Further discussion of the synchronization barrier

devblogs.microsoft.com/oldnewthing/20160729-00

July 29, 2016

Raymond Chen

The synchronization barrier is apparently more confusing than I expected.

The basic idea of the synchronization barrier is that you want to each participant in some coordinated activity to wait until every participant has reached a particular state. Once the last participant reaches the desired state, all the participants are released to go on to the next step, and one of the participants is designated the "winner" for the step that just completed. (Usually being the "winner" means that you have to do some one-time final clean-up.)

A synchronization barrier is completely a user-mode concept. It is not a kernel object that you can pass to `WaitForSingleObject`, destroy with `CloseHandle`, or duplicate with `DuplicateHandle`. instead, there is a special entry function `EnterSynchronization-Barrier`, and a special cleanup function `DeleteSynchronizationBarrier`,

The analogy here is with critical sections, which are also user-mode objects that use a special entry function `EnterCriticalSection`, and a special cleanup function `DeleteCritical-Section`. You can't pass critical sections to `WaitForSingleObject`, `CloseHandle`, or `DuplicateHandle`.

You can think of a synchronization barrier as having enough tokens to track a specific number of threads (specified at its creation). Each token can be in one of the following states:

- Available.
- A thread is entering.
- A thread is leaving.

When a thread tries to enter a synchronization barrier, it takes an available token, and transitions the token to *entering*, and then waits. When all the tokens reach the *entering* state, then they all transition to the *leaving* state simultaneously. When the thread resumes execution, it clears the *leaving* state and returns the token to *available*.

(In reality, it doesn't work like this. There aren't any actual tokens. The synchronization barrier merely keeps track of the number of tokens of each kind. No wait, it doesn't even do that! We'll discuss more about the implementation later.)

It’s important that you not try to enter a synchronization barrier until you are sure that there is an available token, because the “take an available token” code doesn’t actually know whether there are tokens available; it just assumes that there is one.

If the same set of threads participates in the synchronization barrier, then this requirement is easily met, because each thread leaves the synchronization barrier before it enters it again. But if you keep shifting the set of threads in the synchronization barrier, then the incoming thread can’t enter the synchronization barrier until the outgoing thread leaves it. You can arrange for this by having the outgoing thread be the one to tell the incoming thread that it’s okay to enter the synchronization barrier.

Warning: Implementation details. Remember that this information is for educational purposes and is not contractual. Future versions of the synchronization barrier may be implemented differently.

The current implementation of a synchronization barrier uses two manual-reset events (which we will call *incoming* and *outgoing*) and a counter which records the number of available tokens.

- As threads enter the synchronization barrier, they claim a token, and if the available token count is still nonzero, they wait on the *incoming* event.
- When a thread claims the last token, the synchronization barrier changes modes:
    - It exchanges the two event handles, so that the former *outgoing* event is now *incoming*, and vice versa.
    - It resets the new *incoming* event handle.
    - It sets the available token count back to the maximum.
    - It signals the *outgoing* event handle (formerly the *incoming* event handle), which releases all the waiting threads.

From this implementation, you can see why it’s important that outgoing threads leave the synchronization barrier before new arrivals enter. If new threads arrive before the outgoing threads have exited, then it’s possible for the token count to drop to zero while there are still threads trying to get out. The result is that the synchronization barrier starts to “turn the barrier the other way” before all the threads have finished getting out. Those threads end up trapped inside the synchronization barrier for an extra cycle because the event they are using to get out got reused before they were finished with them.

Raymond Chen

Follow