

Multiprocessor Synchronization and Consistency

CSE 240B

Dean Tullsen

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data
- Issues for Synchronization:
 - Need an uninterruptable instruction to read and update memory (atomic operation);
 - User level synchronization operations are then built using this primitive;
 - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization needed

CSE 240B

Dean Tullsen

Uninterruptable Instruction to Fetch and Update Memory

- **Atomic exchange:** interchange a value in a register for a value in memory
 - 0 => synchronization variable is free
 - 1 => synchronization variable is locked and unavailable
 - Set register to 1 & swap
 - New value in register determines success in getting lock
 - 0 if you succeeded in setting the lock (you were first)
 - 1 if other processor had already claimed access
 - Write 0 to release lock
 - Key is that exchange operation is indivisible
 - Can be used to do more powerful things than implement locks.

CSE 240B

Dean Tullsen

Uninterruptable Instruction to Fetch and Update Memory

- **Test-and-set:** reads a value and sets it atomically
 - Special case of atomic exchange
 - Most common sync primitive
 - 0 means lock free, 1 means locked
 - Test-and-set reads the lock variable, and sets it to one. If the value read was 0, you have acquired the lock. If it was 1, you did not.
 - Write 0 to release lock
 - Pretty much just used to enable locks.


CSE 240B

Dean Tullsen

Uninterruptable Instruction to Fetch and Update Memory

- **Fetch-and-increment**: it returns the value of a memory location and atomically increments it
 - 0 => synchronization variable is free, >0 implies locked
 - Write 0 to release lock
 - Can do more powerful things than implement locks

Example

<pre>lock: lw R1, lockaddress bnez R1, lock lw R2, varaddress addi R2, R2, 1 sw R2, varaddress add R1, R0, R0 sw R1, lockaddress</pre>		<pre>lock: addi R1, R0, 1 bnez R1, lock lw R2, varaddress addi R2, R2, 1 sw R2, varaddress add R1, R0, R0 sw R1, lockaddress</pre>
--	---	--

- This works because *test-and-set* is atomic
- Notice this could be done with one instruction if we have *fetch-and-increment*.

Uninterruptable *Instructions* to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- **Load linked** (or load locked) + **store conditional**
 - Load linked returns the initial value
 - Store conditional only completes the store if no other store to same memory location since preceding load load linked. The SC returns 1 if it succeeds and 0 otherwise.

Uninterruptable *Instructions* to Fetch and Update Memory

- Example doing atomic swap with LL & SC:


```
try: mov   R3,R4           ; mov exchange value
      ll    R2,0(R1) ; load linked
      sc    R3,0(R1) ; store
      beqz  R3,try         ; branch store fails
      mov   R4,R2         ; put load value in R4
```
- Example doing fetch & increment with LL & SC:


```
try: ll    R2,0(R1) ; load linked
      addi  R2,R2,#1   ; increment (OK if reg-reg)
      sc    R2,0(R1)   ; store
      beqz  R2,try     ; branch store fails
```
- This is an example of something called *non-locking (lock-free) synchronization*. Why? What's the big advantage?

User Level Synchronization—Operation Using These Primitives

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
lockit:    li    R2,#1
          exch  R2,0(R1)    ;atomic exchange
          bnez  R2,lockit   ;already locked?
```

- What about MP with cache coherency?
 - Want to spin on cache copy to avoid full memory latency
 - Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

User Level Synchronization—Operation Using These Primitives

- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):

```
try:      li    R2,#1
lockit:   lw    R3,0(R1)    ;load var
          bnez  R3,lockit   ;not free=>spin
          exch  R2,0(R1)    ;atomic exchange
          bnez  R2,try     ;already locked?
```

Steps for Invalidate Protocol

Step	P0	\$	P1	\$	P2	\$	Bus/Direct activity
1.	Has lock	Sh	spins	Sh	spins	Sh	None
2.	Lock←-0	Ex		Inv		Inv	P0 Invalidates lock
3.		Sh	miss	Sh	miss	Sh	WB P0; P2 gets bus
4.		Sh	waits	Sh	lock = 0	Sh	P2 cache filled
5.		Sh	lock=0	Sh	exch	Sh	P2 cache miss(WI)
6.		Inv	exch	Inv	r=0;l=1	Ex	P2 cache filled; Inv
7.		Inv	r=1;l=1	Ex	locked	Inv	WB P2; P1 cache
8.		Inv	spins	Ex		Inv	None

For Large Scale MPs, Synchronization Can Be a Bottleneck

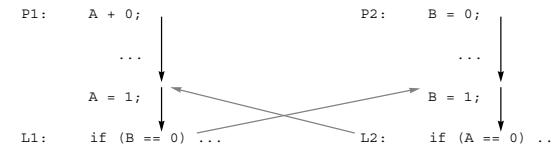
- 20 processors spin on lock held by 1 proc, 50 cycles for bus
 - 1525 bus operations, over 30,000 cycles for 20 processors to pass through the lock
 - Problem is contention for lock and serialization of lock access: once lock is free, all compete to see who gets it (each causing an invalidate storm)
- Alternative: exponential backoff. Why does this help?
- Another alternative: create a list of waiting processors, go through list: called a “queuing lock”

Barrier Synchronization

- A very common synchronization primitive
- Wait until all threads have reached a point in the program before any are allowed to proceed further.

