missing > to my attention. I’ve fixed it in the post.

I’m able to use emplace_back as shown without problem. When using std::vector’s push_back function the way you wrote it would be required, but

emplace_back can directly take constructor arguments and forward them to the vector's type. So calling emplace_back as done above should be suitable and should create a new element in place with the worker function and a pointer to 'this'. I'm using Visual Studio 2015, in case maybe it's an environment issue you're seeing. Hope that helps.

brenton

May 21, 2016 at 3:58 AM

My bad – I have a glitch somewhere else. the emplace back does seem to work

---

Cowlumbus

June 11, 2016 at 2:14 PM

Thanks for sharing this code, it's a good example of one approach to write a task based system.

I have two questions:
1. How would you explain the extra 5 milliseconds spend on the tasks? I have changed each task to a 100 millisecond sleep and spawned 210 of them. The total execution time then is roughly 3.019 seconds, which translates into a 19 milliseconds overhead. Is this caused by context switching? How would you improve this?
2. I've tried to convert the code to make it compatible with c++11 (Visual Studio 2013). It (almost) compiled when I provided explicit return types (using "->" notation) and writing my own move constructors. It only complained about the arguments passed to the submitJob() function. Do you think there is a viable way to rewrite the code for C++11? Or does that require a lot of template magic or ugly code?

Cowlumbus

June 13, 2016 at 10:11 AM

After a bit of profiling, I think I can answer the first question myself:

* Most of the extra milliseconds comes from the fact that sleep_for() does not wake up the thread exactly after the specified amount of (milli)seconds.
* There is also a tiny bit of overhead caused by the mutexes.
* After reading up on context switches, I don't think they contribute much to the additional overhead.

WillP

June 13, 2016 at 11:53 PM

Hey,

Good to see you've done some profiling already. Yep, std::this_thread::sleep_for is specified to block the executing thread for "at least" the time specified. Usually the scheduler will do a decent job waking up a sleeping thread when the time is up, but you can never be exactly certain.

Yes, using mutexes is another potential choke point in this implementation. There are a few resources online that talk about lock-free, work-stealing queues that aim to alleviate this, but they can be difficult to get right, and I wanted to first make sure I had a reliable implementation, even if it isn't absolutely as fast as possible.

In the example from the post, context switching should be minimal since I'm not oversaturating my physical cores, but in your case it could add to the execution time since several tasks could be contending for resources on the same core.

Dynamic memory allocation is another potential slowdown in the implementation from the post. If you look in the ThreadPool.submit function, you'll see make_unique being called on the packaged task wrappers. A

potentially better solution would be to use a pre-allocated memory pool to fetch and return chunks of memory from as they're needed, but that was beyond the scope of this article. In a production real-time application, this is a route I would explore further, especially as the number of tasks being submitted grew.

All of these items are trade-offs between complexity and performance gains. For the application I'm working on, I have 3 large "main" tasks that run constantly (divvying up work inside frames to render) and the costs of the dynamic allocations and mutex uses pale in comparison to the performance improvements on CPU-bound scenes. Long-running background tasks such as loading new textures in, etc., don't bother the workflow, and the main tasks can even spawn off sub-tasks pretty efficiently and still be a net gain for performance.

As with anything else – you have to profile and find the best solution for your project. :)

Edit: As far as the being C++14-specific stuff – the expanded use of 'auto' for return type deduction is a big help in keeping the implementation clean and easy to read. I feel like template magic could get you most of the way there, but I haven't tried going that route. (Also, make_unique is technically a C++14 feature, but I think VS has had it since 2013 if I recall correctly.)

-Will

Mohamed Samir

August 23, 2016 at 9:55 AM

Thank you for the information you provided in this post. I would like to mention that thread.join() is going to
hang the application exist on VS2013 because of this bug
https://connect.microsoft.com/VisualStudio/feedback/details/747145

Apparently, thread::join() is being called after the main() function exists and this tries to run a mutex initialization
code after the main function ends.
a workaround as suggested on the provided link is to call windows API
“_Cnd_do_broadcast_at_thread_exit()”
before creating any threads. This will ensure that mutexes are created before any object tries to access them
and also ensure that mutexes will live until the last moment of application life.

