

Saturday, April 7, 2018 **Latest:** [C++11 Multithreading Tutorial via Q&A – Thread Management Basics](#)

[HOME](#)[CAREER](#)[CODE CRAFT](#)[PERSONAL FINANCE](#)[ABOUT](#)

A CODERS JOURNEY

LIFE LESSONS FROM A CODER



RECENT POSTS

[35 things I learnt at Game Developer Conference \(GDC\) 2018](#)

[System Design Interview Concepts – Consistent Hashing](#)

[Top 20 C++ multithreading mistakes and how to avoid them](#)

[Coding Interview Question – Optimizing toy purchases](#)

[C++11 Multithreading Tutorial via Q&A – Thread Management Basics](#)

[6 Tips to supercharge C++11 vector performance](#)

[Event based synchronization of threads with main game loop](#)

Archives

[April 2018 \(1\)](#)

[October 2017 \(1\)](#)

[August 2017 \(2\)](#)

[ALL](#) [CODE CRAFT](#)

Top 20 C++ multithreading mistakes and how to avoid them

August 17, 2017 Deb Haldar 21 Comments C++, C++ 11, C++ 14, Synchronization, Threads

Threading is one of the most complicated things to get right in programming, especially in C++. I've made a number of mistakes myself over the years. Most of these mistakes were luckily caught in code review and testing ; however, some arcane ones did slip through and make it into production code and we had to patch live systems, which is always expensive.

In this article, I've tried to catalog all the mistakes I know of, with potential solutions. If you know any more pitfalls, or have

alternative suggestions for some of the mistakes – please leave a comment below and I'll factor them into the article.

Mistake # 1: Not using join() to wait for background threads before terminating an application

If we forgot to join a thread or detach it(make it unjoinable) before the main program terminates, it'll cause in a program crash.

In the example below, we forgot to join t1 to the main thread.

```
1.  #include "stdafx.h"
2.  #include <iostream>
3.  #include <thread>
4.
5.  using namespace std;
6.
7.  void LaunchRocket()
8.  {
9.      cout << "Launching Rocket" << endl;
10. }
11.
12. int main()
13. {
14.     thread t1(LaunchRocket);
15.     //t1.join(); // somehow we forgot to join this to
    main thread - will cause a crash.
16.     return 0;
17. }
```

Why does it crash ??? This is because, at the end of the main function, thread t1 goes out of scope and the thread destructor is called. Inside the destructor, a check is performed to see if thread t1 is joinable. A joinable thread is a thread that has not been detached. If the thread is joinable, we call **std::terminate**. Here's what MSVC++ compiler does.

```
1.  ~thread() _NOEXCEPT
2.  { // clean up
3.      if (joinable())
4.          _XSTD terminate();
5.  }
```

There are two ways to fix this depending on your needs.

1. Join the thread t1 to the main thread.

```
1.  int main()
2.  {
3.      thread t1(LaunchRocket);
4.      t1.join(); // join t1 to the main thread
```

January 2017 (1)

November 2016 (1)

October 2016 (2)

August 2016 (1)

June 2016 (2)

May 2016 (4)

April 2016 (2)

March 2016 (1)

February 2016 (3)

January 2016 (3)

```
5.     return 0;
6. }
```

2. Detach the thread t1 from the main thread and let it continue as a daemon thread

```
1. int main()
2. {
3.     thread t1(LaunchRocket);
4.     t1.detach(); // detach t1 from main thread
5.     return 0;
6. }
```

Mistake # 2: Trying to join a thread that has been previously detached

If you have detached a thread and at some point, you cannot rejoin it to the main thread. This is a very obvious error – what makes it problematic is that sometimes you might detach a thread and then write another few hundred lines of code and then try to join the same thread. After all, who remembers what they wrote 300 lines back right?

