Why does SystemParametersInfo hang when I pass the SPIF_SENDCHANGE flag?

devblogs.microsoft.com/oldnewthing/20050310-00

March 10, 2005

Raymond Chen

If you pass the `SPIF_SENDCHANGE` flag to the `SystemParametersInfo` function, it will broadcast the `WM_SETTINGCHANGE` message with the wParam equal to the system parameter code you passed. For example, if you call

```
SystemParametersInfo(SPI_SETDOUBLECLICKTIME,
      500, 0, SPIF_UPDATEINIFILE | SPIF_SENDCHANGE);
```

then the system will broadcast the message

```
SendMessage(HWND_BROADCAST, WM_SETTINGCHANGE,
            SPI_SETDOUBLECLICKTIME, 0);
```

If there is a window that isn’t responding to messages, then this broadcast will hang until that unresponsive window finally resumes responding to messages or is killed.

If you’d rather not be victimed by unresponsive windows, you have a few options, but it also may affect your program’s expectations.

You could issue the `SystemParametersInfo` call on a background thread. Then your background thread is the one that blocks instead of your UI thread.

With this message, the background thread can notify the main thread when the broadcast finally completes, at which point your program now knows that all windows have received their notifications and are on board with the new setting.

You could issue the `SystemParametersInfo` call without the `SPIF_SENDCHANGE` flag, then manually broadcast the change via

```
DWORD dwResult;
SendMessageTimeout(HWND_BROADCAST, WM_SETTINGCHANGE,
            SPI_SETDOUBLECLICKTIME, 0,
            SMTO_ABORTIFHUNG | SMTO_NOTIMEOUTIFNOTHUNG,
            5000, &dwResult);
```

This does mean that unresponsive windows will not receive the notification that a system parameter has changed. This is acceptable if you decide that your change in settings was minor enough that a program missing the notification is no big deal. In other words, when the unresponsive program finally wakes up, it will not know that the setting has changed since it missed the notification.

You can combine the above two methods: Use a background thread and send the message with a timeout.

Perhaps the best technique is to use the `SendNotifyMessage` function. As we learned earlier, the `SendNotifyMessage` function is like `SendMessage` except that it doesn't wait for a response. This lets your program get back work while not messing up programs that were momentarily unresponsive when you decided to broadcast the notification.

```
SendNotifyMessage(HWND_BROADCAST, WM_SETTINGCHANGE,
            SPI_SETDOUBLECLICKTIME, 0);
```

The downside is that you don't know when all windows have finally received and processed the notification. All you know is that someday, they will eventually find out. Usually you don't care about this aspect of the broadcast, so this lack of information is not an impediment.

Raymond Chen

Follow