CS252 Spring 2017
Graduate Computer Architecture

Lecture 15:
Synchronization and Memory Models Part 2

Lisa Wu, Krste Asanovic
http://inst.eecs.berkeley.edu/~cs252/sp17
Project Proposal

• Should be able to answer the following questions:
  • What problem are you solving?
  • What solutions exist from your related work search?
  • What idea/solution are you proposing?
  • What is the potential upside of your idea/solution?
  • How will you evaluate your idea/solution?
  • What are the intermediary milestones to measure your progress?

• 1-2 page proposal (not including references) in the CAL submission format [https://www.computer.org/web/cal/submit](https://www.computer.org/web/cal/submit)

• A reading list containing 10-20 papers in a bibliography format loosely outlining your related work search. The reading list should be prioritized with the most relevant work listed at the top.

• A 15-20 min presentation with a 5-10 min Q&A in class

• Due dates:
  • Proposal and reading list due on Sunday 3/19 by midnight
  • Presentation delivered in class on Thursday 3/23
Last Time in Lecture 14

• Multithreading
  • OoO SMT
  • Historical systems that implement some form of MT
    • Danelcor HEP, Tera MTA, MIT Alewife, IBM PowerPC, IBM Power4 and Power5, Oracle/Sun Niagara, Oracle T and M series, Intel Pentium 4 Hyperthreading

• Synchronization
  • Producer-consumer paradigm: e.g Tera MTA using full/empty bit, hardware retry, and software traps
  • Memory models: sequential consistency, relaxed memory models
Relaxed Consistency Models

Relaxed consistency models allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent.

~ H&P Fifth Edition
Classification of Consistency Models

- Memory operation ordering X→Y reads “operation X must complete before operation Y is done”
- Sequential consistency requires R→W, R→R, W→R, and W→W
- Relaxed models allow certain ordering to be relaxed or violated
  - Relaxing W→R ordering: TSO (Total Store Ordering) or PC (Processor Consistency). Retains orders between writes.

Motivation for relaxing consistency model is to improve performance and hide memory latency (the write latency in this case)
TSO and Processor Consistency
Examples of relaxed consistency models defining precisely what it means for a write to complete

• Assumption: Processor P0 and the rest of the processors Px form a SMP system; A and B are different memory addresses.

TSO: P0 can reorder its own reads and writes to execute read B ahead of write A. P1 and P2 can only read A when every processor in the system can see the new value of A.

PC: P0 can reorder its own reads and writes to execute read B ahead of write A. P1 and P2 can read A when they can see the new value of A without having to wait for everyone else.
Partial Store Ordering (PSO)
Examples of a relaxed consistency model relaxing the $W\rightarrow R$ and the $W\rightarrow W$ ordering

- Assumption: Processor P0 and P1 form a SMP system.

PSO: P0 can reorder its own writes and P1 may observe the new value of flag before the new value of A.

```
P0:  P1:
A = 1;  while (flag == 0);
flag = 1;  print A;
```
Weak Ordering (WO) and Release Consistency (RC)
Examples of relaxed consistency models relaxing all four types of memory operation orderings

• Processor supports special synchronization operations such as atomic hardware primitives

• Memory accesses before the memory barrier instruction (such as FENCE; also called sync) need to complete before the memory barrier instruction is issued.

• Memory accesses after the memory barrier instruction cannot begin until the memory barrier instruction is completed.

reorderable reads and writes
...
SYNC
...
reorderable reads and writes
...
SNYC
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 $c_1$ and $c_2$. Initially, both $c_1$ and $c_2$ are 0 (not busy)

**Process 1**

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

**Process 2**

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

What is wrong? **Deadlock!**
Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets $c_1$ 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

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

### Process 2

```
... L: c2=1;  
    if c1=1 then  
        { c2=0; go to L}  
    < critical section>  
    c2=0
```
Mutual Exclusion
At most one processor is executing critical section at any point

• Avoid deadlock
  • Deadlock: All processors are waiting for some other processor to release the lock.
  • Enforce that if a processor is in its entry section of the code, then later some processor will make it to its critical section of the code.

• Avoid starvation
  • Starvation: An unlucky processor may never get to enter the critical section of the code.
  • Enforce that if a processor is in its entry section of the code, then later the same processor will make it to its critical section of the code.

• Bounded waiting
  • Bound how many times other processors can enter their critical sections while one processor is waiting.
A protocol based on 3 shared variables c1, c2 and turn. Initially, both c1 and c2 are 0 (not busy)

- 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!
ISA Support for Mutual-Exclusion Locks

- Regular loads and stores in SC model (plus fences in weaker model) sufficient to implement mutual exclusion, but inefficient and complex code
- 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

Assumes SC memory model

```
// Both threads execute:
li xone, 1

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

Thread 1

Thread 2

Memory
Lock for Mutual-Exclusion with Relaxed MM

// Both threads execute:
li xone, 1

spin: amoswap xlock, xone, (xlockp)
bnez xlock, spin
fence.r.r

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

fence.w.w
sd x0, (xlockp)
Release Consistency

- 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

Ensure critical section updates visible before release visible

Ensure acquire happened before critical section reads data
Release Consistency Adopted

- 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

- ARM v8.1 and RISC-V ISA adopt release consistency semantics on AMOs
Nonblocking Synchronization using CAS (Compare-and-Swap)

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

status is an implicit argument

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)
Load Linked or Load Locked and Store Conditional (LL-SC)

• A pair of memory instructions where the second instruction returns a value that indicates whether the pair of instructions were executed as if the instructions were atomic.

• If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails.

• If the processor does a context switch between the two instructions, then the store conditional fails.
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 $(m), R$:
- $if <\text{flag, adr}> == <1, m>$
  - $then$ cancel other proc’s’ reservation on $m$;
  - $M[m] \leftarrow R$;
  - $status \leftarrow succeed$;
- $else$ $status \leftarrow fail$;

try:

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