The problem is that this'll not cause a compilation error (which would have been nice!); instead it'll crash your program. For example:

```
1. #include "stdafx.h"
2. #include <iostream>
3. #include <thread>
4.
5. using namespace std;
6.
7. void LaunchRocket()
8. {
9.     cout << "Launching Rocket" << endl;
10. }
11.
12. int main()
13. {
14.     thread t1(LaunchRocket);
15.     t1.detach();
16.     //..... 100 lines of code
17.     t1.join(); // CRASH !!!
18.     return 0;
19. }
```

The solution is to always check if a thread is join-able before trying to join it to the calling thread.

```
1. int main()
```

```

2.  {
3.      thread t1(LaunchRocket);
4.      t1.detach();
5.      //..... 100 lines of code
6.
7.      if (t1.joinable())
8.      {
9.          t1.join();
10.     }
11.
12.     return 0;
13. }

```

334
Shares

217

Mistake # 3: Not realizing that `std::thread::join()` blocks the main thread

In real world applications, you often need to fork worker threads doing long running operations handling network I/O or waiting for a user input from the user etc. Calling `join` on these worker threads in your main application (UI thread handling rendering) can cause the application to freeze. Often there are better ways to avoid this.

For example, in a GUI application, a worker thread that finishes can send a message to the UI thread. The UI thread itself has a message processing loop that can also receive the messages from the worker threads and can react to them without the necessity of making a blocking `join` call.

For this very reason, the new [WinRT platform](#) from Microsoft has made nearly all actions noticeable to a human user asynchronous and synchronous alternatives are not available. These choices were made to ensure that developers consistently picked APIs that deliver great end-user experiences. Please refer to [Modern C++ and Windows Store Apps](#) for a detailed treatment of the subject.

Mistake # 4: Thinking that thread function arguments are pass by reference by default

Thread function arguments are by default pass by value. So if you need the change persisted in the arguments passed in, you'll need to pass them by reference using `std::ref()`.

Please see items 12 and 13 in this article for details and code examples: <http://www.acodersjourney.com/2017/01/c11-multithreading-tutorial-via-faq-thread-management-basics/>

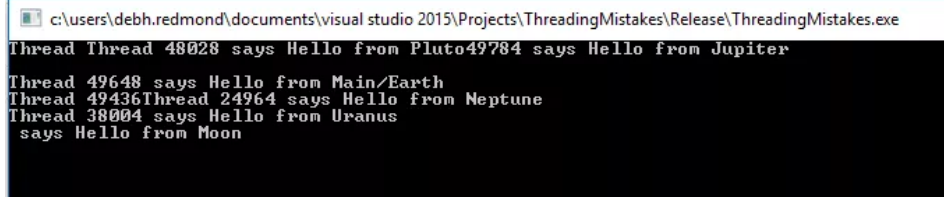
Mistake # 5: Not protecting shared data or shared resources with a critical section (eg. mutex)

In a multithreaded environment, more than one thread is often competing for a resource or shared data. This often results in undefined behavior for the resource or data, unless the resource or data is protected using some mechanics that only allows ONE thread to act on it at a time.

In the example below, **std::cout** is a shared resource that is shared by 6 threads (t1-t5 + main).

```
1.  #include "stdafx.h"
2.  #include <iostream>
3.  #include <string>
4.  #include <thread>
5.  #include <mutex>
6.
7.  using namespace std;
8.
9.  std::mutex mu;
10.
11. void CallHome(string message)
12. {
13.     cout << "Thread " << this_thread::get_id() << " says "
14.     << message << endl;
15. }
16.
17. int main()
18. {
19.     thread t1(CallHome, "Hello from Jupiter");
20.     thread t2(CallHome, "Hello from Pluto");
21.     thread t3(CallHome, "Hello from Moon");
22.
23.     CallHome("Hello from Main/Earth");
24.
25.     thread t4(CallHome, "Hello from Uranus");
26.     thread t5(CallHome, "Hello from Neptune");
27.
28.     t1.join();
29.     t2.join();
30.     t3.join();
31.     t4.join();
32.     t5.join();
33.
34.     return 0;
35. }
```

