CS 152
Computer Architecture and Engineering

Lecture 23 – Synchronization

2005-11-17

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

TAs: David Marquardt and Udam Saini

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.
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 Tuesday: Multiprocessor memory system design and synchronization issues.

Tuesday 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:

Words in Memory

Tail  Head

Higher Address Numbers

T1 code (producer)

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:

Words in Memory

Tail  Head

Higher Address Numbers

x
Producer adding \textit{y} to the queue ...

\begin{itemize}
  \item \textbf{T1 code (producer)}
  \begin{align*}
    \text{ORi R1, R0, yval} & ; \quad \text{Load } y \text{ value into } R1 \\
    \text{LW R2, tail(R0)} & ; \quad \text{Load tail pointer into } R2 \\
    \text{SW R1, 0(R2)} & ; \quad \text{Store } y \text{ 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*}
\end{itemize}

Before:

\begin{itemize}
  \item Head
  \item Tail
  \item \hspace{2cm} x
  \item \hspace{2cm} x
  \item \hspace{2cm} x
  \item \hspace{2cm} x
\end{itemize}

After:

\begin{itemize}
  \item Head
  \item Tail
  \item y
  \item x
  \item x
  \item x
  \item x
\end{itemize}

\begin{itemize}
  \item Words in Memory
  \item Higher Address Numbers
  \item Tail
  \item Head
\end{itemize}
Consumer reading the queue ...

Before:

```
  y  x
```

```
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:

```
  y
```

T2 code (consumer)
What can go wrong?

Before: \( y \ x \)  

After: \( y \)  

T1 code (producer)

1. ORi R1, R0, x ; Load x value into R1  
2. SW R2, 0(R2) ; Store x into queue  
3. LW R3, head(R0) ; Load head pointer into R3  
4. LW R5, 0(R3) ; Read x from queue into R5

T2 code (consumer)

1. LW R2, tail(R0) ; Load tail pointer into R2  
2. ADDi R2, R2, 4 ; Shift tail by one word  
3. LW R4, tail(R0) ; Load tail pointer into R4  
4. BEQ R4, R3, spin ; If queue empty, wait  
5. ADDi R3, R3, 4 ; Shift head by one word  
6. 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!
Sequential Consistency: As if each thread takes turns executing, and instructions in each thread execute in program order.

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.

MEMBAR is expensive, but you only pay for it when you use it.

Many MEMBAR variations for efficiency (versions that only effect loads or stores, certain memory regions, etc).

