Why did my thread pool stop processing work once it hit a long-running work item?

devblogs.microsoft.com/oldnewthing/20170215-00

February 15, 2017

Raymond Chen

A customer found that occasionally, their program's thread pool stopped processing work items queued with the `WT_EXECUTEINPERSISTENTHREAD` flag. They would queue up the work items to the thread pool, but the work items would not get dispatched. Naturally, this caused problems with the program because certain background actions would simply stall.

After some investigation, the problem was traced to this work item that appears to be preventing the thread pool from processing any work:

```
ntdll!NtWaitForMultipleObjects+0xc
KERNELBASE!WaitForMultipleObjectsEx+0xcc
kernel32!WaitForMultipleObjects+0x19
contoso!WidgetMonitor::WidgetNotificationCallback+0xfd
contoso!std::tr1::_Callable_fun<void (__stdcall*const)
    (std::tr1::shared_ptr<WidgetService>),0>::_ApplyX+0x1b
contoso!std::tr1::_Impl_no_alloc1<std::tr1::_Callable_fun<
    void (__stdcall*const)(std::tr1::shared_ptr<WidgetService>),0>,
    void,std::tr1::shared_ptr<WidgetService> &>::_Do_call+0x2d
contoso!std::tr1::_Function_impl1<void,std::tr1::shared_ptr<
    WidgetMonitor::WidgetNotificationContext> &>::operator()+0x1e
contoso!Win32Adapters::Threading::Callback<std::tr1::shared_ptr<
    WidgetMonitor::WidgetNotificationContext> >::
    ExecuteCallbackTarget+0x3f
contoso!Win32Adapters::Threading::Callback<std::tr1::shared_ptr<
    WidgetMonitor::WidgetNotificationContext> >::
    DefaultThreadProc+0xd
ntdll!RtlpTpWorkCallback+0xef
ntdll!TppWorkerThread+0x4f3
kernel32!BaseThreadInitThunk+0x24
ntdll!__RtlUserThreadStart+0x2f
ntdll!_RtlUserThreadStart+0x1b
```

(I inserted line breaks for readability.)

Once they closed the widget monitor, the thread pool woke up and the work items that targeted the persistent thread started running again.

Okay, first things first: For expository purposes, let's remove all of the `std::tr1` stuff and pretend that the stack was this:

```
ntdll!NtWaitForMultipleObjects+0xc
KERNELBASE!WaitForMultipleObjectsEx+0xcc
kernel32!WaitForMultipleObjects+0x19
contoso!WidgetMonitor::WidgetNotificationCallback+0xfd
ntdll!RtlpTpWorkCallback+0xef
ntdll!TppWorkerThread+0x4f3
kernel32!BaseThreadInitThunk+0x24
ntdll!__RtlUserThreadStart+0x2f
ntdll!_RtlUserThreadStart+0x1b
```

That gets rid of the project's internal callback scaffolding and lets us focus on the interaction with the operating system.

The problem isn't really visible in the stack trace. We'll have to go to the code.

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

 RAII_HKEY hkey = ...;
 RAII_HANDLE registryEvent = ...;
 bool keepWaiting = true;
 while (keepWaiting) {
  if (RegNotifyChangeKeyValue(hkey, false, REG_NOTIFY_CHANGE_LAST_SET,
                              registryEvent, TRUE) == ERROR_SUCCESS) {
   HANDLE handles[2] = { registryEvent, context->shutdownEvent };
   DWORD waitResult = WaitForMultipleObjects(2, handles, FALSE, INFINITE);
   switch (waitResult) {
   case WAIT_OBJECT_0: // the registry key changed
    ...
    break;
   case WAIT_OBJECT_0+1: // we are being asked to shut down
    ...
    keepWaiting = false;
    break;
   default: // Something unexpected happened
    ...
    keepWaiting = false;
    break;
   }
  }
 }
}
```

The deal is that the callback function processes the callback, and then goes into a loop monitoring a registry key. It continues monitoring the key until the shutdown event is signaled.

Okay, so this looks a little weird, holding a thread pool thread hostage for an extended period of time, which is sort of contrary to the intent of a thread pool, which is to reuse a thread for multiple short work items. But it's technically legal, and you are encouraged to pass the `WT_EXECUTELONGFUNCTION` flag to tell the thread pool, "This function will take a long time, so you may want to schedule work onto other threads more aggressively instead of sitting around waiting for this work item to finish."

But the problem is that the program didn't pass only the `WT_EXECUTELONGFUNCTION` flag. It did this:

```
BOOL WidgetMonitor::StartMonitoringChangeNotifications()
{
  WidgetNotificationContext context = ...;
  return QueueUserWorkItem(
    WidgetMonitor::WidgetNotificationCallback,
    context, WT_EXECUTELONGFUNCTION | WT_EXECUTEINPERSISTENTTHREAD);
}
```

Notice that they requested that the callback run in the persistent thread. But the documentation for that flag says

> This flag should be used only for short tasks…

So we have a contradiction. One flag says, "Run this callback in a persistent thread, and I promise I don't take a long time." The other flag says, "I'm going to take a long time."

The original thread pool was a bit too trusting and assumed that nobody would be so crazy as to explicitly declare their intent to break the rules.[1] I mean, if you're going to break the rules, you are probably going to be sneaky about it, right? It so happened that the way the thread pool code was written, the `WT_EXECUTEINPERSISTENTHREAD` flag takes precedence. The callback runs in the persistent thread, even though it runs long.[2]

And that's why the thread pool persistent thread grinds to a halt. The persistent thread is running the callback function, and the callback function is stuck. As a result, the persistent thread can't do anything else, and the thread pool makes no progress. This also explains why shutting down widget notifications caused everything to wake up: Shutting down widget notifications causes the `WidgetConfig::WidgetNotificationCallback` function to break out of its loop and finally exit. This releases the persistent thread to run more work items.

Okay, so we've diagnosed the problem. Next time, we'll speculate as to why the developers chose to combine contradictory threads and (perhaps more important) suggest a solution.

[1] Actually, what's happening is that the two flags are targeting different parts of the thread pool. The "persistent thread" flag is an instruction to the thread pool work item dispatcher, telling it to dispatch the work item to a persistent thread. The "long function" flag is an

instruction to the thread pool throughput manager to let it know that it should prefer to start a new thread instead of waiting for the work item to complete. Neither component on its own noticed anything wrong.

[2] If we had a time machine, we could go back and make this combination of flags cause `QueueUserWorkItem` fail with `ERROR_INVALID_PARAMETER`, but unfortunately that option is not available to us. We're stuck with the existing behavior of allowing the contradictory flags and ignoring the `WT_EXECUTELONGFUNCTION` flag.

Raymond Chen

Follow