Agenda

• Brief motivation
• Consistency vs Coherence
• Synchronization
  – Fences
  – Mutexs, locks, semaphores
  – Hardware
• Coherence
• Snoopy
  – MSI, MESI
Symmetric Multiprocessors

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

**symmetric**
Why Would We Want Asymmetry?
Cache Coherence vs. Memory Consistency

- A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors, for one memory address
  - i.e., updates are not lost

- No guarantee of when an update should be seen
- No guarantee of what order of updates (of different addresses) should be seen

- A cache coherence protocol is not enough to ensure sequential consistency
  - But if sequentially consistent, then caches must be coherent
Cache Coherence vs. Memory Consistency

- A memory consistency model gives the rules on when a write by one processor can be observed by a read on another, across different addresses
  - As previously seen with examples

- Combination of cache coherence protocol plus processor memory reorder buffer used to implement a given architecture’s memory consistency model
Synchronization

The need for synchronization arises whenever there are concurrent processes in a system (even in a uniprocessor system)

Two classes of synchronization:

**Producer-Consumer**: A consumer process must wait until the producer process has produced data

**Mutual Exclusion**: Ensure that only one process uses a resource at a given time
A Producer-Consumer Example \textit{continued}

Producer posting Item x:

1. Load $R_{\text{tail}}$, (\textit{tail})
2. Store ($R_{\text{tail}}$), x
3. $R_{\text{tail}} = R_{\text{tail}} + 1$
4. Store (\textit{tail}), $R_{\text{tail}}$

Can the tail pointer get updated before the item x is stored?

Consumer:

1. Load $R_{\text{head}}$, (\textit{head})
2. \textit{spin:}
3. Load $R_{\text{tail}}$, (\textit{tail})
4. if $R_{\text{head}} == R_{\text{tail}}$ goto \textit{spin}
5. Load $R$, (\textit{R}_{\text{head}})
6. $R_{\text{head}} = R_{\text{head}} + 1$
7. Store (\textit{head}), $R_{\text{head}}$
8. process(R)

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:

1. 2, 3, 4, 1
2. 4, 1, 2, 3
Sequential Consistency

A Memory Model

“A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

*Leslie Lamport*

Sequential Consistency =

arbitrary *order-preserving interleaving*

of memory references of sequential programs
Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y  (initially X = 0, Y = 10)

T1:
- Store (X), 1  (X = 1)
- Store (Y), 11  (Y = 11)

T2:
- Load R₁, (Y)
- Store (Y'), R₁  (Y' = Y)
- Load R₂, (X)
- Store (X'), R₂  (X' = X)

what are the legitimate answers for X' and Y' ?

(X', Y') ∈ {(1,11), (0,10), (1,10), (0,11)}  ?

If y is 11 then x cannot be 0
Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies.

What are these in our example?

T1:
- Store (X), 1 (X = 1)
- Store (Y), 11 (Y = 11)

T2:
- Load R₁, (Y)
- Store (Y’), R₁ (Y’ = Y)
- Load R₂, (X)
- Store (X’), R₂ (X’ = X)

additional SC requirements

Does (can) a system with caches or out-of-order execution capability provide a sequentially consistent view of the memory?

more on this later
Implementation of SC is complicated by two issues

- **Out-of-order execution capability**
  
  Load(a); Load(b) yes
  
  Load(a); Store(b) yes if a ≠ b
  
  Store(a); Load(b) yes if a ≠ b
  
  Store(a); Store(b) yes if a ≠ b

- **Caches**
  
  Caches can prevent the effect of a store from being seen by other processors

  No common commercial architecture has a sequentially consistent memory model!
Memory Fences

Instructions to sequentialize memory accesses

Processors with *relaxed or weak memory models* (i.e., permit Loads and Stores to different addresses to be reordered) need to provide *memory fence* instructions to force the serialization of memory accesses.

