Synchronization

Instructor: Josep Torrellas
CS533
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 synch primitives are easily implementable out of locks
• Usually: Not used directly by the user → libraries
Small Machines

• Un-interruptible instruction or instruction sequence
  \(\rightarrow\) capable of atomic read-modify-write
  – Atomic exchange
  – Fetch-and-increment
  – Test&Set

• Non-atomic sequence of instructions that detect if
  intervening access \(\rightarrow\) usually works
  – Load-linked and Store-conditional
Primitives

- **Atomic Exchange**: exchanges value in register with memory
  
  \[
  \begin{array}{ll}
  0: & \text{free} \\
  1: & \text{not free}
  \end{array}
  \]

  \[
  \begin{align*}
  \text{Reg} & \leftarrow 1 \\
  \text{Exchange Reg, Mem} \\
  \text{/* if Reg has 0, got it; else keep trying */}
  \end{align*}
  \]

  To unlock: \( \text{Mem} \leftarrow 0 \)

- **Test&Set**

  \[
  \begin{align*}
  \text{Test&Set Reg, Mem} \\
  \text{/* if Reg has 0, got it; else keep trying */}
  \end{align*}
  \]
Primitives

• Caches cause invalidations: Use Test&Test&Set

• Fetch&Increment: returns the value of mem, and increments it
  \[ \text{Reg} \leftarrow \text{Fetch}&\text{Increment}(\text{Mem}) \]
  /* The returned value tells us how many are waiting */
Cost: Bus Traffic $O(N^2)$

- Suppose $N$ processors are spin-waiting with Test&Set
- Bus traffic for all $N$ procs to gain access to lock: $O(N^2)$

- Why? Each time lock is unset, all processors issue an access, but only 1 is successful
Reducing Implementation Complexity

• Use 2 instructions, where the 2\textsuperscript{nd} 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 /*returns 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: compares a constant w/ value of a location; if same, set a condition code; else replace the value
HEP

- 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
HEP (Continued)

• Advantages:
  – Very efficient for low level dependences (compare to locks)

• Disadvantages: HW required
  – F/E bits
  – Logic to initialize the bits
  – Support to queue a process if test fails
  – Logic to implement indivisible ops
NYU Ultracomputer

• 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 contention
Example

```
F&A(X,3)  Switch  Memory
F&A(X,1)  3

F&A(X,4)  Switch  Memory
  3       5

5  Switch  Memory
  3       9

5  Switch  Memory
  8       9
```
Message Combining

• Advantages:
  – Multiple requests in parallel
  – Less traffic (scalable)

• Disadvantages:
  – Very complex network
  – Slows down the rest of the messages
NYU Ultra (Cont)

• Hardware required:
  – For Fetch&Add: adder in each memory module
  – For message combining:
    • Special, complex queuing logic at each switch in the network
IBM RP3

• 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
Illinois Cedar

• 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 operation

• HW required: special processor at each mem module
Alternative Approach

• No fancy hardware
• Software smarts: distribute the lock and spin locally
• Use locks and barriers
Simple Test&Set

• Processor tests and sets it. Returns old value
• Polling loop around a boolean variable

While (Test&Set(lock)) {} 

• Disadvantages:
  – Contention for the lock
  – Frequent accesses using the expensive RMW
Test&Test&Set

• Issue a Test&Set only when a previous read has indicated that the Test&Set may succeed

• Advantages
  – Reduce the number of RMW
  – Reduce the number of cache misses

• Disadvantage:
  – Still several processors may attempt the Test&Set
Test&Set With Delay

- Reduce the use of caches/networks by delaying consecutive probes of the lock after failure
- Example:
  - Constant delay
  - Backoff on unsuccessful probes
  - Exponential backoff (best)
Example

type lock = (unlocked, locked)
procedure acquire_lock (L: ^lock)
    integer delay :=1
    while (Test&Set(L) == locked) {
        pause (delay)
        delay := delay*2
    }

procedure release_lock(L: ^lock)
    lock^ := unlocked
Ticket Lock

- In Test&Test&Set, still a lot of RMW is possible (e.g. every processor every lock release)
- Use Ticket Lock:

    ![Diagram of R and T]

    - R: # Requests to Acquire the lock
    - T: # Times the lock Has been released

    **Acquire:** ticket = Fetch&Increment( R)
    wait until T == ticket
    ...
    T++
Ticket Lock (II)

• Advantages:
  – Only 1 Fetch&Op per lock operation (probing is done with reads only)
  – FIFO scheme: grant lock to processors in order they requested it → fair, no starvation

• Disadvantages:
  – Still a lot of cache or network contention through polling