What is the difference between task and thread?

Asked 8 years, 8 months ago Active 21 days ago Viewed 195k times

352

137

In C# 4.0, we have `Task` in the *System.Threading.Tasks* namespace. What is the true difference between `Thread` and `Task`. I did some sample program(help taken from MSDN) for my own sake of learning with

```
Parallel.Invoke
Parallel.For
Parallel.ForEach
```

but have many doubts as the idea is not so clear.

I have initially searched in Stackoverflow for a similar type of question but may be with this question title I was not able to get the same. If anyone knows about the same type of question being posted here earlier, kindly give the reference of the link.

c# multithreading c#-4.0 task-parallel-library

edited Oct 4 '17 at 7:45

fat

2,630 3 31 54

asked Nov 9 '10 at 3:14

user372724

6 threads run tasks – pm100 Aug 13 '15 at 22:15

7 Answers

293

A task is something you want done.

A thread is one of the many possible workers which performs that task.

In .NET 4.0 terms, a Task represents an asynchronous operation. Thread(s) are used to complete that operation by breaking the work up into chunks and assigning to separate threads.

edited Mar 18 '14 at 22:06

Daniel Eagle
1,006 1 13 14

answered Nov 9 '10 at 3:15

Mitch Wheat
261k 36 411 504

428

In computer science terms, a `Task` is a *future* or a *promise*. (Some people use those two terms synomymously, some use them differently, nobody can agree on a *precise* definition.) Basically, a `Task<T>` "promises" to return you a `T`, but not right now honey, I'm kinda busy, why don't you come back later?

A `Thread` is a way of fulfilling that promise. But not every `Task` needs a brand-new `Thread`. (In fact, creating a thread is often undesirable, because doing so is much more expensive than re-using an existing thread from the threadpool. More on that in a moment.) If the value you are waiting for comes from the filesystem or a database or the network, then there is no need for a thread to sit around and wait for the data when it can be servicing other requests. Instead, the `Task` might register a callback to receive the value(s) when they're ready.

In particular, the `Task` does *not* say *why* it is that it takes such a long time to return the value. It *might* be that it takes a long time to compute, or it might that it takes a long time to fetch. Only in the former case would you use a `Thread` to run a `Task`. (In .NET, threads are freaking expensive, so you generally want to avoid them as much as possible and really only use them if you want to run multiple heavy computations on multiple CPUs. For example, in Windows, a thread weighs 12 KiByte (I think), in Linux, a thread weighs as little as 4 KiByte, in Erlang/BEAM even just 400 Byte. In .NET, it's 1 MiByte!)

edited Nov 11 '14 at 18:36

rianjs
5,056 5 17 33

answered Nov 9 '10 at 3:47

Jörg W Mittag
300k 64 369 566

28 Interestingly in the early preview releases of TPL (Task Parallel Library) there was Task and Future<T>. Future<T> was then renamed to Task<T>. :) – Lee Campbell Feb 19 '12 at 10:20

22 How did you calculate 1 MB for .NET? – dvallejo Dec 3 '12 at 22:32

5 @DanVallejo: That number was mentioned in an interview with the TPL design team. I can't tell you who said it or which interview it was, I watched that years ago. – Jörg W Mittag Dec 3 '12 at 23:07

9 @RIPUNJAYTRIPATHI Sure, but it doesn't need to be *another* thread, it could be the thread that requested the work in the first place. – Chris Pitman Jun 14 '13 at 15:43

5 .NET just uses Windows threads on Windows, so the size is the same - by default, usually 1 MiB of virtual memory for both. Physical memory is used as needed in page-sized chunks (usually 64 kiB), the same with native code. The minimum thread stack size depends on the OS - 256 kiB for Vista, for example. On x86 Linux, the default is usually 2 MiB - again, allocated in page-sized chunks. (simplification) Erlang only uses one system thread per process, those 400 bytes refer to something similar to .NETs `Task`. – Luaan Jun 9 '16 at 16:13

31

Thread

The bare metal thing, you probably don't need to use it, you probably can use a `LongRunning` task and take the benefits from the TPL - Task Parallel Library, included in .NET Framework 4 (february, 2002) and above (also .NET Core).

