Last Time: NVidia 8800, a unified GPU

Thread processor sets shader type of each CPU

Streams loop around...

128 Shader CPUs

1.35 GHz Shader CPU Clock, 575 MHz core clock
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 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 ...

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>Tail</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Words in Memory

Higher Address Numbers

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:

![Queue Diagram Before](image)

T1 code (producer):

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

After:

![Queue Diagram After](image)
Producer adding \( y \) to the queue ...

Before:

\begin{verbatim}
ORI R1, R0, yval ; Load y value into R1
LW R2, tail(R0) ; Load tail pointer into R2
SW R1, 0(R2) ; Store y into queue
ADDI R2, R2, 4 ; Shift tail by one word
SW R2 0(tail) ; Update tail memory addr
\end{verbatim}

After:

\begin{verbatim}
\end{verbatim}

T1 code (producer)
Consumer reading the queue ...

Before:

Tail ≡ y

x

Head

Words in Memory

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

T2 code (consumer)

After:

Tail ≡ y

x

Head

Words in Memory
What can go wrong? (single-threaded LW/SW "contract")

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><strong>ORI R1, R0, x</strong> ; Load x value into R1</td>
<td></td>
</tr>
<tr>
<td><strong>LW R2, tail(R0)</strong> ; Load queue tail into R2</td>
<td></td>
</tr>
<tr>
<td><strong>SW R1, 0(R2)</strong> ; Store x into queue</td>
<td></td>
</tr>
<tr>
<td><strong>ADDI R2, R2, 4</strong> ; Shift tail by one word</td>
<td></td>
</tr>
<tr>
<td><strong>SW R3 head(R0)</strong> ; Update head memory addr</td>
<td></td>
</tr>
<tr>
<td><strong>LW R3, head(R0)</strong> ; Load queue head into R3</td>
<td></td>
</tr>
<tr>
<td><strong>spin:</strong> <strong>LW R4, tail(R0)</strong> ; Load queue tail into R4</td>
<td></td>
</tr>
<tr>
<td><strong>BEQ R4, R3, spin</strong> ; If queue empty, wait</td>
<td></td>
</tr>
<tr>
<td><strong>LW R5, 0(R3)</strong> ; Read x from queue into R5</td>
<td></td>
</tr>
<tr>
<td><strong>ADDI R3, R3, 4</strong> ; Shift head by one word</td>
<td></td>
</tr>
<tr>
<td><strong>SW R2 0(tail)</strong> ; Update tail memory addr</td>
<td></td>
</tr>
</tbody>
</table>

Sequentially Consistent: 1, 2, 3, 4 or 1, 3, 2, 4... but not 2, 3, 1, 4 or 2, 3, 4, 1!

Sequentially 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.

```
ORI R1, R0, x  
LW R2, tail(R0) 
SW R1, 0(R2) ; 1
MEMBAR
ADDI R2, R2, 4 
SW R2 0(tail) ; 2
```

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).
Producer/consumer memory fences

**T1 code (producer)**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ORI R1, R0, x</td>
<td>Load x value into R1</td>
</tr>
<tr>
<td>2</td>
<td>LW R2, tail(R0)</td>
<td>Load queue tail into R2</td>
</tr>
<tr>
<td>3</td>
<td>SW R1, 0(R2)</td>
<td>Store x into queue</td>
</tr>
<tr>
<td>4</td>
<td>MEMBAR</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ADDI R2, R2, 4</td>
<td>Shift tail by one word</td>
</tr>
<tr>
<td>6</td>
<td>SW R2 0(tail)</td>
<td>Update tail memory addr</td>
</tr>
</tbody>
</table>

**T2 code (consumer)**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LW R3, head(R0)</td>
<td>Load queue head into R3</td>
</tr>
<tr>
<td>2</td>
<td>spin: LW R4, tail(R0)</td>
<td>Load queue tail into R4</td>
</tr>
<tr>
<td>3</td>
<td>BEQ R4, R3, spin</td>
<td>If queue empty, wait</td>
</tr>
<tr>
<td>4</td>
<td>MEMBAR</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LW R5, 0(R3)</td>
<td>Read x from queue into R5</td>
</tr>
<tr>
<td>6</td>
<td>ADDI R3, R3, 4</td>
<td>Shift head by one word</td>
</tr>
<tr>
<td>7</td>
<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.
Sharing Write Access
One producer, two consumers ...

