Real-Time 101

Lockless synchronization and memory accesses in multi-core systems

A **brief** introduction

Charles Steinkuehler

Machinekit
Moves. Controls. Things.

Agenda

- Brief history of CPU development
- Basic synchronization methods
- Problems caused by modern systems
- Compiler barriers
- Memory fences
- Cache coherency / thrashing
- Questions

How we got here

A brief history of CPU development

- In the beginning (microprocessors)
  - One thing at a time, everything in order
- The need for speed (caches)
  - Now “real” memory doesn't match “logical” memory
- Multiple bus masters (DMA and multi-core CPUs)
  - What happens when more than one agent is accessing memory?

Basic Synchronization Methods

- Atomic access (asm, C11, gcc, libraries)
  - Simple, the foundation of all other methods
  - Up to 64-bit data types on modern CPUs
- Spinlocks / Locks / Semaphores / Critical sections
- Request / Acknowledge (4-step 'dance')
- Ring buffer / Queue / FIFO
- Others

Ring buffer basics

- One writer, one reader, zero locks
- Write:
  - Check for space (compare read and write pointers)
  - Write data into buffer
  - Update write pointer
- Read:
  - Check for data available (compare read and write pointers)
  - Read data from buffer
  - Update read pointer
- Safe operation if read or write pointer is “old”
- Lock free operation assuming atomic pointer updates!
- What could possibly go wrong?

Problems – What can go wrong

- Various corner cases result in incorrect logic (use a library!)
  - Non power-of-two size, full bit, etc.
- Memory access ordering changed by:
  - Compiler optimizations changing load/store order
  - Speculative reads
  - Read promotions (posted writes)
  - Hardware optimizations
    - DRAM bank / page optimizations
    - Multichannel memory
    - Path congestion, etc.

Barriers and Fences

- Guarantee required ordering
- Vary in details and “expense” by architecture from very light-weight to “invalidate cache”
- Compiler barrier – optimization boundary only
- Guarantee ordering of memory accesses
- Guarantee completion of memory accesses
- You can specify direction (r/w) and domain (advanced!)

Ring buffer basics revisited

- Write:
  - Check for space (compare read and write pointers)
  - Write data to buffer
  - **Memory fence!** Insure all data is written before the write pointer is updated!
  - Update write pointer

- Read:
  - Check for data available (compare read and write pointers)
  - Read data from buffer
  - **Memory fence!** Insure all data is read before the read pointer is updated!
  - Update read pointer

- Other tweaks (add fence to insure pointer updates are immediately pushed out to memory)

Cache effects

- Each CPU core typically has independent caches
- Caches negotiate for “ownership” of chunks of data, typically a complete cache line
- Data alignment and access patterns can significantly impact performance
- Single core: Pack data to fit as much as possible into cache
  - Can cause horrible performance on multi-core systems!
- Multi-core: Insure data written by different cores resides in different cache lines

References

- Linux kernel memory barrier documentation
  https://www.kernel.org/doc/Documentation/memory-barriers.txt

- From Weak to Weedy: Effective Use of Memory Barriers in the ARM Linux Kernel
  presentation by W. Deacon, ARM
  https://www.youtube.com/watch?v=6ORn6_35kKo

- Slides for above W. Deacon presentation
  http://elinux.org/images/7/73/Deacon-weak-to-weedy.pdf

- Ordering and Barriers:
  Linux Kernel Development Second Edition Ch. 9, Sec. 10
  http://www.makelinux.net/books/lkd2/ch09lev1sec10.html

- Linus on barrier()
  http://yarchive.net/comp/linux/memory_barriers.html

License