The case of the unexpected ERROR_ACCESS_DENIED when calling MapViewOfFile

devblogs.microsoft.com/oldnewthing/20161205-00

December 5, 2016

Raymond Chen

A customer was trying to figure out how to use shared memory, but even their simplest program couldn't work. The customer shared their code and asked for help.

The first process creates a named file mapping object backed by the page file. The second process opens the file mapping object by name, and then maps a view of that file mapping object. But the attempt to map the view always fails with `ERROR_ACCESS_DENIED`. The file mapping object was created by the first process as read/write, and it was opened by the second process as read/write. The two processes are running in the same session as the same user. And yet, the second process can't get access. What's wrong?

To simplify presentation, error checking has been deleted. Instead, we will describe what happened with comments.

```
// code in italics is wrong
//
// Program 1

#include <windows.h>

int main(int, char**)
{
 // This succeeds with a non-null handle.
 HANDLE fileMapping = CreateFileMapping(
    INVALID_HANDLE_VALUE, // backed by page file
    nullptr,              // default security
    PAGE_READWRITE,       // read-write access
    0,                    // high part of size
    65536,                // low part of size
    L"Local\\FileMappingTest"); // name

 // This succeeds with a non-null pointer.
 void* view = MapViewOfFile(
    fileMapping,
    FILE_MAP_READ | FILE_MAP_WRITE, // desired access
    0, 0,                   // file offset zero
    0);                     // map the whole thing

  Sleep(5000); // pause to let user run second process

  UnmapViewOfFile(view);
  CloseHandle(fileMapping);

  return 0;
}

// Program 2
#include <windows.h>

int main(int, char**)
{
 // This succeeds with a non-null handle.
 HANDLE fileMapping = OpenFileMapping(
    PAGE_READWRITE,       // read-write access
    FALSE,                // don't inherit this handle
    L"Local\\FileMappingTest"); // name

 // This fails with a null pointer.
 // GetLastError() returns ERROR_ACCESS_DENIED.
 void* view = MapViewOfFile(
    fileMapping,
    FILE_MAP_READ | FILE_MAP_WRITE, // desired access
    0, 0,                   // file offset zero
    0);                     // map the whole thing

  UnmapViewOfFile(view);
```

```
  CloseHandle(fileMapping);

  return 0;
}
```

The customer added that the second process successfully opened the file mapping object, so presumably the handle does have read/write access. Otherwise, the `OpenFileMapping` would have failed with `ERROR_ACCESS_DENIED` right away, rather than waiting for the `MapViewOfFile`.

Study these programs and see if you can find the problem.

(Time passes.)

The problem is that the first parameter to `OpenFileMapping` is not supposed to be a `PAGE_*` value. It's supposed to be a `FILE_MAP_*` value. This is easily overlooked because you are tempted to just do a copy/paste of the `CreateFileMapping` call's parameters, and just delete the parameters related specifically to creation, like file size and security descriptor.

However, it is a common[1] pattern that `Create` functions return a handle with full access and do not have an explicit access mask parameter, whereas `Open` functions accept an access mask parameter that controls what level of access the returned handle has.

The numeric value of `PAGE_READWRITE` is 4, which happens to match the numeric value of `FILE_MAP_READ`. Therefore, the second program successfully opened the file mapping for read, but when it tried to map it for read and write, it got `ERROR_ACCESS_DENIED` because it's trying to obtain a mapping for writing, even though the mapping was opened only for read.

This is one of the nasty pitfalls of using plain old integers for flags. There's no type safety: Integers look the same.

[1] Note that the pattern is common but not not universal. The most notable exception is `CreateFile`, which takes an explicit access mask. But if you think about it some more, `CreateFile` is an open-like function, because if the file already exists, `CreateFile` opens a handle to it, and it uses the requested access mask to evaluate whether your attempt to open that handle will succeed.

Raymond Chen

Follow