Cheers..Samir

Boris

September 7, 2016 at 4:54 PM

Thank you for the hard work on this!

How would I move a unique_ptr into this queue?

I tried the following:

```
auto taskFuture = DefaultThreadPool::submitJob(
[](int _n, unique_ptr _message)
{
doTask(_n, move(_message));
}, n, move(message));
```

But I get:

Error C2672: ‘std::invoke’: no matching overloaded function found (1)

Do you have any suggestions?

WillP

September 13, 2016 at 6:31 PM

Hi,

An unfortunate result of using std::bind is that movable-only types aren't able to be passed easily if you're looking to transfer ownership (otherwise a const reference would suffice).

A sloppy way to do get around the limitation would be to change the lambda to take a raw pointer of your type and call release() on the original when submitting the task. Then immediately wrap it back up in a unique_ptr inside the lambda body. Again – sloppy and ugly, but it technically works, and from the body of the lambda you could move it elsewhere (your doTask function, for example).

C++ 17's std::apply (http://en.cppreference.com/w/cpp/utility/apply) will likely be a better fit in the future, and it should allow move-only types to be used as expected.

Another potential solution you could explore is presented here: http://talesofcpp.fusionfenix.com/post-14/true-story-moving-past-bind

---

Aust

November 30, 2016 at 12:48 PM

Hi,

I am having trouble using your example. I want to use it to do a single consumer thread that takes an input and process it and send an notification or a result to another thread.

Here is my code :

```
void Notify(int a){
std::cout << "Hello again" << a << std::endl;
}

class Tests{
public:
void run(int a){
std::this_thread::sleep_for(std::chrono::seconds(1));
std::cout << "Hello " << a << std::endl;

DefaultThreadPool::submitJob([](int b)
{
Notify(b);
}, a);
}
};

int main( int argc, char* argv[] )
{
auto i{0};
Tests tests;

while(true){
DefaultThreadPool::submitJob([](Tests& obj, int a)
{
obj.run(a);
},std::ref(tests), i);

i++;
}
return 0;
}
```

I don't know if i am doing it right. It's like a cascading queue. But when I try to build it I have this error :
In file included from /src/main_test.cc:2:0:
/src/ThreadPool.h: In function ‘auto DefaultThreadPool::submitJob(Func&&, Args&& ...)’:
/src/ThreadPool.h:222:30: error: there are no arguments to ‘getThreadPool’ that depend on a template parameter, so a declaration of ‘getThreadPool’ must be available [-fpermissive]
return getThreadPool().submit(std::forward(func), std::forward(args)...);
^
/src/ThreadPool.h:222:30: note: (if you use ‘-fpermissive’, G++ will accept your code, but allowing the use of an undeclared name is deprecated)
src/CMakeFiles/test_thread.dir/build.make:62: recipe for target ‘src/CMakeFiles/test_thread.dir/main_test.cc.o’ failed

I am using G++ 5.4

Thanks for your help.

WillP

December 7, 2016 at 11:10 AM

Interesting, I’m running your example copy/pasted in VS2015 with no errors or warnings. Are you enabling the c++14 flag for g++? (g++-5.4.0 -std=c++14 filename.cpp)

Adam

December 3, 2016 at 12:03 PM

Hey WillP,

Thanks for the great threading article! I've started to have a play around with the code and I've run into an issue. If I submit a task from within a method and then try and fall out of that method whilst the long lived background task is underway then I get a hang at the end of the method.

This is caused by the m_future.get() line in the destructor which blocks the main (calling) thread.

Is this down to the fact that the future was created in the method and therefore the destructor gets called as the method goes out of scope which in turn blocks (because the future is still in use)? If so, is there any way I can work around this?

Thanks again!

Adam

WillP

December 7, 2016 at 11:17 AM

Hey Adam,
Yep, that was a design decision based on how I'm using the thread pool in my specific application. If you don't want it to block when the future goes out of scope, you have a few options. You can return the future to the calling method, store it somewhere else (std::move), or you could flesh the TaskFuture class out a little more and add a detach() method that could set a flag to check in the destructor before calling .get().
-Will