

Thread-Safe Initialization of a Singleton

30 August 2016

Like 6 Share 16 Like 6



(<http://www.facebook.com/rainer.grimm.31>)



(<https://plus.google.com/u/0/109576694736160795401/posts?pr>)



(<https://www.linkedin.com/in/rainergrimm>)



(https://twitter.com/rainer_grimm)



(https://www.xing.com/profile/Rainer_Grimm12)

Contents [Show]

There are a lot of issues with the singleton pattern. I'm totally aware of that. But the singleton pattern is an ideal use case for a variable, which has only to be initialized in a thread safe way. From that point on you can use it without synchronization. So in this post I discuss different ways to initialize a singleton in a multithreading environment. You get the performance numbers and can reason about your use cases for the thread safe initialization of a variable.

There are a lot of different ways to initialize a singleton in C++11 in a thread safe way. From a birds eye you can have guarantees from the C++ runtime, locks or atomics. I'm totally curious about the performance implications.

My strategy

I use as reference point for my performance measurement a singleton object which I sequentially access 40 million times. The first access will initialize the object. In contrast, the access from the multithreading program will be done by 4 threads. Here I'm only interested in the performance. The program will run on two real PCs. My Linux PC has four, my Windows PC has two cores. I compile the program with maximum and without optimization. For the translation of the program with maximum optimization I have to use a volatile variable in the static method getInstance. If not the compiler will optimize away my access to the singleton and my program becomes too fast.

I have three questions in my mind:

1. How is the relative performance of the different singleton implementations?
2. Is there a significant difference between Linux (gcc) and Windows (cl.exe)?
3. What's the difference between the optimized and non-optimized versions?

Finally, I collect all numbers in a table. The numbers are in seconds.

The reference values

The both compilers

The command line gives you the details to the compiler. Here are the gcc and the cl.exe.

```
rainer@linux:~$ g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib64/gcc/x86_64-suse-linux/4.8/lto-wrapper
Target: x86_64-suse-linux
Configured with: ../configure --prefix=/usr --infodir=/usr/share/info --mandir=/usr/share/man --libdir=/usr/lib64 --libexecdir=/usr/lib64 --enable-languages=c,c++,objc,fortran,obj-c++,java,ada --enable-checking=release --with-gx-include-dir=/usr/include/c++/4.8 --enable-ssp --disable-libssp --disable-pg-lugin --with-bugurl=http://bugs.opensuse.org/ --with-pkgversion='SUSE Linux' --disable-libgomp --disable-libmudflap --with-slibdir=/usr/lib64 --with-system-zlib --enable-cxx-atomic --enable-libstdc++-allocator=new --disable-libstdc++-pch --enable-version-specific-runtime-libs --enable-linker-build-id --enable-linux-futex --program-suffix=-4.8 --without-system-libunwind --with-arch=x86_64 --with-tune=generic --build=x86_64-suse-linux --host=x86_64-suse-linux
Thread model: posix
gcc version 4.8.3 20140627 [gcc-4.8-branch revision 212064] (SUSE Linux)
rainer@linux:~$
```

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0>cl.exe
Microsoft (R) C/C++-Optimierungscompiler Version 19.00.23506 für x86
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Syntax: cl [ Option... ] Dateiname... [ /link Linkeroption... ]

C:\Program Files (x86)\Microsoft Visual Studio 14.0>
```

The reference code

At first, the single threaded case. Of course without synchronization.

```

1 // singletonSingleThreaded.cpp
2
3 #include <chrono>
4 #include <iostream>
5
6 constexpr auto tenMill= 10000000;
7
8 class MySingleton{
9 public:
10     static MySingleton& getInstance(){
11         static MySingleton instance;
12         // volatile int dummy{};
13         return instance;
14     }
15 private:
16     MySingleton()= default;
17     ~MySingleton()= default;
18     MySingleton(const MySingleton&)= delete;
19     MySingleton& operator=(const MySingleton&)= delete;
20
21 };
22
23 int main(){
24
25     constexpr auto fortyMill= 4* tenMill;
26
27     auto begin= std::chrono::system_clock::now();
28
29     for ( size_t i= 0; i <= fortyMill; ++i){
30         MySingleton::getInstance();
31     }
32
33     auto end= std::chrono::system_clock::now() - begin;
34
35     std::cout << std::chrono::duration<double>(end).count() << std::endl;
36
37 }

```


<http://www.facebook.com/rainer.gruber>

<https://plus.google.com/u/0/109576694736160795401>

<https://www.linkedin.com/in/rainergruber>

https://twitter.com/rainer_gruber

https://www.xing.com/profile/Rainer_Gruber
[Hunting http://www.hunterspeak.com/](http://www.hunterspeak.com/)

I use in the reference implementation the so called Meyers Singleton. The elegance of this implementation is that the singleton object instance in line 11 is a static variable with block scope. Therefore, instance will exactly be initialized, when the static method `getInstance` (line 10 - 14) will be executed the first time. In line 14 the `volatile` variable `dummy` is commented out. When I translate the program with maximum optimization that has to change. So the call `MySingleton::getInstance()` will not be optimized away.

Now the raw numbers on Linux and Windows.

Without optimization

