Beyond Mutexes, Semaphores, and Critical Sections

Serge Plagnol
Green Hills Software
serge@ghs.com

1. Abstract

The traditional approach to multitasking synchronization has been to use Mutexes, Semaphores, and Critical sections. However, those primitives can lead to inefficiency or, even worse, to error conditions such as, for example, dead or live locks or priority inversion. The problems with those primitive are particularly vivid with real-time systems. Also, with the rapid deployment of multi-core systems, those traditional mechanisms are showing new classes of issues. This talk will discuss how the use of non-blocking algorithms through atomic and barrier operations can lead to more robust, deterministic and higher performance systems.

2. Definitions

Even through many definitions exist, an algorithm is generally considered non-blocking if designed to avoid requiring a critical section. Therefore, this type of algorithms generally allow multiple tasks to make progress without ever blocking each other. For some operations, these algorithms provide valuable alternatives to locking mechanisms.

3. The problems with locks

The traditional approach to multi-tasked programming is to use locks to synchronize access to shared resources. Synchronization primitives such as mutexes, semaphores, and critical sections are all mechanisms by which a programmer can ensure that certain sections of code do not execute concurrently if doing so would corrupt shared memory structures. If one task attempts to acquire a lock that is already held by another task, the task will block until the lock is free.

Blocking a task is undesirable for many reasons. An obvious reason is that while the task is blocked, it cannot accomplish anything. Also with the growing popularity of multi-core systems, blocking is becoming more and more costly as locks effectively serialize operations distributed among several tasks, practically reducing the opportunities for parallelism and the effective use of hardware computing resources.

Other problems are less obvious. Certain interactions between locks can lead to error conditions such as deadlock, livelock, and priority inversion. For example, priority inversion occurs if a high-priority task is blocked by a lower-priority one, violating priority-based scheduling rules making RMA difficult if not impossible. This problem can partially be circumvented using, for example, priority-Inheritance Mutexes. However, situations such as chain-blocking can still occur.

Using locks also involves a trade-off between coarse-grained locking and fine-grained locking. Coarse-grain can significantly reduce opportunities for parallelism, again more and more important with multi-core system, while fine-grained requires more careful design, increases overhead and is more prone to bugs.

Global data structures protected by mutual exclusion cannot safely be accessed by interrupt handlers, as the lock may be already held by a task when an interrupt is serviced. In many embedded OS, this is typically circumvented by allowing tasks to disable interrupts. However, this can have devastating effects on the general system behavior since all real-time events can be arbitrarily postponed by user-code. On the opposite, non-blocking algorithms are also safe for use in interrupt handlers.

Non-blocking algorithms have the potential to remove the risk of priority inversion, as no task is forced to wait for another task to complete.

4. Non-blocking algorithms Implementation

The synchronization primitives provided by most modern architectures, such test-and-set (TAS) compare-and-swap (CAS) or load-locked/store-conditional (LL/SC) are powerful enough to achieve Non-blocking algorithms. It is interesting to note that Mutexes, Semaphores and Critical Sections are almost always implemented using these primitives.

Until recently, all non-blocking algorithms had to be written "natively" with the underlying primitives to achieve acceptable performance. However, each CPU architecture implements differently TAS, CAS and/or LL/SC operations. Unfortunately, some of the simplest processors don't even have such important primitives.

```
TestAndSet::
    lwarx     r12 <test>, r0, r3 <addr>
    cmplw     r12 <test>, r4 <old>
    bne       failure
    stwcx.    r5 <new>, r0, r3 <addr>
    bne       TestAndSet
    li        r3, 0
    blr
failure:
    li        r3, 1
    blr
```

fig 1. Sample TestAndSet on Power

```
TestAndSet::
    movl       4(%esp), %ecx  <addr>
    movl       8(%esp), %eax  <old>
    movl       12(%esp), %edx  <new>
    lock
    cmpxchgl %edx <new>, 0(%ecx) <addr>
        /* eax <old> in implicitely used */
    setne      %al
    movzbl     %al, %eax
    ret
```

fig 2. Sample TestAndSet on x86

Luckily, Modern RTOS do provide the above synchronization primitives in a CPU-independent fashion event supporting processors without dedicated instructions.

```
Error TestAndSet(volatile uint32_t *addr, uint32_t old, uint32_t new);
```

fig 3. Sample TestAndSet C prototype

Much research has also been done in providing non-blocking algorithms for basic data structures such as stacks, queues, sets, and hash tables. These allow programs to easily exchange data between tasks asynchronously.

