CS211
Advanced Computer Architecture
L16 Synchronization

Chundong Wang
November 11th, 2021
Previously in CS211

• Memory Consistency Model (MCM) describes what values are legal for a load to return

• Sequential Consistency is most intuitive model, but almost never implemented in actual hardware
  • Single global memory order where all individual thread memory operations appear in local program order

• Stronger versus Weaker MCMs
  • TSO is strongest common model, allows local hardware thread to see own stores before other hardware threads, but otherwise no visible reordering
  • PSO

• Fences are used to enforce orderings within local thread, suffice for TSO and weak memory models
Multi-Copy Atomic models

- Each hardware thread must view its own memory operations in program order, but can buffer these locally and reorder accesses around the buffer.
- But once a local store is made visible to one other hardware thread in the system, all other hardware threads must also be able to observe it (this is what is meant by “atomic”)

![Diagram of Multi-Copy Atomic models]

- Point of global visibility
Hierarchical Shared Buffering

- Common in large systems to have shared intermediate buffers on path between CPUs and global memory
- Potential optimization is to allow some CPUs see some writes by a CPU before other CPUs
- Shared memory stores are **not** seen to happen atomically by other threads (non multi-copy atomic)
Non-Multi-Copy Atomic

Initially $X = Y = 0$

P1: 
li x1, 1
sw x1, X

P2: 
lw x1, X
sw x1, Y

P3: 
lw x1, Y
fence r,r
lw x2, X

Can P3.x1 = 1, and P3.x2 = 0?

• In general, Non-MCA is very difficult to reason about
• Software in one thread cannot assume all data it sees is visible to other threads, so how to share data structures?
• Adding local fences to require ordering of each thread’s accesses is insufficient – need a more global memory barrier to ensure all writes are made visible
Conflicting data accesses

• Two memory accesses by different processors **conflict** if
  • They access to the same memory location
  • At least one is a write
  • Unordered by synchronization operations

• Unsynchronized program
  • Conflicting accesses not ordered by synchronization (e.g., a fence, operation with release/acquire semantics, barrier, etc.)
  • Unsynchronized programs contain **data races**: the output of the program depends on relative speed of processors (non-deterministic program results)
Synchronized programs

• Synchronized programs yield SC results on non-SC systems
  • Synchronized programs are data-race-free

• If there are no data races, reordering behavior doesn’t matter
  • Accesses are ordered by synchronization, and synchronization forces sequential consistency

• In practice, most programs you encounter will be synchronized (via locks, barriers, etc. implemented in synchronization libraries)
  • Very few programmers do programming that relies on SC
  • Rather than via ad-hoc reads/writes to shared variables like in the example programs
Relaxed Memory Models

• Motivation
  • To obtain higher performance by allowing reordering of memory operations (reordering is not allowed by sequential consistency)

• Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies where needed

• Which dependencies are dropped depends on the particular memory model
  • IBM370, TSO, PSO, WO, PC, Alpha, RMO, ...
  • Some ISAs allow several memory models, some machines have switchable memory models

• How to introduce needed dependencies varies by system
  • Explicit FENCE instructions (sometimes called sync or memory barrier instructions)
  • Implicit effects of atomic memory instructions

How on earth are programmers supposed to work with this????
Do not forget Compiler and Language!

• Compiler can reorder/remove memory operations:
  • Instruction scheduling, move loads before stores if to different address
  • Register allocation, cache load value in register, don’t check memory
• Prohibiting these optimizations would result in very poor performance

```python
//Thread 1
X = 0
For i in (1:100):
    X = 1
    Print X
111111111111...  # Thread 1
111111011111...  # Thread 2

//Thread 2
X = 0
For i in (1:100):
    Print X
111111111111...  # Thread 2
111111000000...  # Thread 2
```
Language-Level Memory Models