```

rainer : bash - Konsole
File Edit View Bookmarks Settings Help
rainer@linux:~> singletonSingleThreaded
0.0913362
rainer@linux:~>

```

```

Eingabeaufforderung
C:\Users\Rainer> singletonSingleThreaded
0.0923164
C:\Users\Rainer>

```

Maximum Optimization

```

rainer: bash - Konsole <9>
File Edit View Bookmarks Settings Help
rainer@linux:~> singletonSingleThreaded
0.0288182
rainer@linux:~>
rainer: bash

```

```

C:\Users\Rainer> singletonSingleThreaded.exe
0.0220456
C:\Users\Rainer>

```


<http://www.facebook.com/rainer.gri>

<https://plus.google.com/u/0/109576694736160795401>

<https://www.linkedin.com/in/rainer>

https://twitter.com/rainer_gri

https://www.xing.com/profile/Rainer_Gri
<http://www.hunterspeak.com/>

Guarantees of the C++ runtime

I already presented the details to the thread safe initialization of variables in the post [initialization-of-data](#)

Meyers Singleton

The beauty of the Meyers Singleton in C++11 is that it's automatically thread safe. That is guaranteed by the standard: Static variables with block scope. ([/index.php/thread-safe-initialization-of-data](#)) The Meyers Singleton is a static variable with block scope, so we are done. It's still left to rewrite the program for four threads.

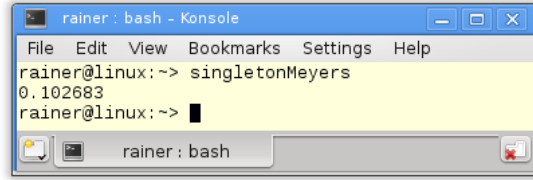
```

1 // singletonMeyers.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <future>
6
7 constexpr auto tenMill= 10000000;
8
9 class MySingleton{
10 public:
11     static MySingleton& getInstance(){
12         static MySingleton instance;
13         // volatile int dummy{};
14         return instance;
15     }
16 private:
17     MySingleton()= default;
18     ~MySingleton()= default;
19     MySingleton(const MySingleton&)= delete;
20     MySingleton& operator=(const MySingleton&)= delete;
21
22 };
23
24 std::chrono::duration<double> getTime(){
25
26     auto begin= std::chrono::system_clock::now();
27     for ( size_t i= 0; i <= tenMill; ++i){
28         MySingleton::getInstance();
29     }
30     return std::chrono::system_clock::now() - begin;
31
32 };
33
34 int main(){
35
36     auto fut1= std::async(std::launch::async,getTime);
37     auto fut2= std::async(std::launch::async,getTime);
38     auto fut3= std::async(std::launch::async,getTime);
39     auto fut4= std::async(std::launch::async,getTime);
40
41     auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
42
43     std::cout << total.count() << std::endl;
44
45 }

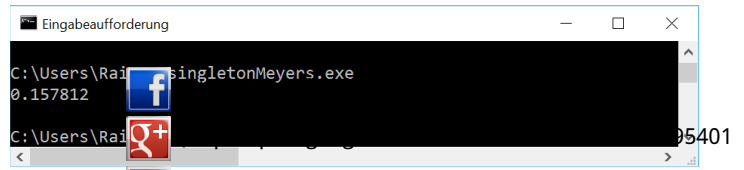
```

I use the singleton object in the function `getTime` (line 24 - 32). The function is executed by the four promise ([/index.php/asynchronous-function-calls](#)) in line 36 - 39. The results of the associate futures ([/index.php/asynchronous-function-calls](#)) are summed up in line 41. That's all. Only the execution time is missing.

Without optimization

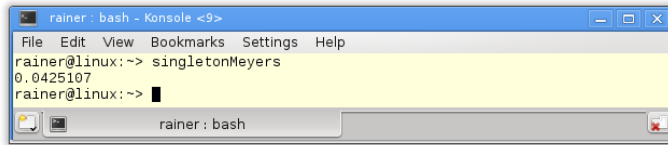


