C++ Thread, shared data

Asked 16 years, 3 months ago Modified 12 years, 1 month ago Viewed 35k times

30

I have an application where 2 threads are running... Is there any certanty that when I change a global variable from one thread, the other will notice this change? I don't have any syncronization or Mutual exclusion system in place... but should this code work all the time (imagine a global **bool** named **dataUpdated**):

Thread 1:

```
while(1) {
    if (dataUpdated)
        updateScreen();
    doSomethingElse();
}
```

Thread 2:

```
while(1) {
    if (doSomething())
        dataUpdated = TRUE;
}
```

Does a compiler like gcc optimize this code in a way that it doesn't check for the global value, only considering it value at compile time (because it nevers get changed at the same thred)?

PS: Being this for a game-like application, it really doen't matter if there will be a read while the value is being written... all that matters is that the change gets noticed by the other thread.

c++ multithreading synchronization mutex

Share
Improve this question
Follow

edited Sep 23, 2008 at 0:29

Tall Jeff
9,984 ● 7 ● 46 ● 61

asked Sep 22, 2008 at 23:22

fabiopedrosa
2,581 ● 8 ● 29 ● 42

10 Answers

Sorted by: Highest score (default)

Yes. No. Maybe.

25

First, as others have mentioned you need to make dataUpdated volatile; otherwise the compiler may be free to lift reading it out of the loop (depending on whether or not it can see that doSomethingElse doesn't touch it).

Secondly, depending on your processor and ordering needs, you may need memory barriers. volatile is enough to guarentee that the other processor will see the change eventually, but not enough to guarentee that the changes will be seen in the order they were performed. Your example only has one flag, so it doesn't really show this phenomena. If you need and use memory barriers, you should no longer need volatile

Volatile considered harmful and Linux Kernel Memory Barriers are good background on the underlying issues; I don't really know of anything similar written specifically for threading. Thankfully threads don't raise these concerns nearly as often as hardware peripherals do, though the sort of case you describe (a flag indicating completion, with other data presumed to be valid if the flag is set) is exactly the sort of thing where ordering matterns...

Share Improve this answer Follow

answered Sep 22, 2008 at 23:56

puetzk
10.8k ● 3 ● 30 ● 33

This is additional good information to consider in addition to the points I made in my answer here. – Tall Jeff Sep 23, 2008 at 0:27

"If you need and use memory barriers, you should no longer need volatile" Um... wouldn't you still need volatile to prevent the compiler from caching the variable in a register for the loop that reads the value, for example? From the same answer: "you need to make dataUpdated volatile; otherwise the compiler may be free to lift reading it out of the loop" I don't see how a memory barrier helps with this. – James Johnston Oct 12, 2011 at 18:27

Memory Barriers have to be implemented with the compiler's help anyway. If you ask the compiler to make sure the CPU can't executes the load/store in exactly this order, then the compiler has to start by honoring the order itself. So the barrier implies everything volatile does, and more. – puetzk Aug 26, 2013 at 13:22

7

Here is an example that uses boost condition variables:

```
bool _updated=false;
boost::mutex _access;
boost::condition _condition;

bool updated()
{
  return _updated;
}

void thread1()
{
```

```
  boost::mutex::scoped_lock lock(_access);
  while (true)
  {
    boost::xtime xt;
    boost::xtime_get(&xt, boost::TIME_UTC);
    // note that the second parameter to timed_wait is a predicate function
that is called - not the address of a variable to check
    if (_condition.timed_wait(lock, &updated, xt))
      updateScreen();
    doSomethingElse();
  }
}

void thread2()
{
  while(true)
  {
    if (doSomething())
      _updated=true;
  }
}
```

Share

Improve this answer

Follow

edited Oct 5, 2008 at 21:26

answered Sep 22, 2008 at 23:46

1800 INFORMATION

135k ● 30 ● 163 ● 242

7

Use a lock. Always always use a lock to access shared data. Marking the variable as volatile will prevent the compiler from optimizing away the memory read, but will not prevent other problems such as memory re-ordering. Without a lock there is no guarantee that the memory writes in doSomething() will be visible in the updateScreen() function.

The only other safe way is to use a memory fence, either explicitly or an implicitly using an Interlocked* function for example.

Share

Improve this answer

Follow

edited Nov 4, 2012 at 13:30

ks1322

35.6k ● 14 ● 116 ● 171

answered Sep 22, 2008 at 23:51

Niall

5,121 ● 1 ● 22 ● 12

6

Use the ***volatile*** keyword to hint to the compiler that the value can change at any time.

```
volatile int myInteger;
```

The above will guarantee that any access to the variable will be to and from memory without any specific optimizations and as a result all threads running on the same

processor will "see" changes to the variable with the same semantics as the code reads.

Chris Jester-Young pointed out that coherency concerns to such a variable value change may arise in a multi-processor systems. This is a consideration and it depends on the platform.

Actually, there are really two considerations to think about relative to platform. They are coherency and atomicity of the memory transactions.

Atomicity is actually a consideration for both single and multi-processor platforms. The issue arises because the variable is likely multi-byte in nature and the question is if one thread could see a partial update to the value or not. ie: Some bytes changed, context switch, invalid value read by interrupting thread. For a single variable that is at the natural machine word size or smaller and naturally aligned should not be a concern. Specifically, an ***int*** type should always be OK in this regard as long as it is aligned - which should be the default case for the compiler.

