Introduction to the Windows Threadpool (Part 1)

Hari Pulapaka - MSFT 10 Oct 2010 10:12 PM 10

Download source code from MSDN Code Gallery.

I regularly receive feedback that the Win32 Threadpool API is complex and there is a need for better examples. To improve this situation, I decided to create three wrapper C++ classes which provide the following; queuing work items, associating callbacks with events and timer functionality. You can directly use these wrapper classes or look at the source to understand how to use the Threadpool APIs. These classes are header only code in the ***windowsthreadpool*** namespace and can be used by including the header file "WindowsThreadPool.h". The entire source code is available on MSDN Code Gallery.

A little background on threadpools:

Threads are the basic abstraction used to schedule work on the CPU in Windows. With the increase in the number of cores/CPUs, developers need to architect their applications to run asynchronously and exploit maximum performance out of the system. There are usually two approaches to running code asynchronously; explicitly creating threads to run code asynchronously or using system provided facilities like threadpool which manage thread lifetimes.

Explicitly managing thread lifetimes is cumbersome from a coding perspective and can degrade system and application performance if the lifetimes are managed poorly. In addition, creating and destroying threads is an expensive operation; however having too many threads and not enough work is not ideal either, as it increases system memory utilization and context-switch overhead.

So how does a developer decide the ideal number of threads and get it right on different machines with different number of cores? That's where Windows threadpool comes into the picture; it frees the developer from managing thread lifetimes and provides a pool of worker threads appropriate for the hardware. The developer queues work-items to the threadpool which executes them asynchronously. As long as there are free CPUs to execute those work-items the threadpool will create new threads to run them and once there is no more work, the threadpool will destroy threads based on its internal timeout heuristics.

Process-wide Threadpool

Every user-mode process on Windows has a default threadpool available to it, the application does not need to initialize it and can submit work items by calling TrySubmitThreadpoolCallback or SubmitThreadpoolWork. System components within Windows use this to execute work items within the process.
You can use the class windowsthreadpool::SimpleThreadPool to queue work items to the default threadpool.

Note: All classes in windowsthreadpool namespace only support callback functions with the following signature. A function accepting a single PVOID pointer (can be NULL) and returns nothing.

```
void CALLBACK FunctionName(PVOID state)
```

This is conveniently typedef-ed to THREADPOOLCALLBACK in the windowsthreadpool namespace. In the example below, ***PrintI*** uses the PVOID parameter whereas ***PrintRand*** does not.

Example Usage:

```
#include <tchar.h>
#include <iostream>
#include <Windows.h>
#include <assert.h>
#include <functional>

#include "WindowsThreadPool.h"

void CALLBACK PrintI(PVOID state)
{
	int *i = reinterpret_cast<int *> (state);
	cout << *i << " ," << endl;
}
void CALLBACK PrintRand(PVOID state)
{
	UNREFERENCED_PARAMETER(state);
```

```
	cout << rand() << " ," << endl;
}

int _cdecl _tmain()
{
	using namespace windowsthreadpool;

	SimpleThreadPool stp;

	int *arr = new int[10];
	for(int i=0; i<10; ++i)
		arr[i] = i;

	for(int i=0; i<10; ++i)
	{
	    stp.QueueUserWorkItem(PrintI, &arr[i]);
	    stp.QueueUserWorkItem(PrintRand);
	}

	stp.WaitForAll();
}
```

You queue work items to the threadpool using the QueueUserWorkItem function. This overloaded function can accept up to two parameters; the function pointer (PrintI) and the data (array element) that needs to be passed into the function. If you don't need to pass in any data to the function pointer (PrintRand), then you don't need to pass anything. To wait for all the work items to complete, you need to call WaitForAll. To wait for currently running work items and cancel any queued work items which haven't started, use WaitForAllCurrentlyRunning.

Notice you didn't use any of the win32 threadpool APIs or create any PTP_WORK objects; all you had to do was define your callback function and pass that function pointer to QueueUserWorkItem function. To understand how the SimpleThreadPool class works let take a look at the internal helper class SimpleCallBack. This is the key piece which submits work items to the process-wide threadpool and hides all the PTP_ parameters.

```
template <class Function>
class SimpleCallBack
{
private:
	const Function m_Func;
	PVOID state;

	static void CALLBACK callback (PTP_CALLBACK_INSTANCE Instance, PVOID Param)
	{
		UNREFERENCED_PARAMETER(Instance);
		SimpleCallBack<Function> *pc = reinterpret_cast<SimpleCallBack<Function>*>
(Param);
		pc->m_Func(pc->state);
		delete pc;
		return;
	}
public:
	SimpleCallBack(const Function Func, PVOID st, PTP_CALLBACK_ENVIRON pEnv) :
m_Func(Func), state(st)
	{
		if (!TrySubmitThreadpoolCallback(callback, this, pEnv))
		{
			throw "Error: Could not submit work item.";
		}
	}
};
```

The user provided callback function is executed in the callback function when the line pc->m_Func (pc->state) is executed. If the user does not provide any parameter for the callback, the state value is NULL. The TrySubmitThreadpoolCallback function can take in a pointer to an environment block or it can be NULL in which case the work item would run in the default environment. In this case, it's taking the environment block which is initialized in the class SimpleThreadPool and this environment block is associated with all work items queued to SimpleThreadPool.

