How can I check whether the user has disconnected from the session?

devblogs.microsoft.com/oldnewthing/20201202-00

December 2, 2020

Raymond Chen

If another user signs onto a Windows system without the previous user signing out, then the previous user goes into a state known as Disconnected. Programs in the previous user's session continue to run, but what if you want your program to check whether the session has been disconnected?

You can subscribe to session state changes by using the `WTSRegisterSessionNotification` function, passing a window handle to receive the `WM_WTSSESSION_CHANGE` message. The `wParam` of the message describes the what happened, and you can use that value to decide to, say, pause various operations while the session is disconnected.

If you would rather poll, you can call `WTSQuerySessionInformation` and ask for the `WTSConnectState`.

Or you can go a completely different route and ask whether the current desktop is the target of user input.

```
bool IsOnInputDesktop()
{
 auto desktop = GetThreadDesktop(GetCurrentThreadId());
 if (!desktop) return false;

 BOOL input = FALSE;
 if (!GetUserObjectInformation(desktop, UOI_IO,
         &input, sizeof(input), nullptr)) return false;

 return !!input;
}
```

Note that the handle returned by `GetThreadDesktop` does not need to be (and shouldn't be) closed.

Note that this is not the same as disconnectedness. For example, if the user has hit `Ctrl`+`Alt`+`Del`, then there is an input desktop switch to the secure desktop, but the session is not disconnected.

If you are a batch file or a PowerShell script, then your options are more limited. There's a little program with the presumptuous name `query` that displays information about Terminal Services. In particular you can say `query session` to get a list of sessions, who is signed into each session, and whether the session is connected.

There is a PowerShell module called PSTerminalServices that parses the output of `query session` into PowerShell objects.

A customer was hoping for an environment variable that provided this information so it could be consumed from their batch file. Even without doing a `SET` in a command prompt, you should be able to determine that no such variable exists: Environment variables are captured at process creation and are private to the process. The only way an environment variable can change is if the process changes it.

The customer didn't realize that environment variables are local to the process, but if they thought about it, they may have realized that they were relying on it without realizing it: If environment variables could be globally modified, then their own batch files would stop working! When their batch file performs a `SET MYVAR=42`, they don't expect the variable `MYVAR` to be set globally. They want the batch file to have a variable named `MYVAR`, different from the `MYVAR` variable in an unrelated command prompt.

Raymond Chen

Follow