*Examples of processors with relaxed memory models:*
  - Sparc V8 (TSO,PSO): Membar
  - Sparc V9 (RMO):
    - Membar #LoadLoad, Membar #LoadStore
    - Membar #StoreLoad, Membar #StoreStore
  - PowerPC (WO): Sync, EIEIO
  - ARM: DMB (Data Memory Barrier)
  - X86/64: mfence (Global Memory Barrier)

*Memory fences are expensive operations, however, one pays the cost of serialization only when it is required.*
N-process Mutual Exclusion

*Lamport’s Bakery Algorithm*

*Process i*

**Entry Code**

```
choosing[i] = 1;
num[i] = max(num[0], ..., num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++) {
    while( choosing[j] );
    while( num[j] &&
        (( num[j] < num[i] ) ||
        ( num[j] == num[i] && j < i )));
}
```

**Exit Code**

```
um[i] = 0;
```
Locks or Semaphores
E. W. Dijkstra, 1965

A *semaphore* is a non-negative integer, with the following operations:

\[ \text{P}(s): \text{if } s > 0, \text{ decrement } s \text{ by } 1, \text{ otherwise wait} \]

\[ \text{V}(s): \text{ increment } s \text{ by } 1 \text{ and wake up one of the waiting processes} \]

P’s and V’s must be executed atomically, i.e., without
- *interruptions* or
- *interleaved accesses to* \( s \) *by other processors*

**Process i**

\[
\begin{align*}
\text{P}(s) & \quad \text{P(s)} \\
\text{<critical section>} & \quad <\text{critical section}> \\
\text{V}(s) & \quad \text{V(s)}
\end{align*}
\]

*Initial value of* \( s \) *determines the maximum no. of processes in the critical section*
Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:

*atomic read-modify-write instructions*

Examples: *m is a memory location, R is a register*

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

Fetch&Add (m), R\text{, } R\text{, } R:\n\[
R \leftarrow M[m]; \\
M[m] \leftarrow R + R\text{, V;}
\]

Swap (m), R:
\[
R_t \leftarrow M[m]; \\
M[m] \leftarrow R; \\
R \leftarrow R_t;
\]
Multiple Consumers Example

*using the Test&Set Instruction*

**P:**
- Test&Set (mutex), $R_{temp}$
- if ($R_{temp}! = 0$) goto P

**spin:**
- Load $R_{head}$, (head)
- Load $R_{tail}$, (tail)
- if $R_{head} == R_{tail}$ goto spin
- Load $R$, ($R_{head}$)
- $R_{head} = R_{head} + 1$
- Store (head), $R_{head}$

**V:**
- Store (mutex), 0
- process(R)

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P’s and V’s

What if the process stops or is swapped out while in the critical section?
Nonblocking Synchronization

Compare&Swap(m), R_t, R_s:
  if (R_t == M[m])
    then M[m] = R_s;
    R_s = R_t;
    status ← success;
  else status ← fail;

status is an implicit argument

try:
  Load R_{head}, (head)
spin:
  Load R_{tail}, (tail)
  if R_{head} == R_{tail} goto spin
  Load R, (R_{head})
  R_{newhead} = R_{head} + 1
  Compare&Swap(head), R_{head}, R_{newhead}
  if (status == fail) goto try
process(R)
Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserve $R, (m)$:
- $\langle \text{flag, addr} \rangle \leftarrow \langle 1, m \rangle$;
- $R \leftarrow M[m]$;

