How efficient is locking and unlocked mutex? What is the cost of a mutex?

Asked 14 years, 3 months ago Modified 2 years, 3 months ago

Viewed 110k times

227

In a low level language (C, C++ or whatever): I have the choice in between either having a bunch of mutexes (like what pthread gives me or whatever the native system library provides) or a single one for an object.

How efficient is it to lock a mutex? I.e. how many assembler instructions are there likely and how much time do they take (in the case that the mutex is unlocked)?

How much does a mutex cost? Is it a problem to have really **a lot** of mutexes? Or can I just throw as much mutex variables in my code as I have `int` variables and it doesn't really matter?

(I am not sure how much differences there are between different hardware. If there is, I would also like to know about them. But mostly, I am interested about common hardware.)

The point is, by using many mutex which each cover only a part of the object instead of a single mutex for the whole object, I could safe many blocks. And I am wondering how far I should go about this. I.e. should I try to safe any possible block really as far as possible, no matter how much more complicated and how many more mutexes this means?

[WebKits blog post (2016) about locking](#) is very related to this question, and explains the differences between a spinlock, adaptive lock, futex, etc.

multithreading locking mutex blocking

Share

Improve this question

Follow

edited Sep 14, 2022 at 22:41

Alexis Wilke
20.7k ● 11 ● 104 ● 174

asked Sep 6, 2010 at 14:00

Albert
67.9k ● 67 ● 251 ● 400

---

1 This is going to be implementation and architecture-specific. Some mutexes will cost almost nothing if there is native hardware support, others will cost a lot. It's impossible to answer without more information. – Gian Sep 6, 2010 at 14:01

8 @Gian: Well, of course I imply this subquestion in my question. I would like to know about common hardware but also notable exceptions if there are any. – Albert Sep 6, 2010 at 14:04

I really don't see that implication anywhere. You ask about "assembler instructions" - the answer could be anywhere from 1 instruction to ten thousand instructions depending on what architecture you're talking about. – Gian Sep 6, 2010 at 14:10

33 @Gian: Then please give exactly this answer. Please say what it is actually on x86 and amd64, please give an example for an architecture where it is 1 instruction and give one where it is 10k. Isn't it clear that I want to know that from my question? – Albert Sep 6, 2010 at 21:41

6 Answers

Sorted by: Highest score (default)

169

> I have the choice in between either having a bunch of mutexes or a single one for an object.

If you have many threads and the access to the object happens often, then multiple locks would increase parallelism. At the cost of maintainability, since more locking means more debugging of the locking.

> How efficient is it to lock a mutex? I.e. how much assembler instructions are there likely and how much time do they take (in the case that the mutex is unlocked)?

The precise assembler instructions are the least overhead of a mutex - the memory/cache coherency guarantees are the main overhead. And less often a particular lock is taken - better.

Mutex is made of two major parts (oversimplifying): (1) a flag indicating whether the mutex is locked or not and (2) wait queue.

Change of the flag is just few instructions and normally done without system call. If mutex is locked, syscall will happen to add the calling thread into wait queue and start the waiting. Unlocking, if the wait queue is empty, is cheap but otherwise needs a syscall to wake up one of the waiting processes. (On some systems cheap/fast syscalls are used to implement the mutexes, they become slow (normal) system calls only in case of contention.)

Locking unlocked mutex is really cheap. Unlocking mutex w/o contention is cheap too.

> How much does a mutex cost? Is it a problem to have really a lot of mutexes? Or can I just throw as much mutex variables in my code as I have int variables and it doesn't really matter?

You can throw as much mutex variables into your code as you wish. You are only limited by the amount of memory you application can allocate.

Summary. User-space locks (and the mutexes in particular) are cheap and not subjected to any system limit. But too many of them spells nightmare for debugging. Simple table:

1. Less locks means more contentions (slow syscalls, CPU stalls) and lesser parallelism
2. Less locks means less problems debugging multi-threading problems.
3. More locks means less contentions and higher parallelism
4. More locks means more chances of running into undebugable deadlocks.

A balanced locking scheme for application should be found and maintained, generally balancing the #2 and the #3.

