Programmatically controlling which handles are inherited by new processes in Win32

devblogs.microsoft.com/oldnewthing/20111216-00

December 16, 2011

Raymond Chen

In unix, file descriptors are inherited by child processes by default. This wasn't so much an active decision as it was a consequence of the fork/exec model. To exclude a file descriptor from being inherited by children, you set the `FD_CLOEXEC` flag on the file descriptor.

Win32 sort of works like that, but backwards, and maybe a little upside-down. And in high heels.

In Win32, handles default to *not inherited*. Ways to make a handle inherited during `CreateProcess` have grown during the evolution of Win32.

As far as I can tell, back in the old days, inheritability of handles was established at handle creation time. For most handle creation functions, you do this by passing a `SECURITY_ATTRIBUTES` structure with `bInheritHandle` set to `TRUE`. Functions which created handles from existing objects don't have a `SECURITY_ATTRIBUTES` parameter, so they instead have an explicit `bInheritHandle` parameter. (For examples, see `OpenEvent` and `DuplicateHandle`.)

But just marking a handle as inheritable isn't good enough to get it inherited. You also have to pass `TRUE` as the `bInheritHandles` parameter to `CreateProcess`. A handle will be inherited only if if the `bInheritHandles` parameter is `TRUE` and the handle is marked as inheritable. Miss either of those steps, and you don't get your inheritance. (To make sure you get your inheritance IRL, be nice to your grandmother.)

In Windows 2000, Win32 gained the ability to alter the inheritability of a handle after it is created. The `SetHandleInformation` function lets you turn the `HANDLE_FLAG_INHERIT` flag on and off on a handle.

But all this inheritability fiddling still had a fatal flaw: What if two threads within the same process both call `CreateProcess` but disagree on which handles they want to be inherited? For example, suppose you have a function `CreateProcessWithSharedMemory` whose job it is to launch a process, passing it a custom-made shared memory block. Suppose two threads run this function simultaneously.

A	B
CreateFileMapping(inheritable=TRUE)	CreateFileMapping(inheritable=TRUE)
returns handle H1	returns handle H2
CreateProcess(“A”, bInheritHandles=TRUE)	CreateProcess(“B”, bInheritHandles=TRUE)
CloseHandle(H1)	CloseHandle(H2)

What just happened? Since inheritability is a property of the handle, processes A and B inherited *both* handles H1 and H2, even though what we wanted was for process A to inherit handle H1 and for process B to inherit handle H2.

For a long time, the answer to this problem was the unsatisfactory “You’ll just have to serialize your calls to `CreateProcessWithSharedMemory` so that thread B won’t accidentally cause a handle from thread A to be inherited by process B.” Actually, the answer was even worse. You had to serialize all functions that created inheritable handles from the time the handle was created, through the call to `CreateProcess`, and waiting until after all those inheritable handles were made no longer inheritable.

This was a serious problem since who knows what other parts of your program are going to call `CreateProcess` with `bInheritHandles` set to `TRUE`? Sure you can control the calls that your own code made, but what about calls from plug-ins or other unknown components? (This is another case of kernel-colored glasses.)

Windows Vista addresses this problem by allowing you to pass an explicit list of handles you want the `bInheritHandles` parameter to apply to. (If you pass an explicit list, then you must pass `TRUE` for `bInheritHandles`.) And as before, for a handle to be inherited, it must be also be marked as inheritable.

Passing the list of handles you want to inherit is a multi-step affair. Let’s walk through it:

```
BOOL CreateProcessWithExplicitHandles(
  __in_opt     LPCTSTR lpApplicationName,
  __inout_opt  LPTSTR lpCommandLine,
  __in_opt     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  __in_opt     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in         BOOL bInheritHandles,
  __in         DWORD dwCreationFlags,
  __in_opt     LPVOID lpEnvironment,
  __in_opt     LPCTSTR lpCurrentDirectory,
  __in         LPSTARTUPINFO lpStartupInfo,
  __out        LPPROCESS_INFORMATION lpProcessInformation,
    // here is the new stuff
  __in         DWORD cHandlesToInherit,
  __in_ecount(cHandlesToInherit) HANDLE *rgHandlesToInherit)
{
 BOOL fSuccess;
 BOOL fInitialized = FALSE;
 SIZE_T size = 0;
 LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList = NULL;
 fSuccess = cHandlesToInherit < 0xFFFFFFFF / sizeof(HANDLE) &&
            lpStartupInfo->cb == sizeof(*lpStartupInfo);
 if (!fSuccess) {
  SetLastError(ERROR_INVALID_PARAMETER);
 }
 if (fSuccess) {
  fSuccess = InitializeProcThreadAttributeList(NULL, 1, 0, &size) ||
             GetLastError() == ERROR_INSUFFICIENT_BUFFER;
 }
 if (fSuccess) {
  lpAttributeList = reinterpret_cast<LPPROC_THREAD_ATTRIBUTE_LIST>
                                (HeapAlloc(GetProcessHeap(), 0, size));
  fSuccess = lpAttributeList != NULL;
 }
 if (fSuccess) {
  fSuccess = InitializeProcThreadAttributeList(lpAttributeList,
                    1, 0, &size);
 }
 if (fSuccess) {
  fInitialized = TRUE;
  fSuccess = UpdateProcThreadAttribute(lpAttributeList,
                    0, PROC_THREAD_ATTRIBUTE_HANDLE_LIST,
                    rgHandlesToInherit,
                    cHandlesToInherit * sizeof(HANDLE), NULL, NULL);
 }
 if (fSuccess) {
  STARTUPINFOEX info;
  ZeroMemory(&info, sizeof(info));
  info.StartupInfo = *lpStartupInfo;
  info.StartupInfo.cb = sizeof(info);
  info.lpAttributeList = lpAttributeList;
  fSuccess = CreateProcess(lpApplicationName,
                           lpCommandLine,
                           lpProcessAttributes,
                           lpThreadAttributes,
                           bInheritHandles,
                           dwCreationFlags | EXTENDED_STARTUPINFO_PRESENT,
                           lpEnvironment,
                           lpCurrentDirectory,
                           &info.StartupInfo,
                           lpProcessInformation);
 }
 if (fInitialized) DeleteProcThreadAttributeList(lpAttributeList);
 if (lpAttributeList) HeapFree(GetProcessHeap(), 0, lpAttributeList);
 return fSuccess;
}
```

After some initial sanity checks, we start doing real work.

Initializing a `PROC_THREAD_ATTRIBUTE_LIST` is a two-step affair. First you call `InitializeProcThreadAttributeList` with a `NULL` attribute list in order to determine how much memory you need to allocate for a one-entry attribute list. After allocating the memory, you call `InitializeProcThreadAttributeList` a second time to do the actual initialization.

After creating the attribute list, you set the one entry by calling `UpdateProcThread-AttributeList`.

And then it's off to the races. Put that attribute list in a `STARTUPINFOEX` structure, set the `EXTENDED_STARTUPINFO_PRESENT` flag, and hand everything off to `CreateProcess`.

Raymond Chen

Follow