```
computation;  
barrier()  
communication;  
barrier()  
repeat:
```

Another MP Issue: **Memory Consistency Models**



- Impossible for both if statements L1 & L2 to be true?
 - What if write (or invalidate) is delayed & processor continues?
- Memory consistency models: what are the rules for such cases?
- **Sequential consistency**: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved.
 - SC: delay all memory accesses until all invalidates done
- **Coherence** guaranteed some ordering of accesses to A, and of accesses to B, but provided no guarantees for ordering of A wrt B.

Sequential Consistency is a Huge Burden

- A write, including all invalidate messages and acknowledgments, must complete before any subsequent memory operation (incl. loads) begins.
- Involves more than just accesses to the same location.
- Modern ILP processors violate SC every chance they get!
- Simplifying observation: most well-written parallel programs are *synchronized* if they want to get the correct values. That is, they don't rely on SC.

Memory Consistency Model

- A program is synchronized if all access to shared data are ordered by synchronization operations

```
write(x)  
...  
release(s) {unlock}  
...  
acquire(s) {lock}  
...  
read(x)
```
- Only those programs willing to be nondeterministic are not synchronized
- There exist several Relaxed Models for Memory Consistency since most programs are synchronized: characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Relaxed (or weak) Consistency Models

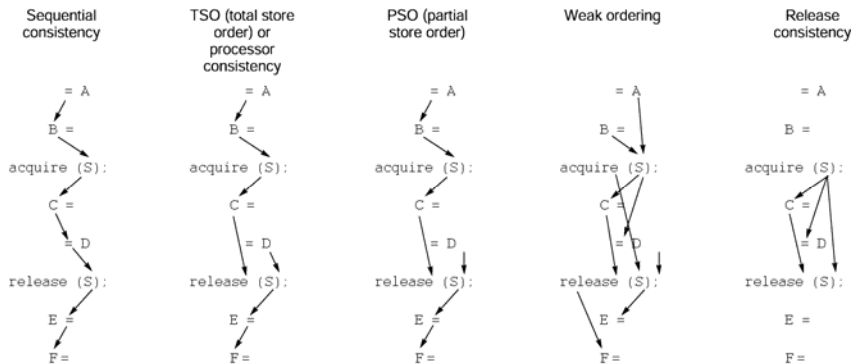
- Differ according to what guarantees they give the programmer in regards to memory access ordering.
- Depend on, and must be communicated to, the programmer.
- Consistency models that require the programmer to change behavior are doomed to failure.

Consistency Models

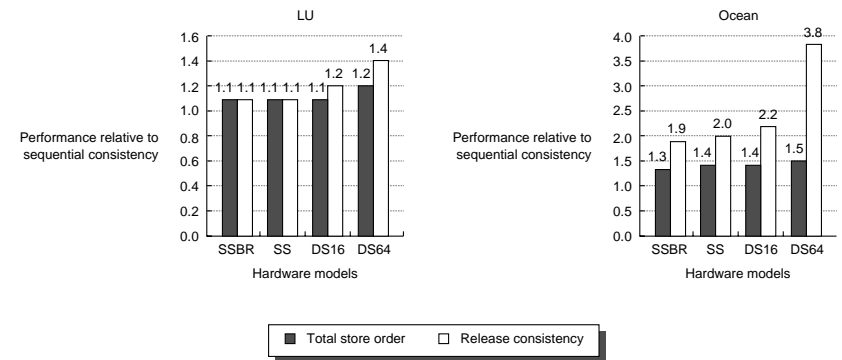
Model	Used In	Ordinary Orderings	Synchronization Orderings
Sequential Consistency	Most machines in an optional mode	R->R, R->W, W->R, W->W	S->W, S->R, R->S, S->S
Total Store Order (Processor Consistency)	IBM S/370, DEC VAX, SPARC	R->R, R->W, W->W	S->W, S->R, R->S, W->S, S->S
Partial Store Order	SPARC	R->R, R->W	S->W, S->R, R->S, W->S, S->S
Weak Ordering	PowerPC		S->W, S->R, R->S, W->S, S->S
Release Consistency	Alpha, MIPS		Sa->W, Sa->R, R->Sa, W->Sr, Sa->Sa, Sa->Sr, Sr->Sa, Sr->Sr

Orderings preserved by various consistency models

Consistency Models



Consistency Models



Key Points

- High-performance synchronization should conserve memory/interconnect bandwidth
- Sequential consistency is attractive as a programming model, but performance is unacceptable.
- Relaxed consistency models allow memory operations to proceed out of order, by guaranteeing ordering of memory operations with regards to synchronization, but not necessarily with each other.