Lecture: Synchronization, Consistency Models

- Topics: efficient synchronization primitives, need for sequential consistency, fences
Test-and-Test-and-Set

- lock:   test   register, location
  bnz   register, lock
  t&s   register, location
  bnz   register, lock
  CS
  st   location, #0
Load-Linked and Store Conditional

• LL-SC is an implementation of atomic read-modify-write with very high flexibility

• LL: read a value and update a table indicating you have read this address, then perform any amount of computation

• SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was “effectively” atomic)

• SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set
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?
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 + $i$ (or 1) read-miss requests +
  $i$ (or 1) responses + 1 write by acquirer + 0 ($i$-1 failed SCs) +
  $i$-1 (or 1) read-miss requests + $i$-1 (or 1) responses

(The $i$/$i$-1 read misses can be reduced to 1)
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

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

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.

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.
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) { };
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, $A = B = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 1$</td>
<td>$B = 1$</td>
</tr>
<tr>
<td>if ($B == 0$)</td>
<td>if ($A == 0$)</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
</tbody>
</table>

Initially, $A = B = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if ($A == 1$)</td>
<td>$B = 1$</td>
<td>if ($B == 1$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>register = $A$</td>
</tr>
</tbody>
</table>

Initially, $Head = Data = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Data = 2000$</td>
<td>while ($Head == 0$)</td>
</tr>
<tr>
<td>$Head = 1$</td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>... = Data</td>
</tr>
</tbody>
</table>
# Sequential Consistency

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 $.....$
Problem 1

• What are possible outputs for the program below?

Assume \(x=y=0\) at the start of the program

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = 10)</td>
<td>(y=20)</td>
</tr>
<tr>
<td>(y = x+y)</td>
<td>(x = y+x)</td>
</tr>
<tr>
<td>Print (y)</td>
<td></td>
</tr>
</tbody>
</table>
Problem 1

• What are possible outputs for the program below?

Assume x=y=0 at the start of the program

Thread 1                           Thread 2
A    x = 10                   a    y=20
B    y = x+y                   b    x = y+x
C    Print y

Possible scenarios: 5 choose 2 = 10

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>10</th>
<th>B</th>
<th>20</th>
<th>Print y</th>
<th>20</th>
<th>C</th>
<th>30</th>
<th>b</th>
<th>30</th>
<th>a</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCab</td>
<td>10</td>
<td>20</td>
<td>ABaCb</td>
<td>20</td>
<td>AaBCb</td>
<td>30</td>
<td>AAabC</td>
<td>30</td>
<td>abABC</td>
<td>30</td>
<td>AAAbC</td>
<td>30</td>
</tr>
<tr>
<td>AabBC</td>
<td>50</td>
<td>30</td>
<td>aABCb</td>
<td>30</td>
<td>aABbC</td>
<td>50</td>
<td>aAbBC</td>
<td>50</td>
<td>abABC</td>
<td>50</td>
<td>aAbABC</td>
<td>50</td>
</tr>
</tbody>
</table>
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 - 1

• An ooo core will see no dependence between instructions dealing with A and instructions dealing with B; those operations can therefore be re-ordered; this is fine for a single thread, but not for multiple threads

Initially A = B = 0

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ← 1</td>
<td>B ← 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>if (B == 0)</td>
<td>if (A == 0)</td>
</tr>
<tr>
<td>Crit.Section</td>
<td>Crit.Section</td>
</tr>
</tbody>
</table>

The consistency model lets the programmer know what assumptions they can make about the hardware’s reordering capabilities
Consistency Example - 2

Initially, $A = B = 0$

P1  P2  P3
A = 1

if ($A == 1$)

    B = 1

if ($B == 1$)

    register = A

If a coherence invalidation didn’t require ACKs, we can’t confirm that everyone has seen the value of $A$. 
Sequential Consistency

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

• 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.

• This is very slow... alternatives:
  - Add optimizations to the hardware (e.g., verify loads)
  - Offer a relaxed memory consistency model and fences
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
{  
    Region of code  
    with no races  
}
Fence
Acquire_lock
Fence
{  
    Racy code  
}
Fence
Release_lock
Fence

P2
{  
    Region of code  
    with no races  
}
Fence
Acquire_lock
Fence
{  
    Racy code  
}
Fence
Release_lock
Fence