If we run the program above, we get the following output.



```
c:\users\debh.redmond\documents\visual studio 2015\Projects\ThreadingMistakes\Release\ThreadingMistakes.exe
Thread 48028 says Hello from Pluto49784 says Hello from Jupiter
Thread 49648 says Hello from Main/Earth
Thread 49436 says Hello from Neptune
Thread 24964 says Hello from Uranus
Thread 38004 says Hello from Moon
```

This is because the five threads get the **std::cout** resource in a random fashion. To make the output more deterministic, the solution is to protect the access to std::cout resource using a **std::mutex**. Just change the CallHome() to acquire a mutex before using std::cout and release it after it's done.

```
1. void CallHome(string message)
2. {
3.     mu.lock();
4.     cout << "Thread " << this_thread::get_id() << " says "
        << message << endl;
5.     mu.unlock();
6. }
```


Mistake # 6: Forgetting to release locks after a critical section

In the previous section, you saw how to protect a critical section with a mutex. However, calling lock() and unlock() on mutex is not preferable because you might forget to relinquish a lock that you're holding. What happens then ? Well, all the other threads that are waiting on that resource will be blocked indefinitely and the program might hang.

In our toy example, if we forget to unlock the mutex in CallHome function, we'll print out the first message from thread t1 and the program will hang. This is because thread t1 gets hold of the mutex and all the other threads are essentially waiting to acquire the mutex.

```
1. void CallHome(string message)
2. {
3.     mu.lock();
4.     cout << "Thread " << this_thread::get_id() << " says "
        << message << endl;
5.     //mu.unlock(); ASSUMING WE FORGOT TO RELEASE THE LOCK
6. }
```

The output of running the above code is below – it'll hang on the console screen and not terminate:



```
c:\users\debh.redmond\documents\visual studio 2015\Projects\ThreadingMistakes\Release\ThreadingMistakes.exe
Thread 23840 says Hello from Jupiter
```

Programming errors happen and for this reason it is never preferable to use the lock/unlock syntax on a mutex directly. Instead, you should use **`std::lock_guard`** which uses RAII style to manage the duration of mutex lock. When the lock_guard object is created, it attempts to take ownership of the mutex. When the lock_guard object goes out of scope, the lock_guard object is destroyed which releases the mutex.

We'd modify our CallHome method like this to use the **`std::lock_guard`** object:

```
1. void CallHome(string message)
2. {
3.     std::lock_guard<std::mutex> lock(mu); // Acquire the
        mutex
4.     cout << "Thread " << this_thread::get_id() << " says "
        << message << endl;
5. } // lock_guard object is destroyed and mutex mu is
        released
```

Mistake # 7: Not keeping critical sections as compact and small as possible

When one thread is executing inside the critical section, all other threads trying to enter the critical section are essentially blocked. So we should keep the instructions inside a critical section as small as possible. To illustrate, here's a bad piece of critical section code.

```
1. void CallHome(string message)
2. {
3.     std::lock_guard<std::mutex> lock(mu); // Start of
        Critical Section - to protect std::cout
4.
5.     ReadFiftyThousandRecords();
6.
7.     cout << "Thread " << this_thread::get_id() << " says "
        << message << endl;
8.
9. } // lock_guard object is destroyed and mutex mu is
        released
```

The method ReadFiftyThousandRecords() is a read only operation. There is no reason for it to be inside a lock. If it takes us 10 seconds to read fifty thousand records from a DB, all other threads are blocked for that period of time unnecessarily. This can seriously affect the throughput of the program.