Last but not least, some data structures are weak enough to be implemented without special atomic primitives, at least if only accessed by a single CPU. A classic example of such a simple and yet very useful data structure is the single-reader single-writer ring buffer FIFO.

```
// FIFO data storage
volatile int FifoData[FIFO_SIZE];

Error FifoGet(int *val)
{
    if(FifoWrite == FifoRead)
       // FIFO is empty
       return Failure;

    /* Read value from current FIFO slot */
    *val = FifoData[FifoRead % FIFO_SIZE];

    /* Increment read index */
    FifoRead++;

    return Success;
}
```

```
Error FifoPut(int val)
{
    if(FifoWrite + 1 == FifoRead)
       /* FIFO is full */
       return Failure;

    /* Store value in next FIFO slot */
    FifoData[FifoWrite % FIFO_SIZE] = val;

    /* Increment write index */
    FifoWrite++;

    return Success;
}
```

fig 4. Single-Reader Single-Writer Ring Buffer FIFO sample implementation

This example algorithm works because all synchronization is done by comparing (i.e. reading) FifoRead and FifoWrite. only the writer task is calling FifoPut() and therefore modifying FifoWrite. Symmetrically, only the reader task is calling FifoGet() and therefore modifying FifoRead. The problem is reduced to one Task reading a value than can only be modified by another task. Therefore there is no possible race condition on the variables used for gating. However, are we always sure that the data is actually put/read into/from the FIFO before FifoWrite/FifoRead is updated?

5. Memory Access Ordering

Most processors reorder memory accesses to improve performance, but they do so in such a way that most programs execute correctly even when the programs were not developed with a full understanding of how processors can reorder memory accesses. However, programs that interact with devices, or with other processors must be explicitly aware of issues of memory ordering.

6. Memory Access Reordering Problems

While some processors provide more memory access ordering guarantees than others, many processors provide only very weak ordering guarantees, and few processors include hardware to prevent memory accesses from appearing to be reordered. Further, the guarantees provided by most processors are very intricate and in some cases difficult to apply correctly.

We define the "program order" of a sequence of memory accesses as the order in which the memory access instructions are actually executed by a processor. When dealing with hand-written assembly code, program order is the same as the order in which the memory accesses occur in source code. However, when dealing with C code, program order may be very different from source order because compilers can reorder accesses to memory objects not accessed with the volatile qualifier.

In order to improve performance some processors may also dynamically reorder instructions and memory accesses. However, all processors guarantee that, for each memory location X accessed by a program, all accesses to X by the program are performed in program order. This rule is what enables programs that do not interact with devices or with other processors to execute correctly without being aware that memory reordering can take place.

In general, you should assume that a processor is free to arbitrarily reorder memory accesses to distinct memory locations. This has significant implications on the above algorithms and even further implications for device drivers and multiprocessor environments (whether AMP or SMP).

In the above example we have a guarantee that access to FifoRead will be in program order. Similarly with FifoWrite or the FifoData array elements. However, we have no guarantee that all accesses to all variables will be ordered.

Let's take another similar example. if a device driver writes to a buffer in memory, then sets a valid bit in a buffer descriptor in memory, and finally writes a device register informing the device that another buffer descriptor is valid, the processor may write to the device register first, then write to the buffer descriptor, and then write to the buffer.

There are various hardware implementation details that can lead to memory accesses being reordered, for example:

- Prefetch buffers and speculative execution can both cause data to be loaded by the processor before it is definitively requested by a program.
- Lockup-free (hit-under-miss) caches allow data to be loaded from memory before earlier memory accesses are complete.
- Banked caches can delay the processing of some invalidations, causing loads to be reordered. Unlike the previous two mechanisms, banked caches can even cause dependent loads to be reordered (where the result of an earlier load is used to determine the address of a subsequent load).
- Store buffers can cause stores to be delayed, reordered, or combined into larger accesses.
- Non-broadcast memory interconnects can defy causality if cache invalidation messages are delayed.

