What possible use are those extra bits in kernel handles? Part 3: New object types

devblogs.microsoft.com/oldnewthing/20080829-00

August 29, 2008

Raymond Chen

Last time, we saw how those extra bits can be used to multiplex `HANDLE` with other values. That was a specific case of a more general scenario: Expanding the handle namespace to include things that aren't handles. (You can also view today's example as a generalization of the sentinel value problem, where we need to generate an arbitrary number of sentinel values dynamically. Actually, multiplexing `HANDLE` with `HRESULT` is also just another special case: We expanded the handle namespace to include error codes too.)

As I noted in the base article, the people who are most interested in this sort of thing are people writing low-level class libraries or wrapping kernel objects inside a larger framework.

Suppose, for example, you are writing a library that manipulates kernel objects, but you also have private types of objects (say, a handle to a remote computer) that you also want this library to be able to manipulate. One way of doing this is to wrap everything inside some base class that virtualizes away what type of handle you're working on:

```
class ExtendedHandle {
public:
  virtual ExtendedHandleType GetType() = 0;
  virtual ~ExtendedHandle() { }
};


class KernelExtendedHandle : public ExtendedHandle {
public:
  static KernelExtendedHandle *Create(…);
  ExtendedHandleType GetType() { return KernelHandle; }
  HANDLE GetHandle() { return Handle; }
private:
  KernelExtendedHandle(…);
  HANDLE Handle;
};


class RemoteComputerExtendedHandle : public ExtendedHandle {
public:
  static RemoteComputerExtendedHandle *Create(…);
  ExtendedHandleType GetType() { return RemoteComputer; }
  LPCTSTR GetComputerName() { … }
  … other remote computer methods …
private:
  RemoteComputerExtendedHandle(…);
  … stuff necessary for tracking remote computers …
};
```

Now, your library spends only 1% of its time manipulating these private object types; most of the time, it's dealing with regular kernel handles. In other words, 99% of your objects are of type `KernelExtendedHandle`. What used to be just a `HANDLE` (4 bytes) is now a `EHANDLE` that in turn points to an 8-byte structure (4 bytes for the vtable and 4 bytes for the `HANDLE`). Your memory requirements have tripled *and* you added another level of indirection (costing you locality), just for that 1% case.

But you can do better if you have those extra bits to play with. Since 99% of the objects you're wrapping are just plain old kernel handles, you can say that if the bottom bit is clear, then it's just a kernel handle, and if the bottom bit is set, then the upper bits tell us what we are really operating on.

```
typedef void *EHANDLE;


BOOL IsKernelHandle(EHANDLE Handle)
{
  return (!((INT_PTR)Handle & 1));
}


EHANDLE CreateHandleFromKernelHandle(HANDLE Handle)
{
  // if this assertion fires, then somebody tried to
  // use an invalid kernel handle!
  assert((!((INT_PTR)Handle & 1));
  return (EHANDLE)Handle;
}


EHANDLE CreateHandleFromExtendedHandle(ExtendedHandle Handle)
{
  // if this assertion fires, then somebody allocated
  // a misaligned ExtendedHandle!
  assert(!((INT_PTR)Handle & 1));
  return (EHANDLE)((INT_PTR)Handle | 1));
}


ExtendedHandle *GetExtendedHandle(EHANDLE Handle)
{
  assert(!IsKernelHandle(Handle));
  return (ExtendedHandle*)((INT_PTR)Handle & ~1);
}


ExtendedHandleType GetType(EHANDLE Handle)
{
 if (IsKernelHandle(Handle)) {
   return KernelHandleType;
 } else {
   return GetExtendedHandle(Handle)->GetType();
 }
}


void ECloseHandle(EHANDLE Handle)
{
  if (IsKernelHandle(Handle))
  {
    CloseHandle(GetKernelHandle(Handle));
  } else {
   delete GetExtendedHandle(Handle);
  }
}
```

Now the cost of a wrapped kernel handle is just 4 bytes: 4 bytes for the `EHANDLE`, which also doubles as the actual kernel handle. The cost of wrapped pseudo-handles is the same as before (4 bytes for the `EHANDLE`, plus the size of the corresponding `XxxExtendedHandle` class). We used the trick from last time in order to pack 33 bits into only 32 bits: Since we know that the bottom bit of both kernel `HANDLE`s and `ExtendedHandle` pointers are zero, we can use it as a discriminator.

If you are not confident that your `ExtendedHandle` classes all use aligned pointers, you can use a different packing mechanism by using your own handle translation table:

```
ExtendedHandle *ExtendedHandleTable;


ExtendedHandle *GetExtendedHandle(EHANDLE Handle)
{
  assert(!IsKernelHandle(Handle));
  return ExtendedHandleTable[(INT_PTR)Handle >> 1];
}
```

Using this technique, the upper 31 bits of an `EHANDLE` which refers to an `ExtendedHandle` is an index into a privately-managed table of `ExtendedHandle` objects.

This *secondary handle table* technique is entirely analogous to the trick which one Posix library uses to distinguish "real" process IDs from "virtual" process IDs, except that they are relying on undocumented behavior because the bottom bits of process IDs are not guaranteed to be zero!

So there you have it, three scenarios where you can take advantage of knowing that the bottom bits of kernel handles are always zero.

Raymond Chen

Follow