```
rainer : bash - Konsole
File Edit View Bookmarks Settings Help
rainer@linux:~> singletonMeyers
0.102683
rainer@linux:~> █
```

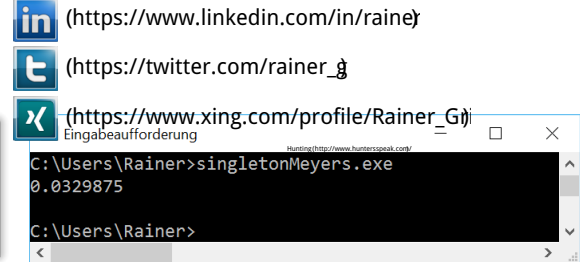


```
Eingabeaufforderung
C:\Users\Rainer> singletonMeyers.exe
0.157812
C:\Users\Rainer> █
```

Maximum optimization



```
rainer : bash - Konsole <9>
File Edit View Bookmarks Settings Help
rainer@linux:~> singletonMeyers
0.0425107
rainer@linux:~> █
```



```
Eingabeaufforderung
C:\Users\Rainer> singletonMeyers.exe
0.0329875
C:\Users\Rainer> █
```

The next step is the function `std::call_once` in combination with the flag `std::once_flag`.

The function `std::call_once` and the flag `std::once_flag`

You can use the function `std::call_once` (</index.php/thread-safe-initialization-of-data>) to register a callable which will be executed exactly once. The flag `std::call_once` in the following implementation guarantees that the singleton will be thread safe initialized.

```

1 // singletonCallOnce.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <future>
6 #include <mutex>
7 #include <thread>
8
9 constexpr auto tenMill= 10000000;
10
11 class MySingleton{
12 public:
13     static MySingleton& getInstance(){
14         std::call_once(initInstanceFlag, &MySingleton::initSingle
15         // volatile int dummy{};
16         return *instance;
17     }
18 private:
19     MySingleton()= default;
20     ~MySingleton()= default;
21     MySingleton(const MySingleton&)= delete;
22     MySingleton& operator=(const MySingleton&)= delete;
23
24     static MySingleton* instance;
25     static std::once_flag initInstanceFlag;
26
27     static void initSingleton(){
28         instance= new MySingleton;
29     }
30 };
31
32 MySingleton* MySingleton::instance= nullptr;
33 std::once_flag MySingleton::initInstanceFlag;
34
35 std::chrono::duration<double> getTime(){
36
37     auto begin= std::chrono::system_clock::now();
38     for ( size_t i= 0; i <= tenMill; ++i){
39         MySingleton::getInstance();
40     }
41     return std::chrono::system_clock::now() - begin;
42 };
43
44 int main(){
45
46     auto fut1= std::async(std::launch::async,getTime);
47     auto fut2= std::async(std::launch::async,getTime);
48     auto fut3= std::async(std::launch::async,getTime);
49     auto fut4= std::async(std::launch::async,getTime);
50
51     auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
52
53     std::cout << total.count() << std::endl;
54
55 }
56

```

<http://www.facebook.com/rainer.gr><https://plus.google.com/u/0/109576694736160795401>https://www.linkedin.com/in/rainer_ghttps://twitter.com/rainer_ghttps://www.xing.com/profile/Rainer_GHunting(<http://www.hunterspeak.com/>)

Here are the numbers.

Without optimization

```

rainer: bash - Konsole <2>
File Edit View Bookmarks Settings Help
rainer@linux:~> singletonCallOnce
1.91897
rainer@linux:~>

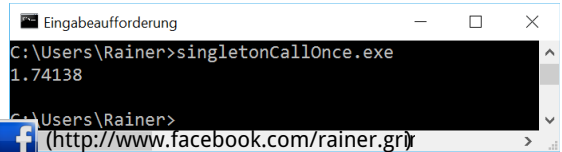
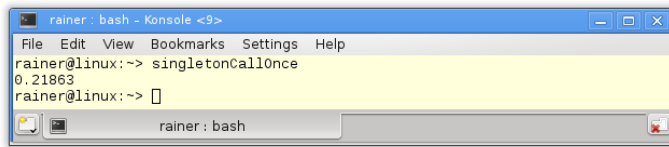
```

