

CS533: Synchronization

Josep Torrellas

University of Illinois in Urbana-Champaign

Synchronization

- To ensure consistency of shared data structures . . . need hardware-supported atomic primitives
- Classes of synchronization primitives
 - Blocking: preempt waiting process
 - Busy-wait: process repeatedly tests shared variables to determine when it can proceed

When is Busy Waiting Preferred?

- *Scheduling overhead* $>$ *expected waiting time*
- Processor resources are not needed for other tasks
- Network/Cache can tolerate hot spot
- Cannot be pre-empted (OS)

Common Synchronization Primitives

- Locks: grant access to one process only
- Barriers: no process advances beyond it until all have arrived
- Semaphores
- Monitors
- ...

- All syn primitives are easily implementable out of locks
- Usually: Not used directly by the users $\implies$ libraries

- Un-interruptible instruction or instruction sequence $\implies$ capable of atomic read-modify-write (RMW)
 - Atomic exchange
 - Fetch-and-increment
 - Test & Set
- Non-atomic sequence of instructions that detect if intervening access $\implies$ usually works
 - Load-linked and Store-conditional

Atomic Exchange: exchanges value in register with memory

```
Reg  $\leftarrow$  1  
Exchange Reg, Mem  
/* if Reg has 0, got it; else keep trying */
```

Test & Set

```
Test & Set Reg, Mem  
/* if Reg has 0, got it; else keep trying */
```

- Caches cause invalidations: Use Test & Test & Set
- Fetch & Increment: returns the value of mem, and increments it

```
Reg  $\leftarrow$  Fetch & Increment (Mem)  
/* The returned value tells us how many are waiting */
```

Cost: Bus Traffic $\mathcal{O}(N^2)$

- Suppose N processors are spin-waiting with Test & Test & Set
- Bus traffic for all N processors to gain access to lock: $\mathcal{O}(N^2)$
- Why? Each time lock is unset, all processors issue an access, but only 1 is successful

Reducing Implementation Complexity

- Use 2 instructions, where the 2nd one returns a value from which it can be deduced whether the pair was executed as if atomic
- MIPS/SGI: Load-linked (LL), Store-conditional (SC)
 - LL: returns the value of the location
 - SC:
 - If contents of location have been changed between LL and SC:

```
SC fails /* returns 0, does not update */
```

- Else

```
SC succeeds /* return 1, updates location */
```

Example

Exchange $R4$ with location $0(R1)$:

```
try:    mov    R3, R4 /* want to store R4 */  
        ll     R2, 0(R1)  
        sc     R3, 0(R1)  
        beqz   R3, try /* if 0, failed, no update */  
        mov    R4, R2 /* only if success */
```

- SC also fails if processor context switches between LL and SC
- Can be used to implement other primitives like Fetch & increment

Fancier HW to Atomically RMW

Atomic Fetch & Phi:

- Test & Set: return old value, set variable
- Test & Test & Set: read it; if not set, try Test & Set
- Fetch & Store: get old value; store new
- Fetch & Add: get the old value; add a constant to the var
- Compare & Swap (CAS):
 - Takes two values (V_1 and V_2 and memory address)
 - If mem location has a value equal to V_1 , then it stores V_2 to that location
 - Else nothing is done

- Each word in memory has Full/Empty (F/E) bit
- Bit is tested in hardware before a RD/WR if special symbol is prepended to the var name
- The RD/WR blocks until the test succeeds:
 - RD until full
 - WR until empty
- When test succeeds, the bit is set to the opposite value, indivisibly with the RD/WR

Advantages:

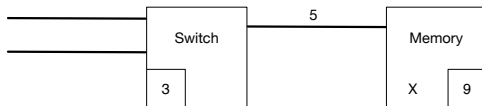
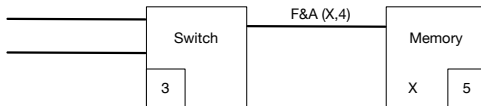
- Very efficient for low level dependences (compare to locks)

Disadvantages:

- F/E bits
- Logic to initialize the bits
- Support to queue a process if test fails
- Logic to implement indivisible ops

- Atomic Fetch & Add: Send a message to a memory location with a constant
- Advantages:
 - Useful in certain cases: get the next iteration of a loop
 - If the network has hardware to combine messages to the same location, primitive tolerates contentions

Example



Message Combining

Advantages:

- Multiple requests in parallel
- Less traffic (scalable)

Disadvantages:

- Very complex network
- Slows down the rest of the messages

Hardware required:

- For Fetch & Add: adder in each memory module
- For message combining:
 - Special, complex queuing logic at each switch in the network

- Atomic Fetch & Phi:
 - Add, And, Or
 - Min, Max, Store
 - Store if zero
- Hardware required: logic in the shared memory to implement the 7 atomic operations

- General atomic instruction that operates on synchronization variables
- Synch var is 2 words: Key and Value
- Synch instruction:

```
{addr; (cond); op on key; op on value}  
  if * in condition: spin until true
```

```
{X; (X.key == 1)*; decrement; fetch}  
  this is F/E bit test for a read option
```

- HW required: special processor at each mem module