Accidentally creating a choke point for what was supposed to hand work off quickly to a background task, part 2

devblogs.microsoft.com/oldnewthing/20190214-00

February 14, 2019

Raymond Chen

Last time, we were looking at a function that wanted to kick work off to a background thread but inadvertently ended up blocking the main threads for about as long as the background tasks were running.

We had previously diagnosed one problem: The code used a lock around an increment operation but kept the lock active for too long, causing the creating of the background task to be serialized inadvertently.

But that by itself should not have caused the main threads to block for about as long as the background tasks were running. Sure, the queueing of the background tasks is serialized, but `QueueUserWorkItem` is relatively quick because it merely schedules the work to run. However, the observation was that the code was actually waiting for the tasks to run. What's going on?

The culprit for the bigger problem is the code that waits for the task to start running before releasing the main thread. The purpose of this wait was to ensure that the MTA was not prematurely torn down. But it had a side effect of making the code that queues task inadvertently end up waiting for them.

The thread pool is designed to maximize throughput, not to minimize latency. If you throw a lot of tasks at the thread pool, it will methodically retire them a few at a time, rather than spinning up a ton of threads and having them all running tasks at the same time, because having a ton of threads all doing CPU-intensive work causes them all to contend with each other and gives you a worse total throughput than just creating one thread per processor and running the tasks one at a time on each thread.

Let's see what happens when a thread tries to create twenty background tasks on a 4-processor system. For simplicity, let's assume that the thread pool has already reached its ideal state of having four threads.

The first task is queued, and it starts running immediately. The task releases the main thread, so the main thread returns quickly.

The second through fourth tasks also start running immediately, so the main thread resumes quickly in those cases as well.

The fifth task is different, though. All of the thread pool threads are busy, so the fifth task doesn't start right away. It's waiting for a thread to become available. Eventually, the first task (say) completes, and the fifth task can now start. Only after the fifth task starts does the main thread become released.

The process repeats with the subsequent tasks. Instead of queuing quickly and returning, the main thread sits and waits for its task to to start before it can return. As a result, the main thread spends most of its time waiting for earlier tasks to complete, which defeats the purpose of queueing them to the background thread!

Here's what the code was hoping to accomplish:

Main thread	**Thread pool thread 1**	**Thread pool thread 2**	**Thread pool thread 3**	**Thread pool thread 4**
Queue tasks 1–20	Task 1	Task 2	Task 3	Task 4
Available to do other stuff	Task 5			
				Task 6
			Task 7	
		Task 8		
				Task 9
	Task 10			
		Task 11		
				Task 12
			Task 13	
		Task 14		
	Task 15			
			Task 16	
				Task 17

			Task 18	
		Task 19		
	Task 20			Idle
		Idle		

But since the code in the main thread waits for the task to start, it means that the main thread doesn't get to do other stuff right away. It has to wait for the task it requested to start running on a thread. This means that what you actually get is this:

Main thread	**Thread pool thread 1**	**Thread pool thread 2**	**Thread pool thread 3**	**Thread pool thread 4**
Queue tasks 1–4	Task 1	Task 2	Task 3	Task 4
Queue task 5	Task 5			
Queue task 6				Task 6
Queue task 7			Task 7	
Queue task 8		Task 8		
Queue task 9				
				Task 9
Queue task 10	Task 10			
Queue task 11		Task 11		
Queue task 12				Task 12
Queue task 13			Task 13	
Queue task 14		Task 14		
Queue task 15	Task 15			
Queue task 16			Task 16	
Queue task 17				Task 17
Queue task 18			Task 18	
Queue task 19		Task 19		
Queue task 20	Task 20			Idle

Available to do other stuff		Idle		

The tasks cannot queue instantly. Instead, each attempt to queue a task stalls until the task starts running somewhere. If the thread pool happens to be very busy at the moment, then you'll have to wait a long time. In practice, what happens is that the main thread waits around until all but the last three tasks have completed.

Okay, now that we understand what the true bottleneck is, we'll try to address it next time.

Raymond Chen

Follow