The correct way is to just keep the **`std::cout`** under the critical section.

```

1. void CallHome(string message)
2. {
3.     ReadFiftyThousandRecords(); // Don't need to be in
        critical section because it's a read only operation
4.
5.     std::lock_guard<std::mutex> lock(mu); // Start of
        Critical Section - to protect std::cout
6.
7.     cout << "Thread " << this_thread::get_id() << " says "
        << message << endl;
8.
9. } // lock_guard object is destroyed and mutex mu is
        released

```

Mistake # 8 : Not acquiring multiple locks in the same order

This is one of the most common causes of DEADLOCK, a situation where threads block indefinitely because they are waiting to acquire access to resources currently locked by other blocked threads. Let's see an example:

Thread 1	Thread 2
Lock A	Lock B
//.. Do some processing	//..do some processing
Lock B	Lock A
// .. Do some more processing	//..Do some more processing
Unlock B	Unlock A
Unlock A	Unlock B

In some situations, what's going to happen is that when Thread 1 tries to acquire Lock B, it gets blocked because Thread 2 is already holding lock B. And from Thread 2's perspective, it is blocked on acquiring lock A , but cannot do so because Thread 1 is holding lock A. Thread 1 cannot release lock A unless it has acquired lock B and so on. In other words, your program is hanged at this point.

Here's a code snippet if you want to try to simulate a deadlock:

```

1. #include "stdafx.h"
2. #include <iostream>
3. #include <string>
4. #include <thread>
5. #include <mutex>
6.
7. using namespace std;

```



```
8.
9.     std::mutex muA;
10.    std::mutex muB;
11.
12.    void CallHome_AB(string message)
13.    {
14.        muA.lock();
15.        //Some additional processing
16.
17.        std::this_thread::sleep_for(std::chrono::milliseconds(100));
18.        muB.lock();
19.
20.        cout << "Thread " << this_thread::get_id() << " says "
21.        << message << endl;
22.
23.        muB.unlock();
24.        muA.unlock();
25.    }
26.
27.    void CallHome_BA(string message)
28.    {
29.        muB.lock();
30.        //Some additional processing
31.
32.        std::this_thread::sleep_for(std::chrono::milliseconds(100));
33.        muA.lock();
34.
35.        cout << "Thread " << this_thread::get_id() << " says "
36.        << message << endl;
37.
38.        muA.unlock();
39.        muB.unlock();
40.    }
41.
42.    int main()
43.    {
44.        thread t1(CallHome_AB, "Hello from Jupiter");
45.        thread t2(CallHome_BA, "Hello from Pluto");
46.
47.        t1.join();
48.        t2.join();
49.
50.        return 0;
51.    }
```

If you run this, it'll hang. Go ahead and break into debugger to look at the threads window and you'll see that Thread 1 (calling function CallHome_Th1()) is trying to acquire mutex B while thread 2 (calling function CallHome_Th2()) is trying to acquire mutex A. None of them is making any progress because of the deadlock! See screenshot below.

```

std::mutex muA;
std::mutex muB;

void CallHome_Th1(string message)
{
    muA.lock();
    //Some additional processing
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    muB.lock();

    cout << "Thread " << this_thread::get_id() << " says " << message << endl;

    muB.unlock();
    muA.unlock();
}

void CallHome_Th2(string message)
{
    muB.lock();
    //Some additional processing
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    muA.lock();

    cout << "Thread " << this_thread::get_id() << " says " << message << endl;

    muA.unlock();
    muB.unlock();
}

```

So, what can you do about it ? The best thing to do is to structure your code in such a way that all locks are acquired in the same order.

Depending on your situation, you can also employ the following strategies:

1. Acquire locks together if both need to be acquired :

```
1. std::scoped_lock lock{muA, muB};
```

2. You can use a **timed mutex** where you can mandate that a lock be released after a timeout if it's not already available.

Mistake # 9: Trying to acquire a `std::mutex` twice

Trying to acquire a mutex twice will cause undefined behavior. In most debug implementations, it'll likely result in a crash. For example, in the code below, `LaunchRocket()` locks a mutex and then calls `StartThruster()`. What's interesting is that there will be no issue in the normal code path – the problem will only happen when the exception codepath is triggered, in which case we'll get in an undefined state/crash.

```

1. #include "stdafx.h"
2. #include <iostream>
3. #include <thread>
4. #include <mutex>
5.
6. std::mutex mu;
7.
8. static int counter = 0;