(*) The problem with less very often locked mutexes is that if you have too much locking in your application, it causes to much of the inter-CPU/core traffic to flush the mutex memory from the data cache of other CPUs to guarantee the cache coherency. The cache flushes are like light-weight interrupts and handled by CPUs transparently - but they do introduce so called stalls (search for "stall").

And the stalls are what makes the locking code to run slowly, often without any apparent indication why application is slow. (Some arch provide the inter-CPU/core traffic stats, some not.)

To avoid the problem, people generally resort to large number of locks to decrease the probability of lock contentions and to avoid the stall. That is the reason why the cheap user space locking, not subjected to the system limits, exists.

Share Improve this answer Follow

answered Sep 6, 2010 at 15:00

Dummy00001
17.4k ● 5 ● 42 ● 63

Thanks, that mostly answers my question. I didn't knew that the kernel (e.g. the Linux kernel) handles mutexes and you control them via syscalls. But as the Linux itself manages the scheduling and context switches, this makes sense. But now I have a rough imagination about what the mutex lock/unlock will do internally. – Albert Sep 6, 2010 at 21:55

5 @Albert: Oh. I forgot the context switches... Context switches are too drain on the performance. If lock acquisition *fails* and thread has to wait, that is too sort of half of the context switch. CS itself is fast, but since CPU might be used by some other process, the caches would be filled with alien data. After thread finally acquires the lock, chances are that to CPU would have to reload pretty much everything from RAM anew. – Dummy00001 Sep 7, 2010 at 9:41

@Dummy00001 Switching to another process means you have to change the memory mappings of the CPU. That isn't so cheap. – curiousguy Dec 8, 2019 at 23:52

2 Many small locks does not make things more complicated, particularly when they are held for a very short time. Whereas having fewer, bigger locks makes things more complicated when you inevitably have to nest them. Therefore I really have to disagree with "More locks means more chances of running into undebugable deadlocks". – VoidStar Nov 2, 2021 at 4:52

1 @ijustlovemath : wait queues are normally in kernel, since are part of scheduler implementation. For Linux, look at the

implementation of "futex" syscall and how it is used in glibc. Wait queues themselves are bit trickier, since are very fundamental building blocks: use LDD3 as intro, and then Linux kernel API for e.g. wait_event() function.
– Dummy00001 Jul 17 at 15:57

45

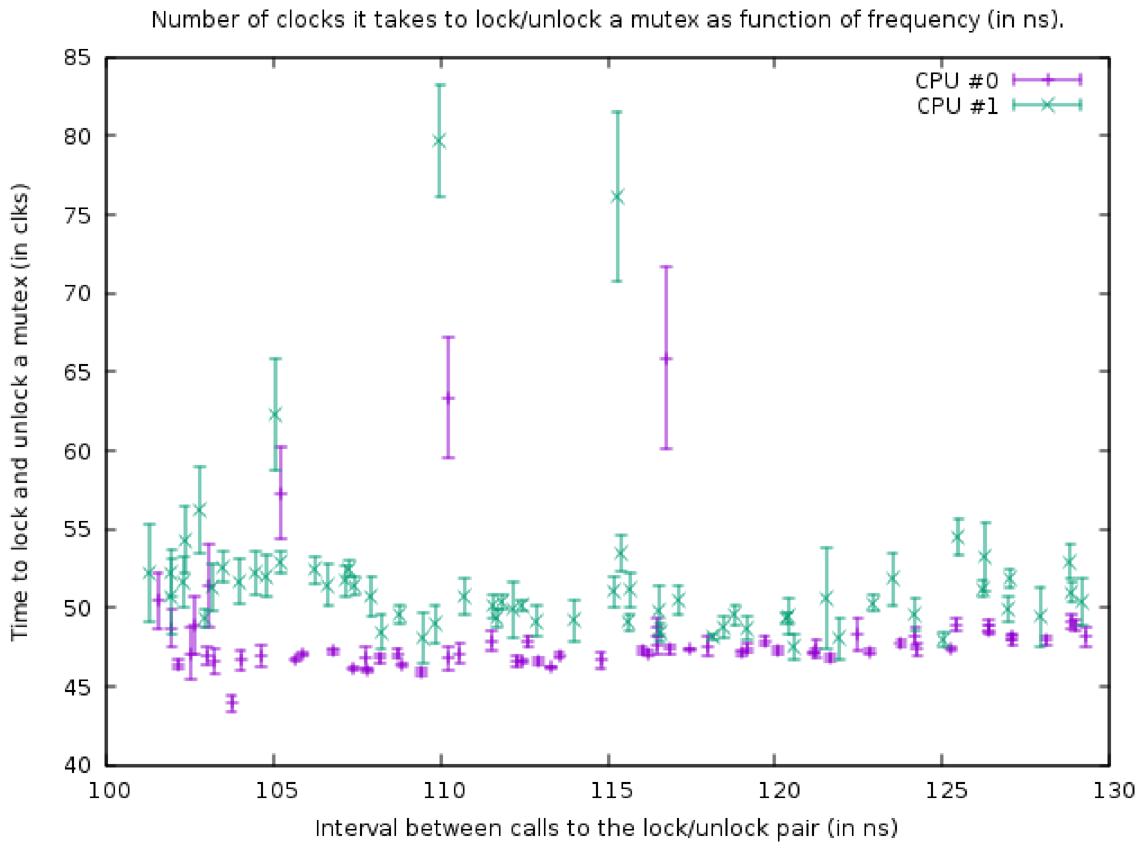
I wanted to know the same thing, so I measured it. On my box (AMD FX(tm)-8150 Eight-Core Processor at 3.612361 GHz), locking and unlocking an unlocked mutex that is in its own cache line and is already cached, takes 47 clocks (13 ns).

