Gotcha: Don’t forget to shut down your dispatcher queues

devblogs.microsoft.com/oldnewthing/20240222-11

February 22, 2024

Raymond Chen

If you need a `DispatcherQueue`, you can create a `DispatcherQueueController` and then read the `DispatcherQueue` property to obtain the associated `DispatcherQueue`.

But don’t throw away that `DispatcherQueueController`!

```
// C#
DispatcherQueueController controller =
    DispatcherQueueController.CreateOnDedicatedThread();
DispatcherQueue queue = controller.DispatcherQueue;

// C++/WinRT
DispatcherQueueController controller =
    DispatcherQueueController::CreateOnDedicatedThread();
DispatcherQueue queue = controller.DispatcherQueue();

// C++/CX
DispatcherQueueController^ controller =
    DispatcherQueueController::CreateOnDedicatedThread();
DispatcherQueue^ queue = controller->DispatcherQueue;

// C++/WRL
ComPtr<IDispatcherQueueControllerStatics> statics;
THROW_IF_FAILED(
    GetActivationFactory(HStringReference(
    RuntimeClass_Windows_System_DispatcherQueueController).Get(),
    &statics));

ComPtr<IDispatcherQueueController> controller;
THROW_IF_FAILED(
    statics->CreateOnDedicatedThread(&controller));

ComPtr<IDispatcherQueue> queue;
THROW_IF_FAILED(
    controller->get_DispatcherQueue(&queue));
```

The `DispatcherQueue` runs until it is shut down by a call to `DispatcherQueueController.ShutdownQueueAsync`. If you throw away the `DispatcherQueueController`, then you have no way of shutting down the `DispatcherQueue`, and it will run forever, long after the destruction of any objects which posted work to the `DispatcherQueue`. It just sits there patiently waiting for new work which will never arrive, until it finally gets the rug pulled out from under it at process termination.

If you create a dispatcher queue each time some action occurs, make sure to shut down that dispatcher queue when you are finished. Otherwise, you're going to leak dispatcher queues and eventually run out of window manager resources.

```
// C#

class MyThing
{
    public MyThing()
    {
        // Oops
        m_queue =
            DispatcherQueueController.
            CreateOnDedicatedThread().
            DispatcherQueue;
    }

    DispatcherQueue m_queue;
}

// C++/WinRT

class MyThing
{
public:
    MyThing()
    {
        // Oops
        m_queue =
            DispatcherQueueController::
            CreateOnDedicatedThread().
            DispatcherQueue();
    }

private:
    DispatcherQueue m_queue{ nullptr };
};

// C++/CX

class MyThing
{
public:
    MyThing()
    {
        // Oops
        m_queue =
            DispatcherQueueController::
            CreateOnDedicatedThread()->
            DispatcherQueue;
    }

private:
    DispatcherQueue^ m_queue;
};
```

```
// C++/WRL

class MyThing
{
public:
    MyThing()
    {
        // Oops
        ComPtr<IDispatcherQueueControllerStatics> statics;
        THROW_IF_FAILED(
            GetActivationFactory(HStringReference(
            RuntimeClass_Windows_System_DispatcherQueueController).Get(),
            &statics));

        ComPtr<IDispatcherQueueController> controller;
        THROW_IF_FAILED(
            statics->CreateOnDedicatedThread(&controller));

        THROW_IF_FAILED(
            controller->get_DispatcherQueue(&m_queue));
    }

private:
    ComPtr<IDispatcherQueue> m_queue;
};
```

In the above examples, we create a dispatcher queue and throw away the controller, leaving us no way to shut down the queue when the `MyThing` destructs.

You have to save the `DispatcherQueueController` so that you can call `ShutdownQueueAsync`[1] to clean up the dispatcher queue.

One way to do this is to have an explicit `ShutDownAsync` method that must be called prior to destruction.

```
// C#

class MyThing
{
    public MyThing()
    {
        m_controller =
            DispatcherQueueController.
            CreateOnDedicatedThread();
        m_queue = m_controller. DispatcherQueue;
    }

    public Task ShutDownAsync()
    {
        return m_controller.ShutdownQueueAsync();
    }

    DispatcherQueueController m_controller;
    DispatcherQueue m_queue;
}

// C++/WinRT

class MyThing
{
public:
    MyThing()
    {
        m_controller =
            DispatcherQueueController::
            CreateOnDedicatedThread();
        m_queue = m_controller.DispatcherQueue();
    }

    IAsyncAction ShutDownAsync()
    {
        return m_controller.ShutdownQueueAsync();
    }

private:
    DispatcherQueueController m_controller{ nullptr };
    DispatcherQueue m_queue{ nullptr };
};

// C++/CX

class MyThing
{
public:
    MyThing()
    {
```

```
        m_controller =
            DispatcherQueueController::
            CreateOnDedicatedThread();
        m_queue = m_controller->DispatcherQueue;
    }

    IAsyncAction ShutDownAsync()
    {
        return m_controller->ShutdownQueueAsync();
    }

private:
    DispatcherQueueController^ m_controller;
    DispatcherQueue^ m_queue;
};

// C++/WRL

class MyThing
{
public:
    MyThing()
    {
        ComPtr<IDispatcherQueueControllerStatics> statics;
        THROW_IF_FAILED(
            GetActivationFactory(HStringReference(
            RuntimeClass_Windows_System_DispatcherQueueController).Get(),
            &statics));

        THROW_IF_FAILED(
            statics->CreateOnDedicatedThread(&m_controller));

        THROW_IF_FAILED(
            controller->get_DispatcherQueue(&m_queue));
    }

    HRESULT ShutDownAsync(IAsyncAction** operation)
    {
        return m_controller->ShutdownQueueAsync(operation);
    }

private:
    ComPtr<IDispatcherQueueController> m_controller;
    ComPtr<IDispatcherQueue> m_queue;
};
```

This sounds great, but there's a catch. We'll look into it next time.

[1] The method name is spelled incorrectly. It should be `ShutDownQueueAsync`, because “shut down” (two words) is a verb, whereas “shutdown” (one word) is a noun.