Multiprocessor Synchronization and Consistency

Why Synchronize?
Need to know when it is safe for different processes to use shared data

Issues for Synchronization:
- Need an uninterruptable instruction to fetch and update memory (atomic operation);
- User level synchronization operations are then built using this primitive;
- For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization needed

Uninterruptable Instruction to Fetch and Update Memory

- **Atomic exchange**: interchange a value in a register for a value in memory
  
  0 ➔ synchronization variable is free
  
  1 ➔ synchronization variable is locked and unavailable
  
  - Set register to 1 & swap
  
  - New value in register determines success in getting lock
    
    0 if you succeeded in setting the lock (you were first)
    
    1 if other processor had already claimed access
  
  - Key is that exchange operation is indivisible

- **Test-and-set**: tests a value and sets it if the value passes the test

- **Fetch-and-increment**: it returns the value of a memory location and atomically increments it
  
  0 ➔ synchronization variable is free

- **Load linked (or load locked) + store conditional**:
  
  - Load linked returns the initial value
  
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceeding load) and 0 otherwise

- Example doing atomic swap with LL & SC:
  
  try: mov R3,R4 ; mov exchange value
  
  ll R2,0(R1) ; load linked
  
  sc R3,0(R1) ; store
  
  beqz R3,try ; branch store fails
  
  mov R4,R2 ; put load value in R4

- Example doing fetch & increment with LL & SC:
  
  try: ll R2,0(R1) ; load linked
  
  addi R2,R2,#1 ; increment (OK if reg-reg)
  
  sc R2,0(R1) ; store
  
  beqz R2,try ; branch store fails

- This is an example of something called **non-locking (lock-free) synchronization**. Why? What's the big advantage?
User Level Synchronization—Operation Using this Primitive

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock
  ```
  li R2,#1
  lockit:  exch R2,0(R1) ; atomic exchange
  bnez R2,lockit ; already locked?
  ```

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables

- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):
  ```
  try:  li R2,#1
  lockit:  lw R3,0(R1) ; load var
  bnez R3,lockit ; not free->spin
  exch R2,0(R1) ; atomic exchange
  bnez R2,try ; already locked?
  ```

Steps for Invalidate Protocol

<table>
<thead>
<tr>
<th>Step</th>
<th>$P0$</th>
<th>$P1$</th>
<th>$P2$</th>
<th>Bus/Direct activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Has lock</td>
<td>Sh</td>
<td>spins</td>
<td>Sh</td>
</tr>
<tr>
<td>2.</td>
<td>Lock←0</td>
<td>Ex</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>3.</td>
<td>Sh</td>
<td>miss</td>
<td>Sh</td>
<td>miss</td>
</tr>
<tr>
<td>4.</td>
<td>Sh</td>
<td>waits</td>
<td>Sh</td>
<td>lock=0</td>
</tr>
<tr>
<td>5.</td>
<td>Sh</td>
<td>lock=0</td>
<td>Sh</td>
<td>exch</td>
</tr>
<tr>
<td>6.</td>
<td>Inv</td>
<td>exch</td>
<td>Inv</td>
<td>r=0;l=1</td>
</tr>
<tr>
<td>7.</td>
<td>Inv</td>
<td>r=1;l=1</td>
<td>Ex</td>
<td>locked</td>
</tr>
<tr>
<td>8.</td>
<td>Inv</td>
<td>spins</td>
<td>Ex</td>
<td></td>
</tr>
</tbody>
</table>

For Large Scale MPs, Synchronization Can Be a Bottleneck

- 20 procs spin on lock held by 1 proc, 50 cycles for bus
  - 1525 bus operations, over 30,000 cycles for 20 processors to pass through the lock
  - Problem is contention for lock and serialization of lock access: once lock is free, all compete to see who gets it (each causing an invalidate storm)

- Alternative: exponential backoff. Why does this help?

- Another alternative: create a list of waiting processors, go through list: called a “queuing lock”

Barrier Synchronization

- A very common synchronization primitive
- Wait until all threads have reached a point in the program before any are allowed to proceed further.

```
  computation;
  barrier()
  communication;
  barrier()
  repeat:
```
Another MP Issue: Memory Consistency Models

Impossible for both if statements L1 & L2 to be true?
– What if write (or invalidate) is delayed & processor continues?

Memory consistency models: what are the rules for such cases?

Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved.
– SC: delay all memory accesses until all invalidates done

Sequential Consistency is a Huge Burden

– A write, including all invalidate messages and acknowledgments, must complete before any subsequent memory operation (incl. loads) begins.
– Involves more than just accesses to the same location.
– Modern ILP processors violate SC every chance they get!
– Simplifying observation: most well-written parallel programs are synchronized if they want to get the correct values. That is, they don’t rely on SC.

Memory Consistency Model

– A program is synchronized if all access to shared data are ordered by synchronization operations
  - write (x)
  - ... release (s) {unlock}
  - ...
  - acquire (s) {lock}
  - ...
  - read(x)
– Only those programs willing to be nondeterministic are not synchronized
– Several Relaxed Models for Memory Consistency since most programs are synchronized: characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Relaxed (or weak) Consistency Models

– Differ according to what guarantees they give the programmer in regards to memory access ordering.
– Depend on, and must be communicated to, the programmer.
– Consistency models that require the programmer to change behavior are doomed to failure.
### Consistency Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Used In</th>
<th>Ordinary Orderings</th>
<th>Synchronization Orderings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak Ordering</td>
<td>PowerPC</td>
<td>S-&gt;W, S-&gt;R, R-&gt;S</td>
<td>W-&gt;S, S-&gt;S</td>
</tr>
<tr>
<td>Release Consistency</td>
<td>Alpha, MIPS</td>
<td>Sa-&gt;W, Sa-&gt;R, R-&gt;Sa, W-&gt;Sr, Sa-&gt;Sa, Sa-&gt;Sr</td>
<td>Sr-&gt;Sa, Sr-&gt;Sr</td>
</tr>
</tbody>
</table>

Orderings preserved by various consistency models

---

**Consistency Models**

![Performance relative to sequential consistency](image1)

![Performance relative to sequential consistency](image2)

---

**Key Points**

- High-performance synchronization should conserve memory/interconnect bandwidth
- Sequential consistency is attractive as a programming model, but performance is unacceptable.
- Relaxed consistency models allow memory operations to proceed out of order, by guaranteeing ordering of memory operations with regards to synchronization, but not necessarily with each other.

---

**FIGURE 8.41** The performance of relaxed consistency models on a variety of hardware mechanisms, varying from quite reasonable to highly ambitious.