How WRL squeezes a weak reference and a reference count into a single integer

devblogs.microsoft.com/oldnewthing/20200817-00

August 17, 2020

Raymond Chen

Objects created with the WRL template library support weak references by default.[1]

Normally, this would require the allocation of a *control block* to do the weak reference bookkeeping. The control block contains the number of weak references and the number of strong references. But allocating this control block is wasteful if no weak reference is every requested.

WRL avoids the problem by allocating the control block on demand. Here's what an object looks like initially.

`refCount_`	strong reference count
other members	
weak reference count assumed zero	

At this point, no control block has been created, which means that the number of weak references is zero, and the reference count field contains the number of strong references.

Once a weak reference is created, the layout changes:

`refCount_`	→	`vtable`	`IWeakReference`
other members		`refCount_`	weak reference count
		`strongRefCount_`	strong reference count
		`unknown_`	pointer to original object

The reference count field becomes a pointer to the control block, and it is the control block which keeps track of the number of weak and strong references. As long as the main object is alive, the number of weak references will be at least one, because there's a weak reference in the main object.

In order to distinguish these two cases, WRL uses a trick: The high bit of the `refCount_` is a flag. If the high bit is clear, then the reference count field is the strong reference count, and there are no weak references. If the high bit is set, then the reference count field is a pointer to the control block, but shifted right by one position. Shifting right one position is safe because the control block will be at least four-byte aligned, so the bottom two bits of the address will always be zero.

In the common case where no weak reference is ever requested, this avoids allocating the control block. Since there can be a lot of objects (imagine how many XAML elements exist in a complicated UI tree), saving a control block for nearly all of those objects comes out to a considerable savings.

Managing this design involves a lot of clever lock-free programming. There are a lot of race conditions, like if two threads both try to allocate the control block, or if one thread tries to increment the reference count while another thread is moving the reference count from its initial location into the control block.

There are also some sneaky optimizations: The control block is created with a `refCount_` of 2, instead of the usual value of 1. That's because creation of the control block is triggered by a request for a weak reference, so there will be two weak references when the transition is complete: One weak reference is being returned to the caller, and another weak reference is stored in the main object.

There's a similar game with reference counts when the weak reference is converted to a strong reference: The strong reference count is atomically incremented from nonzero, to confirm that the object is still alive. This extra reference then needs to be released to keep things back in sync.

Bonus chatter: There is another optimization opportunity when resolving the weak reference to a strong one:

```
STDMETHOD(Resolve)(REFIID riid, IInspectable **ppvObject)
{
    ... increment the strong reference count ...

    HRESULT hr = unknown_->QueryInterface(riid, ppvObject);
    unknown_->Release();
    return hr;
}
```

There is an extra `unknown_->Release()` to counteract the increment of the strong reference count that was performed to ensure that the target is alive. But we can fold that reference count into the one that is performed by the `QueryInterface`, provided we know the concrete type of the target.

```
hr = unknown_->CanCastTo(riid, ppvObject);
if (FAILED(hr)) {
    unknown_->Release();
}
return hr;
```

We use `CanCastTo`, which is like `QueryInterface` except that it doesn't update the reference count. This isn't quite complete, though, because we also need to deal with the possibility of an overridden `CustomQueryInterface` or a composable type (don't ask), which adds to the complexity. It also forces unique code generation for each weak pointer type, so the code expansion is probably not worth saving an increment/decrement pair.

¹ You can prevent it by passing the `InhibitWeakReference` flag.

Raymond Chen

Follow