Due to synchronization between two cores (I used CPU #0 and #1), I could only call a lock/unlock pair once every 102 ns on two threads, so once every 51 ns, from which one can conclude that it takes roughly 38 ns to recover after a thread does an unlock before the next thread can lock it again.

The program that I used to investigate this can be found here: https://github.com/CarloWood/ai-statefultask-testsuite/blob/b69b112e2e91d35b56a39f41809d3e3de2f9e4b8/src/mutex_test.cxx

Note that it has a few hardcoded values specific for my box (xrange, yrange and rdtsc overhead), so you probably have to experiment with it before it will work for you.

The graph it produces in that state is:



This shows the result of benchmark runs on the following code:

```
uint64_t do_Ndec(int thread, int loop_count)
{
  uint64_t start;
  uint64_t end;
  int __d0;

  asm volatile ("rdtsc\n\tshl $32, %%rdx\n\tor
%%rdx, %0" : "=a" (start) : : "%rdx");
  mutex.lock();
  mutex.unlock();
  asm volatile ("rdtsc\n\tshl $32, %%rdx\n\tor
%%rdx, %0" : "=a" (end) : : "%rdx");
  asm volatile ("\n1:\n\tdecl %%ecx\n\tjnz 1b" :
"=c" (__d0) : "c" (loop_count - thread) : "cc");
  return end - start;
}
```

The two rdtsc calls measure the number of clocks that it takes to lock and unlock `mutex' (with an overhead of 39 clocks for the rdtsc calls on my box). The third asm is a delay loop. The size of the delay loop is 1 count smaller for thread 1 than it is for thread 0, so thread 1 is slightly faster.

The above function is called in a tight loop of size 100,000. Despite that the function is slightly faster for thread 1, both loops synchronize because of the call to the mutex. This is visible in the graph from the fact that the number of clocks measured for the lock/unlock pair is slightly larger for thread 1, to account for the shorter delay in the loop below it.

In the above graph the bottom right point is a measurement with a delay loop_count of 150, and then following the points at the bottom, towards the left, the loop_count is reduced by one each measurement. When it becomes 77 the function is called every 102 ns in both threads. If subsequently loop_count is reduced even further it is no longer possible to synchronize the threads and the mutex starts to be actually locked most of the time, resulting in an increased amount of clocks that it takes to do the lock/unlock. Also the average time of the function call increases because of this; so the plot points now go up and towards the right again.

From this we can conclude that locking and unlocking a mutex every 50 ns is not a problem on my box.

All in all my conclusion is that the answer to question of OP is that adding more mutexes is better as long as that results in less contention.

