sign up log in tour help stack overflow careers

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required. Take the 2-minute tour ×

Are spinlocks a good choice for a memory allocator?

I've suggested to the maintainers of the D programming language runtime a few times that the memory allocator/garbage collector should use spinlocks instead of regular OS critical sections. This hasn't really caught on. Here are the reasons I think spinlocks would be better:

1. At least in synthetic benchmarks that I did, it's several times faster than OS critical sections when there's contention for the memory allocator/GC lock. Edit: Empirically, using spinlocks didn't even have measurable overhead in a single-core environment, probably because locks need to be held for such a short period of time in a memory allocator.
2. Memory allocations and similar operations usually take a small fraction of a timeslice, and even a small fraction of the time a context switch takes, making it silly to context switch in the case of contention.
3. A garbage collection in the implementation in question stops the world anyhow. There won't be any spinning during a collection.

Are there any good reasons **not** to use spinlocks in a memory allocator/garbage collector implementation?

memory-management concurrency garbage-collection mutex d

edited Dec 15 '09 at 21:53

asked Dec 15 '09 at 21:46

dsimcha
34.5k 27 118 251

6 Answers

On Windows anyway, critical section objects already have the option of doing this (http://msdn.microsoft.com/en-us/library/ms682530.aspx):

> A thread uses the InitializeCriticalSectionAndSpinCount or SetCriticalSectionSpinCount function to specify a spin count for the critical section object. Spinning means that when a thread tries to acquire a critical section that is locked, the thread enters a loop, checks to see if the lock is released, and if the lock is not released, the thread goes to sleep. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0 (zero). On multiprocessor systems, if the critical section is unavailable, the calling thread spins dwSpinCount times before performing a wait operation on a semaphore that is associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

Hopefully other platforms will follow suit if they don't already.

edited Dec 15 '09 at 22:09

answered Dec 15 '09 at 21:56

Michael Burr
194k 24 264 492

Nice. Everything in the GC implementation is done with the OS APIs abstracted away, using synchronized blocks. From reading the disassembly, these call the OS critical section code, but I'm not sure with what parameters. – dsimcha Dec 15 '09 at 22:00

I'd like note that in my experience Win32 CriticalSections are still considerably slower than manual spinlock implementations. My application gained a 20% performance boost when I replaced the CRITICAL_SECTION objects (spinlocks or not) with simple InterlockedExchange loops (which are compiler intrinsics for some compilers). – CyberShadow Dec 16 '09 at 23:04

@CyberShadow - that's interesting. I haven't looked at the actual Win32 implementation of a `CRITICAL_SECTION` in a debugger in a long, long time, but I would have expected that the variant that uses a spinlock would have implemented the spin using a pretty simple `InterlockedExchange()` loop. I might have to look at this in WinDbg tonight... – Michael Burr Dec 16 '09 at 23:35

Anyone know where to find equivalent functionality on other OS's? – dsimcha Dec 17 '09 at 2:12

coverity® Coverity is the leader in development testing, Use static analysis to find issues earlier. LEARN MORE

1. Obviously the worst-case behavior of a spinlock is awful (the OS scheduler just sees 30 CPU-bound threads, so it tries to give them all some CPU time; 29 of them spin like mad while the thread that holds the lock sleeps), so you should avoid them if you can. Plenty of people smarter than me claim spinlocks have *no* userspace use cases because of this.

2. System mutexes should spin a little before putting the thread to sleep (or indeed making any kind of system call), so they can sometimes perform exactly the same as spinlocks even when there's some contention.

3. An allocator can often practically eliminate lock contention by using the lock only to allocate pages to threads. Each thread is then responsible for partitioning its own pages. You end up acquiring the lock only once every N allocations, and you can configure N to whatever you like.

I consider 2 and 3 to be strong arguments that can't be effectively countered by synthetic benchmarks. You would need to show that the performance of a real-world program suffers.

edited Dec 17 '09 at 16:09

answered Dec 15 '09 at 22:05
Jason Orendorff
17.6k 1 28 59

Spinlocks are absolutely worthless on systems with only one CPU/core, or - more generally - in high-contention situtations (when you have many threads waiting on the lock).

answered Dec 15 '09 at 21:49
CyberShadow
15.5k 2 32 65

Can you support your assertion that they're useless? References? Arguments? – JSBձոգչ Dec 15 '09 at 21:54

They're pointless on single threaded programs, but even a single core computer running multiple threads needs locks. Using a spinlock is so cheap that a single thread program would have no slow down (it's a few instructions) – Martin Dec 15 '09 at 22:01

On single CPU/core systems a spinlock should be implemented by the system as a regular lock, since spinlocks are pointless on those systems. For example, in the Windows kernel the kernel mode spinlocks simply raise the IRQL (effectively disabling scheduling) on single core systems. – Michael Burr Dec 15 '09 at 22:13

3 @JS Bangs in a single core system the blocker cannot execute until the blockee stops executing, a spin lock will always spin at least until the scheduler forces a context switch. It would be faster to use some other locking mechanism and immediately release the processor to other threads. – ScottS Dec 16 '09 at 0:52

> Are there any good reasons not to use spinlocks in a memory allocator/garbage collector implementation?

When some threads are compute-bound (CPU-bound) and other threads are memory-allocator-bound, then using spinlocks takes CPU cycles which could otherwise be used either by the compute-bound threads and/or used by threads which belong to other processes.

answered Dec 15 '09 at 22:12
ChrisW
38.3k 5 58 134

Not sure if i agree, as memory allocations CAN take a very long time (the only way it doesnt if you preallocate all the memory and then re issue it) ..You really need to try the same allocations and deallocations with multi gig heap sizes with millions of entries , with many applications hit the allocation critical section (note applications and not threads) and with

disk trashing/swapping from not enough memory. You also can get disk swapping issues durring allocation and doing a spin lock waiting for a disk request is certainly not appropriate.

And as CyberShadow mentioned on single threaded CPU you end up going to a normal lock with an overhead. Now a language may run on many embedded CPUS which are all single threaded.

Also if you can get away with an interlocked exchange , that is best (as it is lockless though still stalls the CPU and raises LOCK# for multi core memory) but most locks use this anyway (but need to do more) . However the structure of a heap normally means an interlocked exchance is not enough and you end up creating a critical section. Note it is possible in a (Generational) Mark Sweep Nursery with a GC to do allocations as a interlocked compare and add of the pointer. I do this for the Cosmos C# OS GC and it makes for stack speed allocations.

edited Jan 10 '10 at 8:39

answered Jan 10 '10 at 8:30

ben
125 2

One of the perf bugs in the Glasgow Haskell Compiler's garbage collector is so annoying that it has a name, the "last core slowdown". This is a direct consequence of their inappropriate use of spinlocks in their GC and is excacerbated on Linux due to its scheduler but, in fact, the effect can be observed whenever other programs are competing for CPU time.

The effect is clear on the second graph here and can be seen affecting more than just the last core here, where the Haskell program sees performance degradation beyond only 5 cores.

answered Jun 11 '10 at 23:54

Jon Harrop
28.7k 7 94 200
