CS211
Advanced Computer Architecture
L17 Synchronization

Chundong Wang
November 30th, 2020
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
  - 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 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 \(\Rightarrow\) starvation

**Process 1**

\[
\begin{align*}
L_1 & : \ c_1 = 1; \\
& \text{if } c_2 = 1 \text{ then} \\
& \{ \ c_1 = 0; \ \text{go to } L_1 \} \\
& < \text{critical section}> \\
& c_1 = 0
\end{align*}
\]

**Process 2**

\[
\begin{align*}
L_2 & : \ c_2 = 1; \\
& \text{if } c_1 = 1 \text{ then} \\
& \{ \ c_2 = 0; \ \text{go to } L_2 \} \\
& < \text{critical section}> \\
& c_2 = 0
\end{align*}
\]
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

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

Process 2

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

- 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)
// Both threads execute:

li one, 1 %%% lock is 0 initially

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

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

sd x0, (lock_p)  Release Lock

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

// 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

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

**try:**
- Load \(R_{head}\), (head)

**spin:**
- 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)**

status is an implicit argument
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

• 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):
<flag, adr> ← <1, m>;
R ← M[m];

Store-conditional (m), R:
if <flag, adr> == <1, m>
then cancel other procs’ reservation on m;
M[m] ← R;
status ← succeed;
else status ← fail;

try:
Load-reserved R_{head}, (head)
Load R_{tail}, (tail)
if R_{head} == R_{tail} goto spin
Load R, (R_{head})
R_{head} = R_{head} + 1
Store-conditional (head), R_{head}
if (status==fail) goto try
process(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
Transactional Memory (TM)

- 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
Programing with transactions

void deposit(Acct account, int amount) {
    lock(account.lock);
    int tmp = bank.get(account);
    tmp += amount;
    bank.put(account, tmp);
    unlock(account.lock);
}

void deposit(Acct account, int amount) {
    XBEGIN;
    int tmp = bank.get(account);
    tmp += amount;
    bank.put(account, tmp);
    XEND;
}

• Programmer states what to do (maintain atomicity of this code), not how to do
  • No explicit use or management of locks

• System implements synchronization as necessary to ensure atomicity
  • System could implement XBEGIN/XEND using locks
  • Implementation discussed today uses optimistic concurrency: maintain serialization only in situations of true contention (R-W or W-W conflicts)
TM

• Memory transaction
  • An atomic and isolated sequence of memory accesses
  • Inspired by database transactions

• Atomicity (all or nothing)
  • Upon transaction commit, all memory writes in transaction take effect at once
  • On transaction abort, none of the writes appear to take effect (as if transaction never happened)

• Isolation
  • No other processor can observe writes before transaction commits

• Serializability
  • Transactions appear to commit in a single serial order
  • But the exact order of commits is not guaranteed by semantics of transaction
TM implementation basics

• TM systems must provide atomicity and isolation
  • While maintaining concurrency as much as possible

• Two key implementation questions
  • Data versioning policy: How does the system manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions?
  • Conflict detection policy: how/when does the system determine that two concurrent transactions conflict?
Data versioning policy

• Manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions

1. Eager versioning (undo-log based)
   - Update memory location directly
   - Maintain undo info in a log
   - Fast commits
   - Slow aborts

2. Lazy versioning (write-buffer based)
   - Buffer data until commit in a write buffer
   - Update actual memory locations at commit
   - Fast aborts
   - Slow commits
Eager versioning

Begin txn

Thread (executing txn)

X: 10
Memory

X: 10
Undo log

Write X ⟷ 15

Thread (executing txn)

X: 15
Memory

X: 15
Undo log

Commit txn

Thread (executing txn)

X: 15
Memory

X: 15
Undo log

Abort txn

Thread (executing txn)

X: 10
Memory

X: 10
Undo log

Update memory *immediately*, maintain “undo log” in case of abort
Lazy versioning

Begin txn

Thread (executing txn)

X: 10

Memory

Write buffer

Write X ← 15

Thread (executing txn)

X: 10

Memory

Write buffer

Commit txn

Thread (executing txn)

X: 15

Memory

Write buffer

Abort txn

Thread (executing txn)

X: 15

Memory

Write buffer

Log memory updates in transaction write buffer, flush buffer on commit
Data versioning

• Goal
  • Manage uncommitted (new) and committed (old) versions of data for concurrent transactions

• Eager versioning (undo-log based)
  • Update memory location directly on write
  • Maintain undo information in a log (incurs per-store overhead)
  • Good: faster commit (data is already in memory)
  • Bad: slower aborts, fault tolerance issues (consider crash in middle of transaction)

• Lazy versioning (write-buffer based)
  • Buffer data in a write buffer until commit
  • Update actual memory location on commit
  • Good: faster abort (just clear log), no fault tolerance issues
  • Bad: slower commits
Conflict detection

• Must detect and handle conflicts between transactions
  • Read-write conflict: transaction A reads address X, which was written to by pending (but not yet committed) transaction B
  • Write-write conflict: transactions A and B are both pending, and both write to address X

• System must track a transaction’s read set and write set
  • Read-set: addresses read during the transaction
  • Write-set: addresses written during the transaction
Pessimistic detection

• Check for conflicts (immediately) during loads or stores
  • Philosophy: “I suspect conflicts might happen, so let’s always check to see if one has occurred after each memory operation... if I’m going to have to roll back, might as well do it now to avoid wasted work.”

• “Contention manager” decides to stall or abort transaction when a conflict is detected
  • Various policies to handle common case fast
Pessimistic detection examples

Note: diagrams assume “aggressive” contention manager on writes: writer wins, so other transactions abort
Optimistic detection

• Detect conflicts when a transaction attempts to commit
  • Intuition: “Let’s hope for the best and sort out all the conflicts only when the transaction tries to commit”

• On a conflict, give priority to committing transaction
  • Other transactions may abort later on
Optimistic detection examples

Case 1
- T_0: rd A, wr B, wr C
- T_1: wr A
- Commit
- Check
- Success

Case 2
- T_0: wr A
- T_1: rd A, commit
- Check
- Restart

Case 3
- T_0: rd A, wr A
- T_1: Commit
- Check
- Success

Case 4
- T_0: rd A, wr A
- T_1: Restart
- Check
- Forward progress
Conflict detection trade-offs

• Pessimistic conflict detection (a.k.a. “eager”)
  • Good: detect conflicts early (undo less work, turn some aborts to stalls)
  • Bad: no forward progress guarantees, more aborts in some cases
  • Bad: fine-grained communication (check on each load/store)
  • Bad: detection on critical path

• Optimistic conflict detection (a.k.a. “lazy” or “commit”)
  • Good: forward progress guarantees
  • Good: bulk communication and conflict detection
  • Bad: detects conflicts late, can still have fairness problems
TM implementation space (examples)

• Hardware TM systems
  • Lazy + optimistic: Stanford TCC
  • Lazy + pessimistic: MIT LTM, Intel VTM
  • Eager + pessimistic: Wisconsin LogTM
  • Eager + optimistic: not practical

• Software TM systems
  • Lazy + optimistic (rd/wr): Sun TL2
  • Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  • Eager + optimistic (rd)/pessimistic (wr): Intel STM
  • Eager + pessimistic (rd/wr): Intel STM

• Optimal design remains an open question
  • May be different for HW, SW, and hybrid
Conclusion

• Synchronization

• Transactional memory
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)