Try to lock mutexes as short as possible. The only reason to put them -say- outside a loop would be if that loop loops faster than once every 100 ns (or rather, number of threads that want to run that loop at the same time times 50 ns) or when 13 ns times the loop size is more delay than the delay you get by contention.

EDIT: I got a lot more knowledgable on the subject now and start to doubt the conclusion that I presented here. First of all, CPU 0 and 1 turn out to be hyper-threaded; even though AMD claims to have 8 real cores, there is certainly something very fishy because the delays between two other cores is much larger (ie, 0 and 1 form a pair, as do 2 and 3, 4 and 5, and 6 and 7). Secondly, the std::mutex is implemented in way that it spin locks for a bit before actually doing system calls when it fails to immediately obtain the lock on a mutex (which no doubt will be extremely slow). So what I have measured here is the absolute most ideal situtation and in practise locking and unlocking might take drastically more time per lock/unlock.

Bottom line, a mutex is implemented with atomics. To synchronize atomics between cores an internal bus must be locked which freezes the corresponding cache line for several hundred clock cycles. In the case that a lock can not be obtained, a system call has to be performed to put the thread to sleep; that is obviously extremely slow (system calls are in the order of 10 mircoseconds). Normally that is not really a problem because that thread has to sleep anyway-- but it could be a problem with high contention where a thread can't obtain the lock for the time that it normally spins and so does the system call, but CAN take the lock shortly there after. For example, if several threads lock and unlock a mutex in a tight loop and each keeps the lock for 1 microsecond or so, then they might be slowed down enormously by the fact that they are constantly put to sleep and woken up again. Also, once a thread sleeps and another thread has to wake it up, that thread has to do a system call and is delayed ~10 microseconds; this delay thus happens while unlocking a mutex when another thread is waiting for that mutex in the kernel (after spinning took too long).

Share Improve this answer Follow

edited Dec 3, 2019 at 21:48

answered Apr 7, 2018 at 23:32

Carlo Wood
6,663 ● 2 ● 43 ● 54

---

1 Bulldozer line of CPUs have 2 integer cores, and one FPU per module. You get nearly, but not quite double integer performance per module, though floating point isn't faster. IIRC using both parts of a module is about 8% slower than using 2 separate cores. There is debate whether it is SMT or not. SMT normally nets around 5-20% boost, while the module approach around 90-95% for non floating point

operation (later versions were better). – Xeridea Nov 17, 2021 at 23:10

16

This depends on what you actually call "mutex", OS mode and etc.

At **minimum** it's a cost of an interlocked memory operation. It's a relatively heavy operation (compared to other primitive assembler commands).

However, that can be very much higher. If what you call "mutex" a kernel object (i.e. - object managed by the OS) and run in the user mode - every operation on it leads to a kernel mode transaction, which is **very** heavy.

For example on Intel Core Duo processor, Windows XP. Interlocked operation: takes about 40 CPU cycles. Kernel mode call (i.e. system call) - about 2000 CPU cycles.

If this is the case - you may consider using critical sections. It's a hybrid of a kernel mutex and interlocked memory access.

Share Improve this answer Follow

edited Feb 16, 2014 at 9:39
user4035
23.7k ● 11 ● 65 ● 99

answered Sep 6, 2010 at 14:11
valdo
12.9k ● 2 ● 40 ● 69

8 Windows critical sections are much closer to mutexes. They have regular mutex semantics, but they are process-local. The last part makes them a lot faster, as they can be handled entirely within your process (and thus user-mode code). – MSalters Sep 6, 2010 at 15:01

2 The number would be more useful if amount of CPU cycles of common operations (e.g. arithmetic/if-else/cache-miss/indirection) are also provided for comparison. .... It would be even great if there are some reference of the number. In the internet, it is very hard to find such information. – javaLover May 14, 2017 at 2:06

@javaLover Operations don't run on cycles; they run on arithmetic units for a number of cycles. It's very different. The cost of a any instruction in time isn't a defined quantity, only the cost on resources use. These resources are shared. The impact of memory instructions depend a lot of caching, etc. – curiousguy Dec 3, 2019 at 1:37

