How can I find out how many threads are active in the CLR thread pool?

devblogs.microsoft.com/oldnewthing/20170724-00

July 24, 2017

Raymond Chen

A customer was looking for a way to determine programmatically how many threads are active in the CLR thread pool.

There is no method that returns this information directly, but you can snap two blocks together:[1]

```
int max, max2;
ThreadPool.GetMaxThreads(out max, out max2);
int available, available2;
ThreadPool.GetAvailableThreads(out available, out available2);
int running = max - available;
```

But even though we answered the question, we don't know what the customer's problem is. The customer was kind enough to explain:

> We have an issue where we exhaust the thread pool, causing our latency to skyrocket. We are investigating possible mitigations, and knowing when we are close to saturating the thread pool would tell us when we need to take more drastic measures. The thread pool threads are not CPU-bound; they are blocked on SQL queries. We have a long-term plan to use `async` / `await`, but that is a large change to our code base that will take time to implement, so we're looking for short-term mitigations to buy ourselves some time.

A colleague pointed out that if your thread pool threads are all blocked on SQL queries against the same server, then adding more threads won't help because the bottleneck is not the thread pool. The bottleneck is the SQL server. Any new thread pool threads you add will eventually block on SQL queries to the same unresponsive server.

Now, if your workload consists entirely of work items that access the database, then the database is your bottleneck, and there's not much you can do on the client to make it go faster. But if your workload is a mix of work items that access the database and work items that don't access the database, then you at least don't want the non-database work items to be blocked behind database work items.

If this were a Win32 application, you could create a second thread pool and queue database work items to that thread pool. Non-database work items go to the default thread pool. When the second thread pool runs out of threads, it stalls the processing of other database work items, but the non-database work items are not affected because they are running on a different thread pool.

But the CLR doesn't let you create a second thread pool, so your database work items and non-database work items have to learn to live in harmony.

Rewriting the code to be "`async` all the way down" may not be practical in the short term, but you could make it `async` at the top. Suppose your database work item looks like this:

```
ThreadPool.QueueUserWorkItem(() =>
{
  DoDatabaseStuff(x, y, z);
  MoreDatabaseStuff(1, 2, 3);
});
```

Add a single `async` at the top:

```
ThreadPool.QueueUserWorkItem(async () =>
{
  using (await AccessToken.AcquireAsync()) {
    DoDatabaseStuff(x, y, z);
    MoreDatabaseStuff(1, 2, 3);
  }
});
```

The purpose of the `AccessToken` class is to control how many threads are doing database stuff. We put it in a `using` so that it will be `Dispose`d when control exits the block. This ensures that we don't leak tokens.

Since the `AcquireAsync` method is async, it means that work items do not consume a thread while they are waiting for a token. By controlling the number of tokens, you can control how many thread pool threads are doing database work. In particular, you can make sure that database work items don't monopolize the thread pool threads, leaving enough thread pool threads for your non-database work items.

[1] Maoni Stephens pointed out that there's also a managed debugging library called ClrMD which gives you a lot of information about the thread pool. You may want to start with the ClrThread class.

Raymond Chen

Follow