• Programming languages have memory models too
• Hide details of each ISA’s memory model underneath language standard
  • c.f. C function declarations versus ISA-specific subroutine linkage convention
• Language memory models: C/C++, Java
  • Modern (C11, C++11) and not-so-modern (Java 5) languages guarantee sequential consistency for data-race-free programs (“SC for DRF”)
  • Compilers will insert the necessary synchronization to cope with the hardware memory model
• Describe legal behaviors of threaded code in each language and what optimizations are legal for compiler to make
• E.g., C11/C++11: `atomic_load(memory_order_seq_cst)` maps to RISC-V `fence rw,rw; lw; fence r,rw`
Release Consistency [Garachorloo 1990]

- Observation that consistency only matters when processes communicate data
- Only need to have consistent view when one process shares its updates to other processes
- Other processes only need to ensure they receive updates after they acquire access to shared data
Release Consistency Adopted

• Only care about inter-processor memory ordering at thread synchronization points, not in between
• Can treat all synchronization instructions as the only ordering points

• Memory model for C/C++ and Java uses release consistency
• Programmer has to identify synchronization operations, and if all data accesses are protected by synchronization, appears like SC to programmer

• ARMv8 and RISC-V ISA adopt release consistency semantics on atomic memory operations (AMOs)
  • AMOs, such as compare-and-swap, fetch-and-add, etc. can be used to implement lock- and wait-free algorithms and data structures
  • Lock-free algorithms are supposed to allow an arbitrary number of threads to share a resource without the need for serial execution on a lock
Recap: Synchronization

The need for synchronization arises whenever there are concurrent processes in a system (even in a uniprocessor system).

Two classes of synchronization:

• **Producer-Consumer:** A consumer process must wait until the producer process has produced data

• **Mutual Exclusion:** Ensure that only one process uses a resource at a given time
Simple Mutual-Exclusion Example

// Both threads execute:
ld data, (data_p)
add data, 1
sd data, (data_p)

Is this correct?
Mutual Exclusion Using Load/Store (assume SC)
A protocol based on two shared variables c1 and c2. Initially, both c1 and c2 are 0 (not busy)

Process 1

...  
c1=1;
L: if c2=1 then go to L  
< critical section>
c1=0;

Process 2

...  
c2=1;
L: if c1=1 then go to L  
< critical section>
c2=0;

What is wrong?
Mutual Exclusion: second attempt

To avoid deadlock, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

• Deadlock is not possible but with a low probability a livelock may occur.

• An unlucky process may never get to enter the critical section ⇒ starvation
A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and turn. Initially, both c1 and c2 are 0 (not busy)

Process 1

... 
c1=1;
turn = 1;
L: if c2=1 & turn=1
then go to L
< critical section>
c1=0;

Process 2

... 
c2=1;
turn = 2;
L: if c1=1 & turn=2
then go to L
< critical section>
c2=0;

• turn = i ensures that only process i can wait
• variables c1 and c2 ensure mutual exclusion

Solution for n processes was given by Dijkstra and is quite tricky!
Analysis of Dekker’s Algorithm

Scenario 1

Process 1
\[
\begin{align*}
  c1 &= 1; \\
  \text{turn} &= 1; \\
  L: & \text{ if } c2 = 1 \text{ & turn} = 1 \\
  & \text{ then go to } L \\
  & \text{< critical section>} \\
  c1 &= 0;
\end{align*}
\]

Process 2
\[
\begin{align*}
  c2 &= 1; \\
  \text{turn} &= 2; \\
  L: & \text{ if } c1 = 1 \text{ & turn} = 2 \\
  & \text{ then go to } L \\
  & \text{< critical section>} \\
  c2 &= 0;
\end{align*}
\]

Scenario 2

Process 1
\[
\begin{align*}
  c1 &= 1; \\
  \text{turn} &= 1; \\
  L: & \text{ if } c2 = 1 \text{ & turn} = 1 \\
  & \text{ then go to } L \\
  & \text{< critical section>} \\
  c1 &= 0;
\end{align*}
\]

Process 2
\[
\begin{align*}
  c2 &= 1; \\
  \text{turn} &= 2; \\
  L: & \text{ if } c1 = 1 \text{ & turn} = 2 \\
  & \text{ then go to } L \\
  & \text{< critical section>} \\
  c2 &= 0;
\end{align*}
\]
ISA Support for Mutual-Exclusion Locks

