Trying to make the thread pool more responsive to a large queue of long-running work items

devblogs.microsoft.com/oldnewthing/20170623-00

June 23, 2017

Raymond Chen

A customer had a question about the thread pool. Specifically, they found that when they queued a work item with the `System.Threading.ThreadPool.QueueUserWorkItem` method, the work item didn't start running until eight minutes later. They used `GetMinThreads()` and `GetMaxThreads()` to determine that the thread pool is configured with a minimum or 40 worker threads and maximum of 32767.

The system thread pool intentionally delays the creation of new threads to reduce the likelihood of "wakeup storms". Here's a Channel9 video that looks inside the thread pool. The thread pool is designed to maximize throughput, not to minimize latency.

The customer replied that the delay, while well-intentioned, is resulting in significant delays in the completion of work items. The customer wanted to know if there was some way to tell the thread pool to be more aggressive about creating new threads to help drain the backlog. Or should they just create a thread for each work item, so that it starts running immediately? The customer explained that the work items are relatively long-running because they communicate with a Web service, which means that most of the work item's running time is just waiting for a response from the service.

While there may be some CPU-bound portions of the operation, most of the time, the work item is just waiting for a response. The customer should formulate the work items as C# tasks and use the `async` infrastructure to yield threads back to the thread pool when they are waiting for the server to return with an answer.

A colleague demonstrated with a sample program that creates a thousand work items, each of which simulates a network call that takes 500ms to complete. In the first demonstration, the network calls were blocking synchronous calls, and the sample program limited the thread pool to ten threads in order to make the effect more apparent. Under this configuration, the first few work items were quickly dispatched to threads, but then the latency started to build as there were no more threads available to service new work items, so the remaining work items had to wait longer and longer for a thread to become available to service it. The average latency to the start of the work item was over two minutes.

The second version of the sample increased the number of threads in the thread pool to 200 but made no changes to the work items themselves. Under this configuration, there were more threads available to retire work items, which meant that there were five times as many threads working to retire work items. In this configuration, the average latency to the start of the work item was only four seconds.

The third version of the sample converted the work item to a C# task. Instead of performing a synchronous wait for the response from the server, the work item used the `await` keyword to initiate the I/O operation, requested to be called back when the I/O completed (by attaching a continuation to the task), and then returned the thread back to the thread pool immediately. When the simulated I/O operation completed, the task continuation ran, which took the simulated I/O results and did some imaginary work with them. In this version, even with a thread pool cap of only 10 threads, the average latency to the start of the work item was 0.0005 milliseconds. That's because the work items got in, did their work, and when they found themselves waiting on something, they got out, thereby releasing the thread pool thread to work on a new task, which could be a newly-queued work item, or the continuation of one that had already started.

The point of this exercise was to show that breaking the work item into tasks improves the efficient of the thread pool threads since they are always busy doing something rather than sitting around waiting for something to do. It's like that movie where there was a thread that had to *speed* around the work item queue, keeping its *speed* above 100% CPU, and if its *speed* dropped, it would explode. I think it was called *The thread that couldn't slow down*.

The customer replied, "This is amazing. My understanding is: enlarging min threads in thread pool and switching to Task can definitely improve the performance?"

The customer got half the point.

Increasing the number of threads in the task pool mitigates the original problem, because you increase the number of threads that can work to retire work items. Your improvement is linear in the number of threads.

Switching to C# tasks solves the problem entirely. Notice that when we switched to C# tasks, we were able to process a thousand work items in under a second, even though we had only ten threads. Indeed, even if we dropped the thread pool maximum thread count to two, the program can still process 1000 tasks in one second. The improvement is superlinear, at no additional thread cost.

Tasks allow a small number of threads to process multiple work items in pseudo-parallel. Ten threads can be juggling 100 tasks each, if the tasks spend most of their time waiting.

Enlarging the thread pool allows more threads to work at draining the task queue. Each thread can drain one task, so 200 threads can drain 200 tasks. However, if you have 200 threads in your thread pool, it's really not a thread pool any more. The purpose of a thread pool is to amortize the overhead cost of creating threads by having a single thread do multiple pieces of work. Creating 200 threads is effectively abandoning the idea that you're trying to minimize thread creation overhead. (Context switching, memory for stacks, etc.)

Imagine that each work item is a customer in line at a fast-food restaurant. Each person gets to the counter, and then stops and says, "Hm, I wonder what to get. Let me look at the menu." In version 1, each cashier has to stand there and wait for the customer to decide what they want to order.

Using tasks means that when the customer gets to the front of the line and starts looking at the menu, the cashier can say, "Let me know when you're ready," and then call "Next!" and start helping the next person in line. If that person stops and stares at the menu, then the cashier can keep calling "Next!" and have 20 people all standing there at the counter staring at the menu. Eventually, a customer will say, "Okay, I'm ready," as soon as a cashier finishes with the customer they are currently serving, that cashier can help the customer who is now ready to order. A single cashier can serve 20 customers simultaneously because through most of the transaction, the customer is staring at the menu trying to decide what to do next. During that time, the cashier can be doing something else. Furthermore, when the customer becomes ready, any cashier can resume the transaction. It doesn't have to be the cashier that started the transaction.