Processor memory ordering is restricted to program order by using memory barrier instructions. A memory barrier instruction informs the processor that some or all of the accesses associated with instructions executed before the barrier instruction should not be reordered with accesses associated with subsequent instructions.

Memory barrier instructions, also known as membar or fence instructions, cause a processor or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction.

Many processors implement a variety of barrier instructions, each of which orders distinct sets of accesses and has a distinct performance impact. The behavior of barrier instructions is often very complex, so it is required that the OS provides efficient, architecture-independent barrier functions that abstract away most of the complexity.

7. Using FullBarrier() to Order Accesses

Typically the RTOS should provide a function that we will call FullBarrier() that shall be the strongest memory barrier function. It can be used to order any memory accesses or device register accesses, regardless of whether the device registers are memory-mapped.

Using the example of the device driver in the previous section, the following could correctly order the writes to the memory buffer, to the buffer descriptor, and to the device register:

```
memcpy(buffer, user_data, len);
FullBarrier();
bd->valid = true;
FullBarrier();
*device_command_register = DEVICE_GO;
```

If we go back to our original FIFO example, the code can be made safe as follow:

```
Error FifoGet(int *val)
{
    if(FifoWrite == FifoRead)
       // FIFO is empty
       return Failure;

    /* Read value from current FIFO slot */
    *val = FifoData[FifoRead % FIFO_SIZE];

    /* We need to make sure that the */
    /* FifoData slot is read before */
    /* we mark the FIFO slot as empty */
    FullBarrier();

    /* Increment read index */
    FifoRead++;

    return Success;
}
```

```
Error FifoPut(int val)
{
    if(FifoWrite + 1 == FifoRead)
       /* FIFO is full */
       return Failure;

    /* Store value in next FIFO slot */
    FifoData[FifoWrite % FIFO_SIZE] = val;

    /* We need to make sure that the */
    /* FifoData slot is written before */
    /* we mark the FIFO slot as full */
    FullBarrier();

    /* Increment write index */
    FifoWrite++;

    return Success;
}
```

fig 4. Single-Reader Single-Writer Ring Buffer FIFO with strong access ordering

8. Using MemoryBarrier() to Order All Memory Accesses

While FullBarrier() is the strongest barrier function, it is also the most expensive. Usually only loads and stores need to be ordered.

When only loads and stores need to be ordered, it can be significantly less expensive to invoke weaker barriers. For this we introduce MemoryBarrier(). Unlike FullBarrier(), MemoryBarrier() requires the caller to indicate what kinds of accesses are to be ordered by providing 2 parameters, the first parameters describing the type of accesses that we want to be committed before the barrier with respect to the type of accesses described by the second parameters. The MemoryBarrier() implementation will then use the appropriate required target barrier instruction, if any, the ensure that the requested ordering in enforced.

The simplest form would be invoking MemoryBarrier(MEMORY_BARRIER_ALL, MEMORY_BARRIER_ALL), which emits a barrier that ensures all memory accesses before the call to MemoryBarrier() are performed before all memory accesses after the call to MemoryBarrier().

9. Ordering Accesses on Processors with dedicated IO buses

Not all device registers are memory-mapped. For example, x86 processors have separate memory and IO buses. Similarly, PowerPC 4xx cores have a separate Device Control Register (DCR) bus on which on-chip peripherals can have registers. Accesses to these IO registers usually cannot be reordered with other accesses to such registers, but they can be reordered with memory accesses.

FullBarrier() must be used to order IO accesses with memory accesses. For example, if you want to write memory-mapped device registers A and B, then IO registers Q and R, and then memory-mapped device register C, you could do the following:

```
*device_register_A = VALUE_A;
MemoryBarrier(MEMORY_BARRIER_ALL, MEMORY_BARRIER_ALL);
*device_register_B = VALUE_B;
FullBarrier();
write_IO_Q(VALUE_Q);
write_IO_R(VALUE_R);
FullBarrier();
*device_register_C = VALUE_C;
```

10. Optimizing Accesses to Regular memory

In many cases, processors can provide inexpensive barriers for ordering loads and stores to regular memory. Consequently, the performance of code that makes frequent use of MemoryBarrier() may be improved by specifying that only accesses to regular memory need to be ordered.

