What is the difference between mutex and critical section?

Please explain from Linux, Windows perspectives?

I am programming in C#, would these two terms make a difference. Please post as much as you can, with examples and such....

Thanks

windows linux multithreading programming-languages

edited Jul 6 '12 at 9:52
Inge Henriksen
3,280 1 20 46

asked Apr 29 '09 at 0:23
ultraman

6 Why on earth is this tagged `language-agnostic` *and* `c#` ? – Puppy Dec 14 '11 at 17:18

3 Good read for the person that commented and possibly the 5 people that upvoted if they had enough rep: blogs.msdn.com/b/oldnewthing/archive/2012/10/17/10360184.aspx – Brian R. Bondy Dec 14 '12 at 21:15

Awesome link @BrianR.Bondy thanks. – Wodzu Dec 28 '13 at 6:50

9 Answers

For Windows, critical sections are lighter-weight than mutexes.

Mutexes can be shared between processes, but always result in a system call to the kernel which has some overhead.

Critical sections can only be used within one process, but have the advantage that they only switch to kernel mode in the case of contention - Uncontended acquires, which should be the common case, are incredibly fast. In the case of contention, they enter the kernel to wait on some synchronization primitive (like an event or semaphore).

I wrote a quick sample app that compares the time between the two of them. On my system for 1,000,000 uncontended acquires and releases, a mutex takes over one second. A critical section takes ~50 ms for 1,000,000 acquires.

Here's the test code, I ran this and got similar results if mutex is first or second, so we aren't seeing any other effects.

```
HANDLE mutex = CreateMutex(NULL, FALSE, NULL);
CRITICAL_SECTION critSec;
InitializeCriticalSection(&critSec);

LARGE_INTEGER freq;
QueryPerformanceFrequency(&freq);
LARGE_INTEGER start, end;

// Force code into memory, so we don't see any effects of paging.
EnterCriticalSection(&critSec);
LeaveCriticalSection(&critSec);
QueryPerformanceCounter(&start);
for (int i = 0; i < 1000000; i++)
{
    EnterCriticalSection(&critSec);
    LeaveCriticalSection(&critSec);
}

QueryPerformanceCounter(&end);

int totalTimeCS = (int)((end.QuadPart - start.QuadPart) * 1000 / freq.QuadPart);
```

```
// Force code into memory, so we don't see any effects of paging.
WaitForSingleObject(mutex, INFINITE);
ReleaseMutex(mutex);

QueryPerformanceCounter(&start);
for (int i = 0; i < 1000000; i++)
{
    WaitForSingleObject(mutex, INFINITE);
    ReleaseMutex(mutex);
}

QueryPerformanceCounter(&end);

int totalTime = (int)((end.QuadPart - start.QuadPart) * 1000 / freq.QuadPart);

printf("Mutex: %d CritSec: %d\n", totalTime, totalTimeCS);
```

edited Apr 29 '09 at 1:21
Zifre
14.6k 3 59 91

answered Apr 29 '09 at 0:38
Michael
35.4k 4 66 106

beats me - maybe you should post your code. I voted you up one if it makes you feel better – 1800 INFORMATION Apr 29 '09 at 1:04

1 Well done. Upvoted. – ApplePieIsGood Apr 29 '09 at 3:18

1 Not sure if this relates or not (since I haven't compiled and tried your code), but I've found that calling WaitForSingleObject with INFINITE results in poor performance. Passing it a timeout value of 1 then looping while checking it's return has made a huge difference in the performance of some of my code. This is mostly in the context of waiting for an external process handle, however... Not a mutex. YMMV. I'd be interested in seeing how mutex performs with that modification. The resulting time difference from this test seems bigger than should be expected. – Troy Howard Jul 23 '09 at 5:37

3 @TroyHoward aren't you basically just spin locking at that point? – dss539 Feb 21 '13 at 14:54

The reasons for this distinction are probaly mainly historical. It is not hard to implement locking that is as fast as CriticalSection in the uncontended case (few atomic instructions, no syscalls), yet works across processes (with a piece of shared memory). See e.g. Linux futexes. – regnarg Jan 20 at 9:55

From a theoretical perspective, a critical section is a piece of code that must not be run by multiple threads at once because the code accesses shared resources.

