Various ways of performing an operation asynchronously after a delay

devblogs.microsoft.com/oldnewthing/20121129-00

November 29, 2012

Raymond Chen

Okay, if you have a UI thread that pumps messages, then the easiest way to perform an operation after a delay is to set a timer. But let's say you don't have a UI thread that you can count on.

One method is to burn a thread:

```
#define ACTIONDELAY (30 * 60 * 1000) // 30 minutes, say
DWORD CALLBACK ActionAfterDelayProc(void *)
{
 Sleep(ACTIONDELAY);
 Action();
 return 0;
}
BOOL PerformActionAfterDelay()
{
 DWORD dwThreadId;
 HANDLE hThread = CreateThread(NULL, 0, ActionAfterDelayProc,
                              NULL, 0, &dwThreadId);
 BOOL fSuccess = hThread != NULL;
 if (hThread) {
  CloseHandle(hThread);
 }
 return fSuccess;
}
```

Less expensive is to borrow a thread from the thread pool:

```
BOOL PerformActionAfterDelay()
{
 return QueueUserWorkItem(ActionAfterDelayProc, NULL,
                          WT_EXECUTELONGFUNCTION);
}
```

But both of these methods hold a thread hostage for the duration of the delay. Better would be to consume a thread only when the action is in progress. For that, you can use a thread pool timer:

```
void CALLBACK ActionAfterDelayProc(void *lpParameter, BOOLEAN)
{
 HANDLE *phTimer = static_cast<HANDLE *>(lpParameter);
 Action();
 DeleteTimerQueueTimer(NULL, *phTimer, NULL);
 delete phTimer;
}
BOOL PerformActionAfterDelay()
{
 BOOL fSuccess = FALSE;
 HANDLE *phTimer = new(std::nothrow) HANDLE;
 if (phTimer != NULL) {
  if (CreateTimerQueueTimer(
     phTimer, NULL, ActionAfterDelayProc, phTimer,
     ACTIONDELAY, 0, WT_EXECUTEONLYONCE)) {
   fSuccess = TRUE;
  }
 }
 if (!fSuccess) {
  delete phTimer;
 }
 return fSuccess;
}
```

The timer queue timer technique is complicated by the fact that we want the timer to self-cancel, so it needs to know its handle, but we don’t know the handle until after we’ve scheduled it, at which point it’s too late to pass the handle as a parameter. In other words, we’d ideally like to create the timer, and then once we get the handle, go back in time and pass the handle as the parameter to `CreateTimerQueueTimer`. Since the Microsoft Research people haven’t yet perfected their time machine, we solve this problem by passing the handle by address: The `CreateTimerQueueTimer` function fills the address with the timer, so that the callback function can read it back out.

In practice, this additional work is no additional work at all, because you’re already passing some data to the callback function, probably an object or at least a pointer to a structure. You can stash the timer handle inside that object. In our case, our object is just the handle itself. If you prefer to be more explicit:

```
struct ACTIONINFO
{
 HANDLE hTimer;
};
void CALLBACK ActionAfterDelayProc(void *lpParameter, BOOLEAN)
{
 ACTIONINFO *pinfo = static_cast<ACTIONINFO *>(lpParameter);
 Action();
 DeleteTimerQueueTimer(NULL, pinfo->hTimer, NULL);
 delete pinfo;
}
BOOL PerformActionAfterDelay()
{
 BOOL fSuccess = FALSE;
 ACTIONINFO *pinfo = new(std::nothrow) ACTIONINFO;
 if (pinfo != NULL) {
  if (CreateTimerQueueTimer(
     &pinfo->hTimer, NULL, ActionAfterDelayProc, pinfo,
     ACTIONDELAY, 0, WT_EXECUTEONLYONCE)) {
   fSuccess = TRUE;
  }
 }
 if (!fSuccess) {
  delete pinfo;
 }
 return fSuccess;
}
```

The threadpool functions were redesigned in Windows Vista to allow for greater reliability and predictability. For example, the operations of creating a timer and setting it into action are separated so that you can preallocate your timer objects (inactive) at a convenient time. Setting the timer itself cannot fail (assuming valid parameters). This makes it easier to handle error conditions since all the errors happen when you preallocate the timers, and you can deal with the problem up front, rather than proceeding ahead for a while and then realizing, "Oops, I wanted to set that timer but I couldn't. Now how do I report the error and unwind all the work that I've done so far?" (There are other new features, like *cleanup groups* that let you clean up multiple objects with a single call, and being able to associate an execution environment with a library, so that the DLL is not unloaded while it still has active thread pool objects.)

The result is, however, a bit more typing, since there are now two steps, creating and setting. On the other hand, the new threadpool callback is explicitly passed the `PTP_TIMER`, so we don't have to play any weird time-travel games to get the handle to the callback, like we did with `CreateTimerQueueTimer`.

```
void CALLBACK ActionAfterDelayProc(
    PTP_CALLBACK_INSTANCE, PVOID, PTP_TIMER Timer)
{
 Action();
 CloseThreadpoolTimer(Timer);
}
BOOL PerformActionAfterDelay()
{
 BOOL fSuccess = FALSE;
 PTP_TIMER Timer = CreateThreadpoolTimer(
                      ActionAfterDelayProc, NULL, NULL);
 if (Timer) {
  LONGLONG llDelay = -ACTIONDELAY * 10000LL;
  FILETIME ftDueTime = { (DWORD)llDelay, (DWORD)(llDelay >> 32) };
  SetThreadpoolTimer(Timer, &ftDueTime, 0, 0); // never fails!
  fSuccess = TRUE;
 }
 return fSuccess;
}
```

Anyway, that’s a bit of a whirlwind tour of some of the ways of arranging for code to run after a delay.

Raymond Chen

Follow