Lecture 26 – Synchronization

2005-4-26

John Lazzaro

(www.cs.berkeley.edu/~lazzaro)

TAs: Ted Hong and David Marquardt

www-inst.eecs.berkeley.edu/~cs152/
2. Forwarding engine determines the next hop for the packet, and returns next-hop data to the line card, together with an updated header.
Recall: Two CPUs sharing memory

In earlier lectures, we pretended it was easy to let several CPUs share a memory system. In fact, it is an architectural challenge. Even letting several threads on one machine share memory is tricky.
Today: Hardware Thread Support

Producer/Consumer: One thread writes A, one thread reads A.

Locks: Two threads share write access to A.

On Thursday: Multiprocessor memory system design and synchronization issues.

Thursday is a simplified overview -- graduate-level architecture courses spend weeks on this topic ...
How 2 threads share a queue ...

We begin with an empty queue ...

Thread 1 (T1) adds data to the tail of the queue.

"Producer" thread

Thread 2 (T2) takes data from the head of the queue.

"Consumer" thread
Producer adding $x$ to the queue ...

Before:

```
ORi R1, R0, xval ; Load x value into R1
LW R2, tail(R0)  ; Load tail pointer into R2
SW R1, 0(R2)     ; Store x into queue
ADDi R2, R2, 4   ; Shift tail by one word
SW R2 0(tail)    ; Update tail memory addr
```

After:

|   |   |   |   | x |   |

Words in Memory

Higher Address Numbers
Producer adding \( y \) to the queue ...

Before:

\[ \begin{array}{c|c|c|c|c} & & & & \\
\text{Tail} & & x & & \\
\text{Head} & & & & \end{array} \]

Words in Memory

Higher Address Numbers

T1 code (producer)

\[
\begin{align*}
\text{ORi R1, R0, yval} ; & \quad \text{Load \( y \) value into R1} \\
\text{LW R2, tail(R0)} ; & \quad \text{Load tail pointer into R2} \\
\text{SW R1, 0(R2)} ; & \quad \text{Store \( y \) into queue} \\
\text{ADDi R2, R2, 4} ; & \quad \text{Shift tail by one word} \\
\text{SW R2 0(tail)} ; & \quad \text{Update tail memory addr}
\end{align*}
\]

After:

\[ \begin{array}{c|c|c|c|c} & & y & & \\
\text{Tail} & & & & \\
\text{Head} & & x & & \end{array} \]

Words in Memory

Higher Address Numbers
Consumer reading the queue ...

Before:

```
     Tail
     y   x
     Head
```

**Words in Memory**

T2 code (consumer)

```
LW R3, head(R0)  ; Load head pointer into R3
spin: LW R4, tail(R0)  ; Load tail pointer into R4
BEQ R4, R3, spin  ; If queue empty, wait
LW R5, 0(R3)     ; Read x from queue into R5
ADDi R3, R3, 4   ; Shift head by one word
SW R3 head(R0)   ; Update head pointer
```

After:

```
     Tail
     y
     Head
```

**Words in Memory**

Higher Address Numbers
### What can go wrong?

<table>
<thead>
<tr>
<th>Before:</th>
<th>After:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
</tbody>
</table>

#### Higher Addresses

**T1 code (producer)**

- **ORi R1, R0, x**: Load x value into R1
- **LW R2, tail(R0)**: Load tail pointer into R2
- **SW R1, 0(R2)**: Store x into queue
- **ADDi R2, R2, 4**: Shift tail by one word
- **SW R2 0(tail)**: Update tail pointer

**T2 code (consumer)**

- **LW R3, head(R0)**: Load head pointer into R3
- **LW R4, tail(R0)**: Load tail pointer into R4
- **BEQ R4, R3, spin**: If queue empty, wait
- **LW R5, 0(R3)**: Read x from queue into R5
- **ADDi R3, R3, 4**: Shift head by one word
- **SW R3 head(R0)**: Update head pointer

---

**What if order is 2, 3, 4, 1?** Then, x is read before it is written! **The CPU running T1 has no way to know its bad to delay 1!**
Leslie Lamport: Sequential Consistency

Sequential Consistency: As if each thread takes turns executing, and instructions in each thread execute in program order.

