# Lock free many-producer/single-consumer patterns: A work queue where the last one wins

devblogs.microsoft.com/oldnewthing/20161122-00

November 22, 2016

Raymond Chen

Continuing our miniseries on lock-free many-producer/single-consumer patterns, we're going to look at the case where you have a work queue where there can be multiple work items, but each work item replaces the previous one.

You see this pattern when the work is something like *Update*. If you update with one set of data, and then update with a second set of data, then the second set of data overwrites the first set; you may as well not have bothered with the first one.

This is similar to our previous case, except that this time the tasks are not identical, so we can't just use a flag. But that's okay. We can use the data itself as the flag!

```
Data* PendingData;

void RequestUpdate(Data* data)
{
 Data* previousData =
    InterlockedExchangePointer(&PendingData, data);
 if (previousData == nullptr) {
  // You provide the WakeConsumer() function.
  WakeConsumer();
 } else {
  delete previousData;
 }
}

// You call this function when the consumer receives the
// signal raised by WakeConsumer().
void ConsumeUpdate()
{
 Data* currentData;
 while ((currentData = InterlockedExchangePointer(&PendingData,
                                          nullptr)) != nullptr) {
  Update(currentData);
  delete currentData;
 }
}
```

This is similar to the previous case, except that we're using a pointer instead of a flag. Again, we wake the consumer only on the transition from null to non-null. There is an extra wrinkle in that we need to delete the previous data if our update replaced a pending update that hasn't yet begun processing.

The idea here is that the `PendingData` variable contains the data that is being transferred from the producer to the consumer. The consumer goes into a loop checking if there is any pending data, and if so, udpating with it. If the producer can replace the data before the consumer sees it, then the producer has basically saved the consumer the work of updating with data that is already stale.

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

That was a little trickier. Maybe we'll take a break and do something slightly less tricky next time.

Raymond Chen

**Follow**