Introduction to Locks

Computer Science 432 — Lecture 11 — Duane Bailey

March 16, 2022

Announcements

- Work continues on Lab 5, the COW optimization. Due on Friday.
- Small group D will meet tomorrow, at the usual time. Topic: Lab 3.
- Office Hours (this week, only): W1-4, F9-10:30
- O/S Conference topic contracts available. Return by Wednesday, April 6. Happy to discuss.
- April 6/7: SEM Microscope lab during small group meetings (optional, but: really?).

(From Monday:) Timer Interrupt Handling

- Scratch area set off for machine-mode interrupts in `start()` (`kernel/start.c`):
  - Saves only a few registers
  - Sets the machine interrupt vector to `timervec` (`kernel/kernelvec.c`)

- Periodically, a timer goes off, reminding the machine to swap tasks.
  - Interrupts are machine-level.
  - Very simple approach
    - Schedule another timer event
    - Raise a *software interrupt* (landing in trap.c's `devintr`).
  - Allows kernel to carefully control scheduling, through `yield()`

Critical Sections

- There is concurrency in xv6. For example, we have three processors.
- Sometimes you're interested in *exclusively* executing sections of code that are *critical*. Examples:
  - Modifying the free list in the kernel page allocator.
  - Modifying the struct proc structure in proc.c
- One approach to managing access to critical sections is the use of *locks*.
- Failure to appropriately manage access to critical sections could lead to:
  - A race condition.
  - Deadlock.
  - Livelock.
  - Starvation.
- Today, we'll look at lock design.

Sharing Memory Between Processors

- The memory in xv6 is shared among all cpu's/processors:
  - Every processor can see all of memory.
  - Every processor can execute the kernel.
  - Processes can freely move between processors, over time.

- RISC-V has a number of instructions that support *atomic access* to memory:
  - Atomic operators, "AMOs".
  - Load-reserved, store-conditional operators, "LR/SC".

Atomicity Operators

- By definition, the atomicity operators:
  - Are complex. You might attempt to do the same things with 2 or more instructions.
  - Access memory. Since internal state of the CPU is not shared, the only concern is memory.
  - Guaranteed atomicity. Concurrent instructions that involve AMOs guarantee the outcome is the same as executing the AMO before or after the other instructions.
  - In short: the AMOs act like operations that happen directly in memory.

- Example: amoswap rd, rs1, rs2 (Similar to: csrrw rd, rs1, rs2, but involving memory.)
  - rd is assigned value at memory pointed to bey rs1.
  - memory at rs1 gets value in rs2.

- AMO guarantees require consistency protocols in the cache.
  - L1$ and L2$ are dedicated to the CPU.
  - L3$ is shared. This level of cache is responsible for managing atomic access to memory.

Load-Reserved/Store-Conditional

- The load-reserved (LR) and store-conditional (SC) operations work together:
  - LR loads a value from memory location and places a “reservation” on that address
  - SC stores a value to a memory location, *but only if there is a reservation there.*
  - Regular loads and stores clear the reservation at their addresses.

The Atomicity Axiom

Volume I: RISC-V Unprivileged ISA V20191213

A.3.3 Atomicity Axiom

> Atomicity Axiom (for Aligned Atomics): If r and w are paired load and store operations generated by aligned LR and SC instructions in a hart h, s is a store to byte x, and r returns a value written by s, then s must precede w in the global memory order, and there can be no store from a hart other than h to byte x following s and preceding w in the global memory order.

The Progress Axiom

A.3.4 Progress Axiom

> Progress Axiom: No memory operation may be preceded in the global memory order by an infinite sequence of other memory operations.

Implementing Compare-and-Swap with LR/SC

- An example atomic operation, built with LR/SC.
- `compareAndSwap(a0,a1,a2)` checks to see if `*a0 == a1` and, if so, sets `*a0 = a2` and returns `0` (success). Otherwise, returns `1`.

```
    # a0 holds address of memory location
    # a1 holds expected value
    # a2 holds desired value
    # a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0)          # Load original value.
    bne t0, a1, fail       # Doesn't match, so fail.
    sc.w t0, a2, (a0)      # Try to update.
    bnez t0, cas           # Retry if store-conditional failed.
    li a0, 0               # Set return to success.
    jr ra                  # Return.
fail:
    li a0, 1               # Set return to failure.
    jr ra                  # Return.
```

Figure 8.1: Sample code for compare-and-swap function using LR/SC.