Lock free many-producer/single-consumer patterns: A work queue of distinct events, order not important

devblogs.microsoft.com/oldnewthing/20161124-00

November 24, 2016

Raymond Chen

Almost done with our miniseries on lock-free many-producer/single-consumer patterns. Today, we'll look at the case of multiple producers generating distinct work items which cannot be coalesced, but for which the order of processing is not important.

As I noted last time, you can view this as a scenario built on top of the previous one. In the previous scenario, there was a counter of the number of work items to be done, but the work itself was fixed. You can pair this with another data structure that contains a collection of things to do. In that case, the `DoSomeWork()` function pulls a work item out of the collection.

We're going to go one step further: We're going to let the work item be its own counter.

```
SLIST_HEADER WorkQueue;

struct alignas(MEMORY_ALLOCATION_ALIGNMENT)
WorkItem : SLIST_ENTRY
{
    ... other stuff ...
};

void Initialize()
{
 InitializeSListHeader(&WorkQueue);
}

void RequestWork(WorkItem* work)
{
 if (InterlockedPushEntrySList(&WorkQueue, work)
                                                  == nullptr) {
  // You provide the WakeConsumer() function.
  WakeConsumer();
 }
}

// You call this function when the consumer receives the
// signal raised by WakeConsumer().
void ConsumeWork()
{
 PSLIST_ENTRY entry;
 while ((entry = InterlockedPopEntrySList(&WorkQueue))
                                                  != nullptr) {
  ProcessWorkItem(static_cast<WorkItem*>(entry));
  delete entry;
 }
}
```

We use the lock-free linked list data structure `LIST_HEADER` to manage our work queue. To request some work, we push an item onto the front of the list. The `InterlockedPushEntrySList` function returns the previous list head. If that returns `nullptr`, then it means that the list was empty, so we wake up the consumer. If the list was not empty, then it means that somebody else woke the consumer, so we won't do it (to avoid a spurious wake).

On the consumer side, we atomically pop work off the list and process them, and we stop when there is no more work. Since the order in which work items are processed is presumed to be unimportant, we can process them LIFO.

Remember, there is only one consumer, so if `WakeConsumer` is called while `ConsumeWork` is still running, the wake will not start a new consumer immediately. It will wait for the existing `ConsumeWork` to complete before starting a new `ConsumeWork`.

Next time, we'll wrap up by looking at the case where work items must be processed FIFO.

Raymond Chen

Follow