By regular memory we mean RAM that is typically cacheable and does not have any specific attribute. This is typically where C variables go. It is also the type of memory returned my malloc(). Most application programs only access regular memory therefore this case is very relevant to all programs and programmers.

While some processors can order accesses to regular memory regardless of whether the accesses are loads or stores, some processors can only order accesses efficiently in some of the four possible combinations of prior loads/stores from regular memory before subsequent loads/stores from regular memory. Consequently, in order to improve the performance of barriers that order accesses only to regular memory, it is sometimes necessary (and generally advisable) to inform MemoryBarrier() of exactly what is being ordered.

Keeping the same device driver example, suppose that the buffer and buffer descriptor reside in regular memory (assuming that there is hardware cache coherency between the processor and the device). The following code could be used to ensure that the accesses are performed in the right order:

```
memcpy(buffer, user_data, len);
MemoryBarrier(MEMORY_BARRIER_STORE, MEMORY_BARRIER_STORE);
bd->valid = true;
```

Note that the MemoryBarrier() call here only orders stores to regular memory before stores to regular memory. Many processors have an efficient barrier for ordering this.

As a second example, suppose that a valid flag and a buffer reside in regular memory that is shared with a device or another processor, and that the buffer is only valid if the valid flag is non-zero. The following code could be used to check the valid flag and access the buffer, ordering the accesses properly:

```
if (shared->valid_flag) {
    MemoryBarrier(MEMORY_BARRIER_LOAD, MEMORY_BARRIER_LOAD);
    memcpy(my_copy, shared->buffer, len);
    ...
}
```

This example could be extended slightly by having the program clear the valid flag as an indication that it is done reading the buffer. This would require an additional barrier:

```
if (shared->valid_flag) {
    MemoryBarrier(MEMORY_BARRIER_LOAD, MEMORY_BARRIER_LOAD);
    memcpy(my_copy, shared->buffer, len);
    MemoryBarrier(MEMORY_BARRIER_LOAD, MEMORY_BARRIER_STORE);
    shared->valid_flag = 0;
}
```

It is also meaningful to order multiple kinds of accesses with one barrier. For example, suppose the above example is extended even further by having the program also copy new data into the buffer before clearing the valid flag (where some device or some other processor waits for the valid flag to be cleared, reads the data from the buffer, processes the data, writes resulting data to the buffer, and then sets the valid flag again):

```
if (shared->valid_flag) {
    MemoryBarrier(MEMORY_BARRIER_LOAD, MEMORY_BARRIER_LOADSTORE);
    memcpy(my_output, shared->buffer, len);
    memcpy(shared->buffer, my_next_input, len);
    MemoryBarrier(MEMORY_BARRIER_LOADSTORE, MEMORY_BARRIER_STORE);
    shared->valid_flag = 0;
}
```

Note: MEMORY_BARRIER_LOADSTORE only causes loads and stores to regular memory to be ordered, while MEMORY_BARRIER_ALL causes all loads and stores to be ordered. Thus, MEMORY_BARRIER_LOADSTORE is weaker than MEMORY_BARRIER_ALL.

11. Optimizing Accesses to Memory of Other Types

Some processors (such as PowerPCs) have special barrier instructions that can be used to efficiently order accesses to non-regular memory.

Here are the three main other memory types:

- Volatile memory: Non-cacheable, and speculative loads (including prefetching) are prohibited
- Uncacheable memory: Non-cacheable, but speculative loads (including prefetching), load combining, and store combining are permitted
- Write-through memory: Cacheable but write-through required

Let’s reuse the example of the device driver that is writing to a buffer, a buffer descriptor, and a device register, and assume that the device does not have hardware coherency with the processor's caches. The buffer is expected to reside in normal (writeback-cacheable) memory, so you will need to instruct the caches to write back any dirty blocks corresponding to the contents of the buffer (using some kind of FlushCaches() API). The buffer descriptor will be placed in some write-through memory so that stores to the buffer descriptor are coherent with the device. The device register must be mapped volatile.