```

```
9.
10.
11. void StartThruster()
12. {
13.     try
14.     {
15.         // Some operation to start thruster
16.     }
17.     catch (...)
18.     {
19.         std::lock_guard<std::mutex> lock(mu);
20.         std::cout << "Launching rocket" << std::endl;
21.     }
22. }
23.
24. void LaunchRocket()
25. {
26.     std::lock_guard<std::mutex> lock(mu);
27.     counter++;
28.     StartThruster();
29. }
30.
31. int main()
32. {
33.     std::thread t1(LaunchRocket);
34.     t1.join();
35.     return 0;
36. }
```

The fix is to structure your code in such a way that it does not try to acquire a previously locked mutex. A superficial solution might be to just use a **`std::recursive_mutex`** — but this is almost always indicative of a bad design.

Mistake # 10: Using mutexes when `std::atomic` types will suffice

When you have simple data types that needs to be updated, for example, a simple bool or a integer counter, using `std::atomic` will almost yield better performance than using a mutex.

For example, instead of doing :

```
1. int counter;
2. ....
3. mu.lock();
4. counter++;
5. mu.unlock();
```

Try:

```
1. std::atomic<int> counter;
2. ...
```

```
3. counter++;
```

For a detailed analysis of using mutex vs atomics, please refer to <https://www.arangodb.com/2015/02/comparing-atomic-mutex-rwlocks/>

Mistake # 11: Creating and Destroying a lot of threads

directly when using a thread pool is available

Creating and deleting threads are expensive in terms of the CPU time. Imagine trying to create a thread when the system is trying to perform a complex process like rendering graphics or calculating game physics. A technique often employed is to create a pool of preallocated threads that can handle routine tasks like logging to disks or sending data across network throughout the life of the process.

The other benefit of using threadpool threads instead of spinning your own is that you don't have to worry about oversubscription whereby you can affect system performance.

Also, all the gory details of thread lifecycle management are taken care off for you, which would mean less code and less BUGS !

Two of the most popular libraries that implements thread pools are [Intel Thread Building Blocks\(TBB\)](#) and [Microsoft Parallel Patterns Library\(PPL\)](#).

Mistake # 12: Not handling exceptions in background threads

Exceptions thrown in one thread cannot be caught in another thread. Let's assume we have a function that can throw an exception. If we execute this function in a seperate thread forked from main and expect to catch any exception from this thread in the main thread, it's not going to work. Here's an example:

```
1. #include "stdafx.h"
2. #include<iostream>
3. #include<thread>
4. #include<exception>
5. #include<stdexcept>
6.
7. static std::exception_ptr teptr = nullptr;
8.
9. void LaunchRocket ()
10. {
```

```
11.     throw std::runtime_error("Catch me in MAIN");
12. }
13.
14. int main()
15. {
16.     try
17.     {
18.         std::thread t1(LaunchRocket);
19.         t1.join();
20.     }
21.     catch (const std::exception &ex)
22.     {
23.         std::cout << "Thread exited with exception: " <<
ex.what() << "\n";
24.     }
25.
26.     return 0;
27. }
```

The above program will crash and the catch block in main() will do nothing to handle the exception thrown thread t1.

The solution is to use the C++11 feature `std::exception_ptr` to capture exception thrown in a background thread. Here are the steps you need to do:

1. Create a global instance of `std::exception_ptr` initialized to `nullptr`
2. Inside the function that executes in the forked thread, catch any exception and set the `std::current_exception()` to the `std::exception_ptr` in step#1
3. Inside the main thread, check if the global exception pointer is set.
4. If yes, use `std::rethrow_exception(exception_ptr p)` to rethrow the exception referenced by `exception_ptr` parameter.

Rethrowing the referenced exception does not have to be done in the same thread that generated the referenced exception in the first place, which makes this feature perfectly suited for handling exceptions across different threads.

