Developing the method for taking advantage of the fact that the OVERLAPPED associated with asynchronous I/O is passed by address

devblogs.microsoft.com/oldnewthing/20101220-01

December 20, 2010

Raymond Chen

You can take advantage of the fact that the `OVERLAPPED` associated with asynchronous I/O is passed by address, but there was some confusion about how this technique could "work" when kernel mode has no idea that you are playing this trick.

Whether kernel mode is in on the trick is immaterial since it is not part of the trick.

Let's start with a version of the code which does not take advantage of the `OVERLAPPED` structure address in the way described in the article. This is a technique I found in a book on advanced Windows programming:

```
#define MAX_OVERLAPPED 10 // let's do 10 I/O's at a time
// data to associate with each OVERLAPPED
struct OTHERDATA { ... };
OVERLAPPED MasterOverlapped[MAX_OVERLAPPED];
OTHERDATA OtherData[MAX_OVERLAPPED];
OTHERDATA* FindOtherDataFromOverlapped(OVERLAPPED *lpOverlapped)
{
 ptrdiff_t index = lpOverlapped - MasterOverlapped;
 return &OtherData[index];
}
// I/O is issued via
// ReadFileEx(hFile, lpBuffer, nNumberOfBytesToRead,
//             &MasterOverlapped[i], CompletionRoutine);
void CALLBACK CompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    LPOVERLAPPED lpOverlapped)
{
 OTHERDATA *lpOtherData =
                      FindOtherDataFromOverlapped(lpOverlapped);
 ... do stuff with lpOverlapped and lpOtherData ...
}
```

This version of the code uses the address of the `OVERLAPPED` structure to determine the location in the `MasterOverlapped` table and uses the corresponding entry in the parallel array at `OtherData` to hold the other data.

Let's make this code worse before we make it better:

```
OTHERDATA* FindOtherDataFromOverlapped(OVERLAPPED *lpOverlapped)
{
 for (int index = 0; index < MAX_OVERLAPPED; index++) {
  if (&MasterOverlapped[index] == lpOverlapped) {
   return &OtherData[index];
  }
 }
 FatalError(); // should never be reached
}
```

Instead of doing simple pointer arithmetic to recover the index, we walk the array testing the pointers. This is naturally worse than doing pointer arithmetic, but watch what this step allows us to do: First, we reorganize the data so that instead of two parallel arrays, we have a single array of a compound structure.

```
struct OVERLAPPEDEX
{
 OVERLAPPED Overlapped;
 OTHERDATA OtherData;
};
OVERLAPPEDEX Master[MAX_OVERLAPPED];
OTHERDATA* FindOtherDataFromOverlapped(OVERLAPPED *lpOverlapped)
{
 for (int index = 0; index < MAX_OVERLAPPED; index++) {
  if (&Master[index].Overlapped == lpOverlapped) {
   return &Master[index].OtherData;
  }
 }
 FatalError(); // should never be reached
}
// I/O is issued via
// ReadFileEx(hFile, lpBuffer, nNumberOfBytesToRead,
//            &Master[i].Overlapped, CompletionRoutine);
```

All we did was consolidate the parallel arrays into a single array.

Now that it's an array of compound structures, we don't need to carry two pointers around (one to the `OVERLAPPED` and one to the `OTHERDATA`). We can just use a single `OVERLAPPEDEX` pointer and dereference either the `Overlapped` or the `OtherData` part.

```
OVERLAPPEDEX* FindOverlappedExFromOverlapped(
    OVERLAPPED *lpOverlapped)
{
 for (int index = 0; index < MAX_OVERLAPPED; index++) {
  if (&Master[index].Overlapped == lpOverlapped) {
   return &Master[index];
  }
 }
 FatalError(); // should never be reached
}
void CALLBACK CompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    LPOVERLAPPED lpOverlapped)
{
    OVELRAPPEDEX *lpOverlappedEx =
                    FindOverlappedExFromOverlapped(lpOverlapped);
    ... do stuff with lpOverlappedEx ...
}
```

Finally, we can optimize the `FindOverlappedExFromOverlapped` function that we de-optimized earlier. Observe that the de-optimized loop is an example of the “for/if” anti-pattern.

The “for/if” anti-pattern goes like this:

```
for (int i = 0; i < 100; i++) {
 if (i == 42) do_something(i);
}
```

This can naturally be simplified to

```
do_something(42);
```

Our `FindOverlappedExFromOverlapped` function is a special case of this anti-pattern. It becomes more evident if we do some rewriting. Start with

```
&Master[index].Overlapped == lpOverlapped
```

Apply `CONTAINING_RECORD` to both sides.

```
CONTAINING_RECORD(&Master[index].Overlapped, OVERLAPPEDEX, Overlapped) ==
    CONTAINING_RECORD(lpOverlapped, OVERLAPPEDEX, Overlapped)
```

The left-hand side of the comparison simplifies to

```
    &Master[index]
```

resulting in

```
&Master[index] ==
   CONTAINING_RECORD(lpOverlapped, OVERLAPPEDEX, Overlapped)
```

Recall that `a[b]` is equivalent to `*(a+b)`, and therefore `&a[b]` is equivalent to `a+b`.

```
Master + index ==
   CONTAINING_RECORD(lpOverlapped, OVERLAPPEDEX, Overlapped)
```

Now subtract `Master` from both sides:

```
index == CONTAINING_RECORD(lpOverlapped, OVERLAPPEDEX, Overlapped) - Master
```

We have transformed the test into a clear case of the for/if anti-pattern, and the function can be simplified to

```
OVERLAPPEDEX* FindOverlappedExFromOverlapped(
    OVERLAPPED *lpOverlapped)
{
 ptrdiff_t index =
   CONTAINING_RECORD(lpOverlapped, OVERLAPPEDEX, Overlapped) - Master;
 return &Master[index];
}
```

Again, rewrite `&a[b]` as `a+b`:

```
 return Master + index;
```

Substitute the value of `index` computed on the previous line:

```
 return Master +
   CONTAINING_RECORD(lpOverlapped, OVERLAPPEDEX, Overlapped) - Master;
```

The two occurrences of `Master` cancel out, leaving

```
OVERLAPPEDEX* FindOverlappedExFromOverlapped(
    OVERLAPPED *lpOverlapped)
{
 return CONTAINING_RECORD(lpOverlapped, OVERLAPPEDEX, Overlapped);
}
```

And there you have it. By a series of purely mechanical transformations, we have rediscovered the technique of extending the `OVERLAPPED` structure.

Raymond Chen

Follow