```

Eingabeaufforderung
C:\Users\Rainer>singletonCallOnce.exe
4.66457
C:\Users\Rainer>

```

Maximum optimization



Of course the most obvious way is it protect the singleton with a lock.

Lock

The mutex wrapped in a lock (/index.php/prefer-locks-to-mutexes) guarantees that the singleton will be thread safe initialized.

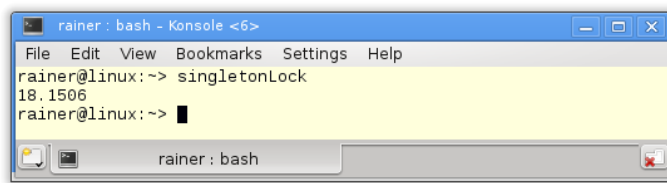
```

1 // singletonLock.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <future>
6 #include <mutex>
7
8 constexpr auto tenMill= 10000000;
9
10 std::mutex myMutex;
11
12 class MySingleton{
13 public:
14     static MySingleton& getInstance(){
15         std::lock_guard<std::mutex> myLock(myMutex);
16         if ( !instance ){
17             instance= new MySingleton();
18         }
19         // volatile int dummy{};
20         return *instance;
21     }
22 private:
23     MySingleton()= default;
24     ~MySingleton()= default;
25     MySingleton(const MySingleton&)= delete;
26     MySingleton& operator=(const MySingleton&)= delete;
27
28     static MySingleton* instance;
29 };
30
31 MySingleton* MySingleton::instance= nullptr;
32
33 std::chrono::duration<double> getTime(){
34
35     auto begin= std::chrono::system_clock::now();
36     for ( size_t i= 0; i <= tenMill; ++i){
37         MySingleton::getInstance();
38     }
39     return std::chrono::system_clock::now() - begin;
40 };
41
42 };
43
44 int main(){
45
46     auto fut1= std::async(std::launch::async,getTime);
47     auto fut2= std::async(std::launch::async,getTime);
48     auto fut3= std::async(std::launch::async,getTime);
49     auto fut4= std::async(std::launch::async,getTime);
50
51     auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
52
53     std::cout << total.count() << std::endl;
54 }

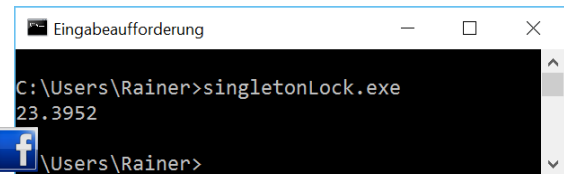
```

How fast is the classical thread safe implementation of the singleton pattern?





Without optimization



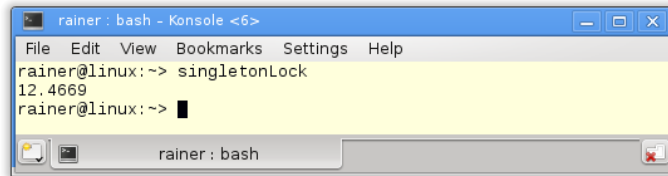
```
rainer: bash - Konsole <6>
File Edit View Bookmarks Settings Help
rainer@linux:~> singletonLock
18.1506
rainer@linux:~> █
```



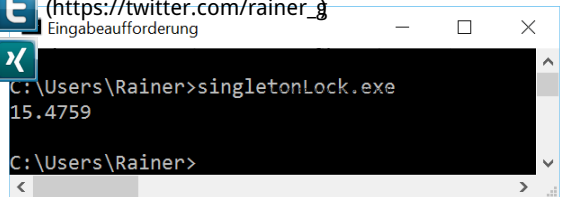
```
Eingabeaufforderung
C:\Users\Rainer>singletonLock.exe
23.3952
C:\Users\Rainer> █
```

 <https://plus.google.com/u/0/109576694736160795401>
 <https://www.linkedin.com/in/rainer>
 https://twitter.com/rainer_g


Maximum optimization



```
rainer: bash - Konsole <6>
File Edit View Bookmarks Settings Help
rainer@linux:~> singletonLock
12.4669
rainer@linux:~> █
```



```
Eingabeaufforderung
C:\Users\Rainer>singletonLock.exe
15.4759
C:\Users\Rainer> █
```

Not so fast. Atomics should make the difference.

Atomic variables

With atomic variables my job becomes extremely challenging. Now I have to use the C++ memory model (</index.php/c-memory-model>). I base my implementation on the well-known double-checked locking pattern. (</index.php/thread-safe-initialization-of-data>)

Sequential consistency

The handle to the singleton is an atomic. Because I didn't specify the C++ memory model the default applies: Sequential consistency. (</index.php/sequential-consistency>)

