Thread pool for executing arbitrary tasks with different priorities

Asked 16 years, 3 months ago Modified 7 years, 1 month ago Viewed 3k times

I'm trying to come up with a design for a thread pool with a lot of design requirements for my job. This is a real problem for working software, and it's a difficult task. I have a working implementation but I'd like to throw this out to SO and see what interesting ideas people can come up with, so that I can compare to my implementation and see how it stacks up. I've tried to be as specific to the requirements as I can.

The thread pool needs to execute a series of tasks. The tasks can be short running (<1sec) or long running (hours or days). Each task has an associated priority (from 1 = very low to 5 = very high). Tasks can arrive at any time while the other tasks are running, so as they arrive the thread pool needs to pick these up and schedule them as threads become available.

The task priority is completely independant of the task length. In fact it is impossible to tell how long a task could take to run without just running it.

Some tasks are CPU bound while some are greatly IO bound. It is impossible to tell beforehand what a given task would be (although I guess it might be possible to detect while the tasks are running).

The primary goal of the thread pool is to maximise throughput. The thread pool should effectively use the resources of the computer. Ideally, for CPU bound tasks, the number of active threads would be equal to the number of CPUs. For IO bound tasks, more threads should be allocated than there are CPUs so that blocking does not overly affect throughput. Minimising the use of locks and using thread safe/fast containers is important.

In general, you should run higher priority tasks with a higher CPU priority (ref: SetThreadPriority). Lower priority tasks should not "block" higher priority tasks from running, so if a higher priority task comes along while all low priority tasks are running, the higher priority task will get to run.

The tasks have a "max running tasks" parameter associated with them. Each type of task is only allowed to run at most this many concurrent instances of the task at a time. For example, we might have the following tasks in the queue:

- A - 1000 instances - low priority - max tasks 1
- B - 1000 instances - low priority - max tasks 1

- C - 1000 instances - low priority - max tasks 1

A working implementation could only run (at most) 1 A, 1 B and 1 C at the same time.

It needs to run on Windows XP, Server 2003, Vista and Server 2008 (latest service packs).

For reference, we might use the following interface:

```
namespace ThreadPool
{
    class Task
    {
    public:
        Task();
        void run();
    };

    class ThreadPool
    {
    public:
        ThreadPool();
        ~ThreadPool();

        void run(Task *inst);
        void stop();
    };
}
```

c++ windows multithreading

Share

Improve this question

Follow

edited Sep 1, 2008 at 22:23

asked Sep 1, 2008 at 21:57

1800 INFORMATION

135k ● 30 ● 163 ● 242

5 Answers

Sorted by: Highest score (default)

5

So what are we going to pick as the basic building block for this. Windows has two building blocks that look promising :- I/O Completion Ports (IOCPs) and Asynchronous Procedure Calls (APCs). Both of these give us FIFO queuing without having to perform explicit locking, and with a certain amount of built-in OS support in places like the scheduler (for example, IOCPs can avoid some context switches).

APCs are perhaps a slightly better fit, but we will have to be slightly careful with them, because they are not quite "transparent". If the work item performs an alertable wait (::SleepEx, ::WaitForXxxObjectEx, etc.) and we accidentally dispatch an APC to the thread then the newly dispatched APC will take over the thread, suspending the previously executing APC until the new APC is finished. This is bad for our concurrency requirements and can make stack overflows more likely.

Share
Improve this answer
Follow

edited Nov 7, 2017 at 16:31

Vineet Jain
1,565 ● 4 ● 21 ● 32

answered Sep 2, 2008 at 18:51

DrPizza
18.3k ● 7 ● 42 ● 53

1

> It needs to run on Windows XP, Server 2003, Vista and Server 2008 (latest service packs).

What feature of the system's built-in thread pools make them unsuitable for your task? If you want to target XP and 2003 you can't use the new shiny Vista/2008 pools, but you can still use QueueUserWorkItem and friends.

Share Improve this answer Follow

answered Sep 1, 2008 at 21:58

DrPizza
18.3k ● 7 ● 42 ● 53

0

@DrPizza - this is a very good question, and one that strikes right to the heart of the problem. There are a few reasons why QueueUserWorkItem and the Windows NT thread pool was ruled out (although the Vista one does look interesting, maybe in a few years).

Firstly, we wanted to have greater control over when it starts up and stops threads. We have heard that the NT thread pool is reluctant to start up a new thread if it thinks that the tasks are short running. We could use the WT_EXECUTELONGFUNCTION, but we really have no idea if the task is long or short

Secondly, if the thread pool was already filled up with long running, low priority tasks, there would be no chance of a high priority task getting to run in a timely manner. The

NT thread pool has no real concept of task priorities, so we can't do a QueueUserWorkItem and say "oh by the way, run this one right away".

Thirdly, (according to MSDN) the NT thread pool is not compatible with the STA apartment model. I'm not sure quite what this would mean, but all of our worker threads run in an STA.

Share Improve this answer Follow

answered Sep 1, 2008 at 22:18

1800 INFORMATION

135k ● 30 ● 163 ● 242

0

> @DrPizza - this is a very good question, and one that strikes right to the heart of the problem. There are a few reasons why QueueUserWorkItem and the Windows NT thread pool was ruled out (although the Vista one does look interesting, maybe in a few years).

Yeah, it looks like it got quite beefed up in Vista, quite versatile now.

OK, I'm still a bit unclear about how you wish the priorities to work. If the pool is currently running a task of type A with maximal concurrency of 1 and low priority, and it gets given a new task also of type A (and maximal concurrency 1), but this time with a high priority, what should it do?

Suspending the currently executing A is hairy (it could hold a lock that the new task needs to take, deadlocking the system). It can't spawn a second thread and just let it run alongside (the permitted concurrency is only 1). But it can't wait until the low priority task is completed, because the runtime is unbounded and doing so would allow a low priority task to block a high priority task.

My presumption is that it is the latter behaviour that you are after?

Share Improve this answer Follow

answered Sep 1, 2008 at 22:37

DrPizza

18.3k ● 7 ● 42 ● 53

0

@DrPizza:

> OK, I'm still a bit unclear about how you wish the priorities to work. If the pool is currently running a task of type A with maximal concurrency of 1 and low priority, and it gets given a new task also of type A (and maximal concurrency 1), but this time with a high priority, what should it do?

This one is a bit of a tricky one, although in this case I think I would be happy with simply allowing the low-priority task to run to completion. Usually, we wouldn't see a lot of the same types of tasks with different thread priorities. In our model it is actually possible to safely halt and later restart tasks at certain well defined points (for different reasons than this) although the complications this would introduce probably aren't worth the risk.

Normally, only different types of tasks would have different priorities. For example:

- A task - 1000 instances - low priority
- B task - 1000 instances - high priority

Assuming the A tasks had come along and were running, then the B tasks had arrived, we would want the B tasks to be able to run more or less straight away.

answered Sep 1, 2008 at 22:46

1800 INFORMATION

135k ● 30 ● 163 ● 242