Before:

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

Higher Addresses

After:

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

Higher Addresses

**T1 code (producer)**

\[
\begin{align*}
\text{ORI R1, R0, x} & ; \text{ Load x value into R1} \\
\text{LW R2, tail(R0)} & ; \text{ Load queue tail into R2} \\
\text{SW R1, 0(R2)} & ; \text{ Store x into queue} \\
\text{ADDI R2, R2, 4} & ; \text{ Shift tail by one word} \\
\text{SW R2 0(tail)} & ; \text{ Update tail memory addr} \\
\end{align*}
\]

**T2 & T3 (2 copies of consumer thread)**

\[
\begin{align*}
\text{LW R3, head(R0)} & ; \text{ Load queue head into R3} \\
\text{spin: LW R4, tail(R0)} & ; \text{ Load queue tail into R4} \\
\text{BEQ R4, R3, spin} & ; \text{ If queue empty, wait} \\
\text{LW R5, 0(R3)} & ; \text{ Read x from queue into R5} \\
\text{ADDI R3, R3, 4} & ; \text{ Shift head by one word} \\
\text{SW R3 head(R0)} & ; \text{ Update head memory addr} \\
\end{align*}
\]

**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).

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

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

Example use (initial \( s = 1 \)):

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

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

An example atomic read-modify-write ISA instruction:

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

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

P: \text{ Test\&Set R6, mutex(R0); Mutex check} \\
\text{BNE R6, R0, P} \quad ; \text{ If not 0, spin}

\begin{align*}
\text{spin: LW R4, tail(R0)} & \quad ; \text{ Load queue tail into R4} \\
\text{BEQ R4, R3, spin} & \quad ; \text{ If queue empty,} \\
\text{LW R5, 0(R3)} & \quad ; \text{ Read x from queue into R5} \\
\text{ADDI R3, R3, 4} & \quad ; \text{ Shift head by one word} \\
\text{SW R3 head(R0)} & \quad ; \text{ Update head memory addr}
\end{align*}

V: \text{ 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:

\[
\text{Compare\&Swap}(Rt, Rs, m) \\
\text{if} \ (Rt == M[m]) \text{then} \\
\quad M[m] = Rs; \ Rs = Rt; \ /* \ do \ swap */ \text{else} \\
\quad /* \ do \ not \ swap */
\]

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

\[
\text{try:} \ \text{LW} \ R3, \ \text{head}(R0) \ ; \ \text{Load \ queue \ head \ into \ R3} \\
\text{spin:} \ \text{LW} \ R4, \ \text{tail}(R0) \ ; \ \text{Load \ queue \ tail \ into \ R4} \\
\quad \text{BEQ} \ R4, R3, \text{spin} \ ; \ \text{If \ queue \ empty, \ wait} \\
\quad \text{LW} \ R5, 0(R3) \ ; \ \text{Read \ x \ from \ queue \ into \ R5} \\
\quad \text{ADDI} \ R6, R3, 4 \ ; \ \text{Shift \ head \ by \ one \ word} \\
\quad \text{Compare\&Swap} \ R3, R6, \text{head}(R0); \ \text{Try \ to \ update \ head} \\
\quad \text{BNE} \ R3, R6, \text{try} \ ; \ \text{If \ not \ success, \ try \ again}
\]

If R3 ≠ R6, another thread got here first, so we must try again. If thread swaps out before \text{Compare\&Swap}, 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.