- Regular loads and stores in SC model (plus fences in weaker model) sufficient to implement mutual exclusion, but code is inefficient and complex
- Therefore, atomic read-modify-write (RMW) instructions added to ISAs to support mutual exclusion

- Many forms of atomic RMW instruction possible, some simple examples:
  - Test and set (reg_x = M[a]; M[a]=1)
  - Swap (reg_x=M[a]; M[a] = reg_y)
Test-and-set, compare-and-swap

• Test-and-set
  • A single atomic operation
  • Write `1’ (set) to a memory location and return its old value
  • If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished

• Compare-and-swap
  • A single atomic operation
  • Compare the contents of a memory location with a given value and, only if they are the same, modify the contents of that memory location to a new given value
Lock for Mutual-Exclusion Example

// Both threads execute:
li one, 1 // lock is 0 initially

spin: amoswap lock, one, (lock_p)
  bnez lock, spin

ld data, (data_p)
add data, 1
sd data, (data_p)

sd x0, (lock_p)

Assumes SC memory model
Lock for Mutual-Exclusion with Relaxed MM

```c
// Both threads execute:
li one, 1  // lock is 0 initially

spin: amoswap lock, one, (lock_p)
    bnez lock, spin
    fence r,rw

ld data, (data_p)
add data, 1
sd data, (data_p)

fence rw,w
sd x0, (lock_p)
```

Acquire Lock

Critical Section

Release Lock
RISC-V Atomic Memory Operations

- Atomic Memory Operations (AMOs) have two ordering bits:
  - Acquire (aq)
  - Release (rl)

- If both clear, no additional ordering implied
- If aq set, then AMO “happens before” any following loads or stores
  - No later memory operations in this RISC-V hardware thread can be observed to take place before the AMO
- If rl set, then AMO “happens after” any earlier loads or stores
  - Other RISC-V hardware threads will not observe the AMO before memory accesses preceding the AMO in this RISC-V hardware thread
- If both aq and rl set, then AMO happens in program order
Lock for Mutual-Exclusion using RISC-V AMO

// Both threads execute:
  li one, 1

spin: amoswap.w.aq lock, one, (lock_p)
  bnez lock, spin

ld data, (data_p)
  add data, 1
  sd data, (data_p)

amoswap.w.rl x0, x0, (lock_p)
RISC-V FENCE versus AMO.aq/rl

sd x1, (a1) # Unrelated store
ld x2, (a2) # Unrelated load
li t0, 1
again:
amoswap.w.aq t0, t0, (a0)
bnez t0, again
# ...
# critical section
# ...
amoswap.w.rl x0, x0, (a0)
sd x3, (a3) # Unrelated store
ld x4, (a4) # Unrelated load

sd x1, (a1) # Unrelated store
ld x2, (a2) # Unrelated load
li t0, 1
again:
amoswap.w t0, t0, (a0)
fence r, rw
bnez t0, again
# ...
# critical section
# ...
fence rw, w
amoswap.w x0, x0, (a0)
sd x3, (a3) # Unrelated store
ld x4, (a4) # Unrelated load

AMOs only order the AMO w.r.t. other loads/stores/AMOs
FENCEs order every load/store/AMO before/after FENCE
Executing Critical Sections without Locks

• If a software thread is descheduled after taking lock, other threads cannot make progress inside critical section
• “Non-blocking” synchronization allows critical sections to execute atomically without taking a lock
Nonblocking Synchronization

\[
\text{Compare}\&\text{Swap}(m), R_t, R_s:
\]
\[
\text{if } (R_t == M[m])
\]
\[
\text{then } \text{M}[m] = R_s;
\]
\[
R_s = R_t;
\]
\[
\text{status } \leftarrow \text{success};
\]
\[
\text{else status } \leftarrow \text{fail};
\]

