Lecture 20: Synchronization & Consistency

- Topics: synchronization, consistency models (Sections 4.5-4.6)

Test-and-Test-and-Set

```
lock:   test  register, location
        bnz   register, lock
        t&s   register, location
        bnz   register, lock
        CS
        st    location, #0
```

Spin Lock with Low Coherence Traffic

```
lockit:   LL       R2, 0(R1)    ; load linked, generates no coherence traffic
          BNEZ     R2, lockit   ; not available, keep spinning
          DADDUI R2, R0, #1 ; put value 1 in R2
          SC       R2, 0(R1)    ; store-conditional succeeds if no one
                                ; updated the lock since the last LL
          BEQZ     R2, lockit   ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?
  1 write by the releaser + i read-miss requests +
  i responses + 1 write by acquirer + 0 (i-1 failed SCs) +
  i-1 read-miss requests

Lock Vs. Optimistic Concurrency

```
lockit:  LL      R2, 0(R1)
         BNEZ    R2, lockit
         DADDUI R2, R0, #1
         SC      R2, 0(R1)
         BEQZ    R2, lockit
           Critical Section
         ST      0(R1), #0
```

LL-SC is being used to figure out if we were able to acquire the lock without anyone interfering – we then enter the critical section

```
tryagain: LL      R2, 0(R1)
          DADDUI R2, R2, R3
          SC      R2, 0(R1)
          BEQZ    R2, tryagain
```

If the critical section only involves one memory location, the critical section can be captured within the LL-SC – instead of spinning on the lock acquire, you may now be spinning trying to atomically execute the CS

Further Reducing Bandwidth Needs

- Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable

- Array-Based lock: instead of using a “now-serving” variable, use a “now-serving” array and each process waits on a different variable – fair, low latency, low bandwidth, high scalability, but higher storage

- Queueing locks: the directory controller keeps track of the order in which requests arrived – when the lock is available, it is passed to the next in line (only one process sees the invalidate and update)

Barriers

- Barriers are synchronization primitives that ensure that some processes do not outrun others – if a process reaches a barrier, it has to wait until every process reaches the barrier

- When a process reaches a barrier, it acquires a lock and increments a counter that tracks the number of processes that have reached the barrier – it then spins on a value that gets set by the last arriving process

- Must also make sure that every process leaves the spinning state before one of the processes reaches the next barrier

Barrier Implementation

```
LOCK(bar.lock);
if (bar.counter == 0)
  bar.flag = 0;
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
  bar.counter = 0;
  bar.flag = 1;
}
else
  while (bar.flag == 0)  { };
```

Sense-Reversing Barrier Implementation

```
local_sense = !(local_sense);
LOCK(bar.lock);
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
  bar.counter = 0;
  bar.flag = local_sense;
}
else {
  while (bar.flag != local_sense)  { };
}
```

Coherence Vs. Consistency

- Recall that coherence guarantees (i) that a write will eventually be seen by other processors, and (ii) write serialization (all processors see writes to the same location in the same order)

- The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions

Example Programs

```
Initially, A = B = 0

  P1                          P2
A = 1                      B = 1
if (B == 0)                if (A == 0)
   critical section            critical section
```

```
   P1                    P2
Data = 2000       while (Head == 0)
Head = 1             { }
                  … = Data
```

```
Initially, A = B = 0

 P1              P2               P3
A = 1
             if (A == 1)
               B = 1
                                if (B == 1)
                                  register = A
```

Consistency Example - I

- Consider a multiprocessor with bus-based snooping cache coherence and a write buffer between CPU and cache

```
Initially A = B = 0
 P1                    P2
A ← 1                B ← 1
 …                     …
if (B == 0)          if (A == 0)
  Crit.Section         Crit.Section
```

The programmer expected the above code to implement a lock – because of write buffering, both processors can enter the critical section

The consistency model lets the programmer know what assumptions they can make about the hardware's reordering capabilities

Consistency Example - 2

P1	P2
Data = 2000	while (Head == 0) { }
Head = 1	… = Data

Sequential consistency requires program order

-- the write to Data has to complete before the write to Head can begin

-- the read of Head has to complete before the read of Data can begin

Consistency Example - 3

Initially, A = B = 0

```
  P1            P2                  P3
A = 1
           if (A == 1)
              B = 1
                              if (B == 1)
                                 register = A
```

Sequential consistency can be had if a process makes sure that everyone has seen an update before that value is read – else, write atomicity is violated

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achieveable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion

- The multiprocessors in the previous examples are not sequentially consistent

- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow

Relaxed Consistency Models

- We want an intuitive programming model (such as sequential consistency) and we want high performance

- We care about data races and re-ordering constraints for some parts of the program and not for others – hence, we will relax some of the constraints for sequential consistency for most of the program, but enforce them for specific portions of the code

- Fence instructions are special instructions that require all previous memory accesses to complete before proceeding (sequential consistency)

Relaxing Constraints

- Sequential consistency constraints can be relaxed in the following ways (allowing higher performance):
    - within a processor, a read can complete before an earlier write to a different memory location completes (this was made possible in the write buffer example and is of course, not a sequentially consistent model)
    - within a processor, a write can complete before an earlier write to a different memory location completes
    - within a processor, a read or write can complete before an earlier read to a different memory location completes
    - a processor can read the value written by another processor before all processors have seen the invalidate
    - a processor can read its own write before the write is visible to other processors

Title

- Bullet