Creating a co_await awaitable signal that can be awaited multiple times, part 6

devblogs.microsoft.com/oldnewthing/20210308-00

March 8, 2021

Raymond Chen

So far, we've created an awaitable signal that can be awaited multiple times. I noted last time that there was still a lot left to discuss. So let's discuss those things.

In all of the implementations we've created in this series, if resuming a coroutine raises an exception, the unresumed coroutines remain permanently suspended. Fortunately, that's not a problem in practice with non-generator coroutines. (More on that in a future entry.)

Furthermore, even though we tried to preserve fairness by resuming the waiting coroutines in FIFO order, the unfairness has not been completely eradicated: A late-coming coroutine that waits on the event after it has been signaled will be allowed to pass through immediately instead of waiting for the resumption of the coroutines that had previously been waiting patiently on the event.

Coroutine 1	Coroutine 2	Coroutine 3	Coroutine 4
Wait on event			
	Wait on event		
		Set event	
		Resume waiting coroutines	
			Wait on event
			Continues executing
Continues executing			
	Continues executing		

This is the case where there is a line of people waiting to get into the building, and once the doors are unlocked, some latecomer just runs through the open door, past all the people who had been waiting in line. We could try to fix this by making newly-waiting coroutines check both whether the event is set *and* whether the previous waiters are still being released. If so, then append the current coroutine to the end of the list of coroutines being woken. (This is the coroutine version of sending the latecomer to the back of the line.)

However, we won't do that, because it introduces extra complexity, leads to convoys, and this type of unfairness is probably not entirely unexpected.

All of our implementations resume the coroutines sequentially. This can be a problem if one of the coroutines resumes and begins doing long-running work, since it will starve out the other coroutines awaiting resumption. We can fix this by queueing all the resumptions to the thread pool. We should still queue them in FIFO order so that they are more likely to be resumed in that order.

Which ties into the next observation: The resumption of the coroutines happens on an arbitrary thread. The way we've been coding it up, the resumption occurs on the thread that signals the event. If we queue resumptions to the thread pool, then they run on a thread pool thread. Callers need to be aware that the `co_await` can change thread contexts.

And then there's the matter of abandonment. If a coroutine is destroyed while waiting for the event, we corrupt the linked list. Should we have the awaiter's destructor unlink itself from the event's linked list at destruction?

After thinking about this problem for a while, I eventually convinced myself that we do *not* need to defend against this, or at least don't need to try very hard.

A coroutine can be destroyed only when it is in the suspended state. Therefore, in principle, a caller could destroy the coroutine while it is suspended, waiting for the event to be signaled. However, the caller would have no way of knowing whether it is safe to do so, because immediately before the caller decides to destroy the coroutine, another coroutine might have signaled the event, thereby resuming the about-to-be-destroyed coroutine.

The only way the caller can be sure that destroying the coroutine is safe is if they also control the event and can ensure that the event is definitely not signaled. This means that we don't have to defend against the case where a coroutine is destroyed at the same time that an event is being signaled, which is a good thing because that race condition also turns into a race condition in our bookkeeping.

We'll start generalizing this solution to other types of synchronization objects, and I'll remember to include abandonment in that generalization.

Bonus chatter: Lewis Baker's excellent coroutine library doesn't deal with the case where an awaiting coroutine is destroyed while suspended. So maybe I'm being a bit too paranoid about this?

Raymond Chen

Follow