The code below achieves safe handling of exceptions in background thread.

```
1.     #include "stdafx.h"
2.     #include<iostream>
3.     #include<thread>
4.     #include<exception>
5.     #include<stdexcept>
6.
7.     static std::exception_ptr globalExceptionPtr = nullptr;
8.
```

```
9.  void LaunchRocket ()
10. {
11.     try
12.     {
13.
14.         std::this_thread::sleep_for(std::chrono::milliseconds(100));
15.         throw std::runtime_error("Catch me in MAIN");
16.     }
17.     catch (...)
18.     {
19.         //Set the global exception pointer in case of an
20.         exception
21.         globalExceptionPtr = std::current_exception();
22.     }
23. }
24.
25. int main()
26. {
27.     std::thread t1(LaunchRocket);
28.     t1.join();
29.
30.     if (globalExceptionPtr)
31.     {
32.         try
33.         {
34.             std::rethrow_exception(globalExceptionPtr);
35.         }
36.         catch (const std::exception &ex)
37.         {
38.             std::cout << "Thread exited with exception: " <<
39.             ex.what() << "\n";
40.         }
41.     }
42.
43.     return 0;
44. }
```

Mistake # 13: Using threads to simulate Asyn jobs when std::async will do

If you just need some code executed asynchronously i.e. without blocking execution of Main thread, your best bet is to use the `std::async` functionality to execute the code. The same could be achieved by creating a thread and passing the executable code to the thread via a function pointer or lambda parameter. However, in the later case, you're responsible for managing creation and joining/detaching of them thread , as well as handling any exceptions that might happen in the thread. If you use **`std::async`** , you just get rid of all these hassels and also dramatically reduce chances of getting into a deadlock scenario.

Another huge advantage of using `std::async` is the ability to get the result of the task communicated back to the calling thread via a `std::future` object. For example, assuming we have a function `ConjureMagic` which returns an `int`, we can spin an `async` task that sets a future when it's done and we can extract the result from that future in our calling thread when at an opportune time.

```
1. // spin an async task and get a handle to the future
2. std::future<int> asyncResult2 = std::async(&ConjureMagic);
3.
4. //... do some processing while the future is being set
5.
6. // Get the result from the future
7. int v = asyncResult2.get();
```

On the contrary, getting the result back from a worker thread to a calling thread is much more cumbersome. The two options include:

1. Passing reference to a result variable to the thread in which the thread stores the results.
2. Store the result inside a class member variable of a function object which can be retrieved once the thread has finished executing.

On the performance front, [Kurt Guntheroth](#) found that creating a thread is 14 times more expensive than using an `async`.

To summarize, use **`std::async`** by default unless you can find good justification for using `std::thread` directly.

Mistake # 14: Not using `std::launch::async` if asynchronicity is desired

`std::async` is a bit of a misnomer because the function in its default form may not execute in an asynchronous way !

There are two launch policies for `std::async`:

1. **`std::launch::async`** : The task is launched immediately in a separate thread
2. **`std::launch::deferred`**: The task is not launched immediately, but is deferred until a **`.get()`** or **`.wait()`** call is made on the future returned by the `std::async`. At the point such a call is made, the task is executed synchronously.

When **`std::async`** is launched with default parameters, it's a combination of these two policies which essentially makes the behavior unpredictable. There's a set of other complications that

tags along using `std::async` with default launch parameters as well – these include, inability to predict whether thread local variables are properly accessed, the async task running the risk of not being run at all because `.get()` or `.wait()` may not get called along all codepaths and loops which wait for the future status to be ready never finishing because the future returned by `std::async` may start off in a deferred state.

So, to avoid all these complications, ALWAYS launch `std::async` with the `std::launch::async` launch parameter.

DON'T DO THIS:

```
1. //run myFunction using default std::async policy
2. auto myFuture = std::async(myFunction);
```

DO THIS INSTEAD:

```
1. //run myFunction asynchronously
2. auto myFuture = std::async(std::launch::async,
    myFunction);
```

For a more detailed discussion, please see Scott Meyer's [Effective Modern C++](#).