The following code could be used to order the stores:

```
memcpy(buffer, user_data, len);
FlushCaches(buffer, len);
bd->valid = true;
MemoryBarrier(MEMORY_BARRIER_WT_STORE, MEMORY_BARRIER_VOLATILE_STORE);
*device_command_register = DEVICE_GO;
```

It is assumed here that FlushCaches() implicitly orders the flush before any subsequent memory access performed by the caller, so no barrier is required after the call to FlushCaches() before the store to the buffer descriptor.

If we go back to our FIFO example it should now look like this:

```
Error FifoGet(int *val)
{
    if(FifoWrite == FifoRead)
       /* FIFO is empty */
       return Failure;

    /* Read value from current FIFO slot */
    *val = FifoData[FifoRead % FIFO_SIZE];

    /* We need to make sure that the */
    /* FifoData slot is read before */
    /* we mark the FIFO slot as empty */
    MemoryBarrier(MEMORY_BARRIER_LOAD,
                  MEMORY_BARRIER_STORE);

    /* Increment read index */
    FifoRead++;

    return Success;
}
```

```
Error FifoPut(int val)
{
    if(FifoWrite + 1 == FifoRead)
       /* FIFO is full */
       return Failure;

    /* Store value in next FIFO slot */
    FifoData[FifoWrite % FIFO_SIZE] = val;

    /* We need to make sure that the */
    /* FifoData slot is written before */
    /* we mark the FIFO slot as full */
    MemoryBarrier(MEMORY_BARRIER_STORE,
                  MEMORY_BARRIER_STORE);

    /* Increment write index */
    FifoWrite++;

    return Success;
}
```

fig 5. Single-Reader Single-Writer Ring Buffer FIFO with fine-grain access ordering

In the above, we guarantee that the write to FifoRead/FifiWrite will happen after accessing the FifoData array. However, the compiler or the CPU may actualy optimize and decide to read FifoRead/FifiWrite before accessing FifoData if that is considered more efficient for the CPU.

12. Optimizing Atomic Operations

A successful atomic operation (such as TestAndSet() and AtomicSwap()) involves an atomic load and store to a value in regular memory (other memory types are not supported). In ordering the load and/or store with respect to other memory accesses, better performance can sometimes be achieved by informing MemoryBarrier() when a load and/or store is actually part of an atomic operation. This is because, very often the RTOS atomic primitives will themselves use some form of barrier. First, consider the case of ordering atomic operations in general (whether or not they are used for locks) with respect to other memory accesses. Consider the following example:

```
if (TestAndSet(&shared->state, 0, 1) == Success) {
    MemoryBarrier(MEMORY_BARRIER_ATOMIC_STORE, MEMORY_BARRIER_LOADSTORE);
    memcpy(state0_output_copy, shared->buffer, len);
    memcpy(shared->buffer, my_state2_input, len);
    MemoryBarrier(MEMORY_BARRIER_LOADSTORE, MEMORY_BARRIER_STORE);
    shared->state = 2;
}
```

Here, the program attempts to atomically transition the shared state variable from 0 to 1. Then, if it actually transitioned the shared state from 0 to 1, it accesses the buffer as in an earlier example, and finally sets the shared set to 2 (indicating to a device or another processor that the buffer has data valid for state 2).

The first MemoryBarrier() orders the prior store portion of the prior atomic operation before subsequent loads and stores to regular memory. This ensures that the accesses to the shared buffer are only performed while the shared state is actually 1.

In certain cases, it may also be correct to order only the load portion of the prior atomic operation before subsequent loads and stores to regular memory, which on some processors results in a more efficient barrier:

```
if (TestAndSet(&shared->state, 0, 1) == Success) {
    MemoryBarrier(MEMORY_BARRIER_ATOMIC_LOAD, MEMORY_BARRIER_LOADSTORE);
    memcpy(state0_output_copy, shared->buffer, len);
    memcpy(shared->buffer, my_state2_input, len);
    MemoryBarrier(MEMORY_BARRIER_LOADSTORE, MEMORY_BARRIER_STORE);
    shared->state = 2;
}
```

This barrier allows the processor to perform the accesses to the shared buffer before actually waiting for the transition of the shared state to be committed to the memory system. Because the accesses to the shared buffer are conditioned on the success of the TestAndSet() call, they are still not actually performed unless the shared state is about to be transitioned to 1.

