FIFO Patent Hack

Mark D. Rustad

While working for Apple, I applied for, and was granted two patents. The second patent, #5,671,446, is a method for atomically accessing a FIFO queue. It is general enough to permit multiple enqueuers and multiple dequeuers. It also works across multiple processors, in conjunction with timesharing schedulers and, with a minor limitation, with interrupt routines combined with all of the above cases. Oh, and it never has to disable interrupts. It only requires the equivalent of a compare-and-swap for atomic operations. Philip Rakity and I developed the original patent and thought it was pretty cool.

Of course, the problem with patents is that I cannot use this method now that I am no longer with Apple. Even Microsoft can use it, but I can't. So now it would be really cool to work around the patent in some way.

The patented method makes use of a queue head with a structure like this:

```
struct FIFO
{
     OTLink    * volatile Head;
     volatile OTLIFO LIFO;
     volatile ulong  Lock;/* True if FIFO locked */
};
```

The only real problem with this type of FIFO implementation is that if it is taking the place of a more traditional FIFO, the new queue head structure will take more space in existing data structures that include queue heads. In some cases, this can be a critical consideration. In a caffeine and sleep-deprived state at MacHack, I came up with a solution to this problem that also may work around the patent.

DISCLAIMER: I am not a lawyer. Although what I present here may avoid infringing on the referenced patent, but it also may not. Or there might be some other patent. Consider yourself warned! If it matters, consult your own lawyer.

On to a possible work-around. The original dequeue routine looked something like this:

```
/*
 *   DequeueFIFO - Dequeue from head of FIFO.
 */

OTLink    *DequeueFIFO(FIFO *pFIFO)
{
     OTLink    *tmp;

     do
     {
          /* Extract elements queued normally */

          while (tmp = pFIFO->List)
          {
               if (OTCompareAndSwapPtr(tmp,tmp->fNext, &pFIFO->List))
                    return tmp;
          }

          /* No entry, so check the LIFO */

          if (!OTCompareAndSwap32(0, 1, &pFIFO->Lock))
               return nil;

          /* If something is in LIFO */
          if (tmp = OTLIFOStealList(&pFIFO->LIFO))
          {
               /* Reorder the LIFO and make it the FIFO */
               pFIFO->List = OTReverseList(tmp);
          }

          /* Ensures List is written back before lock is cleared */
          DoSYNCInstruction();
          pFIFO->Lock = 0;              /* Release the lock */
     } while (tmp);

     return tmp;
}
```

If the Lock field could be eliminated, the queue head would be simply two pointers, just like typical FIFO implementations. Fortunately by applying a hack, it can be made so!

If the List field is defined to contain a well-known constant to signify that the lock is set, we have everything we need. As it turns out, this is even fairly convenient to do. Since these structure should all be aligned anyway, we can use the value 1 to denote that the FIFO is locked. Behold:

```
/*
 *    NewDequeueFIFO - Dequeue from head of FIFO.
 */

#define    kFIFOLocked    1

OTLink    *NewDequeueFIFO(FIFO *pFIFO)
{
     OTLink    *tmp;

     do
     {
          /* Extract elements queued normally */

          while ((tmp = pFIFO->List) != 0 && (int)tmp != kFIFOLocked)
          {
               if (OTCompareAndSwapPtr(tmp,tmp->fNext, &pFIFO->List))
                    return tmp;
          }

          /* No entry, so check the LIFO */

          if (tmp == kFIFOLocked)   /* Handle locked case */
               continue;            /* Try again */

          if (!OTCompareAndSwap32(0, kFIFOLocked, &pFIFO->List))
               return nil;

          /* If something is in LIFO */
          if (tmp = OTLIFOStealList(&pFIFO->LIFO))
          {
               /* Reorder the LIFO and make it the FIFO */
               pFIFO->List = OTReverseList(tmp);
          }

          /* Ensures List is written back before lock is cleared */
          DoSYNCInstruction();
     } while (tmp);

     return tmp;
}
```

Notice that the changes eliminated all references to the Lock field in the original FIFO structure. Only the LIFO field is referenced in the enqueue routines.

For reference, here is the enqueue routine:

```
void   EnqueueFIFO(FIFO *pFIFO, OTLink *pEntry)
{
       OTLIFOEnqueue(&pFIFO->LIFO, pEntry);
       return;
}
```

Many variations on these routines are possible. For example, a non-blocking variant of dequeue can be made for use in interrupt routines. Another example would be to encode the cpu owning the lock into the lock value. Then interrupt dequeuers can spin if the lock is owned by a different cpu, and return nil otherwise. Another variation is to permit high-priority messages to effectively be placed at the head of the queue. That is implemented by having a high-priority LIFO that is checked before the normal list in the dequeue routine, and having the high-priority enqueuer place messages on that list.

These methods dovetail very well with the techniques described in the Atomicity paper delivered here at MacHack. Some of the techniques in that paper may or may not infringe on the patent mentioned earlier.