```

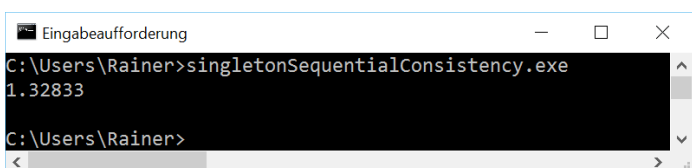
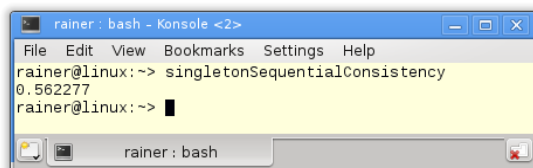
1 // singletonAcquireRelease.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <future>
6 #include <mutex>
7 #include <thread>
8
9 constexpr auto tenMill= 10000000;
10
11 class MySingleton{
12 public:
13     static MySingleton* getInstance(){
14         MySingleton* sin= instance.load();
15         if ( !sin ){
16             std::lock_guard<std::mutex> myLock(myMutex);
17             sin= instance.load();
18             if( !sin ){
19                 sin= new MySingleton();
20                 instance.store(sin);
21             }
22         }
23         // volatile int dummy{};
24         return sin;
25     }
26 private:
27     MySingleton()= default;
28     ~MySingleton()= default;
29     MySingleton(const MySingleton&)= delete;
30     MySingleton& operator=(const MySingleton&)= delete;
31
32     static std::atomic<MySingleton*> instance;
33     static std::mutex myMutex;
34 };
35
36
37 std::atomic<MySingleton*> MySingleton::instance;
38 std::mutex MySingleton::myMutex;
39
40 std::chrono::duration<double> getTime(){
41     auto begin= std::chrono::system_clock::now();
42     for ( size_t i= 0; i <= tenMill; ++i){
43         MySingleton::getInstance();
44     }
45     return std::chrono::system_clock::now() - begin;
46 };
47
48 };
49
50
51 int main(){
52
53     auto fut1= std::async(std::launch::async,getTime);
54     auto fut2= std::async(std::launch::async,getTime);
55     auto fut3= std::async(std::launch::async,getTime);
56     auto fut4= std::async(std::launch::async,getTime);
57
58     auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
59
60     std::cout << total.count() << std::endl;
61
62 }

```

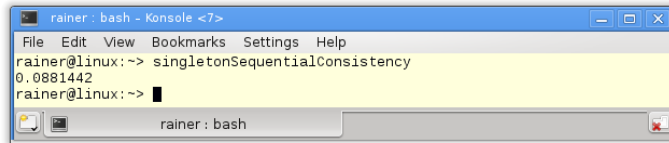
<http://www.facebook.com/rainer.gr><https://plus.google.com/u/0/109576694736160795401>https://www.linkedin.com/in/rainer_https://twitter.com/rainer_ghttps://www.xing.com/profile/Rainer_GHunting(<http://www.hunterspeak.com/>)

Now I'm curious.

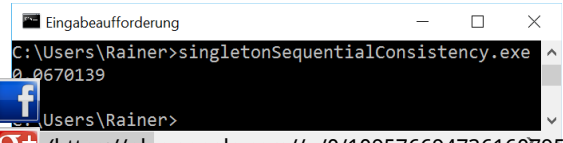
Without optimization



Maximum optimization



```
rainer: bash - Konsole <7>
File Edit View Bookmarks Settings Help
rainer@linux:~> singletonSequentialConsistency
0.0881442
rainer@linux:~>
```



```
Eingabeaufforderung
C:\Users\Rainer>singletonSequentialConsistency.exe
0.08670139
C:\Users\Rainer>
```



(<https://plus.google.com/u/0/109576694736160795401>)

(<https://www.linkedin.com/in/rainer>)



(https://twitter.com/rainer_g)



(https://www.xing.com/profile/Rainer_Gj)

Hunting(<http://www.hunterspeak.com/>)

But we can do better. There is an additional optimization possibility.

Acquire-release Semantic

The reading of the singleton (line 14) is an acquire operation, the writing a release operation (line 20). Because both operations take place on the same atomic I don't need sequential consistency. The C++ standard guarantees that an acquire operation synchronizes with a release operation on the same atomic. This conditions hold in this case therefore I can weaken the C++ memory model in line 14 and 20. Acquire-release semantic (/index.php/acquire-release-semantic) is sufficient.

