Don't forget to include the message queue in your lock hierarchy

devblogs.microsoft.com/oldnewthing/20110418-00

April 18, 2011

Raymond Chen

In addition to the loader lock, the message queue is another resource that people often forget to incorporate into their lock hierarchy. If your code runs on a UI thread, then it implicitly owns the message queue whenever it is running, because messages cannot be dispatched to a thread until it calls a message-retrieval function such as `GetMessage` or `PeekMessage`. In other words, whenever a thread is not checking for a message, it cannot receive a message.

For example, consider the following code:

```
EnterCriticalSection(&g_cs);
for (int i = 0; i < 10; i++) {
  SendMessage(hwndLB, LB_ADDSTRING, 0, (LPARAM)strings[i]);
}
LeaveCriticalSection(&g_cs);
```

If `hwndLB` belongs to another thread, then you have a potential deadlock, because that thread might be waiting for your critical section.

```
case WM_DOESNTMATTERWHAT:
    EnterCriticalSection(&g_cs);
    ... doesn't matter what goes here ...
    LeaveCriticalSection(&g_cs);
    break;
```

If you happen to try to send the message while that other thread is waiting for the critical section, you will deadlock because you are waiting for that thread to finish whatever it’s doing so it can process the message you sent to it, but that thread is waiting for the critical section which you own.

Even if you promise that `hwndLB` belongs to your thread, the possibility of subclassing or window hooks means that you do not have full control over what happens when you try to send that message. A `WH_CALLWNDPROC` window hook may decide to communicate with another thread (for example, to log the message). Boom, what you thought was a simple message sent to a window on your thread turned into a cross-thread message.

There are many actions that generate message traffic that may not be obvious at first glance because they don't involve explicitly sending a message. Invoking a COM method from an STA thread on an object which belongs to another apartment requires the call to be marshaled to the thread that hosts the object. Tinkering with a window's scroll bars can result in the `WS_HSCROLL` or `WS_VSCROLL` style being added or removed, which in turn generates `WM_STYLECHANGING` and `WM_STYLECHANGED` messages. Obtaining the text from a window belonging to another thread in your process results in synchronous message traffic.

A good rule of thumb is basically to avoid anything that involves windows belonging to other threads while holding a critical section or other resource. And even windows which belong to your thread are suspect (due to hooks and subclassing).

Raymond Chen

Follow