A mutex is an algorithm (and sometimes the name of a data structure) that is used to protect critical sections.

Semaphores and Monitors are common implementations of a mutex.

In practice there are many mutex implementation availiable in windows. They mainly differ as consequence of their implementation by their level of locking, their scopes, their costs, and their performance under different levels of contention. See CLR Inside Out - Using concurrency for scalability for an chart of the costs of different mutex implementations.

Availiable synchronization primitives.

- Monitor
- Mutex
- Semaphore
- ReaderWriterLock
- ReaderWriterLockSlim
- Interlocked

The `lock(object)` statement is implemented using a `Monitor` - see MSDN for reference.

In the last years much research is done on non-blocking synchronization. The goal is to implement algorithms in a lock-free or wait-free way. In such algorithms a process helps other processes to finish their work so that the process can finally finish its work. In consequence a process can finish its work even when other processes, that tried to perform some work, hang. Usinig locks, they would not release their locks and prevent other processes from

continuing.

edited Jan 15 '13 at 17:41

answered Apr 29 '09 at 1:14

Daniel Brückner
37.9k 4 49 110

Seeing the accepted answer, I was thinking maybe I remembered the concept of critical sections wrong, till I saw that **Theoretical Perspective** you wrote. :) – Anirudh Ramanathan Oct 11 '12 at 5:17

Practical lock free programming is like Shangri La, except it exists. Keir Fraser's paper (PDF) explores this rather interestingly (going back to 2004). And we're still struggling with it in 2012. We suck. – Tim Post ♦ Oct 11 '12 at 15:07

Critical Section and Mutex are not Operating system specific, their concepts of multithreading/multiprocessing.

Critical Section Is a piece of code that must only run by it self at any given time (for example, there are 5 threads running simultaneously and a function called "critical_section_function" which updates a array... you don't want all 5 threads updating the array at once. So when the program is running critical_section_function(), none of the other threads must run their critical_section_function.

mutex* Mutex is a way of implementing the critical section code (think of it like a token... the thread must have possession of it to run the critical_section_code)

answered Apr 29 '09 at 0:31

The Unknown
4,243 17 47 81

1 Also, mutexes can be shared across processes. – configurator Apr 29 '09 at 1:07

The 'fast' Windows equal of critical selection in Linux would be a futex, which stands for fast user space mutex. The difference between a futex and a mutex is that with a futex, the kernel only becomes involved when arbitration is required, so you save the overhead of talking to the kernel each time the atomic counter is modified. A futex can also be shared amongst processes, using the means you would employ to share a mutex.

Unfortunately, futexes can be very tricky to implement (PDF).

Beyond that, its pretty much the same across both platforms. You're making atomic, token driven updates to a shared structure in a manner that (hopefully) does not cause starvation. What remains is simply the method of accomplishing that.

edited Apr 29 '09 at 14:57

answered Apr 29 '09 at 1:38

Tim Post ♦
22.4k 9 71 129

A mutex is an object that a thread can acquire, preventing other threads from acquiring it. It is advisory, not mandatory; a thread can use the resource the mutex represents without acquiring it.

A critical section is a length of code that is guaranteed by the operating system to not be interupted. In pseudo-code, it would be like:

```
StartCriticalSection();
    DoSomethingImportant();
    DoSomeOtherImportantThing();
EndCriticalSection();
```

answered Apr 29 '09 at 0:28

Zifre
14.6k 3 59 91

2 Am I incorrect? I would appreciate it if down voters would comment with a reason. – Zifre Apr 29 '09 at 1:18

+1 because the down vote confuses me. :p I'd say this is more correct than the statements that hint to Mutex and Critical Section being two different mechanisms for multithreading. Critical section is any

section of code which ought to be accessed only by one thread. Using mutexes is one way to implement critical sections. – Mikko Rantanen Apr 29 '09 at 1:22