```

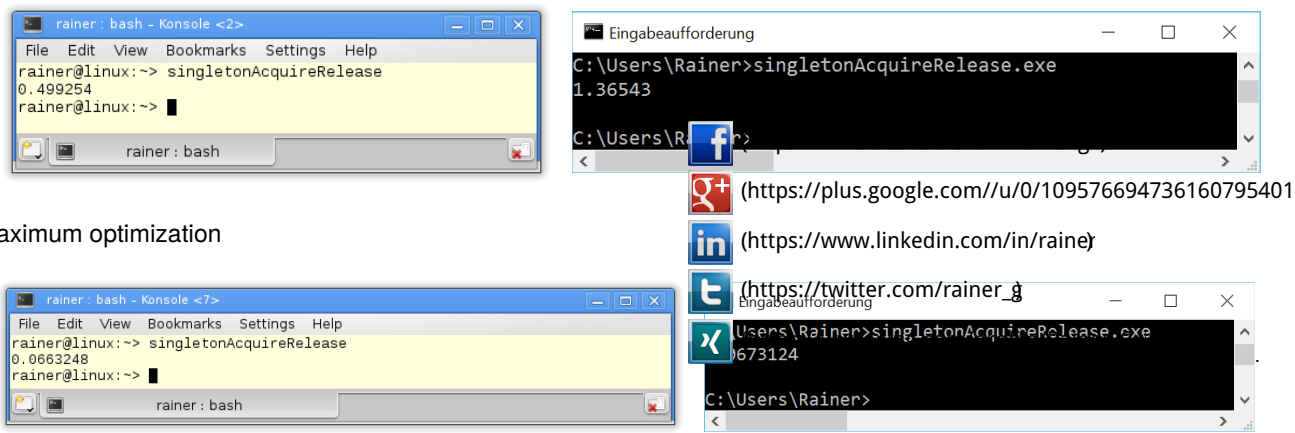
1 // singletonAcquireRelease.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <future>
6 #include <mutex>
7 #include <thread>
8
9 constexpr auto tenMill= 10000000;
10
11 class MySingleton{
12 public:
13     static MySingleton* getInstance(){
14         MySingleton* sin= instance.load(std::memory_order_acquire);
15         if ( !sin ){
16             std::lock_guard<std::mutex> myLock(myMutex);
17             sin= instance.load(std::memory_order_relaxed);
18             if( !sin ){
19                 sin= new MySingleton();
20                 instance.store(sin,std::memory_order_release);
21             }
22         }
23         // volatile int dummy{};
24         return sin;
25     }
26 private:
27     MySingleton()= default;
28     ~MySingleton()= default;
29     MySingleton(const MySingleton&)= delete;
30     MySingleton& operator=(const MySingleton&)= delete;
31
32     static std::atomic<MySingleton*> instance;
33     static std::mutex myMutex;
34 };
35
36
37 std::atomic<MySingleton*> MySingleton::instance;
38 std::mutex MySingleton::myMutex;
39
40 std::chrono::duration<double> getTime(){
41     auto begin= std::chrono::system_clock::now();
42     for ( size_t i= 0; i <= tenMill; ++i){
43         MySingleton::getInstance();
44     }
45     return std::chrono::system_clock::now() - begin;
46 };
47
48 };
49
50
51 int main(){
52
53     auto fut1= std::async(std::launch::async,getTime);
54     auto fut2= std::async(std::launch::async,getTime);
55     auto fut3= std::async(std::launch::async,getTime);
56     auto fut4= std::async(std::launch::async,getTime);
57
58     auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
59
60     std::cout << total.count() << std::endl;
61
62 }

```

<http://www.facebook.com/rainer.gruber><https://plus.google.com/u/0/109576694736160795401><https://www.linkedin.com/in/rainergruber>https://twitter.com/rainer_gruberhttps://www.xing.com/profile/Rainer_GruberHunting(<http://www.hunterspeak.com/>)

The acquire-release semantic has a similar performance as the sequential consistency. That's not surprising, because on x86 both memory models are very similar. We would get totally different numbers on an ARMv7 or PowerPC architecture. You can read the details on Jeff Preshings blog [Preshing on Programming](http://preshing.com/) (<http://preshing.com/>).

Without optimization



Maximum optimization

If I forget an import variant of the thread safe singleton pattern, please let me know and send me the code. I will measure it and add the numbers to the comparison.

All numbers at one glance

Don't take the numbers too seriously. I executed each program only once and the executable is optimized for four cores on my two core windows PC. But the numbers give a clear indication. The Meyers Singleton is the easiest to get and the fastest one. In particular the lock based implementation is by far the slowest one. The numbers are independent of the used platform.

But the numbers show more. Optimization counts. This statements holds not totally true for the `std::lock_guard` based implementation of the singleton pattern.

Operating system (Compiler)	Optimization	Single threaded	Meyers Singleton	std::call_once	std::lock_guard	Sequential consistency	Acquire-release semantic
Linux (GCC)	no	0.09	0.10	1.92	18.15	0.56	0.50
Linux (GCC)	yes	0.03	0.04	0.22	12.47	0.09	0.07
Window (cl.exe)	no	0.09	0.16	4.66	23.40	1.33	1.37
Windows (cl.exe)	yes	0.02	0.03	1.74	15.48	0.07	0.07

What's next?

I'm not so sure. This post is a the translation of german post I wrote half a year ago. My German post get's a lot of reaction. I'm not sure, what will happen this time. A few days letter I'm sure. The next post (/index.php/single-threaded-sum-of-the-elements-of-a-vector) will be about the addition of the elements of a vector. First it takes in one thread.



Go to (<https://leanpub.com/cplusplus>)



