Lock-free algorithms: The try/commit/(hand off) model

devblogs.microsoft.com/oldnewthing/20110415-00

April 15, 2011

Raymond Chen

The last lock-free pattern for this week isn't actually lock-free, but it does run without blocking.

The pattern for what I'll call try/commit/(hand off) is more complicated than the other patterns, so I'll start off by describing it in words rather than in code, because the code tends to make things more complicated.

First, you take the state variable and chop it up into pieces. You need some bits to be used as a lock and as a *work has been handed off* flag. And if the work that has been handed off is complicated, you may need some more bits to remember the details of the handoff. A common way of doing this is to use a pointer-sized state variable, require that the objects being pointed to are suitably aligned, and reusing the bottom bits as flags. For example, if you require that the objects be `DWORD`-aligned, then the two bottom bits will always be zero and you can reuse them as flags.

To perform an operation, you first try to lock the state variable. If you can't because the state variable is already locked, then you record the details of the operation in the state variable and update it atomically.

If you succeed in locking the state variable, then you perform the desired operation, but before you unlock the state variable, you look to see if any work has been handed off. (This hand-off work is the result of attempts to perform the operation while you held the lock.) If there is hand-off work, then you perform that work as well. Of course, while you're doing that, more hand-off work may arrive. You can't unlock the state variable until you've drained off all the pent-up hand-off work.

The code for this pattern tends to be a tangle of loops since there is a lot off backing off and retrying. Every atomic operation is its own loop, draining the hand-off work is another loop, and any time an `InterlockedCompareExchange` fails, you have to undo the work you did and retry—another loop.

I trust only about five people in the world to write code that is this advanced, and I'm not one of them. But just to illustrate the principle (although I will certainly get the details wrong), here's an implementation of a synchronization-like object which I will call a `GroupWait` for lack of any other name. It has the following operations:

- `AddWait` : Register an event handle with the group wait.
- `SignalAll` : Signals all events that are registered with the group wait. Once an event is signalled, it is automatically unregistered from the group wait. If you want the event to be signalled at the next call to `SignalAll` you have to re-add it.

The group wait object is just a linked list of `NODE` s containing the handles being waited on.

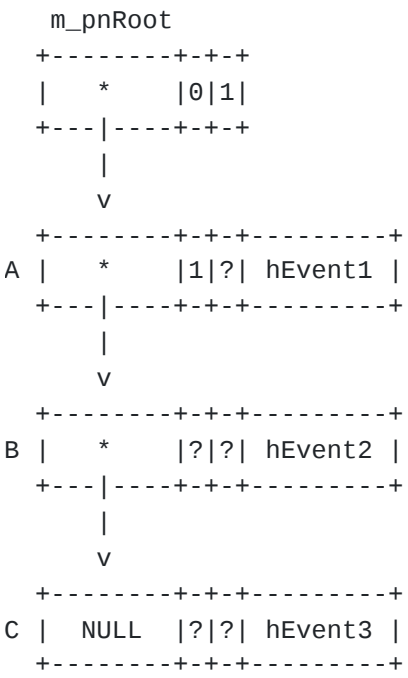
Actually, this type of object doesn't need to use the try/commit/hand off model. It can be implemented in a much more straightforward manner by having `AddWait` atomically prepend the node to a list and having `SignalAll` atomically steal the list. There are even prewritten functions to perform these atomic linked list operations for you. But I'm going to implemented it the complicated way for demonstration purposes. In real life, the code would be much simpler.

Since the bottom two bits of the pointer must be zero due to alignment, we repurpose them as a lock bit and a signal bit. The lock bit is set when the list is locked, and the signal bit is set when a signal was requested but had to be handed off because the list was locked.

```
// WARNING! IF YOU USE THIS CODE YOU ARE AN IDIOT - READ THE TEXT ABOVE
struct NODE;
NODE *Node(LONG_PTR key) { return reinterpret_cast<NODE*>(key); }
enum {
 Locked = 1,
 Signalled = 2,
};
struct NODE {
 NODE *pnNext;
 HANDLE hEvent;
 LONG_PTR Key() { return reinterpret_cast<LONG_PTR>(this); }
 NODE *Ptr() { return Node(Key() & ~(Locked | Signalled)); }
};
#define NODE_INVALID Node(-1)
class GroupWait {
public:
 GroupWait() : m_pnRoot(NULL) { }
 ~GroupWait();
 BOOL AddWait(HANDLE hEvent);
 void SignalAll();
private:
 NODE *m_pnRoot;
};
```

Since I will be viewing the `NODE*` as both a pointer and as a bunch of bits (which I call a *key*), I created some helper methods to save typing. `Node` and `Key` convert back and forth between node pointers and keys, and `Ptr` strips off the tag bits and returns a usable pointer.

For notational purposes, a `NODE*` will be written as the combination `p|S|L` where `p` is a pointer to the next node, `S` is the signalled bit, and `L` is the lock bit. The signalled bit is set to indicate that we need to signal all the nodes in the list starting with the *next* node. (Think of the `S` bit as being attached to the outgoing arrow.) For example, this linked list:



represents a group wait with three registered event handles. The `S` bit is clear on the root pointer, which means that nobody has yet requested that `hEvent1` be signalled. On the other hand, the `S` bit is set on node A, which means that all the events after node A need to be signaled, specifically, `hEvent2` and `hEvent3`. Note that this means that it doesn't matter whether the `S` bit is set on nodes B or C; those events are getting set regardless because the `S` bit on node A already requested it. (In particular, the `S` bit on the last node is meaningless since there are no nodes which come after it.)

The `L` bit is meaningless on all pointers other than `m_pnRoot`.