```assembly
ORi R1, R0, x ;
LW R2, tail(R0) ;
SW R1, 0(R2) ;
MEMBAR
ADDi R2, R2, 4 ;
SW R2 0(tail) ;
```
Producer/consumer memory fences

Before:

Tail

y

x

Head

Higher Addresses

After:

Tail

y

Head

Higher Addresses

T1 code (producer)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORi R1, R0, x</td>
<td>Load x value into R1</td>
</tr>
<tr>
<td>LW R2, tail(R0)</td>
<td>Load queue tail into R2</td>
</tr>
<tr>
<td>SW R1, 0(R2)</td>
<td>Store x into queue</td>
</tr>
<tr>
<td>MEMBAR</td>
<td></td>
</tr>
<tr>
<td>ADDi R2, R2, 4</td>
<td>Shift tail by one word</td>
</tr>
<tr>
<td>SW R2 0(tail)</td>
<td>Update tail memory addr</td>
</tr>
</tbody>
</table>

T2 code (consumer)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R3, head(R0)</td>
<td>Load queue head into R3</td>
</tr>
<tr>
<td>spin: LW R4, tail(R0)</td>
<td>Load queue tail into R4</td>
</tr>
<tr>
<td>BEQ R4, R3, spin</td>
<td>If queue empty, wait</td>
</tr>
<tr>
<td>MEMBAR</td>
<td></td>
</tr>
<tr>
<td>LW R5, 0(R3)</td>
<td>Read x from queue into R5</td>
</tr>
<tr>
<td>ADDi R3, R3, 4</td>
<td>Shift head by one word</td>
</tr>
<tr>
<td>SW R3 head(R0)</td>
<td>Update head memory addr</td>
</tr>
</tbody>
</table>

Ensures 1 happens before 2, and 3 happens before 4.
Reminder: Final Checkoff this Friday!

Final report due following Monday, 11:59 PM

TAs will provide “secret” MIPS machine code tests.

Bonus points if these tests run by end of section. If not, TAs give you test code to use over weekend

Mid-term, project presentations after Thanksgiving.
CS 152: What’s left ...

Monday 11/21: Final report due, 11:59 PM
Class as normal on Tuesday, then Thanksgiving

Tuesday 11/29: Architecture @ Cal.
Team evaluations due 11:59 PM Tuesday ...

Thursday 12/1: Mid-term review in-class

Tuesday 12/6: Mid-term II, 6:00-9:00 PM.
No class 11-12:30 that day.

Thursday 12/8: Final presentations.
If you get done early on Monday ...

CITRIS Distinguished Speaker Series, 306 Soda Hall, Berkeley Campus

- November 21, 2005: 4:00pm - 5:00pm
- Contact: travisr@eecs.berkeley.edu

"High Performance Throughput Computing"

Dr. Marc Tremblay, Sun Microsystems
Sun Fellow and VP Chief Architect, Scalable Systems Group

Abstract:
Throughput Computing, achieved through a new generation of microprocessors composed of multiple multi-threaded cores, can lead to performance improvements that are 10 to 30x those of conventional processors and systems. In this talk I will discuss how the value of a robust, high-performance single thread leads to even higher throughput rates. I will also describe some of the techniques we are implementing in future mainstream processors that accomplish the somewhat conflicting goal of attacking both latency and throughput.

Sun’s chief architect, key player in Niagara ...
Sun announced T1 (Niagara) last week officially.
Designed to be difficult for hobbyists to get inside the box ...
3 IBM PowerPC ISA Cores
In-Order (Static Pipeline)
Dual-Issue Superscalar
Multi-threaded (2-way)
Shared 1 MB L2 Cache

Graphics chip is also north bridge, and contains the DRAM controller.

512 MB RAM, fixed size.
With heat sinks attached ...
Sharing Write Access
One producer, two consumers ...

Before:

|   |   | y | x |   |

Higher Addresses

After:

|   |   | y |   |   |

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
spun: 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

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).

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

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

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!

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

Critical section

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:

\[
\text{Compare\&Swap}(R_t, R_s, m) \\
\text{if } (R_t = M[m]) \\
\text{then} \\
\text{\hspace{1em}} M[m] = R_s; \ R_s = R_t; \ \text{status} = \text{success}; \\
\text{else} \\
\text{\hspace{1em}} \text{status} = \text{fail};
\]

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

try: \text{LW} \ R_3, \text{head}(R_0) \ ; \ \text{Load queue head into R3} \\
spin: \text{LW} \ R_4, \text{tail}(R_0) \ ; \ \text{Load queue tail into R4} \\
BEQ \ R_4, \ R_3, \text{spin} \ ; \ \text{If queue empty, wait} \\
\text{LW} \ R_5, 0(R_3) \ ; \ \text{Read x from queue into R5} \\
\text{ADDi} \ R_6, \ R_3, 4 \ ; \ \text{Shift head by one word} \\
\text{Compare\&Swap} \ R_3, \ R_6, \text{head}(R_0); \ \text{Try to update head} \\
\text{BNE} \ R_3, \ R_6, \text{try} \ ; \ \text{If not success, try again}

If \ R_3 \neq \ R_6, \text{another thread got here first, so we must try again.} \\
If thread swaps out before \text{Compare\&Swap}, \text{no latency problem; this code only "holds" the lock for one instruction!}
Semaphores with just LW & 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 Test&set or Compare&swap instead.
Conclusions: Synchronization

**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.