(<http://www.facebook.com/rainer.grimm>)



(<https://plus.google.com/u/0/109576694736160795401>)



(<https://www.linkedin.com/in/rainer-grimm>)



(https://twitter.com/rainer_g)



(https://www.xing.com/profile/Rainer_Grimm)

Hunting(<http://www.hunterspeak.com/>)

(<https://leanpub.com/cplusplus>) "What every professional C++ programmer should know about the C++ standard library". (<https://leanpub.com/cplusplus>)

Get your e-book. Support my blog.

Tweet



Share

16

Tags: [atomics](/index.php/tag/atomics), [sequential consistency](/index.php/tag/sequential-consistency), [lock](/index.php/tag/lock), [mutex](/index.php/tag/mutex), [static](/index.php/tag/static), [acquire-release semantic](/index.php/tag/acquire-release-semantic), [singleton](/index.php/tag/singleton)

Comments (/index.php/component/jcomments/feed/com_jaggyblog/197)

#1 (</index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-372>) **Anton** 2016-08-30 15:04 0
why you not measured this method
<http://ideone.com/8wePDz>

i know what writing and reading of volatile, can be not atomic, but for aligned register-size data it looks like atomic.

Quote

#2 (</index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-374>) **Rainer Grimm** 2016-08-30 18:37 0

I hope I get your point.
You proposing using volatile as a kind of atomi?. If so volatile has no multithreading semantic. Maybe the compiler guarantees it. With volatile we are in the area of the broken double-checked locking pattern. To measure undefined behaviours makes no sense from my perspective.

Quote

#3 (</index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-376>) **Anton** 2016-08-31 11:35 0

Thank for your answer. I just checked asm listing of your Acquire-release Semantic, and it doesn't contain lock prefixes or fences (like in version with volatile).

Quote

#4 (</index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-648>) **nikitablack** 2016-10-11 07:16 0

Hi. Can you please explain Acquire-release Semantic in a more detail?

Why do you need a mutex on line 16? I know the problem with double check locking but I thought with atomics it's not the case so no need for the mutex.

Is it possible that relaxed read (line 17) move above mutex lock (line 16)? In that case this approach is also broken. Or maybe a mutex lock guarantees a happens-before relationship?

Quote

#5 (</index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-651>) **Rainer Grimm** 2016-10-11 16:20 0

Quoting nikitablack:

Hi. Can you please explain Acquire-release Semantic in a more detail?

Why do you need a mutex on line 16? I know the problem with double check locking but I thought with atomics it's not the case so no need for the mutex.

Is it possible that relaxed read (line 17) move above mutex lock (line 16)? In that case this approach is also broken. Or maybe a mutex lock guarantees a happens-before relationship?

You have no guarantee that the line 19 (`sin= new MySingleton()`) is an atomic operation. Therefore you can have a half initialized singleton. The lock establishes a kind of a barrier. Nothing from inside can cross the border. => See the details in the post: [Acquire-release semantic](#)

Quote

#6 (</index.php/component/jaggyblog/thread-safe-initialization-of-a-singleton#comment-4723>) **centos_linux** 2017-04-25 13:20 0

Your style is very unique in comparison to other people I've read stuff from.
I appreciate you for posting when you have the opportunity, Guess I will just book mark this blog.

Quote

Refresh comments list

RSS feed for comments to this post (/index.php/component/jcomments/feed/com_jaggyblog/197)

Add comment

Name (required)

E-mail (required, but will not display)

Website

☐

Notify me of follow-up comments

Send



(<http://www.facebook.com/rainer.grimm>)



(<https://plus.google.com/u/0/109576694736160795401>)



(<https://www.linkedin.com/in/rainer-grimm>)