The WaitForAll function waits for all work items to complete; it accomplishes this by associating every work item with the same cleanup group. The cleanup group is a convenience which allows us to wait for all work items with one single call to CloseThreadpoolCleanupGroupMembers. It also frees all threadpool structures associated with the cleanup group with one call to CloseThreadpoolCleanupGroup. The second parameter to CloseThreadpoolCleanupGroupMembers controls whether to cancel any work items which haven't started running yet.

```
void WaitForAllCallbacks(bool CancelNotStarted)
{
	assert(InfraInitialized == true);
```

```
        CloseThreadpoolCleanupGroupMembers(CleanupGroup, CancelNotStarted, NULL);
        CloseThreadpoolCleanupGroup(CleanupGroup);
```

Next up, work item priority...

Comments

Kind of old-style.. 13 Oct 2010 12:45 AM #
I suggest you get to know boost libraries. Functors, locally-scoped mutexes and boost::thread implementation makes it easy to write your own thread pool without pointer-to-function and void* argument hassle. It is not windows-bound as well.

Chris 25 Oct 2010 7:02 PM #
@Kind of old-style, how do you know the author doesn't know about the Boost libraries?

The purpose of this article is to introduce reader to Windows thread pools. A library would hide the details.

Some issues 25 Jan 2012 11:31 AM #
Here's a fix to this memory leak waiting to happen: if (!TrySubmitThreadpoolCallback(callback, this, pEnv))

BOOL Queue()

{

  return !TrySubmitThreadpoolCallback(callback, this, m_pEnv); //made last param a class var

}

template <class Function>

bool QueueUserWorkItemInternal(Function cb, PVOID State)

{

InitializeInfra();

windowsthreadpool::internal::SimpleCallBack<Function> *scb = new windowsthreadpool::internal::SimpleCallBack<Function>(cb, State, &CallbackEnvironment);

    if (scb->Queue == FALSE)

    {

      delete scb; //clean up here on failure to create thread

      return false;

    }

return true;

}

Additionally, this isn't a static or singleton class and you fail to create a destructor to do the following:

CloseThreadpoolCleanupGroupMembers

CloseThreadpoolCleanupGroup

DestroyThreadpoolEnvironment

You leave the first two in the user's hands and the last one never gets called at all. Additionally, it might be worth the 3 lines to add:

PTP_POOL m_tpool; //class variable in SimpleThreadPool

m_tpool = CreateThreadpool(NULL); //initialize

CloseThreadpool(m_tpool); //destructor

So that Windows can properly shut it down. Whenever I used this in testing a server with this that it would hang when shutting down the service. Although I haven't fully tested it, I imagine it's because the threadpool endlessly waits instead of forces the threads to shut down.

Hari Pulapaka - MSFT 25 Jan 2012 11:59 AM #
@Some issues, thanks for the changes. I will try to update the samples.

Harvey B 6 Aug 2012 1:05 PM #
Grrrr! I am really frustrated at all the crap I have to do to post a simple message.

Is this code based on the old (Win XP) thread pool code or the 'new' (ie. Vista+) code? This is an important detail but is left out of the article. The details can be found at:

<adilevin.wordpress.com/.../>

<msdn.microsoft.com/.../cc163327.aspx>

I am interested in the Vista+ version of it.

Harvey B 17 Aug 2012 7:26 PM #
Okay, I figured out that this is for the Vista++ thread pool logic. In fact, as presented, it won't work as provided unless either _WIN32_WINNT >= _WIN32_WINNT_WIN7 or else the function SetThreadpoolCallbackPriority() needs to be #if'd out with the same test.

I somewhat get the reason for namespace, but I am a bit puzzled why all the code is templated instead of just plain classes and class methods. You haven't really removed the cast to PVOID or added any logic that is enhanced with the template features. Also since it is in template form, everything has to be in the .h file and no code can be factored out to a .cpp and linked with a single copy in the .obj file(s). This will make for bulkier .obj code with a lot of repetition (that I don't think is reduced if COMDAT folding is enabled).

Also, another minor problem with the source... in SimpleThreadPool.h, method "bool QueueUserWorkItemInternal(Function cb, PVOID State, PTP_CALLBACK_ENVIRON env)" calls "InitializeInfra(false);" in line 194 which should be "InitializeInfra();"

Having said all that, this is really good stuff. Thanks for the effort Hari.

Hari Pulapaka - MSFT 18 Aug 2012 8:18 AM #
@Harvey, yes this is the Vista+ threadpool API. the old threadpool APIs should not be used. You are right I could have avoided templates but when I started this I was thinking of using lambdas but decided against it in the end...

Amit 1 Jun 2014 6:12 AM #
The link to download the source code seems to have broken.

Can you please help me out by fixing it?

Tomas 16 Oct 2014 11:20 PM #
Also:

The link to download the source code seems to have broken.

Can you please help me out by fixing it?

Tomas 16 Oct 2014 11:22 PM #
The link to download the source code seems to have broken.

Can you please help me out by fixing it?