When to use Task.Delay, when to use Thread.Sleep?

Asked 5 years, 8 months ago Active 4 months ago Viewed 180k times

312

62

Are there good rule(s) for when to use Task.Delay versus Thread.Sleep?

- Specifically, is there a minimum value to provide for one to be effective/efficient over the other?
- Lastly, since Task.Delay causes context-switching on a async/await state machine, is there an overhead of using it?

c# multithreading task-parallel-library

edited Feb 5 '14 at 23:26

David Gardiner
10.3k 12 64 109

asked Nov 19 '13 at 21:16

Tom K.
1,809 2 9 11

2 10ms is a lot of cycles in computer world... – Brad Christie Nov 19 '13 at 21:21

How fast should it be? What performance problems do you have? – L.B Nov 19 '13 at 21:28

4 I think the more pertinent question is in what context do you intend to use either of these? Without that information the scope is too broad. What do you mean by effective/efficient? Are you referring to accuracy, power efficiency etc.? I'm very curious to know in what context this matters. – James World Nov 19 '13 at 21:40

3 The minimum is 15.625 msec, values less than the clock interrupt rate have no effect. Task.Delay always burns up a System.Threading.Timer, Sleep has no overhead. You don't worry about overhead when you write code that does nothing. – Hans Passant Nov 19 '13 at 22:11

4 Answers

310

Use `Thread.Sleep` when you want to block the current thread.

Use `Task.Delay` when you want a logical delay without blocking the current thread.

Efficiency should not be a paramount concern with these methods. Their primary real-world use is as retry timers for I/O operations, which are on the order of seconds rather than milliseconds.

answered Nov 19 '13 at 23:44

Stephen Cleary
304k 50 499 631

28 in which cases do we need to block the current thread? – Tom K. Nov 20 '13 at 0:31

3 It's the same primary use case: a retry timer. – Stephen Cleary Nov 20 '13 at 0:42

4 Or when you don't want to chew up CPU in a main loop. – Eddie Parker May 20 '14 at 21:23

5 @RoyiNamir: No. There is no "other thread". Internally, it's implemented with a timer. – Stephen Cleary Nov 3 '14 at 18:34

8 The suggestion not to worry about efficiency is ill-advised. `Thread.Sleep` will block the current thread which causes a context switch. If you're using a thread pool this could also cause a new thread to be allocated. Both operations are quite heavy whereas the cooperative multi-tasking provided by `Task.Delay` etc is designed to avoid all of that overhead, maximize throughput, allow cancellation, and provide cleaner code. – Corillian May 3 '16 at 16:06

194

The biggest difference between `Task.Delay` and `Thread.Sleep` is that `Task.Delay` is intended to run asynchronously. It does not make sense to use `Task.Delay` in synchronous code. It is a VERY bad idea to use `Thread.Sleep` in asynchronous code.

Normally you will call `Task.Delay()` with the `await` keyword:

```
await Task.Delay(5000);
```

or, if you want to run some code before the delay:

```
var sw = new Stopwatch();
sw.Start();
Task delay = Task.Delay(5000);
Console.WriteLine("async: Running for {0} seconds", sw.Elapsed.TotalSeconds);
await delay;
```

Guess what this will print? Running for 0.0070048 seconds. If we move the `await delay` above the `Console.WriteLine` instead, it will print Running for 5.0020168 seconds.

Let's look at the difference with `Thread.Sleep`:

```
class Program
{
```

```
    static void Main(string[] args)
    {
        Task delay = asyncTask();
        syncCode();
        delay.Wait();
        Console.ReadLine();
    }

    static async Task asyncTask()
    {
        var sw = new Stopwatch();
        sw.Start();
        Console.WriteLine("async: Starting");
        Task delay = Task.Delay(5000);
        Console.WriteLine("async: Running for {0} seconds", sw.Elapsed.TotalSeconds);
        await delay;
        Console.WriteLine("async: Running for {0} seconds", sw.Elapsed.TotalSeconds);
        Console.WriteLine("async: Done");
    }

    static void syncCode()
    {
        var sw = new Stopwatch();
        sw.Start();
        Console.WriteLine("sync: Starting");
        Thread.Sleep(5000);
        Console.WriteLine("sync: Running for {0} seconds", sw.Elapsed.TotalSeconds);
        Console.WriteLine("sync: Done");
    }
}
```

Try to predict what this will print...

