When I intentionally create a stack overflow with SendMessage, why do I sometimes not get a stack overflow?

devblogs.microsoft.com/oldnewthing/20180620-00

June 20, 2018

Raymond Chen

Take our scratch program and make these changes:

```
LRESULT CALLBACK
WndProc(HWND hwnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uiMsg) {
    ...
    case WM_APP: return SendMessage(hwnd, WM_APP, 0, 0) + 1;
    }

    return DefWindowProc(hwnd, uiMsg, wParam, lParam);
}

int WINAPI WinMain(HINSTANCE hinst, HINSTANCE hinstPrev,
                   LPSTR lpCmdLine, int nShowCmd)
{
    ...
        ShowWindow(hwnd, nShowCmd);

        SendMessage(hwnd, WM_APP, 0, 0);
        MessageBox(hwnd, TEXT("Will this appear?"), TEXT("Hey"), MB_OK);

        while (GetMessage(&msg, NULL, 0, 0)) {
            ...
        }

        ...
}
```

This program enters infinite recursion upon receipt of the `WM_ APP` message by sending itself another copy of the same message. We add one to the returned value to make sure there's no tail call elimination.

After the program creates the window, it starts the death spiral by sending the first `WM_APP` message. And then after the deadly message, it displays a message.

The question: What happens? Do you see the message?

When you run the program, you might see the message, or you might crash. It depends on which resource runs out first: The user-mode stack or the kernel-mode stack.

Sending a message consumes some stack in user-mode, because you're calling the `SendMessage` function, and that does some work in user mode before calling into kernel mode to do the actual message sending.

Upon entry into kernel mode, some more stack is consumed to do the kernel-mode processing, like looking up which thread should be chosen to handle the message and dispatching any active window hooks. In this case, the destination window belongs to the same thread, so the kernel simulates a call into user mode and transitions back to user mode at a function whose job it is to call the window procedure.

Now we're back in user mode, and the helper function calls the window procedure, which allocates a stack frame, detects that the message is `WM_APP`, and calls `SendMessage`, and the process repeats.

When does this end?

On entry to kernel mode, the kernel checks how much kernel-mode stack is still available, and if there isn't enough to send a message, then the kernel says, "Whoa, I'm not going to be able to do this `SendMessage` thing, so I'm going to fail the call." The kernel-mode portion of the `SendMessage` function returns zero, and that causes `SendMessage` to return zero instead of sending the message.

On the other hand, there is no such preflight check in user mode. User mode just keeps running until it runs out of stack, at which point a `STATUS_STACK_OVERFLOW` exception is raised. Assuming nobody handles this exception, the program crashes.

It comes down to a race to see which limited resource runs out first. If the kernel-mode stack runs out first, then one of the `SendMessage` calls will return zero without actually sending the message, at which point the entire call stack unwinds, and then execution resumes at the call to `MessageBox`. On the other hand, if the user-mode stack runs out first, then you get a stack overflow exception.

Your program has some control over how much user-mode stack is consumed at each recursion, because some of that stack usage comes from your own window procedure. So if you keep your window procedure's stack usage low, then you're more likely to run out of kernel mode stack first.

But wait, there's still another wrinkle that introduces a level of unpredictability to the calculations. We'll dig deeper next time.

Raymond Chen

Follow