1 @curiousguy Agree. I was not clear. I would like answer such as `std::mutex` averagely use duration (in second) 10 times more than `int++`. However, I know it is hard to answer because it vastly depends on a lot of thing. – javaLover Mar 11, 2020 at 13:00

14

I'm completely new to pthreads and mutex, but I can confirm from experimentation that the cost of locking/unlocking a mutex is almost zilch when there is no contention, but when there is contention, the cost of blocking is extremely high. I ran a simple code with a thread pool in which the task was just to compute a sum in a global variable protected by a mutex lock:

```
y = exp(-j*0.0001);
pthread_mutex_lock(&lock);
x += y ;
pthread_mutex_unlock(&lock);
```

With one thread, the program sums 10,000,000 values virtually instantaneously (less than one second); with two threads (on a MacBook with 4 cores), the same program takes 39 seconds.

Share Improve this answer Follow

answered Nov 18, 2018 at 4:50

Grant Petty
1,251 ● 1 ● 19 ● 28

1 I don't think your example demonstrate the cost of locking/unlocking of mutex. I suspect the cause of your result is because your multithreads are accessing a global object `x` and that is very cache unfriendly. It will be nice if you can share the complete code. – Lion Lai Feb 21 at 7:40

9

The cost will vary depending on the implementation but you should keep in mind two things:

- the cost will be most likely be minimal since it's both a fairly primitive operation and it will be optimised as much as possible due to its use pattern (used a *lot*).
- it doesn't matter how expensive it is since you need to use it if you want safe multi-threaded operation. If you need it, then you need it.

On single processor systems, you can generally just disable interrupts long enough to atomically change data. Multi-processor systems can use a test-and-set strategy.

In both those cases, the instructions are relatively efficient.

As to whether you should provide a single mutex for a massive data structure, or have many mutexes, one for each section of it, that's a balancing act.

By having a single mutex, you have a higher risk of contention between multiple threads. You can reduce this risk by having a mutex per section but you don't want to get into a situation where a thread has to lock 180 mutexes to do its job :-)

Share Improve this answer Follow

edited Sep 6, 2010 at 14:38

answered Sep 6, 2010 at 14:05

paxdiablo
880k ● 241 ● 1.6k ● 2k

---

2 Yea, but *how* efficient? Is it a single machine instruction? Or about 10? Or about 100? 1000? More? All of this is still efficient, however can make a difference in extreme situations. – Albert Sep 6, 2010 at 14:10

1 Well, that depends *entirely* on the implementation. You can turn off interrupts, test/set an integer and reactivate interrupts in a loop in about six machine instructions. Test-and-set can be done in about as many since the processors tend to

provide that as a single instruction. – paxdiablo Sep 6, 2010 at 14:14

A bus-locked test-and-set is a single (rather long) instruction on x86. The rest of the machinery to use it is pretty quick ("did the test succeed?" is a question that CPUs are good at doing fast) but it's the bus-locked instruction's length that really matters as it is the part that blocks things. Solutions with interrupts are much slower, because manipulating them is typically restricted to the OS kernel to stop trivial DoS attacks. – Donal Fellows Sep 6, 2010 at 14:19

BTW, don't use drop/reacquire as a means for having a thread yield to others; that's a strategy that sucks on a multicore system. (It's one of the relatively-few things that CPython gets wrong.) – Donal Fellows Sep 6, 2010 at 14:23

1 @Donal: I didn't meant that I want to use it. I just want to know what you mean by that so I can get sure that I am not using it and that I can understand why it is a bad idea to use it. I was basically asking for references about that which give some background/details about it. – Albert Sep 6, 2010 at 23:55

2

I just measured it on my Windows 10 system.

This is testing Single Threaded code with no contention at all.

Compiler: Visual Studio 2019, x64 release, with loop overhead subtracted from measurements.

Using `std::mutex` takes about 74 machine cycles, while using a native Win32 `CRITICAL_SECTION` takes about 53 machine cycles.

So unless 100 machine cycles is a significant amount of time compared to the code itself, the mutexes aren't going to be the source of a performance problem.

Share Improve this answer Follow

edited Mar 27, 2022 at 17:31

answered Mar 27, 2022 at 16:53

Dwedit
744 ● 5 ● 14