Implementing a critical section in terms of WaitOnAddress

devblogs.microsoft.com/oldnewthing/20160825-00

August 25, 2016

Raymond Chen

Last time, we built a synchronization barrier out of `WaitOnAddress`. Today, we'll build a critical section. Remember that this is an exercise just to demonstrate ways of using `WaitOnAddress`; in real life, you should just use `EnterCriticalSection` because it has stuff like spin counts and lock convoy resistance.

Okay, enough with the warnings. Let's try it.

```
struct ALTCS
{
  DWORD OwnerThread;
  LONG ClaimCount;
};

void InitializeAltCS(ALTCS* Cs)
{
    Cs->OwnerThread = 0;
    Cs->ClaimCount = 0;
}
```

Our alternative critical section keeps track of its owner and the number of times the owner thread (if any) has entered the critical section. If the owner thread is zero, then the critical section is available.

```
void EnterAltCs(ALTCS* Cs)
{
  DWORD ThisThread = GetCurrentThreadId();
  while (true) {
    DWORD PreviousOwner = InterlockedCompareExchangeAcquire(
               &Cs->OwnerThread, ThisThread, 0);
    if (PreviousOwner == 0) {
      Cs->ClaimCount++;
      return;
    }

    if (PreviousOwner == ThisThread) {
      Cs->ClaimCount++;
      return;
    }

    WaitOnAddress(&Cs->OwnerThread,
                  &PreviousOwner, sizeof(PreviousOwner), INFINITE);
  }
}
```

To enter the critical section, we try to change the owner from nobody to ourselves. If that succeeds, then we increment the claim count (from zero to one) and we're done. Note that the increment doesn't need to be interlocked, because only the owner thread will manipulate the claim count, and we are the owner.

If the attempt to change the owner from nobody to ourselves fails because we are already the owner, then great! Increment the claim count and we're done.

Otherwise, the critical section is owned by somebody else. This means we have to wait. Use `WaitOnAddress` to wait for the owner to change, after which we go back and try to claim the critical section again. There's a wrinkle here: As written, it looks like we will wake up when the critical section becomes free (owner is zero) or when it becomes claimed by another thread (owner is nonzero). But look at this function: When the `InterlockedCompare-ExchangeAcquire` succeeds, which means that the critical section owner has changed to a nonzero value, we do not call `WakeByAddressXxx`. This means that the `WaitOnAddress` does not wake when the critical section becomes claimed. As we'll see below, we wake the wait only when the critical section becomes available. (Of course, you also have to worry about spurious wakes.)

Next comes the code to leave the critical section:

```
void LeaveAltCs(ALTCS* Cs)
{
  Cs->ClaimCount--;
  if (Cs->ClaimCount != 0) {
     return;
  }

  InterlockedExchange(&Cs->OwnerThread, 0);

  WakeByAddressSingle(&Cs->OwnerThread);
}
```

Only the thread that owns the critical section is allowed to leave it, so we can assume that the `OwnerThread` is the current thread. Decrement the claim count, and if it hasn't dropped to zero, then the critical section remains claimed, and we're done.

If the claim count drops to zero, then we release the critical section by setting the owner to zero. Note that we use an interlocked operation to ensure that the claim count is published as zero before the owner thread is cleared. (Ideally, we'd use `InterlockedExchangeRelease` if such a thing existed. I guess we could have done

```
  InterlockedCompareExchangeRelease(
    &Cs->OwnerThread, 0, Cs->OwnerThread);
```

because we know that no other thread can change the owner.)

Once we've set the owner to zero, indicating that the critical section is available, we use `WakeByAddressSingle` to wake up just one waiting thread (if any).

Next time, we'll peek behind the curtain of `WaitOnAddress` a little bit.

Raymond Chen

Follow