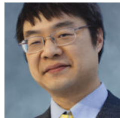
The importance of passing the WT_EXECUTELONGFUNCTION flag to QueueUserWorkItem

devblogs.microsoft.com/oldnewthing/20050722-15

July 22, 2005


Raymond Chen

One of the flags to the `QueueUserWorkItem` function is `WT_EXECUTELONGFUNCTION`. The documentation for that flag reads

> The callback function can perform a long wait. This flag helps the system to decide if it should create a new thread.

As noted in the documentation, the thread pool uses this flag to decide whether it should create a new thread or wait for an existing work item to finish. If all the current thread pool threads are busy running work items and there is another work item to dispatch, it will tend to wait for one of the existing work items to complete if they are "short", because the expectation is that some work item will finish quickly and its thread will become available to run a new work item. On the other hand, if the work items are marked `WT_EXECUTELONGFUNCTION`, then the thread pool knows that waiting for the running work item to complete is not going to be very productive, so it is more likely to create a new thread.

If you fail to mark a long work item with the `WT_EXECUTELONGFUNCTION` flag, then the thread pool ends up waiting for that work item to complete, when it really should be kicking off a new thread. Eventually, the thread pool gets impatient and figures out that you lied to it, and it creates a new thread anyway. But it often takes a while before the thread pool realizes that it's been waiting in vain.

Let's illustrate this with a simple console program.

```
#include <windows.h>
#include <stdio.h>


DWORD g_dwLastTick;


void CALLBACK Tick(void *, BOOLEAN)
{
 DWORD dwTick = GetTickCount();
 printf(“%5d\n”, dwTick – g_dwLastTick);
}


DWORD CALLBACK Clog(void *)
{
 Sleep(4000);
 return 0;
}


int __cdecl
main(int argc, char* argv[])
{
 g_dwLastTick = GetTickCount();
 switch (argc) {
 case 2: QueueUserWorkItem(Clog, NULL, 0); break;
 case 3: QueueUserWorkItem(Clog, NULL, WT_EXECUTELONGFUNCTION); break;
 }
 HANDLE hTimer;
 CreateTimerQueueTimer(&hTimer, NULL, Tick, NULL, 250, 250, 0);
 Sleep(INFINITE);
 return 0;
}
```

This program creates a periodic thread pool work item that fires every 250ms, and which merely prints how much time has elapsed since the timer was started. As a baseline, run the program with no parameters, and observe that the callbacks occur at roughly 250ms intervals, as expected.

```
  251
  501
  751
 1012
^C
```

Next, run the program with a single command line parameter. This causes the “case 2” to be taken, where the “Clog” work item is queued. The “Clog” does what its names does: It clogs up the work item queue by taking a long time (four seconds) to complete. Notice that the first callback doesn’t occur for a whole second.

```
 1001
 1011
 1021
 1021
 1252
 1502
 1752
^C
```

That's because we queued the "Clog" work item without the `WT_EXECUTELONGFUNCTION` flag. In other words, we told the thread pool, "Oh, don't worry about this guy, he'll be finished soon." The thread pool wanted to run the Tick event, and since the Clog work item was marked as "fast", the thread pool decided to wait for it and recycle its thread rather than create a new one. After about a second, the thread pool got impatient and spun up a new thread to service the now-long-overdue Tick events.

Notice that as soon as the first Tick event was processed, three more were fired in rapid succession. That's because the thread pool realized that it had fallen four events behind (thanks to the clog) and had to fire the next three immediately just to clear its backlog. The fifth and subsequent events fire roughly on time because the thread pool has figured out that the Clog really is a clog and should be treated as a long-running event.

Finally, run the program with two command line parameters. This causes the "case 3" to be taken, where we queue up the Clog but also pass the `WT_EXECUTELONGFUNCTION` flag.

```
  251
  511
  761
 1012
^C
```

Notice that with this hint, the thread pool no longer gets fooled by the Clog and knows to spin up a new thread to handle the Tick events.

Moral of the story: If you're going to go wading into the thread pool, make sure you play friendly with other kids and let the thread pool know ahead of time whether you're going to take a long time. This allows the thread pool to keep the number of worker threads low (thus reaping the benefits of thread pooling) while still creating enough threads to keep the events flowing smoothly.

Exercise: What are the consequences to the thread pool if you create a thread pool timer whose callback takes longer to complete than its timer period?

Raymond Chen

Follow