Why is my call to ChangeTimerQueueTimer having no effect? (And another case of looking past the question to solve the problem.)

devblogs.microsoft.com/oldnewthing/20151230-00

December 30, 2015

Raymond Chen

A customer had a question about the `ChangeTimerQueueTimer` function, and they shared a sample program that demonstrated the issue. (Hooray for reduction.)

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

VOID CALLBACK WaitOrTimerCallback(
       PVOID lpParameter,
       BOOLEAN TimerOrWaitFired
    )
{
   std::cout << "tick" << std::endl;
}


int main(int argc, char* argv[])
{
    HANDLE timer;
    if (!CreateTimerQueueTimer(&timer, nullptr,
             WaitOrTimerCallback, nullptr, 0, 0, 0)) abort();

   // This prints "tick" once, and then stops.
   Sleep(1000);

   if (!ChangeTimerQueueTimer(nullptr, timer, 0, 1000)) abort();

   // We expect to print "tick" once every second,
   // but we get nothing.
   Sleep(5000);
   return 0;
}
```

The customer asked, "Why aren't we getting the ticks after calling `ChangeTimerQueue-Timer`?"

This is explained down in the documentation for the `ChangeTimerQueueTimer` function:

> If you call **ChangeTimerQueueTimer** on a one-shot timer (its period is zero) that has already expired, the timer is not updated.

"Aha," they replied. "What we should do is instead of creating a one-shot timer that has already expired, we should create a one-shot timer that has an `INFINITE` due time." The customer explained that they want to create a dormant periodic timer that they could activate later. Their idea was to create a one-shot timer and ignore its first tick. Then later, when they want to activate the periodic timer, they change the one-shot timer to a periodic timer, and then the money starts flowing in.

If your goal is to create a dormant timer that you can turn on and off later, then this is the wrong way to do it.

The `CreateTimerQueueTimer` and `ChangeTimerQueueTimer` functions belong to the old Windows 2000 thread pool functions. Windows Vista introduced a new thread pool that is significantly more flexible, and all the old thread pool functions turned into wrappers around the new thread pool.

In the new model, scheduling tasks on the thread pool is done in two steps: First, you create an object to represent the thread pool task. Second, you activate the object. The task can be deactivated and reactivated as many times as you like, up until the point where you close it.

One nice feature of the two-step model is that all memory allocations take place during object creation. Once the task is created, activating and deactivating it will always succeed. This saves you from writing a bunch of complicated rollback code if you need to create a bunch of things that are dependent on each other. Under the old model, to launch a process and schedule a function to be called when the process exits, you would have to write something like this:

```
PROCESS_INFORMATION pi;
if (!CreateProcess(...,CREATE_SUSPENDED, ..., &pi)) return ERROR;
if (!RegisterWaitForSingleObject(&wait, pi.hProcess, ...)) {
 // roll back the CreateProcess
 TerminateProcess(pi.hProcess, ...);
 CloseHandle(pi.hProcess);
 CloseHandle(pi.hThread);
 return ERROR;
}

// success: The process may start running now.
ResumeThread(pi.hThread);

// The wait handler will close the handles.
```

Under the new model, you can do all the things that can fail up front, before you commit to anything that will be difficult to roll back.

```
wait = CreateThreadpoolWait(...);
if (!wait) return ERROR;

// We have everything we need - now go create the process.
// (Don't need to create suspended now.)
PROCESS_INFORMATION pi;
if (!CreateProcess(...,, &pi)) {
 CloseThreadpoolWait(wait); // not using it after all
 return ERROR;
}

// Activate the wait now that we know what the handle is.
// This cannot fail.
SetThreadpoolWait(wait, pi.hProcess, ...);
```

And we are lucky that it's possible to create a process suspended in the first place, so that rolling back process creation is possible at all. If it were not possible to create a process suspended, then we would be stuck, because we end up in a situation where a process has already started, but we are unable to wait for it, and we are unable to recall the process to say, "No, don't do that thing I asked you to do because it turns out that I can't… oh rats."

Two-phase initialization also avoids race conditions: You can prepare a timer without activating it, then activate it only after the other prerequisites are met.

Anyway, for our customer, the plan would be to write the code like this:

```
VOID CALLBACK TimerCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID Context,
    PTP_TIMER Timer
    )
{
   std::cout << "tick" << std::endl;
}

int main(int argc, char* argv[])
{
    PTP_TIMER timer = CreateThreadpoolTimer(
       TimerProc, nullptr, nullptr);
    if (!timer) abort();

   // This prints nothing.
   Sleep(1000);

   // Activate the timer.
   FILETIME dueTime = { 0, 0 }; // due immediately
   SetThreadpoolTimer(timer, &dueTime, 1000, 100);

   // This prints "tick" every second.
   Sleep(5000);

   // Deactivate the timer.
   SetThreadpoolTimer(timer, nullptr, 0, 0);

   // Nothing gets printed.
   Sleep(5000);

   // Reactivate it.
   SetThreadpoolTimer(timer, &dueTime, 1000, 100);

   // This prints "tick" every second.
   Sleep(5000);

   // All done. (Note race condition mentioned in documentation.)
   CloseThreadpoolTimer(timer);

   return 0;
}
```

Raymond Chen

Follow