Lecture 22: Synchronization & Consistency

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

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
```

Sequential Consistency

P1	P2
Instr-a	Instr-A
Instr-b	Instr-B
Instr-c	Instr-C
Instr-d	Instr-D
…	…

We assume:
- Within a program, program order is preserved
- Each instruction executes atomically
- Instructions from different threads can be interleaved arbitrarily

Valid executions:

abAcBCDdeE… or ABCDEFabGc… or abcAdBe… or
aAbBcCdDeE… or …..

Sequential Consistency

- Programmers assume SC; makes it much easier to reason about program behavior

- Hardware innovations can disrupt the SC model

- For example, if we assume write buffers, or out-of-order execution, or if we drop ACKS in the coherence protocol, the previous programs yield unexpected outputs

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)

Fences

```
  P1                        P2
{                         {
  Region of code            Region of code
  with no races             with no races
}                         }

Fence                     Fence
Acquire_lock              Acquire_lock
Fence                     Fence

{                         {
   Racy code                 Racy code
}                         }

Fence                     Fence
Release_lock              Release_lock
Fence                     Fence
```

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