Okay, let's start be adding a handle to the wait list:

```
BOOL GroupWait::AddWait(HANDLE hEvent)
{
 NODE *pnInsert = new(nothrow) NODE;
 if (pnInsert == NULL) return FALSE;
 pnInsert->hEvent = hEvent;
 NODE *pn;
 NODE *pnNew;
 do {
  pn = InterlockedReadAcquire(&m_pnRoot, NODE_INVALID);
  pnInsert->pnNext = pn;
  pnNew = Node(pnInsert->Key() | (pn->Key() & Locked));
 } while (InterlockedCompareExchangeRelease(&m_pnRoot, pnNew, pn) != pn);
 return TRUE;
}
```

To add a handle to the wait list, we just prepend it to the linked list, being careful to propagate the `L` bit into the new pointer so we don't accidentally release a lock that somebody else took. We add the node with the `S` bit clear on the inbound pointer since nobody has yet asked for this handle to be signalled. After setting up the node, we attempt to insert it into the head of the list, and if we can't (because somebody else beat us to it), then we restart and try again. This is a standard try/commit/try again pattern.

Exercise: Is there an ABA race condition here?

The `AddWait` method illustrates one extreme case of the try/commit/hand off model, where there is really nothing to hand off; we did it all ourselves. Of course, this does make other parts of the code trickier since they have to go back and deal with nodes that were added while the list was locked.

The nasty part of the code is in `SignalAll`. I'll present it in pieces.

```
void GroupWait::SignalAll()
{
 NODE *pnCapture;
 NODE *pnNew;
 do {
  pnCapture = InterlockedReadAcquire(&m_pnRoot, NODE_INVALID);
  if (pnCapture->Key() & Locked) {
   pnNew = Node(pnCapture->Key() | Signaled);
  } else {
   pnNew = Node(Locked);
  }
 } while (InterlockedCompareExchangeAcquire(&m_pnRoot,
                                pnNew, pnCapture) != pnCapture);
 if (pnCapture->Key() & Locked) return;
 ...
```

If the list is locked, then all we do is try to set the `S` bit on the root. If the list is not locked, then we try to lock it and simultaneously detach all the nodes by replacing the root pointer with `NULL|0|1`. Either way, we perform the operation with the try/commit/try again pattern until we finally get through.

If the list was locked, then all we had to do was set the `S` bit on the root pointer. Setting the `S` bit on the root pointer means that all the nodes reachable from this pointer (*i.e.*, all nodes after the root, which is all nodes) should be signalled, which is exactly what we want. Since the list is locked, we leave the actual signalling to the code that unlocks the list. (This is the *hand off* part of *try/commit/hand off*.)

Exercise: What if the `S` bit is already set? Did we lose a signal?

Otherwise, we are the ones to lock the list. We also detach the node list, for if another thread calls `SignalAll`, we don't want that signal to affect the nodes that we're signalling. (Otherwise we might end up double-signalling the event.)

```
...
NODE *pnNext;
NODE *pn;
for (pn = pnCapture->Ptr(); pn; pn = pnNext) {
 SetEvent(pn->hEvent);
 pnNext = pn->pnNext->Ptr();
 delete pn;
}
...
```

That little fragment above is basically what you would do in a naïve implementation that didn't worry about multithreading: It walks the list of nodes, signals each event, and then frees the node. The only trick is sending each node pointer through `->Ptr()` to strip off the tag bits.

Next comes the unlock code. First, a preparatory step:

```
...
pnCapture = pnNew;
...
```

We exchanged `pnNew` into `m_pnRoot` up above, and if that's still the value of `m_pnRoot`, then it means that nobody tried to perform any operations while the list was locked, and we got off easy.

```
 ...
 for (;;) {
  pnNew = Node(pnCapture->Key() & ~Locked);
  if (InterlockedCompareExchangeRelease(&m_pnRoot,
                       pnNew, pnCapture) == pnCapture) {
   return;
  }
 ...
```

We start a new loop whose job is to drain off all the handed-off work items that built up while the list was locked. First, we see whether anything has changed since the last time we looked; if not, then we unlock and we're done. Otherwise, we proceed to pick up all the handed-off work:

```
 ...
  pnCapture = InterlockedReadAcquire(&m_pnRoot, NODE_INVALID);
  NODE *pnNew = Node(pnCapture->Key() & ~(Locked | Signaled));
  NODE **ppn = &pnNew;
  NODE *pn;
  NODE *pnNext;
  BOOL fSignalSeen = FALSE;
  for (pn = pnNew; pn->Ptr(); pn = pnNext) {
   pnNext = pn->Ptr()->pnNext;
   if (fSignalSeen) {
    SetEvent(pn->Ptr()->hEvent);
    delete pn->Ptr();
   } else if (pn->Key() & Signaled) {
    fSignalSeen = TRUE;
    (*ppn) = Node(Locked); // detach but retain lock
    SetEvent(pn->Ptr()->hEvent);
    delete pn->Ptr();
   } else {
    ppn = &pn->Ptr()->pnNext;
   }
  }
 } // retry unlock
} // end of function
```

To drain the handed-off work, we walk the list of nodes, keeping track of whether we've seen an `S` bit. If so, then we signal the event and free the node. And the first time we see an `S` bit, we null out the inbound pointer to detach the list from the chain so we do not double-signal the event in the future.

Once that's done, we go back and try to unlock again. Eventually, there will be no more hand-off work, and we can finally return.

And that's it, a demonstration of the try/commit/hand off model. The basic idea is simple, but getting all the details right is what makes your head hurt.

I leave this sort of thing to the kernel folks, who have the time and patience and brainpower to work it all through. An example of this pattern can be found, for example, in this talk that describes the dismantling of the dispatcher spinlock.

Raymond Chen

Follow