<table>
<thead>
<tr>
<th>T1 code (producer)</th>
<th>T2 code (consumer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORi R1, R0, x ;</td>
<td>spin:</td>
</tr>
<tr>
<td>LW R2, tail(R0) ;</td>
<td>LW R3, head(R0) ;</td>
</tr>
<tr>
<td>SW R1, 0(R2) 1 ;</td>
<td>LW R4, tail(R0) 3</td>
</tr>
<tr>
<td>ADDi R2, R2, 4 ;</td>
<td>BEQ R4, R3, spin ;</td>
</tr>
<tr>
<td>SW R2 0(tail) 2 ;</td>
<td>LW R5, 0(R3) 4 ;</td>
</tr>
<tr>
<td></td>
<td>ADDi R3, R3, 4 ;</td>
</tr>
<tr>
<td></td>
<td>SW R3 head(R0) ;</td>
</tr>
</tbody>
</table>

Legal orders: 1, 2, 3, 4 or 1, 3, 2, 4 or 3, 4, 1, 2 ... but not 2, 3, 1, 4!

Sequential Consistent architectures get the right answer, but give up many optimizations.
Efficient alternative: Memory barriers

In the general case, machine is not sequentially consistent.

When needed, a memory barrier may be added to the program (a fence).

All memory operations before fence complete, then memory operations after the fence begin.

Ensures 1 completes before 2 takes effect.

Many MEMBAR variations for efficiency (versions that only affect loads or stores, certain memory regions, etc).
Producer/consumer memory fences

Before:

```
|   |   | y | x |
```

Higher Addresses

After:

```
|   |   | y |   |
```

Higher Addresses

**T1 code (producer)**

1. ORi R1, R0, x ; Load x value into R1
2. LW R2, tail(R0) ; Load queue tail into R2
3. SW R1, 0(R2) ; Store x into queue
4. MEMBAR ;
5. ADDi R2, R2, 4 ; Shift tail by one word
6. SW R2 0(tail) ; Update tail memory addr

**T2 code (consumer)**

1. LW R3, head(R0) ; Load queue head into R3
2. LW R4, tail(R0) ; Load queue tail into R4
3. BEQ R4, R3, spin ; If queue empty, wait
4. MEMBAR ;
5. LW R5, 0(R3) ; Read x from queue into R5
6. ADDi R3, R3, 4 ; Shift head by one word
7. SW R3 head(R0) ; Update head memory addr

Ensures 1 happens before 2, and 3 happens before 4.
TAs will provide “secret” MIPS machine code tests.

Bonus points if these tests run by 2 PM. If not, TAs give you test code to use over weekend.
**CS 152: What’s left ...**

- **Monday 5/2:** Final report due, 11:59 PM
  Watch email for final project peer review request.

- **Thursday 5/5:** Midterm II, 6 PM to 9 PM, 320 Soda.
  No class on Thursday. Review session in Tuesday 5/2, + HKN (???).

- **Tuesday 5/10:** Final presentations.
  Deadline to bring up grading issues: Tues 5/10@ 5PM. Contact John at lazaro@eecs
Sharing Write Access
One producer, two consumers ...

Before: \[ \text{y} \quad \text{x} \quad \text{Higher Addresses} \]

T1 code (producer)

- ORi R1, R0, x ; Load x value into R1
- LW R2, tail(R0) ; Load queue tail into R2
- SW R1, 0(R2) ; Store x into queue
- ADDi R2, R2, 4 ; Shift tail by one word
- SW R2 0(tail) ; Update tail memory addr

T2 & T3 (2 copies of consumer thread)

- LW R3, head(R0) ; Load queue head into R3
- spin: LW R4, tail(R0) ; Load queue tail into R4
- BEQ R4, R3, spin ; If queue empty, wait
- LW R5, 0(R3) ; Read x from queue into R5
- ADDi R3, R3, 4 ; Shift head by one word
- SW R3 head(R0) ; Update head memory addr