Using threads means that you hire 200 cashiers. Each cashier calls "Next!," and then when the customer stops to look at the menu, the cashier just stands there waiting. Youre paying the overhead for 200 cashiers, but not utilizing them efficiently.

And in fact it's worse than that. Because those 200 cashiers are not true physical cashiers standing there. The ordering system is really a video chat, and there's a person in a remote office taking the order. And the dirty secret is that the remote office has only four employees! When you hired 200 cashiers, you set up 200 video terminals, but there are only four employees at the remote office who are actually taking orders. Those four employees are switching between video chat windows based on which customers are ready. And sometimes, there are five ready customers, and the remote employee has to put an existing customer on hold in order to service the fifth customer. So basically, it's back to the same thing as tasks, but with a lot more overhead, and more rudeness (putting a customer on hold while they are in the middle of placing an order).

(In case you haven't figured it out: The person in the remote office is a CPU core.)

My colleague didn't share the code for their sample program to demonstrate tasks, so I wrote one based on the description. Here it is:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class Program
{
 static int ITERATIONS = 1000;
 static CountdownEvent done = new CountdownEvent(ITERATIONS);
 static DateTime startTime = DateTime.Now;
 static TimeSpan totalLatency = TimeSpan.FromSeconds(0);
 static SynchronizedCollection<string> messages =
  new SynchronizedCollection<string>();

 static void Log(int id, DateTime queueTime, string action)
 {
   var now = DateTime.Now;
   var timestamp = now - startTime;
   var latency = now - queueTime;
   var msg = string.Format("{0}: {1} {2,3}, latency = {3}",
     timestamp, action, id, latency);
   messages.Add(msg);
   System.Console.WriteLine(msg);
 }

 static void OnTaskStart(int id, DateTime queueTime)
 {
   var latency = DateTime.Now - queueTime;
   lock(done) totalLatency += latency;
   Log(id, queueTime, "Starting");
 }

 static void OnTaskEnd(int id, DateTime queueTime)
 {
   Log(id, queueTime, "Finished");
   done.Signal();
 }

 public static void V1()
 {
  ThreadPool.SetMaxThreads(10, 10);
  for (int i = 0; i < ITERATIONS; i++) {
   var queueTime = DateTime.Now;
   int id = i;
   ThreadPool.QueueUserWorkItem((o) => {
    OnTaskStart(id, queueTime);
    Thread.Sleep(500);
    OnTaskEnd(id, queueTime);
   });
   Thread.Sleep(10);
  }
```

```
 }

 public static void V2()
 {
  ThreadPool.SetMinThreads(20, 1);
  ThreadPool.SetMaxThreads(20, 1);
  for (int i = 0; i < ITERATIONS; i++) {
   var queueTime = DateTime.Now;
   int id = i;
   ThreadPool.QueueUserWorkItem((o) => {
    OnTaskStart(id, queueTime);
    Thread.Sleep(500);
    OnTaskEnd(id, queueTime);
   });
   Thread.Sleep(10);
  }
 }

 public static void V3()
 {
  ThreadPool.SetMaxThreads(1, 1);
  for (int i = 0; i < ITERATIONS; i++) {
   var queueTime = DateTime.Now;
   int id = i;
   ThreadPool.QueueUserWorkItem(async (o) => {
    OnTaskStart(id, queueTime);
    await Task.Delay(500);
    OnTaskEnd(id, queueTime);
   });
   Thread.Sleep(10);
  }
 }

 public static void Main(string[] args)
 {
  if (args.Length<1) return;
  switch (args[0]) {
  case "1": V1(); break;
  case "2": V2(); break;
  case "3": V3(); break;
  }
  done.Wait();
  foreach (var message in messages) {
    System.Console.WriteLine(message);
  }
  System.Console.WriteLine(
   "Average latency = {0}",
   TimeSpan.FromMilliseconds(totalLatency.TotalMilliseconds / ITERATIONS));
 }
}
```

This program simulates a workload where a new work item is requested every 10ms, for a total of 1000 work items.

Run the program with the command line parameter `1` to run the original version, where we simply sleep to simulate the network operation, on a thread pool with a maximum of ten threads. This version reports an average latency of around 32 seconds. When you run this version, notice that the only time a work item starts is when a previous work item completes.

If you use the command line parameter `2`, then you get the first alternate, which also sleeps to simulate the network operation, but on a thread pool with twenty threads. In this version, the average latency is significantly better (around 5 seconds) because we have more threads available. However, those theads are still being used inefficiently, because they spend most of their time waiting.

If you use the command line parameter `3`, then you get the second alternate, which uses asynchronous I/O (here simulated as an asynchronous delay). In this version, the average latency is minuscule (cannot even be measured), even though all the tasks are running on a single thread. That one thread runs the task as long as the task is actively doing work, but once it performs an I/O operation, the thread stops running the task and moves to another ready task. When the I/O completes, then the task continuation becomes ready, and it joins the list of tasks that our one thread can process. One thread can juggle a thousand tasks with low latency.

Raymond Chen

Follow