(https://twitter.com/rainer_g)



(https://www.xing.com/profile/Rainer_Grimm)

Hunting (<http://www.hunterspeak.com/>)

JComments (<http://www.joomlatune.com>)

(<http://www.modernescpp.de/>)

Open C++ Seminars

Embedded Programmierung mit modernem C++: 16.01 - 18.01
(<http://www.modernescpp.de/index.php/c/2-c/16-embedded-programmierung-mit-modernem-c>)

C++11 und C++14: 13.03 - 15.03 (<http://www.modernescpp.de/index.php/c/2-c/3-c-11-und-c-14>)

Multithreading mit modernem C++: 08.05 - 09.05
(<http://www.modernescpp.de/index.php/c/2-c/13-multithreading-mit-c>)

Contact

rainer@grimm-jaud.de (</index.php/rainer-grimm>)

[Impressum \(/index.php/impressum\)](/index.php/impressum)

Subscribe to the newsletter (+ pdf bundle)

E-mail

Subscribe



Become a patron

(<https://www.patreon.com>)

/rainer_grimm

>> Start here <<

(</index.php/der-einstieg-in-modernes-c>)

Categories

- Embedded (</index.php/category/embedded>)
- Functional (</index.php/category/functional>)
- Modern C++ (</index.php/category/modern-c>)
- Multithreading (</index.php/category/multithreading>)
- Multithreading - Application (</index.php/category/multithreading-application>)
- Multithreading - C++17 and C++20 (</index.php/category/multithreading-c-17-and-c-20>)
- Multithreading - Memory Model (</index.php/category/multithreading-memory-model>)
- News (</index.php/category/news>)
- Overview (</index.php/category/ueberblick>)
- C++ 17 (</index.php/category/c-17>)
- Pdf bundles (</index.php/category/pdf-bundle>)

All tags

[acquire-release semantic \(/index.php/tag/acquire-release-semantic\)](/index.php/tag/acquire-release-semantic)

[async \(/index.php/tag/async\)](/index.php/tag/async) [atomics \(/index.php/tag/atomics\)](/index.php/tag/atomics)

[atomic_thread_fence \(/index.php/tag/atomic_thread_fence\)](/index.php/tag/atomic_thread_fence) [auto \(/index.php/tag/auto\)](/index.php/tag/auto)

[C++17 \(/index.php/tag/c-17\)](/index.php/tag/c-17) [C++20 \(/index.php/tag/c-20\)](/index.php/tag/c-20)

[concepts \(/index.php/tag/concepts\)](/index.php/tag/concepts) [condition variable \(/index.php/tag/condition-variable\)](/index.php/tag/condition-variable)

[constexpr \(/index.php/tag/constexpr\)](/index.php/tag/constexpr) [CppMem \(/index.php/tag/cppmem\)](/index.php/tag/cppmem)

[enum \(/index.php/tag/enum\)](/index.php/tag/enum) [final \(/index.php/tag/final\)](/index.php/tag/final) [hash tables](/index.php/tag/hash-tables)

C++ Core Guidelines: Function Definitions (/index.php/component/jaggyblog/c-core-guidelines-functions)

Jeremiah

Very nice blog post. I certainly appreciate this website. Keep writing!
Read more... (/index.php/component/jaggyblog/c-core-guidelines-functions#comment-8526)

Strongly-Typed Enums (/index.php/component/jaggyblog/strongly-typed-enums)

Horacio

Excellent post. I was checking constantly this blog and I am impressed! Extremely useful info ...
Read more... (/index.php/component/jaggyblog/strongly-typed-enums#comment-8509)



(http://www.facebook.com/rainer.gruber)



(https://plus.google.com/u/0/109576694736160795401)



(https://www.linkedin.com/in/rainergruber)



(https://twitter.com/rainer_gruber)



(https://www.xing.com/profile/Rainer_Gruber)

Hunting (http://www.hunterspeak.com/)

You are here: [Home \(/index.php\)](#) / [Thread-Safe Initialization of a Singleton](#)

Copyright © 2017 ModernesCpp.com. All Rights Reserved. Designed by Joomla!Art.com (<http://www.joomlaart.com/>).

Joomla! (<https://www.joomla.org>) is Free Software released under the GNU General Public License. (<https://www.gnu.org/licenses/gpl-2.0.html>)

Bootstrap (<http://twitter.github.io/bootstrap/>) is a front-end framework of Twitter, Inc. Code licensed under MIT License. (<https://github.com/twbs/bootstrap/blob/master/LICENSE>)

Font Awesome (<http://fontawesome.github.io/Font-Awesome/>) font licensed under SIL OFL 1.1 (<http://scripts.sil.org/OFL>).