Do not access the disk in your IContextMenu handler, no really, don't do it

devblogs.microsoft.com/oldnewthing/20111003-00

October 3, 2011

Raymond Chen

We saw some time ago that the number one cause of crashes in Explorer is malware.

It so happens that the number one cause of hangs in Explorer is disk access from context menu handlers (a special case of the more general principle, you can't open the file until the user tells you to open it).

That's why I was amused by Memet's claim that "would hit the disk" is not acceptable for me. The feedback I see from customers, either directly from large multinational corporations with 500ms ping times or indirectly from individual users who collectively click *Send Report* millions of times a day, is that "would hit the disk" ruins a lot of people's days. It may not be acceptable to you, but millions of other people would beg to disagree.

The Windows team tries very hard to identify unwanted disk accesses in Explorer and get rid of them. We don't get them all, but at least we try. But if the unwanted disk access is coming from a third-party add-on, there isn't much that can be done aside from saying, "Don't do that" and hoping the vendor listens.

Every so often, a vendor will come back and ask for advice on avoiding disk access in their context menu handler. There's a lot of information packed into that data object that contains information gathered from when the disk was accessed originally. You can just retrieve that cached data instead of going off and hitting the disk again to recalculate it.

I'm going to use a boring console application and the clipboard rather than building a full `IContextMenu`, since the purpose here is to show how to get data from a data object without hitting the disk and not to delve into the details of `IContextMenu` implementation.

```
#define UNICODE
#define _UNICODE
#include <windows.h>
#include <ole2.h>
#include <shlobj.h>
#include <propkey.h>
#include <tchar.h>
void ProcessDataObject(IDataObject *pdto)
{
 ... to be written ...
}
int __cdecl _tmain(int argc, PTSTR *argv)
{
 if (SUCCEEDED(OleInitialize(NULL))) {
  IDataObject *pdto;
  if (SUCCEEDED(OleGetClipboard(&pdto))) {
   ProcessDataObject(pdto);
   pdto->Release();
  }
  OleUninitialize();
 }
}
```

Okay, let's say that we want to check that all the items on the clipboard are files and not directories. The `HDROP` way of doing this would be to get the path to each of the items in the data object, then call `GetFileAttributes` on each one to see if any of them has the `FILE_ATTRIBUTE_DIRECTORY` flag set. But this hits the disk, which makes baby context menu host sad. Fortunately, the `IShellItemArray` interface provides an easy way to check whether any or all the items in a data object have a particular attribute.

```
void ProcessDataObject(IDataObject *pdto)
{
 IShellItemArray *psia;
 HRESULT hr;
 hr = SHCreateShellItemArrayFromDataObject(pdto,
                                          IID_PPV_ARGS(&psia));
 if (SUCCEEDED(hr)) {
  SFGAOF sfgaoResult;
  hr = psia->GetAttributes(SIATTRIBFLAGS_OR, SFGAO_FOLDER,
                                                 &sfgaoResult);
  if (hr == S_OK) {
   _tprintf(TEXT("Contains a folder\n"));
  } else if (hr == S_FALSE) {
   _tprintf(TEXT("Contains no folders\n"));
  }
  psia->Release();
 }
}
```

In this case, we want to see if any item (`SIATTRIBFLAGS_OR`) in the data object has the `SFGAO_FOLDER` attribute. The `IShellItemArray::GetAttributes` method returns `S_OK` if all of the attributes you requested are present in the result. Since we asked for only one attribute, and since we asked for the result to be the logical *or* of the individual attributes, this means that it returns `S_OK` if any item is a folder.

Okay, fine, but what if the thing you want to know is not expressible as a `SFGAO` flag? Well, you can dig into each of the individual items. For example, suppose we want to see the size of each item.

```
#include <strsafe.h>
void ProcessDroppedObject(IDataObject *pdto)
{
 IShellItemArray *psia;
 HRESULT hr;
 hr = SHCreateShellItemArrayFromDataObject(pdto,
                                          IID_PPV_ARGS(&psia));
 if (SUCCEEDED(hr)) {
  IEnumShellItems *pesi;
  hr = psia->EnumItems(&pesi);
  if (SUCCEEDED(hr)) {
   IShellItem *psi;
   while (pesi->Next(1, &psi, NULL) == S_OK) {
    IShellItem2 *psi2;
    hr = psi->QueryInterface(IID_PPV_ARGS(&psi2));
    if (SUCCEEDED(hr)) {
     ULONGLONG ullSize;
     hr = psi2->GetUInt64(PKEY_Size, &ullSize);
     if (SUCCEEDED(hr)) {
      _tprintf(TEXT("Item size is %I64u\n"), ullSize);
     }
     psi2->Release();
    }
    psi->Release();
   }
  }
  psia->Release();
 }
}
```

I went for `IEnumShellItems` here, even though a `for` loop with `IShellItemArray::GetCount` and `IShellItemArray::GetItemAt` would have worked, too.