Mistake # 15: Calling .Get() on a std::future in a time sensitive code path

The following code retrieves the result from the future returned by an async task. However, the while loop will be blocked till the async task finishes(10 seconds in this case). If you consider this as a loop which renders data on screen, it can lead to a very bad user experience.

```
1. #include "stdafx.h"
2. #include <future>
3. #include <iostream>
4.
5. int main()
6. {
7.     std::future<int> myFuture =
8.     std::async(std::launch::async, []()
9.     {
10.         std::this_thread::sleep_for(std::chrono::seconds(10));
11.         return 8;
12.     });
13.     // Update Loop for rendering data
14.     while (true)
```



```
15.     {
16.         // Render some info on the screen
17.         std::cout << "Rendering Data" << std::endl;
18.
19.         int val = myFuture.get(); // this blocks for 10
seconds
20.
21.         // Do some processing with Val
22.     }
23.
24.     return 0;
25. }
```

Note: There is an additional problem with the code above – it tries to poll a future a second time when it has with no shared state – because the state of the future was retrieved on the first iteration of the loop.

The solution is to check if the future is valid before calling `t.get()`. This way we neither block on the completion of async job nor we try to poll an already retrieved future.

Here's the code snippet that achieves this:

```
1.     #include "stdafx.h"
2.     #include <future>
3.     #include <iostream>
4.
5.     int main()
6.     {
7.         std::future<int> myFuture =
std::async(std::launch::async, [] ()
8.         {
9.
std::this_thread::sleep_for(std::chrono::seconds(10));
10.        return 8;
11.    });
12.
13.    // Update Loop for rendering data
14.    while (true)
15.    {
16.        // Render some info on the screen
17.        std::cout << "Rendering Data" << std::endl;
18.
19.        if (myFuture.valid())
20.        {
21.            int val = myFuture.get(); // this blocks for 10
seconds
22.
23.            // Do some processing with Val
24.        }
25.    }
26.
27.    return 0;
28. }
```

Mistake # 16: Not realizing that an exception thrown inside an async task is propagated when `std::future::get()` is invoked.

Imagine you have the following piece of code – what do you think will be the result of calling the **`std::future::get()`** ?

```
1.  #include "stdafx.h"
2.  #include <future>
3.  #include <iostream>
4.
5.  int main()
6.  {
7.      std::future<int> myFuture =
std::async(std::launch::async, [] ()
8.      {
9.          throw std::runtime_error("Catch me in MAIN");
10.         return 8;
11.     });
12.
13.     if (myFuture.valid())
14.     {
15.         int result = myFuture.get();
16.     }
17.
18.     return 0;
19. }
```

If you guessed a crash, you're absolutely correct !

The exception from async tasks is only propagated when we call get on the future. If get is not called, the exception is ignored and discarded when the future goes out of scope.

So, if your async tasks can throw, you should always wrap the call to **`std::future::get()`** in a try/catch block. Here's an example:

```
1.  #include "stdafx.h"
2.  #include <future>
3.  #include <iostream>
4.
5.  int main()
6.  {
7.      std::future<int> myFuture =
std::async(std::launch::async, [] ()
8.      {
9.          throw
std::runtime_error("Catch me in MAIN");
10.         return 8;
11.     });
12.
13.     if (myFuture.valid())
14.     {
```

```

15.                                     try
16.                                     {
17.                                     int
    result = myFuture.get();
18.                                     }
19.                                     catch (const
    std::runtime_error& e)
20.                                     {
21.                                     std::cout
    << "Async task threw exception: " << e.what() <<
    std::endl;
22.                                     }
23.                                     }
24.
25.                                     return 0;
26.     }

```

Mistake # 17: Using `std::async` when you need granular control over thread execution

While using `std::async` should suffice in most cases, there are situations where you'd want more granular control over the thread executing your code. For example, if you want to pin the thread to a specific CPU core in a multi processor system (like Xbox etc.)