\text{status is an } \text{implicit argument}

\text{try: Load } R_{\text{head}}, (\text{head})
\text{spin: Load } R_{\text{tail}}, (\text{tail})
\text{if } R_{\text{head}} == R_{\text{tail}} \text{ goto spin}
\text{Load } R, (R_{\text{head}})
\text{R}_{\text{newhead}} = R_{\text{head}} + 1
\text{Compare}\&\text{Swap}(\text{head}), R_{\text{head}}, \text{R}_{\text{newhead}}
\text{if (status == fail) goto try}
\text{process}(R)
Compare-and-Swap Issues

• Compare and Swap is a complex instruction
  • Three source operands: address, comparand, new value
  • One return value: success/fail or old value

• ABA problem
  • Load(A), Y=process(A), success=CAS(A,Y)
  • What if another task switched A to B, then back to A before process() finished?

• Add a counter, and make CAS access two words

• Double Compare and Swap
  • Five source operands: one address, two comparands, two values
  • Load(<A1,A2>), Z=process(A1), success=CAS(<A1,A2>,<Y,A2+1>)
Load-linked & Store-conditional

• Also known as load-reserved/store-conditional (LR/SC)
  • e.g., RISC-V
  • Or load-locked, or load-link

• Load-linked returns the current value of a memory location

• A subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-link

• If any updates have occurred, the store-conditional is guaranteed to fail, even if the value read by the load-linked has since been restored
Load-reserved & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserved $R, (m)$:
\[
\langle \text{flag}, \text{adr} \rangle \leftarrow \langle 1, m \rangle;
R \leftarrow M[m];
\]

Store-conditional $(m), R$:
\[
\text{if } \langle \text{flag}, \text{adr} \rangle =\langle 1, m \rangle \text{ then cancel other procs’ reservation on } m;
M[m] \leftarrow R;
\text{status} \leftarrow \text{succeed};
\text{else status} \leftarrow \text{fail};
\]

```
try:
```
Load-reserved $R_{\text{head}}, (\text{head})$
Load $R_{\text{tail}}, (\text{tail})$
if $R_{\text{head}}==R_{\text{tail}}$ goto spin
Load $R, (R_{\text{head}})$
$R_{\text{head}} = R_{\text{head}} + 1$
Store-conditional $(\text{head}), R_{\text{head}}$
if (status==fail) goto try
process($R$)
```

```
spin:
```
try: Store-conditional (m), R:
```
Load-Reserved/Store-Conditional using MESI Caches

Load-Reserved ensures line in cache in **Exclusive/Modified** state
Store-Conditional succeeds if line still in **Exclusive/Modified** state

*(In practice, this implementation only works for smaller systems)*
LR/SC Issues

• LR/SC does not suffer from ABA problem, as any access to addresses will clear reservation regardless of value
  • CAS only checks stored values not intervening accesses
• LR/SC non-blocking synchronization can livelock between two competing processors
  • CAS guaranteed to make forward progress, as CAS only fails if some other thread succeeds
• RISC-V LR/SC makes guarantee of forward progress provided code inside LR/SC pair obeys certain rules
  • Can implement CAS inside RISC-V LR/SC
RISC-V Atomic Instructions

• Non-blocking “Fetch-and-op” with guaranteed forward progress for simple operations, returns original memory value in register
• AMOSWAPM[a] = d
• AMOADD M[a] += d
• AMOAND M[a] &= d
• AMOOR M[a] |= d
• AMOXOR M[a] ^= d
• AMOMAX M[a] = max(M[a],d)  # also, unsigned AMOMAXU
• AMOMIN M[a] = min(M[a],d)  # also, unsigned AMOMINU
Conclusion

• Memory Consistency

• Synchronization
Acknowledgements

• These slides contain materials developed and copyright by:
  • Prof. Krste Asanovic (UC Berkeley)
  • Prof. Daniel Sanchez (MIT)
  • Prof. Kayvon Fatahalian (Stanford)
  • Prof. Kunle Olukotun (Stanford)
  • Prof. Stan Scott (Queen’s University Belfast)