Relative to coherency, this is a potential concern in a multi-processor system. The question is if the system implements full cache coherency or not between processors. If implemented, this is typically done with the MESI protocol in hardware. The question didn't state platforms, but both Intel x86 platforms and PowerPC platforms are cache coherent across processors for normally mapped program data regions. Therefore this type of issue should not be a concern for ordinary data memory accesses between threads even if there are multiple processors.

The final issue relative to atomicity that arises is specific to read-modify-write atomicity. That is, how do you guarantee that if a value is read updated in value and the written, that this happen atomically, even across processors if more than one. So, for this to work without specific synchronization objects, would require that all potential threads accessing the variable are readers ONLY but expect for only one thread can ever be a writer at one time. If this is not the case, then you do need a sync object available to be able to ensure atomic actions on read-modify-write actions to the variable.

Share
Improve this answer
Follow

edited Sep 23, 2008 at 15:38

answered Sep 23, 2008 at 0:08

Tall Jeff
9,984 ● 7 ● 46 ● 61

3

Your solution will use 100% CPU, among other problems. Google for "condition variable".

Share Improve this answer Follow

answered Sep 22, 2008 at 23:25

Jeff

that while is only for demonstration purposes – fabiopedrosa Sep 22, 2008 at 23:27

3

Chris Jester-Young pointed out that:

> This only work under Java 1.5+'s memory model. The C++ standard does not address threading, and volatile does not guarantee memory coherency between processors. You do need a memory barrier for this

being so, the only true answer is implementing a synchronization system, right?

Share Improve this answer Follow

answered Sep 22, 2008 at 23:36


fabiopedrosa
2,581 ● 8 ● 29 ● 42

Some of the comments hinted at using interlocked/atomic variables. That's fine: you use compare-and-set functions to work with those, and they can be tricky to use. Otherwise, for maximum ease of use, use a lock. – C. K. Young Sep 22, 2008 at 23:37

somehow, no one agreed because they buried that response and comments... Thanks anyway :D – fabiopedrosa Sep 22, 2008 at 23:38

I voted that response down because it was poorly worded and confusing. I suspect others did the same. – 1800 INFORMATION Sep 22, 2008 at 23:40

I can only chose one answer. Because I now know for a fact that my compiler respects the volatile keyword, I will chose that one. But I keep in mind that implementing locks would be an acceptable solution. – fabiopedrosa Sep 22, 2008 at 23:43

Here's a Microsoft article on Memory Barriers and volatile: msdn.microsoft.com/en-us/library/f20w0x5e(VS.80).aspx It looks like you should be fine if you're using the Microsoft compiler. – Adam Pierce Sep 22, 2008 at 23:56

2

Use the **volatile** keyword to hint to the compiler that the value can change at any time.

```
volatile int myInteger;
```

Share Improve this answer Follow

answered Sep 22, 2008 at 23:24

Adam Pierce
34.3k ● 23 ● 71 ● 89

2 This only work under Java 1.5+'s memory model. The C++ standard does not address threading, and volatile does not guarantee memory coherency between processors. You do need a memory barrier for this. – C. K. Young Sep 22, 2008 at 23:29

The C++ standard itself does currently not define this semantic. But all 'recent' compilers honor volatile as the correct hint. – Christopher Sep 22, 2008 at 23:38

1 No, they do not. volatile ensures that the compiler will perform the load as specified, and so accounts for threading, but it does not guarentee that the underlying memory subsystem will preserve causality in a true multiprocessor system. – puetzk Sep 22, 2008 at 23:56

puetzk: sure, but that's not needed for this case. – wnoise Sep 23, 2008 at 0:07

1 Depends on whether doSomething() is writing data that updateScreen() will read. That seems likely, though he didn't mention it specifically. – puetzk Sep 23, 2008 at 0:16

2

No, it's not certain. If you declare the variable volatile, then the complier is supposed to generate code that always loads the variable from memory on a read.

Share Improve this answer Follow

answered Sep 22, 2008 at 23:25

Lou Franco
89k ● 14 ● 136 ● 198

1

If the scope is right ("extern", global, etc.) then the change will be noticed. The question is when? And in what order?

The problem is that the compiler **can** and frequently **will** re-order your logic to fill all it's concurrent pipelines as a performance optimization.

It doesn't really show in your specific example because there aren't any other instructions around your assignment, but imagine functions declared after your bool assign execute **before** the assignment.

Check-out Pipeline Hazard on wikipedia or search google for "compiler instruction reordering"

Share Improve this answer Follow

answered Sep 22, 2008 at 23:47

kervin

11.9k ● 6 ● 46 ● 60

1

As others have said the `volatile` keyword is your friend. :-)

You'll most likely find that your code would work when you had all of the optimisation options disabled in gcc. In this case (I believe) it treats everything as volatile and as a result the variable is accessed in memory for every operation.

With any sort of optimisation turned on the compiler will attempt to use a local copy held in a register. Depending on your functions this may mean that you only see the change in variable intermittently or, at worst, never.

Using the keyword `volatile` indicates to the compiler that the contents of this variable can change at any time and that it should *not* use a locally cached copy.

With all of that said you may find better results (as alluded to by Jeff) through the use of a semaphore or condition variable.

This is a reasonable introduction to the subject.

Share

Improve this answer

Follow

edited May 23, 2017 at 11:33

Community Bot

1 ● 1

answered Sep 22, 2008 at 23:53

Andrew Edgecombe

40.3k ● 3 ● 38 ● 63