Tasks

Abstraction above the Threads. It **uses the thread pool** (unless you specify the task as a `LongRunning` operation, if so, a new thread is created under the hood for you).

Thread Pool

As the name suggests: a pool of threads. Is the .NET framework handling a limited number of threads for you. Why? Because opening 100 threads to execute expensive CPU operations on a Processor with just 8 cores definitely is not a good idea. The framework will maintain this pool for you, reusing the threads (not creating/killing them at each operation), and executing some of them in parallel, in a way that your CPU will not burn.

OK, but when to use each one?

In resume: always use tasks.

Task is an abstraction, so it is a lot easier to use. I advise you to always try to use tasks and if you face some problem that makes you need to handle a thread by yourself (probably 1% of the time) then use threads.

BUT be aware that:

- **I/O Bound**: For I/O bound operations (database calls, read/write files, APIs calls, etc) avoid using normal tasks, use `LongRunning` tasks (*or threads if you need to*). Because using tasks would lead you to a thread pool with a few threads busy and a lot of another tasks waiting for its turn to take the pool.
- **CPU Bound**: For CPU bound operations just use the normal tasks (that internally will use the thread pool) and be happy.

edited Jul 30 '18 at 16:43

answered Jul 14 '17 at 23:02

fabriciorissetto

6,195 2 42 51

7

You can use `Task` to specify what you want to do then attach that `Task` with a `Thread` . so that `Task` would be executed in that newly made `Thread` rather than on the GUI thread.

Use `Task` with the `TaskFactory.StartNew(Action action)` . In here you execute a delegate so if you didn't use any thread it would be executed in the same thread (GUI thread). If you mention a thread you can execute this `Task` in a different thread. This is an unnecessary work cause you can directly execute the delegate or attach that delegate to a thread and execute that delegate in that thread. So don't use it. it's just unnecessary. If you intend to optimize your software this is a good candidate to be removed.

**Please note that the `Action` is a `delegate` .

edited Sep 17 '15 at 10:47

Tajkia Rahman Toma
443 1 5 16

answered Jul 19 '12 at 23:52

Gryphes
79 1 2

6

In addition to above points, it would be good to know that:

1. A task is by default a background task. You cannot have a foreground task. On the other hand a thread can be background or foreground (Use IsBackground property to change the behavior).
2. Tasks created in thread pool recycle the threads which helps save resources. So in most cases tasks should be your default choice.
3. If the operations are quick, it is much better to use a task instead of thread. For long running operations, tasks do not provide much advantages over threads.

answered Aug 1 '17 at 23:37

user2492339
115 1 3

3

I usually use `Task` to interact with Winforms and simple background worker to make it not freezing the UI. here an example when I prefer using `Task`

```
private async void buttonDownload_Click(object sender, EventArgs e)
{
    buttonDownload.Enabled = false;
    await Task.Run(() => {
        using (var client = new WebClient())
        {
```

```
            client.DownloadFile("http://example.com/file.mpeg", "file.mpeg");
        }
    })
    buttonDownload.Enabled = true;
}
```

VS

```
private void buttonDownload_Click(object sender, EventArgs e)
{
    buttonDownload.Enabled = false;
    Thread t = new Thread(() =>
    {
        using (var client = new WebClient())
        {
            client.DownloadFile("http://example.com/file.mpeg", "file.mpeg");
        }
        this.Invoke((MethodInvoker)delegate()
        {
            buttonDownload.Enabled = true;
        });
    });
    t.IsBackground = true;
    t.Start();
}
```

the difference is you don't need to use `MethodInvoker` and shorter code.

edited Oct 3 '18 at 16:04

answered Jul 22 '18 at 9:53

ewwink

12.8k 2 25 43

Task is like a operation that you wanna perform , Thread helps to manage those operation through multiple process nodes. task is a lightweight option as Threading can lead to a complex code management
I will suggest to read from MSDN(Best in world) always

2

[Task](#)

[Thread](#)

edited Jul 8 at 7:37

answered Mar 27 '17 at 11:00

Malice
3,159 1 30 47

Saurabh
646 8 17