CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 20 Synchronization

Krsti Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152
Last Time in Lecture 19

- 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
  - Weak multi-copy atomic model allows more reordering provided when a store is made visible to other threads, all threads can “see” at same time
  - Very weak non-multi-copy atomic model allows stores from one thread to be observed in different orders by remote threads

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

- Heavyweight barriers are needed for non-multi-copy atomic, across multiple hardware threads
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 xdata, (xdatap)
add xdata, 1
sd xdata, (xdatap)

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;

What is wrong?  Deadlock!

Process 2

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

Scenario 2

... 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;
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)
Lock for Mutual-Exclusion Example

// Both threads execute:

li xone, 1

spin: amoswap xlock, xone, (xlockp)  
      bnez xlock, spin  
      ld xdata, (xdatap)  
      add xdata, 1  
      sd xdata, (xdatap)  
      sd x0, (xlockp)

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

Both threads execute:

```c
li xone, 1
spin: amoswap xlock, xone, (xlockp)
bnez xlock, spin
  fence r,rw
ld xdata, (xdatap)
  add xdata, 1
  sd xdata, (xdatap)
fence rw,w
sd x0, (xlockp)
```

- **Acquire Lock**
- **Critical Section**
- **Release Lock**
CS152 Administrivia

- Midterm 2 in class Wednesday April 11
  - covers lectures 10-17, plus associated problem sets, labs, and readings
- PS 5 out on Wednesday April 11
- Lab 5 in section on Friday April 13
CS252 Administrivia

- Monday April 9th Project Checkpoint, 4-5pm, 405 Soda
  - Prepare 10-minute presentation on current status
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
- If rl set, then AMO “happens after” any earlier loads or stores
- If both aq and rl set, then AMO happens in program order
Lock for Mutual-Exclusion using RISC-V AMO

// Both threads execute:
li xone, 1

spin: amoswap.w.aq xlock, xone, (xlockp)
bnez xlock, spin  

Critical Section

ld xdata, (xdatap)
add xdata, 1
sd xdata, (xdatap)

Release Lock

amoswap.w.rl x0, x0, (xlockp)
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
FENCEes order every load/store/AMO before/after FENCE

AMOs only order the AMO w.r.t. other loads/stores/AMOs
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

Compare&Swap(m), R_t, Rs:
if (R_t == M[m])
    then M[m] = Rs;
    Rs = R_t;
    status ← success;
else status ← fail;

status is an implicit argument

try:
    Load R_head, (head)
    Load R_tail, (tail)
    if R_head == R_tail goto spin
    Load R, (R_head)
    R_newhead = R_head + 1
    Compare&Swap(head), R_head, R_newhead
    if (status == fail) goto try
    process(R)
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 different 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-reserve & Store-conditional

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

Try:

Load-reserve \( R, (m) \):
\[
<\text{flag}, \text{adr}> \leftarrow <1, m>;
R \leftarrow M[m];
\]

Spin:

Load-reserve \( R_{\text{head}}, (\text{head}) \)
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 (head), \( R_{\text{head}} \)
if (status==fail) goto try
process(R)

Store-conditional (m), R:
\[
\text{if } <\text{flag}, \text{adr}> == <1, m> \text{ then}
\text{ cancel other procs’ reservation on } m; \\
M[m] \leftarrow R; \\
\text{ status } \leftarrow \text{succeed}; \\
\text{ else status } \leftarrow \text{ fail};
\]
Load-Reserved ensures line in cache in Exclusive/Modified state
Store-Conditional succeeds if line still in Exclusive/Modified state
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
  - AMOSWAP \( M[a] = d \)
  - AMOADD \( M[a] += d \)
  - AMOAND \( M[a] &= d \)
  - AMOOR \( M[a] |= d \)
  - AMOXOR \( M[a] ^= d \)
  - AMOMAX \( M[a] = \max(M[a],d) \)
  - AMOMIN \( M[a] = \min(M[a],d) \)
Transactional Memory

- Proposal from Knight ['80s], and Herlihy and Moss ['93]
  XBEGIN
  MEM-OP1
  MEM-OP2
  MEM-OP3
  XEND
- Operations between XBEGIN instruction and XEND instruction either all succeed or are all squashed
- Access by another thread to same addresses, cause transaction to be squashed
- More flexible than CAS or LR/SC
- Commercially deployed on IBM POWER8 and Intel TSX extension
Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)