However, this optimization is not correct if another processor might be reading the buffer while the shared state is 0. Specifically, the other processor might read the buffer and read the shared state as 0 and assume that the buffer contents are valid for state 0 (or for some earlier state), when in fact the above program may already have started modifying the buffer's contents.

13. Optimizing Lock Acquisition and Release

If an atomic operation is actually part of a lock acquire or lock release, even better performance can sometimes be achieved by informing MemoryBarrier() of this fact. Using the previous example, suppose that instead of a shared state you have a shared lock. A processor acquires the lock by transitioning (atomically) the lock value from 0 to 1, and releases the lock by transitioning it from 1 to 0. You could potentially improve performance by doing the following:

```
if (TestAndSet(&shared->lock, 0, 1) == Success) {
    MemoryBarrier(MEMORY_BARRIER_IMPORT_ATOMIC_LOAD, MEMORY_BARRIER_LOADSTORE);
    memcpy(state0_output_copy, shared->buffer, len);
    memcpy(shared->buffer, my_state2_input, len); MemoryBarrier(MEMORY_BARRIER_LOADSTORE,
        MEMORY_BARRIER_EXPORT_STORE);
    shared->lock = 0;
}
```

In this example, the first MemoryBarrier() creates an `import` barrier, while the second MemoryBarrier() creates an `export` barrier. An import barrier ensures that all shared data associated with a lock are imported at the point of the barrier, and it does this by ordering the load that sees the lock as being free before the accesses within the critical section. An export barrier ensures that all shared data accesses performed in the critical section are exported at the point of the barrier, and it does this by ordering the accesses within the critical section before the store that frees the lock.

14. Memory Barrier on SMP systems

On an SMP system, we have multiple cores. From any specific core point-of-view, other cores are just like peripherals. However, unlike most peripherals, cores in an SMP system have an identical cache architecture. It also means that the system is design for efficient core synchronization. Because most often cores have local cache(s) they will usually implement the MESI protocol (Modified Exclusive, Shared, Invalid, also known also as Illinois protocol due to its development at the University of Illinois at Urbana-Champaign) , a widely used cache coherency and memory coherence protocol. For a core to “see” memory changes from other cores, we only need to ensure that writes are committed to memory. After this the MESI protocol will ensure coherency between the caches and cores.

15. Non-blocking algorithms on SMP systems

For Non-blocking algorithms to work on SMP systems, we need to add the correct barriers at the various key points in the algorithm, much like we did for interacting with peripherals in the previous examples. However, those SMP-specific barriers are not required on non-SMP system and using barriers will have some impact on performance as the compiler and/or the processor will not be able to do there complete reordering optimizations. Also the barrier instructions will introduce “bubbles” in the CPU pipeline waiting for the barrier conditions to be met. Therefore, SMP barriers should only be used on SMP systems and removed on non-SMP systems for performance and code size reasons.

Practically, SMP barriers are identically to the previously described barriers when dealing with peripherals. We just need to have a way to enable those barriers only when required without having to rewrite our code.

This can be accomplished quite simply by creating a macro that would look somewhat like this:

```
#ifdef  NO_SMP
       /* We are sure we are running on a single core system */
       /* So we don’t need SMP support */
#define SMPMemoryBarrier(x, y)
#else
       /* We may be running on an SMP system */
       /* So let’s add SMP support */
       #define SMPMemoryBarrier(x, y)       MemoryBarrier(x,y)
#endif
```