After: \[ \text{y} \quad \text{Higher Addresses} \]

Critical section: T2 and T3 must take turns running red code.
Abstraction: Semaphores (Dijkstra, 1965)

Semaphore: unsigned int s

s is initialized to the number of threads permitted in the critical section at once (in our example, 1).

\[ P(s) : \text{If } s > 0, \ s-- \text{ and return. Otherwise, sleep. When woken do } s-- \text{ and return.} \]

\[ V(s) : \text{Do } s++, \text{ awaken one sleeping process, return.} \]

Example use (initial s = 1):

\[ P(s); \]

\[ \text{critical section } (s=0) \]

\[ V(s); \]

When awake, \( V(s) \) and \( P(s) \) are atomic: no interruptions, with exclusive access to \( s \).
Spin-Lock Semaphores: Test and Set

An example atomic read-modify-write ISA instruction:

Test&Set(m, R)
R = M[m];
if (R == 0) then M[m]=1;

Note: With Test&Set(), the M[m]=1 state corresponds to last slide’s s=0 state!

Critical section

P: Test&Set R6, mutex(R0); Mutex check
   BNE R6, R0, P ; If not 0, spin

spin: LW R4, tail(R0) ; Load queue tail into R4
      BEQ R4, R3, spin ; If queue empty,
      LW R5, 0(R3) ; Read x from queue into R5
      ADDi R3, R3, 4 ; Shift head by one word
      SW R3 head(R0) ; Update head memory addr

V: SW R0 mutex(R0) ; Give up mutex

Assuming sequential consistency: 3 MEMBARs not shown ...

What if the OS swaps a process out while in the critical section? "High-latency locks", a source of Linux audio problems (and others)
Non-blocking synchronization ...

Another atomic read-modify-write instruction:

\[
\text{Compare\&Swap}(R_t, R_s, m) \\
\text{if }(R_t == M[m]) \\
\text{then} \\
\quad M[m] = R_s; \ R_s = R_t; \ \text{status} = \text{success}; \\
\text{else} \\
\quad \text{status} = \text{fail};
\]

Assuming sequential consistency: \text{MEMBAR}s not shown ...

\[
\begin{align*}
\text{try:} & \quad \text{LW } R_3, \ \text{head}(R_0) \quad ; \quad \text{Load queue head into } R_3 \\
\text{spin:} & \quad \text{LW } R_4, \ \text{tail}(R_0) \quad ; \quad \text{Load queue tail into } R_4 \\
& \quad \text{BEQ } R_4, R_3, \text{spin} \quad ; \quad \text{If queue empty, wait} \\
& \quad \text{LW } R_5, 0(R_3) \quad ; \quad \text{Read x from queue into } R_5 \\
& \quad \text{ADDi } R_6, R_3, 4 \quad ; \quad \text{Shift head by one word} \\
& \quad \text{Compare\&Swap } R_3, R_6, \ \text{head}(R_0); \text{ Try to update head} \\
& \quad \text{BNE } R_3, R_6, \text{try} \quad ; \quad \text{If not success, try again}
\end{align*}
\]

If R_3 \neq R_6, another thread got here first, so we must try again. If thread swaps out before Compare\&Swap, no latency problem; this code only "holds" the lock for one instruction!
Semaphores with just \texttt{LW} & \texttt{SW}? 

Can we implement semaphores with just normal load and stores? Yes!
Assuming sequential consistency ...

In practice, we create sequential consistency by using memory fence instructions ... so, not really “normal”.

Since load and store semaphore algorithms are quite tricky to get right, it is more convenient to use a \texttt{Test\&set} or \texttt{Compare\&swap} instead.
Memset: Memory fences, in lieu of full sequential consistency.

Test&Set: A spin-lock instruction for sharing write access.

Compare&Swap: A non-blocking alternative to share write access.