1 I think the poster was talking about user mode synchronization primitives, like a win32 Critical section object, which just provides mutual exclusion. I don't know about Linux, but Windows kernel has critical regions which behave like you describe - non-interruptable. – Michael Apr 29 '09 at 1:22

1 I don't know why you got downvoted. There's the *concept* of a critical section, which you've described correctly, which is different from the Windows kernel object called a CriticalSection, which is a type of mutex. I believe the OP was asking about the latter definition. – Adam Rosenfield Apr 29 '09 at 1:22

At least I got confused by the language agnostic tag. But in any case this is what we get for Microsoft naming their implementation the same as their base class. Bad coding practice! – Mikko Rantanen Apr 29 '09 at 1:27

In addition to the other answers, the following details are specific to critical sections on windows:

- in the absence of contention, acquiring a critical section is as simple as an `InterlockedCompareExchange` operation
- the critical section structure holds room for a mutex. It is initially unallocated
- if there is contention between threads for a critical section, the mutex will be allocated and used. The performance of the critical section will degrade to that of the mutex
- if you anticipate high contention, you can allocate the critical section specifying a spin count.
- if there is contention on a critical section with a spin count, the thread attempting to acquire the critical section will spin (busy-wait) for that many processor cycles. This can result in better performance than sleeping, as the number of cycles to perform a context switch to another thread can be much higher than the number of cycles taken by the owning thread to release the mutex
- if the spin count expires, the mutex will be allocated
- when the owning thread releases the critical section, it is required to check if the mutex is allocated, if it is then it will set the mutex to release a waiting thread

In linux, I think that they have a "spin lock" that serves a similar purpose to the critical section with a spin count.

answered Apr 29 '09 at 1:03

1800 INFORMATION
60.8k 18 106 185

Unfortunately a Window critical section involves doing a CAS operation *in kernel mode*, which is massively more expensive than the actual interlocked operation. Also, Windows critical sections can have spin counts associated with them. – Promit Apr 29 '09 at 1:10

That is definitly not true. CAS can be done with cmpxchg in user mode. – Michael Apr 29 '09 at 1:12

I thought the default spin count was zero if you called InitializeCriticalSection - you have to call InitializeCriticalSectionAndSpinCount if you want a spin count applied. Do you have a reference for that? – 1800 INFORMATION Apr 29 '09 at 1:24

In Windows, a critical section is local to your process. A mutex can be shared/accessed across processes. Basically, critical sections are much cheaper. Can't comment on Linux specifically, but on some systems they're just aliases for the same thing.

answered Apr 29 '09 at 0:25

Promit
2,759 10 26

Just to add my 2 cents, critical Sections are defined as a structure and operations on them are performed in user-mode context.

```
ntdll!_RTL_CRITICAL_SECTION
   +0x000 DebugInfo        : Ptr32 _RTL_CRITICAL_SECTION_DEBUG
   +0x004 LockCount        : Int4B
   +0x008 RecursionCount   : Int4B
   +0x00c OwningThread     : Ptr32 Void
   +0x010 LockSemaphore    : Ptr32 Void
```

```
+0x014 SpinCount        : Uint4B
```

Whereas mutex are kernel objects (ExMutantObjectType) created in the Windows object directory. Mutex operations are mostly implemented in kernel-mode. For instance, when creating a Mutex, you end up calling nt!NtCreateMutant in kernel.

answered Apr 29 '09 at 1:34

Martin
85 1

What happens when a program that initializes and uses a Mutex object, crashes? Does the Mutex object gets automatically deallocated? No, I would say. Right? – Ankur Oct 26 '09 at 12:30

3 Kernel objects have a reference count. Closing a handle to an object decrements the reference count and when it reaches 0 the object is freed. When a process crashes, all of its handles are automatically closed, so a mutex that only that process has a handle to would be automatically deallocated. – Michael Nov 18 '09 at 17:19

Here is the msdn link.. http://msdn.microsoft.com/en-us/library/windows/desktop/ms682530(v=vs.85).aspx

answered Sep 21 '11 at 23:04

Karthick
696 2 12 31