> async: Starting
> async: Running for 0.0070048 seconds
> sync: Starting
> async: Running for 5.0119008 seconds
> async: Done
> sync: Running for 5.0020168 seconds
> sync: Done

Also, it is interesting to notice that `Thread.Sleep` is far more accurate, ms accuracy is not really a problem, while `Task.Delay` can take 15-30ms minimal. The overhead on both functions is minimal compared to the ms accuracy they have (use `Stopwatch` Class if you need something more accurate). `Thread.Sleep` still ties up your Thread, `Task.Delay` release it to do other work while you wait.

edited Mar 5 at 3:11

Tarik
38k 68 209 310

answered Feb 9 '15 at 15:22

Dorus
4,805 1 22 35

12 Why is it "a VERY bad idea to use Thread.Sleep in asynchronous code"? – sunside Feb 26 '15 at 16:13

54 @sunside One of the major advantages of async code is to allow one thread to work on multiple tasks at once, by avoiding blocking calls. This avoids the need for huge amounts of individual threads, and allows a threadpool to service many requests at once. However, given that async code usually runs on the threadpool, needlessly blocking a single thread with `Thread.Sleep()` consumes an entire thread that could otherwise be used elsewhere. If many tasks are run with Thread.Sleep(), there's a high chance of exhausting all threadpool threads and seriously hindering performance. – Ryan Mar 2 '15 at 23:53

1 Get it. I was missing the notion of asynchronous code in the sense of `async` methods as they are encouraged to be used. It's basically just a bad idea to run `Thread.Sleep()` in a threadpool thread, not a bad idea in general. After all, there's `TaskCreationOptions.LongRunning` when going the (albeit discouraged) `Task.Factory.StartNew()` route. – sunside Mar 3 '15 at 8:58

5 kudos for `await wait` – Eric Wu Oct 6 '16 at 19:27

1 You can use `Thread.Sleep` in async code for very low values, e.g. `Thread.Sleep(50)` (for some HW communication). – xmedeko Nov 13 '17 at 20:31

26

if the current thread is killed and you use `Thread.Sleep` and it is executing then you might get a `ThreadAbortException` . With `Task.Delay` you can always provide a cancellation token and gracefully kill it. Thats one reason I would choose `Task.Delay` . see http://social.technet.microsoft.com/wiki/contents/articles/21177.visual-c-thread-sleep-vs-task-delay.aspx

I also agree efficiency is not paramount in this case.

edited May 19 '14 at 6:12

answered May 19 '14 at 6:00

Balanikas
1,063 8 18

2 Assume we got the following situation: `await Task.Delay(5000)` . When I kill the task I get `TaskCanceledException` (and suppress it) but my thread is still alive. Neat! :) – AndreyWD Jul 1 '17 at 10:11

20

I want to add something. Actually, `Task.Delay` is a timer based wait mechanism. If you look at the source you would find a reference to a `Timer` class which is responsible for the delay. On the other hand `Thread.Sleep` actually makes current thread to sleep, that way you are just blocking and wasting one thread. In async programming model you should always use `Task.Delay()` if you want something(continuation) happen after some delay.

edited Aug 14 '17 at 10:16

answered Jul 6 '16 at 11:44

crypted
8,224 2 33 47

'await Task.Delay()' frees the thread to do other things until the timer expires, 100% clear. But what if I cannot use 'await' since the method is not prefixed with 'async'? Then I can only call 'Task.Delay()'. In that case *the thread is still blocked* but I have the *advantage of canceling the Delay()*. Is that correct? – Erik Stroeken May 4 '17 at 8:43

4 @ErikStroeken You can pass cancellation tokens to both thread and task. Task.Delay().Wait() will block, while Task.Delay() just creates the task if used without await. What you do with that task is up to you, but the thread continues. – Physics-Compute May 6 '17 at 4:38