The following piece of code sets the processor affinity of the thread to core 5 of my system.

```

1.  #include "stdafx.h"
2.  #include <windows.h>
3.  #include <iostream>
4.  #include <thread>
5.
6.  using namespace std;
7.
8.  void LaunchRocket()
9.  {
10.     cout << "Launching Rocket" << endl;
11. }
12.
13. int main()
14. {
15.     thread t1(LaunchRocket);
16.
17.     DWORD result =
    ::SetThreadIdealProcessor(t1.native_handle(), 5);
18.
19.     t1.join();
20.
21.     return 0;
22. }

```

This is made possible by using the `native_handle` of the **`std::thread`**, and passing it to an Win32 thread API function. There's a bunch of

other functionality exposed via the [Win32 Threads API](#) that is not exposed in **`std::thread`** or **`std::async`**. `std::Async` makes these underlying platform features inaccessible which makes it not suitable for more sophisticated work.

The other option is to create a **`std::packaged_task`** and move it to the desired thread of execution after setting thread properties.

Mistake # 18: Creating many more "Runnable" threads than available cores

Threads can be classified into two types from design perspective – Runnable threads and Waitable threads.

Runnable threads consume 100% of the CPU time of the core on which they run. When more than one runnable thread is scheduled on a single core, they effectively time slice the CPU time of the core. There is no performance gain achieved when more than one runnable thread is scheduled on a single core- in fact there is a performance degradation due to additional context switches involved.

Waitable threads consumes only a few cycles of the core they run on while waiting for events or network I/O etc. This leaves majority of the available compute time of the CPU core unused. That's why it's beneficial to schedule multiple waitable threads on a single core because one waitable thread can process data while others are waiting for some event to happen. Scheduling multiple waitable threads on a single core can provide much larger throughput from your program.

So, how do you get the number of runnable threads the system can support ? Use `std::thread::hardware_concurrency()` . This function will generally return the number of processor cores – but it will factor in cores that behave as two or more logical cores due to [hyperthreading](#).

You should use this value from your target platform to plan the max number of Runnable threads your program should concurrently use. You can also designate a core for all your waitable threads and use the remaining number of cores for runnable threads. For example, on a quad-core system, use one core for ALL waitable threads and use three runnable threads for the remaining three cores. Depending on your thread schedulers efficiency, a few of your runnable threads might get context switched out (due to page faults etc.) leaving the core idle for some

amount of time. If you observe this situation during profiling, you should create a few more runnable threads than the number of your cores and tune it for your system.

Mistake # 19: Using "volatile" keyword for synchronization

The "volatile" keyword in front of a variable type declaration does not make the operations on that variable atomic or thread safe in any way. What you probably want is an `std::atomic`.

See this [stackoverflow](#) article discussion for more details.

Mistake # 20: Using a Lock Free architecture unless absolutely needed

There is something about complexity that appeals to every engineer. Lock free programming sounds very sexy when compared to regular synchronization mechanisms such as mutex, condition variables, async etc. However, every seasoned C++ developer I've spoken to has had the opinion that using lock free programming as first resort is a form of premature optimization that can come back to haunt you at the most in opportune time (Think a crash out in production when you don't have the full heap dump !).

In my C++ career, there has been only one piece of tech which needed the performance of lock free code because we're on a resource constrained system where each transaction from our component needed to take no more than 10 micro seconds.

So, before you start thinking going the lock free route, please ask yourself these three questions in order:

1. Have you considered designing your system such that it does not need a synchronization mechanism ? The best synchronization is often "No synchronization" !
2. If you do need synchronization, have you profiled your code to understand the performance characteristics ? If yes, have you tried to optimize the hot code paths?
3. Can you scale out instead of scaling up ?

In a nutshell, for regular application development, please consider lock free programming only when you've exhausted all other alternatives. *Another way to look at it(suggested by one of my readers) is that if you're still making some of the the above 19 mistakes, you should probably stay away from lock free programming 😊*

Please share if you enjoyed this article.

← [Coding Interview Question – Optimizing toy purchases](#)

[System Design Interview Concepts – Consistent Hashing](#) →

Copyright © 2018 [A CODER'S JOURNEY](#). All rights reserved.

Theme: ColorMag by [ThemeGrill](#). Powered by [WordPress](#).