Store-conditional $\langle m, R \rangle$:
- if $\langle \text{flag, addr} \rangle == \langle 1, m \rangle$
  - then cancel other procs’ reservation on $m$;
  - $M[m] \leftarrow R$;
  - status $\leftarrow$ succeed;
- else status $\leftarrow$ fail;

try:
- Load-reserve $R_{\text{head}}, (\text{head})$
- Load $R_{\text{tail}}, (\text{tail})$
- if $R_{\text{head}} == R_{\text{tail}}$ goto spin
- Load $R, (R_{\text{head}})$
- $R_{\text{head}} = R_{\text{head}} + 1$
- Store-conditional $(\text{head}), R_{\text{head}}$
- if (status==fail) goto try
- process($R$)

spin:
Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserve R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];

try:
    Load-reserve \(R_{head}\), (head)
    Load \(R_{tail}\), (tail)
    if \(R_{head} == R_{tail}\) goto spin
    Load R, (\(R_{head}\))
    \(R_{head} = R_{head} + 1\)
    Store-conditional (head), \(R_{head}\)
    if (status==fail) goto try
    process(R)

spin:
    Store-conditional (m), R:
    \(\text{if } <\text{flag, adr}> == <1, m>\)
    \(\text{then cancel other procs’ reservation on m;}
    \ M[m] ← R;
    \text{status ← succeed;}
    \(\text{else status ← fail;}
    \)
Performance of Locks

Blocking atomic read-modify-write instructions
  *e.g., Test&Set, Fetch&Add, Swap*
  vs
Non-blocking atomic read-modify-write instructions
  *e.g., Compare&Swap, Load-reserve/Store-conditional*
  vs
Protocols based on ordinary Loads and Stores

*Performance depends on several interacting factors:*
  degree of contention,
  caches,
  out-of-order execution of Loads and Stores

*later ...*
Amdahl’s Law

Begins with Simple Software Assumption (Limit Arg.)

Fraction $F$ of execution time perfectly parallelizable

No Overhead for Scheduling Communication, Synchronization, etc.

$F$ is the Parallel Part

Fraction $1 - F$ Completely Serial

Time on 1 core $= (1 - F) / 1 + F / 1 = 1$

Time on $N$ cores $= (1 - F) / 1 + F / N$
Suppose CPU-1 updates A to 200.

- **write-back:** memory and cache-2 have stale values
- **write-through:** cache-2 has a stale value

Do these stale values matter?

What is the view of shared memory for programming?
Maintaining Cache Coherence

- Hardware support is required such that
  - only one processor at a time has write permission for a location
  - no processor can load a stale copy of the location after a write
- cache coherence protocols
Use snoopy mechanism to keep all processors’ view of memory coherent
Cache State Transition Diagram

*The MSI protocol*

*Each* cache line has state bits

- **M**: Modified
- **S**: Shared
- **I**: Invalid

---

- **Address tag**
- **State bits**

---

- **Write miss** (P₁ gets line from memory)
- **Other processor reads** (P₁ writes back)
- **Read miss** (P₁ gets line from memory)
- **Read by any processor**
- **Other processor intents to write**
- **Cache state in processor P₁**
MESI: An Enhanced MSI protocol
increased performance for private data

Each cache line has a tag

M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid

Address tag

Cache state in processor \( P_1 \)

Write miss

Read miss, shared

Read by any processor

Other processor reads

\( P_1 \) writes back

\( P_1 \) write or read

Other processor intent to write

\( P_1 \) intent to write

Other processor reads

\( P_1 \) reads

\( P_1 \) read

Read miss, not shared

Other processor intent to write

Other processor intent to write, \( P_1 \) writes back
Optimized Snoop with Level-2 Caches

- Processors often have two-level caches
  - small L1, large L2 (on chip)
- Inclusion property: entries in L1 must be in L2
  invalidation in L2 => invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

What problem could occur?
False Sharing

| state | line addr | data0 | data1 | ... | dataN |

A cache line contains more than one word

Cache-coherence is done at the line-level and not word-level

Suppose $M_1$ writes $\text{word}_i$ and $M_2$ writes $\text{word}_k$ and both words have the same line address.

*What can happen?*
Out-of-Order Loads/Stores & CC

**Blocking caches**
One request at a time + CC ⇒ SC

**Non-blocking caches**
Multiple requests (different addresses) concurrently + CC ⇒ Relaxed memory models

CC ensures that all processors observe the same order of loads and stores to an address
Questions?