A more efficient solution to the problem of a long-running task running on the thread pool persistent thread

devblogs.microsoft.com/oldnewthing/20170217-00

February 17, 2017

Raymond Chen

Last time, we found one solution to the problem of the long-running task on the persistent thread: Namely, put the long-running task on a regular thread. But that's not the best solution, because it still burns a thread.

The better solution is to let the thread pool manage the wait. Instead of dedicating a task pool thread to waiting around for a specific type of work to do, the thread pool can merge the wait with all the other thread pool wait operations onto a single thread. This keeps all task pool threads available for doing actual work.

```
// Error checking elided for expository purposes.

void WidgetMonitor::RegisterNotificationWait(
 WidgetNotificationContext* context)
{
 RegisterWaitForSingleObject(&context->waitHandle,
  context->registryEvent,
  WidgetNotificationWaitCallback,
  context,
  INFINITE,
  WT_EXECUTEONLYONCE | WT_EXECUTEINPERSISTENTTHREAD);
 RegNotifyChangeKeyValue(context->hkey, false,
                         REG_NOTIFY_CHANGE_LAST_SET,
                         context->registryEvent, TRUE);
}

void WidgetMonitor::WidgetNotificationStartCallback(void* parameter)
{
 WidgetNotificationContext* context =
   reinterpret_cast<WidgetNotificationContext*>(parameter);

 context->hkey = ...;
 context->registryEvent = ...;
 RegisterNotificationWait(context);
}

void WidgetMonitor::WidgetNotificationWaitCallback(
    void* parameter, BOOLEAN /* TimerOrWaitFired */)
{
 WidgetNotificationContext* context =
   reinterpret_cast<WidgetNotificationContext*>(parameter);

 ... process the change ...

 RegisterNotificationWait(context);
}

void WidgetMonitor::StartMonitoring()
{
 auto context = new WidgetNotificationContext();
 QueueUserWorkItem(WidgetNotificationStartCallback,
                   context,
                   WT_EXECUTEINPERSISTENTTHREAD);
}

void WidgetMonitor::StopMonitoring(
    WidgetNotificationContext* context)
{
 // WARNING! Massive race conditions here need to be addressed.

 if (context->waitHandle) {
  UnregisterWait(context->waitHandle);
```

```
  context->waitHandle = nullptr;
 }
 ... clean up other resources ...
 delete context;
}
```

The basic idea is that you start the ball rolling by queueing `WidgetNotificationStartCallback` onto the persistent thread. This task opens the registry key and registers the notification. The registration must take place on a persistent thread, and we use the thread pool persistent thread for this purpose, since we are not going to keep a thread captive for the duration of the registration.

When the change occurs, we process it (quickly, since we're on the persistent thread), then register another wait. We use one-time waits because we don't want two sets of changes to be processed simultaneously.

When the client wants to stop receiving notifications, we unregister the wait, which prevents us from reacting to any future changes. And we clean up any other resources before deleting the context. (Of course, you probably would put all of this code into the `WidgetNotificationContext` destructor, but I'm putting it here for explicitness.)

Now, this code has race conditions galore. For example, what if a change is being processed at the moment the client deides to stop notifications? I'll leave closing all the race windows (and adding proper error handling) as an exercise. You may find that the error handling is a lot easier if you switch to functions like `CreateThreadPoolWait`, which let you preallocate all the resources that will be used by a future wait operation, thereby removing an error scenario.

If processing the change notification is slow (for example, because it waits for the client to respond), then we cannot do that work on the persistent thread. Instead, we should queue the wait callback to a regular thread pool thread, and then queue another `WidgetNotificationStartCallback` back to the persistent thread to request the next registry notification. While you're at it, move all the code that initializes the `context` into the `StartMonitoring` method. This solves two problems: First, it lets you handle errors more easily, since you can report them to the code that valled `StartMonitoring`. But more important, it avoids double-initializing the `context`.

Raymond Chen

Follow