# Combining the work queue of distinct events, order not important, with an auto-reset event

devblogs.microsoft.com/oldnewthing/20170616-00

June 16, 2017

Raymond Chen

Some time ago, I described a lock-free pattern for a work queue of distinct events, where the order of event processing is not important. A customer was using a variation of this pattern, where they unlink the entire work queue in the consumer, and combining it with an auto-reset event to signal the consumer thread that there is work to do. The general sketch is like this:

```
SLIST_HEADER WorkQueue;
HANDLE WorkAvailable;

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

void Initialize()
{
 InitializeSListHeader(&WorkQueue);

 // Create an auto-reset event, initially unset.
 WorkAvailable = CreateEvent(nullptr, FALSE, FALSE,
                             nullptr);
}

void RequestWork(WorkItem* work)
{
 if (InterlockedPushEntrySList(&WorkQueue, work)
                                                 == nullptr) {
  SetEvent(WorkAvailable);
 }
}

void ConsumeWork()
{
 while (true) {
  WaitForSingleObject(WorkAvailable, INFINITE);
  PSLIST_ENTRY entry = InterlockedFlushSList(&WorkQueue);
  while (entry) {
   ProcessWorkItem(static_cast<WorkItem*>(entry));
   PSLIST_ENTRY nextEntry = entry->Next;
   delete entry;
   entry = nextEntry;
  }
 }
}
```

The customer was looking for something lighter weight than a kernel event, however.

Enter `WaitOnAddress`. We could use our `ALT_AEVENT` structure as a drop-in replacement for the kernel event, but we can do better.

We can use a `LONG` as our data and use it to signal the consumer thread.

```
SLIST_HEADER WorkQueue;
LONG WorkAvailable;

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

void Initialize()
{
 InitializeSListHeader(&WorkQueue);

 WorkAvailable = 0;
}

void RequestWork(WorkItem* work)
{
 if (InterlockedPushEntrySList(&WorkQueue, work)
                                                 == nullptr) {
  InterlockedIncrement(&WorkAvailable);
  WakeByAddressSingle(&WorkAvailable);
 }
}

void ConsumeWork()
{
 LONG PreviousAvailable = 0;
 while (true) {
  WaitOnAddress(&WorkAvailable,
                &PreviousAvailable,
                sizeof(PreviousAvailable),
                INFINITE);
  PreviousAvailable = WorkAvailable;
  PSLIST_ENTRY entry = InterlockedFlushSList(&WorkQueue);
  while (entry) {
   ProcessWorkItem(static_cast<WorkItem*>(entry));
   PSLIST_ENTRY nextEntry = entry->Next;
   delete entry;
   entry = nextEntry;
  }
 }
}
```

We replace our kernel handle with a `LONG` that contains the number of times the consumer has been notified of work. The precise meaning of the value isn't important; what's important is that it changes whenever we want the consumer to wake up, and zero means that no work has ever been queued.

The consumer waits for the counter to become nonzero, meaning that there might be some work. It captures the updated counter value, drains any available work, and then waits for the counter to change again.

There are many ways this code could be structured, but it is important that we capture the counter *before* draining the work. That way, if the counter changes while we are draining the work, our subsequent `WaitOnAddress` will return immediately rather than waiting for the counter to change yet again.

Note also that the code is resilent to spurious wake-ups. If the `WaitOnAddress` returns prematurely, the code performs a redundant check for work. It won't find any work, and will cycle back to wait for another change.

Raymond Chen

**Follow**