Lecture 22 Synchronization

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

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152
Recap: 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
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 xdata, (xdatap)
add xdata, 1
sd xdata, (xdatap)

Is this correct?
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? **Deadlock!**
Mutual Exclusion: \textit{second attempt}

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

- Deadlock is not possible but with a low probability a \textit{livelock} may occur.

- An unlucky process may never get to enter the critical section $\Rightarrow$ \textit{starvation}

```plaintext
Process 1

... 
L: \texttt{c1}=1;  
\textit{if} \texttt{c2}=1 \textit{then}  
\{ \texttt{c1}=0; \textit{go to L}\}  
< critical section>  
\texttt{c1}=0

Process 2

... 
L: \texttt{c2}=1;  
\textit{if} \texttt{c1}=1 \textit{then}  
\{ \texttt{c2}=0; \textit{go to L}\}  
< critical section>  
\texttt{c2}=0
```
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  
them go to L  
< critical section>  
c1=0;  

Scenario 2

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

Process 2

...  
c2=1;  
turn = 2;  
L: if c1=1 & turn=2  
them 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
// Both threads execute:
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)
CS152 Administrivia

- Lab 5 due on Star Wars day (May the 4th)
- Midterm grades coming soon
  - Will have one week for regrade requests
CS252 Administrivia

- Monday April 27th Project Checkpoint
  - Schedule 10-minute Zoom calls during discussion period
  - Please have status slides, what’s completed, what’s left to do
**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

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

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

dx 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

Compare&Swap(m), R\textsubscript{t}, R\textsubscript{s}:

\begin{align*}
\text{if } (R\textsubscript{t}==M[m]) \\
\quad \text{then } M[m]=R\textsubscript{s}; \\
\quad R\textsubscript{s}=R\textsubscript{t}; \\
\quad \text{status } \leftarrow \text{success}; \\
\text{else } \text{status } \leftarrow \text{fail};
\end{align*}

status is an implicit argument

try:
\begin{align*}
\text{Load } R\textsubscript{head}, (\text{head}) \\
\text{Load } R\textsubscript{tail}, (\text{tail}) \\
\text{if } R\textsubscript{head}== R\textsubscript{tail} \text{ goto spin} \\
\text{Load } R, (R\textsubscript{head}) \\
R\textsubscript{newhead} = R\textsubscript{head}+1 \\
\text{Compare&Swap(head), } R\textsubscript{head}, R\textsubscript{newhead} \\
\text{if (status==fail) goto try} \\
\text{process(R)}
\end{align*}

spin:
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 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

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

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

try:

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

spin:
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
  - AMOSWAP $M[a] = d$
  - AMOADD $M[a] += d$
  - AMOAND $M[a] &= d$
  - AMOOR $M[a] |= d$
  - AMOXOR $M[a] ^= d$
  - AMOMAX $M[a] = \text{max}(M[a],d)$  
    \hspace{1cm} # also, unsigned AMOMAXU
  - AMOMIN $M[a] = \text{min}(M[a],d)$  
    \hspace{1cm} # also, unsigned AMOMINU
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)