How do I find out which process has a file open?

devblogs.microsoft.com/oldnewthing/20120217-00

February 17, 2012

Raymond Chen

Classically, there was no way to find out which process has a file open. A file object has a reference count, and when the reference count drops to zero, the file is closed. But there's nobody keeping track of which processes own how many references. (And that's ignoring the case that the reference is not coming from a process in the first place; maybe it's coming from a kernel driver, or maybe it came from a process that no longer exists but whose reference is being kept alive by a kernel driver that captured the object reference.)

This falls into the category of not keeping track of information you don't need. The file system doesn't care who has the reference to the file object. Its job is to close the file when the last reference goes away.

You do the same thing with your COM object reference counts. All you care about is whether your reference count has reached zero (at which point it's time to destroy the object). If you later discover an object leak in your process, you don't have a magic query "Show me all the people who called `AddRef` on my object" because you never kept track of all the people who called `AddRef` on your object. Or even, "Here's an object I want to destroy. Show me all the people who called `AddRef` on it so I can destroy them and get them to call `Release`."

At least that was the story under the classical model.

Enter the Restart Manager.

The official goal of the Restart Manager is to help make it possible to shut down and restart applications which are using a file you want to update. In order to do that, it needs to keep track of which processes are holding references to which files. And it's that database that is of use here. (Why is the kernel keeping track of which processes have a file open? Because it's the converse of the principle of not keeping track of information you don't need: Now it needs the information!)

Here's a simple program which takes a file name on the command line and shows which processes have the file open.

```
#include <windows.h>
#include <RestartManager.h>
#include <stdio.h>
int __cdecl wmain(int argc, WCHAR **argv)
{
 DWORD dwSession;
 WCHAR szSessionKey[CCH_RM_SESSION_KEY+1] = { 0 };
 DWORD dwError = RmStartSession(&dwSession, 0, szSessionKey);
 wprintf(L"RmStartSession returned %d\n", dwError);
 if (dwError == ERROR_SUCCESS) {
   PCWSTR pszFile = argv[1];
   dwError = RmRegisterResources(dwSession, 1, &pszFile,
                                 0, NULL, 0, NULL);
   wprintf(L"RmRegisterResources(%ls) returned %d\n",
           pszFile, dwError);
  if (dwError == ERROR_SUCCESS) {
   DWORD dwReason;
   UINT i;
   UINT nProcInfoNeeded;
   UINT nProcInfo = 10;
   RM_PROCESS_INFO rgpi[10];
   dwError = RmGetList(dwSession, &nProcInfoNeeded,
                       &nProcInfo, rgpi, &dwReason);
   wprintf(L"RmGetList returned %d\n", dwError);
   if (dwError == ERROR_SUCCESS) {
    wprintf(L"RmGetList returned %d infos (%d needed)\n",
            nProcInfo, nProcInfoNeeded);
    for (i = 0; i < nProcInfo; i++) {
     wprintf(L"%d.ApplicationType = %d\n", i,
                              rgpi[i].ApplicationType);
     wprintf(L"%d.strAppName = %ls\n", i,
                              rgpi[i].strAppName);
     wprintf(L"%d.Process.dwProcessId = %d\n", i,
                              rgpi[i].Process.dwProcessId);
     HANDLE hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION,
                                   FALSE, rgpi[i].Process.dwProcessId);
     if (hProcess) {
      FILETIME ftCreate, ftExit, ftKernel, ftUser;
      if (GetProcessTimes(hProcess, &ftCreate, &ftExit,
                          &ftKernel, &ftUser) &&
          CompareFileTime(&rgpi[i].Process.ProcessStartTime,
                          &ftCreate) == 0) {
       WCHAR sz[MAX_PATH];
       DWORD cch = MAX_PATH;
       if (QueryFullProcessImageNameW(hProcess, 0, sz, &cch) &&
           cch <= MAX_PATH) {
        wprintf(L"  = %ls\n", sz);
       }
      }
      CloseHandle(hProcess);
     }
    }
```

```
   }
  }
  RmEndSession(dwSession);
 }
 return 0;
}
```

The first thing we do is call, no wait, even before we call the `RmStartSession` function, we have the line

```
WCHAR szSessionKey[CCH_RM_SESSION_KEY+1] = { 0 };
```

That one line of code addresses two bugs!

First is a documentation bug. The documentation for the `RmStartSession` function doesn't specify how large a buffer you need to pass for the session key. The answer is `CCH_RM_SESSION_KEY+1`.

Second is a code bug. The `RmStartSession` function doesn't properly null-terminate the session key, even though the function is documented as returning a null-terminated string. To work around this bug, we pre-fill the buffer with null characters so that whatever ends gets written will have a null terminator (namely, one of the null characters we placed ahead of time).

Okay, so that's out of the way. The basic algorithm is simple:

1. Create a Restart Manager session.
2. Add a file resource to the session.
3. Ask for a list of all processes affected by that resource.
4. Print some information about each process.
5. Close the session.

We already mentioned that you create the session by calling `RmStartSession`. Next, we add a single file resource to the session by calling `RmRegisterResources`.

Now the fun begins. Getting the list of affected processes is normally a two-step affair. First, you ask for the number of affected processes (by passing `0` as the `nProcInfo`), then allocate some memory and call a second time to get the data. But this is just a sample program, so I've hard-coded a limit of ten processes. If more than ten processes are affected, I just give up. (You can see this if you ask for all the processes that have open handles to `kernel32.dll`.)

The other tricky part is mapping the `RM_PROCESS_INFO` to an actual process. Since process IDs can be recycled, the `RM_PROCESS_INFO` structure identifies a process by the combination of the process ID and the process creation time. That combination is unique because two processes cannot have the same ID at the same time. We open the handle to the

process via its ID, then confirm that the start times match. (If not, then the ID refers to a process that exited during the time we obtained the list and the time we actually looked at it.) Assuming it all matches, we get the image name and print it.

And that's all there is to enumerating all the processes that have a particular file open. Of course, a more expressive interface for managing files in use is `IFileIsInUse`, which I mentioned some time ago. That interface not only tells you the application that has the file open (in a friendlier format than just an executable path), you can also use it to switch to the application and even ask it to close the file. (Windows 7 first tries `IFileIsInUse`, and if that fails, then it goes to the Restart Manager.)

Raymond Chen

Follow