File system items in data objects cache a bunch of useful pieces of information, such as the last-modified time, file creation time, last-access time, the file size, the file attributes, and the file name (both long and short). Of course, all of these properties are subject to file system support. the shell just takes what's in the `WIN32_FIND_DATA` ; if the values are incorrect (for example, if last-access time tracking is disabled), then the shell is going to cache the incorrect value. But don't say, "Well, if the cache is no good, then I won't use it; I'll just go hit the disk", because if you hit the disk, the file system is going to give you the same incorrect value anyway!

If you just want to order the combo platter, you can ask for `PKEY_FindData`, and out will come a `WIN32_FIND_DATA`. This might be the easiest way to convert your old-style context menu that hits the disk into a new-style context menu that doesn't hit the disk: Take your calls to `GetFileAttributes` and `FindFirstFile` and convert them into calls into the property system, asking for `PKEY_FileAttributes` or `PKEY_FindData`.

Okay, that's the convenient modern way to get information that has been cached in the data object provided by the shell. What if you're an old-school programmer? Then you get to roll up your sleeves and get your hands dirty with the `CFSTR_SHELLIDLIST` clipboard format. (And if your target is Windows XP or earlier, you have to do it this way since the `IShellItemArray` interface was not introduced until Windows Vista.) In fact, the `CFSTR_SHELLIDLIST` clipboard format will get your hands so dirty, I'm writing a helper class to manage it.

First, go back and familiarize yourself with the `CIDA` structure.

```
// these should look familiar
#define HIDA_GetPIDLFolder(pida) (LPCITEMIDLIST)(((LPBYTE)pida)+(pida)->aoffset[0])
#define HIDA_GetPIDLItem(pida, i) (LPCITEMIDLIST)(((LPBYTE)pida)+(pida)-
>aoffset[i+1])
void ProcessDataObject(IDataObject *pdto)
{
 FORMATETC fmte = {
    (CLIPFORMAT)RegisterClipboardFormat(CFSTR_SHELLIDLIST),
    NULL, DVASPECT_CONTENT, -1, TYMED_HGLOBAL };
 STGMEDIUM stm = { 0 }; // defend against buggy data object
 HRESULT hr = pdto->GetData(&fmte, &stm);
 if (SUCCEEDED(hr) && stm.hGlobal != NULL) {
  LPIDA pida = (LPIDA)GlobalLock(stm.hGlobal);
  if (pida != NULL) { // defend against buggy data object
   IShellFolder *psfRoot;
   hr = SHBindToObject(NULL, HIDA_GetPIDLFolder(pida), NULL,
                       IID_PPV_ARGS(&psfRoot));
   if (SUCCEEDED(hr)) {
    for (UINT i = 0; i < pida->cidl; i++) {
     IShellFolder2 *psf2;
     PCUITEMID_CHILD pidl;
     hr = SHBindToFolderIDListParent(psfRoot,
                HIDA_GetPIDLItem(pida, i),
                IID_PPV_ARGS(&psf2), &pidl);
     if (SUCCEEDED(hr)) {
      VARIANT vt;
      if (SUCCEEDED(psf2->GetDetailsEx(pidl, &PKEY_Size, &vt))) {
       if (SUCCEEDED(VariantChangeType(&vt, &vt, 0, VT_UI8))) {
         _tprintf(TEXT("Item size is %I64u\n"), vt.ullVal);
       }
       VariantClear(&vt);
      }
      psf2->Release();
     }
    }
    psfRoot->Release();
   }
   GlobalUnlock(stm.hGlobal);
  }
  ReleaseStgMedium(&stm);
 }
}
```

I warned you it was going to be ugly.

First, we retrieve the `CFSTR_SHELLIDLIST` clipboard format from the data object. This format takes the form of an `HGLOBAL` , which needs to be `GlobalLock` 'd like all `HGLOBAL` s returned by `IDataObject::GetData` . You may notice two defensive measures here. First, there is a defense against data objects which return success when they actually failed. To detect this case, we zero out the `STGMEDIUM` and make sure they returned something non-`NULL` in it. The second defensive measure is against data objects which put an invalid

`HGLOBAL` in the `STGMEDIUM`. One of the nice things about doing things the `IShellItem-Array` way is that the shell default implementation of `IShellItemArray` has all these defensive measures built-in so you don't have to write them yourself.

Anyway, once we get the `CIDA`, we bind to the folder portion, walk through the items, and get the size of each item in order to print it. Same story, different words.

Exercise: Why did we need a separate defensive measure for data objects which returned success but left garbage in the `STGMEDIUM`? Why doesn't the `GlobalLock` test cover that case, too?

Raymond Chen

Follow