We can now make our code SMP-safe and SMP-efficient:

```
Error FifoGet(int *val)
{
    if(FifoWrite == FifoRead)
       /* FIFO is empty */
       return Failure;

    /* Read value from current FIFO slot */
    *val = FifoData[FifoRead % FIFO_SIZE];

    /* We need to make sure that the */
    /* FifoData slot is read before */
    /* we mark the FIFO slot as empty */
    MemoryBarrier(MEMORY_BARRIER_LOAD,
                  MEMORY_BARRIER_STORE);

    /* Increment read index */
    FifoRead++;

    /* We should inform the other */
    /* cores that a slot was freed */
    /* This will force the above write */
    /* to be commited */
    SMPMemoryBarrier(MEMORY_BARRIER_STORE,
MEMORY_BARRIER_ALL);

    return Success;
}
```

```
Error FifoPut(int val)
{
    if(FifoWrite + 1 == FifoRead)
       /* FIFO is full */
       return Failure;

    /* Store value in next FIFO slot */
    FifoData[FifoWrite % FIFO_SIZE] = val;

    /* We need to make sure that the */
    /* FifoData slot is written before */
    /* we mark the FIFO slot as full */
    MemoryBarrier(MEMORY_BARRIER_STORE,
                  MEMORY_BARRIER_STORE);

    /* Increment write index */
    FifoWrite++;

    /* We should inform the other */
    /* cores that new data is available */
    /* This will force the above write */
    /* to be commited */
    SMPMemoryBarrier(MEMORY_BARRIER_STORE,
MEMORY_BARRIER_ALL);

    return Success;
}
```

fig 6. SMP Single-Reader Single-Writer Ring Buffer FIFO

An alternative and probably better implementation may take advantage of the RTOS atomic operations, which should be SMP-aware if required:

```
Error FifoGet(int *val)
{
    if(FifoWrite == FifoRead)
       /* FIFO is empty */
       return Failure;

    /* Read value from current FIFO slot */
    *val = FifoData[FifoRead % FIFO_SIZE];

    /* We need to make sure that the */
    /* FifoData slot is read before */
    /* we mark the FIFO slot as empty */
    MemoryBarrier(MEMORY_BARRIER_LOAD,
              MEMORY_BARRIER_ATOMIC_STORE);

    /* Increment write index */
    AtomicIncrement(&FifoRead);

    /* The correct SMP-barrier should be */
    /*  part of the Atomic operation */
    /* implementation in order to */
    /* guarantee system-wide atomicity */

    return Success;
}
```

```
Error FifoPut(int val)
{
    if(FifoWrite + 1 == FifoRead)
       /* FIFO is full */
       return Failure;

    /* Store value in next FIFO slot */
    FifoData[FifoWrite % FIFO_SIZE] = val;

    /* We need to make sure that the */
    /* FifoData slot is written before */
    /* we mark the FIFO slot as full */
    MemoryBarrier(MEMORY_BARRIER_STORE,
              MEMORY_BARRIER_ATOMIC_STORE);

    /* Increment write index */
    AtomicIncrement(&FifoWrite);

    /* The correct SMP-barrier should be */
    /*  part of the Atomic operation */
    /* implementation in order to */
    /* guarantee system-wide atomicity */

    return Success;
}
```

fig 7. SMP Single-Reader Single-Writer Ring Buffer FIFO using Atomic primitives

16. Conclusion

We have seen that it is possible using the correct atomic and barrier primitives to write efficient lock-free algorithms even on SMP-systems. It is also possible to write CPU independent and nearly optimal code if the underlying RTOS provides a rich set of API implementing various atomic operations and fine-grained barriers. In practice real processor barrier implementations are rarely if ever as fine-grained as the above examples may imply. However, providing a rich barrier semantic usually allows the RTOS, with the help of the right optimizing compiler, to map precisely this abstract semantic to the actual underlying instruction set architecture, nearly avoiding any expensive runtime decisions on the type of barrier to use.

It is also interesting to note that algorithms that use Mutexes, Semaphores, and/or Critical Sections correctly do usually work well on SMP systems even though usually no special care is taken to use the correct barrier operations. The reason is that the RTOS will implement the barriers in the implementation of those synchronization primitives. But because the RTOS implementer has no prior knowledge about the semantic of the code using those primitives, the implementer will often have to default to the strongest barriers even though they can be quite expensive, especially on higher-end processor with sophisticated pipeline and cache architectures. This emphasize the need to use portable SMP-ready non-blocking algorithms in order to achieve the highest throughput and determinism.