When is it appropriate to use the current processor number as an optimization hint?

devblogs.microsoft.com/oldnewthing/20180719-00

July 19, 2018

Raymond Chen

Some time ago, on the topic of avoiding heap contention, I left an exercise that asked whether it would be appropriate to use the current processor number (as reported by `GetCurrentProcessorNumber`) to select which heap to use.

In this case, the answer is no. While using the current processor would avoid contention at allocation, it would make contention at deallocation even worse.

Suppose thread 1 is running on processor 1 and allocates some memory. It allocates it from heap 1. Later, thread 2 is running on processor 1 and allocates some memory. It also allocates it from heap 1.

Time passes, and the two threads are now running simultaneously, say one on processor 1 and another on processor 2. They both go to free the memory, but since you have to free the memory back to the heap from which it was allocated, the fact that they are running on separate processors right now is immaterial. They both have to free the memory back to heap 1, and that creates contention.

If we had assigned heaps based on thread, then they would have allocated from different heaps and freed back to those different heaps. No contention. (Assuming the threads were assigned different heaps.)

Okay, so what guidelines can we infer from this analysis?

If you are going to use the current processor as a hint to avoid contention, the entire scenario needs to be quick. If the processor changes while your scenario is running, then you will have contention if the new thread also tries to perform that same processor-keyed operation.

In the case of memory allocation, the memory is allocated, and then used for a while, possibly a long time, before finally being freed back to the heap from which it was allocated. Since the scenario is a very long one, using the current processor number as a hint is going to run into a lot of cases of accidental contention.

On the other hand, if you had a linked list of available memory blocks, then using the current processor may be helpful. Keep a free list per processor. When it's time to allocate a node, you consult the free list for the current processor. And when you want to free a node, you free it back to the list associated with the processor doing the free.

Unlinking a node from a linked list and pushing a node to the front of a linked list are relatively fast operations, so the processor is unlikely to change out from under you. The important distinction here is that we don't try to free the node to the list it originally came from. We return it to the list that belongs to the current processor *at the time it is freed*.

Of course, if you find that the free list is empty, then you'll have to go create some new nodes. Yes, this introduces the risk of contention, but creating new nodes will be a comparatively slow operation, so the hope is that added risk of contention is not noticeable in practice.

Bonus chatter: In the discussion that followed the exercise, there were a pair apparently contradictory claims:

- The scheduler tries to keep load even across all processors.
- The scheduler tries not to move threads between processors.

Both are right.

When a thread is ready to be scheduled, the scheduler will try to put it back on the processor that it had been running on most recently. But if that processor is not available, then the scheduler will move it to another processor.

In other words, the "try not to move threads between processors" rule is a final tie-breaker if the scheduler has multiple processors available to it. But if the thread becomes ready, and there is only one available processor, then the thread will run on that processor. If the system has decided to shut down a processor to conserve power, then the thread will go to a processor that still has power.

Raymond Chen

Follow