EC 513
Computer Architecture

Synchronization and Sequential Consistency

Prof. Michel A. Kinsy
Architecture Design Challenge

- Relatively easy to get two of three, harder to get all three!

Uniprocessor | ASIC | Superscalar/VLIW

Current application trends demand:
- Performance-Awareness,
- Power-Awareness,
- Timing-Awareness,
- Security-Awareness
Architecture Taxonomy

Processor Organizations

- Single instruction, single data stream (SISD)
  - Uniprocessor
- Single instruction multiple data stream (SIMD)
  - Vector Processor
- Multiple instruction, single data stream (MISD)
  - Array Processor
- Multiple instruction, multiple data stream (MIMD)
  - Shared Memory (Tightly Coupled)
  - Distributed Memory (Loosely Coupled)
  - Symmetric Multiprocessor (SMP)
  - Nonuniformed Memory Access (NUMA)
  - Cluster

Parallelism Paradigms: Instruction level, Data level and Task level Parallelisms
Architecture Taxonomy

Processor Organizations

- Single instruction, single data stream (SISD)
  - Uniprocessor
- Single instruction multiple data stream (SIMD)
  - Vector Processor
  - Array Processor
- Multiple instruction, single data stream (MISD)
- Multiple instruction, multiple data stream (MIMD)

- Shared Memory (Tightly Coupled)
  - Symmetric Multiprocessor (SMP)
- Distributed Memory (Loosely Coupled)
  - Nonuniformed Memory Access (NUMA)
  - Cluster

Parallelism Paradigms: Instruction level, Data level and Task level Parallelisms

Presented RISC-V Architecture
Symmetric Multiprocessors (SMP)

- A collection of processors and a collection of memory connected through some interconnect
- Symmetric because latency for any processor to access any memory is constant – uniform memory access (UMA)
Shared Memory Architectures

- Key differentiating feature: the address space is shared, i.e., any processor can directly address any memory location and access them with load/store instructions.
Distributed Memory Multiprocessors

- Processor 1
- Cache
- Memory 1

- Processor 2
- Cache
- Memory 2

- ... (repeat for M memories)
- Processor N
- Cache
- Memory M

Interconnection Network

Department of Electrical & Computer Engineering
Distributed Memory Multiprocessors

- Each processor has local memory that is accessible through a fast interconnect
- The different nodes are connected as I/O devices with (potentially) slower interconnect
- Local memory access is a lot faster than remote memory
  - Nonuniform memory access (NUMA)
Parallel Architectures

- The parallel computers are classified as
  - Shared memory
  - Distributed memory
- Both shared and distributed memory systems have:
  - Processors: now generally commodity processors
  - Memory: now general commodity DRAM/DDR
  - Network/interconnect: between the processors or memory
Synchronization

- The need for synchronization arises whenever there are parallel processes in a system (even in a uniprocessor system)
  - Forks and Joins: A parallel process may want to wait until several events have occurred
  - Producer-Consumer: A consumer process must wait until the producer process has produced data
  - Exclusive use of a resource: Operating System has to ensure that only one process uses a resource at a given time
The program is written assuming instructions are executed in order

- Problems?
Producer-Consumer

Producer posting Item x:

1. Load $R_{tail}$, (tail)
2. Store (R_tail), x
   $R_{tail} = R_{tail} + 1$
3. Store tail, $R_{tail}$

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

- Programmer assumes that if 3 happens after 2, then 4 happens after 1
- Problem sequences are
  - 2, 3, 4, 1
  - 4, 1, 2, 3

Consumer:

1. Load $R_{head}$, (head)
2. spin:
   - Load $R_{tail}$, (tail)
   - if $R_{head} == R_{tail}$ goto spin
3. Load $R$, (R_head)
   $R_{head} = R_{head} + 1$
4. Store head, $R_{head}$
   process(R)
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 R_1, (Y)
- Store Y', R_1 (Y' = Y)
- Load R_2, (X)
- Store X', R_2 (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
Multiple Consumer Example

Producer posting item x:
Load $R_{tail}$, (tail)
Store ($R_{tail}$), x
$R_{tail} = R_{tail} + 1$
Store tail, $R_{tail}$

Consumer:

Consumer 1:
Load $R_{head}$, (head)

Consumer 2:
Load $R_{tail}$, (tail)
if $R_{head} == R_{tail}$ goto spin
Load R, ($R_{head}$)
$R_{head} = R_{head} + 1$
Store head, $R_{head}$

process(R)

What is wrong with this code?
Locks or Semaphores

- E. W. Dijkstra, 1965

- A semaphore is a non-negative integer, with the following operations
  - P(s): if s > 0, decrement s by 1, otherwise wait V(s)
    - Increment s by 1 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

<table>
<thead>
<tr>
<th>Process i</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(s)</td>
</tr>
<tr>
<td>&lt;critical section&gt;</td>
</tr>
<tr>
<td>V(s)</td>
</tr>
</tbody>
</table>

*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 ← M[m];
if R==0 then
M[m] ← 1;

**Swap (m), R:**

R_t ← M[m];
M[m] ← R;
R ← R_t;

**Fetch&Add (m), R_v, R:**

R ← M[m];
M[m] ← R + R_v;
Multiple Consumers Example

- Using the Test&Set Instruction

P:
- Test&Set (mutex), \( R_{temp} \)
- if \( (R_{temp} \neq 0) \) goto P

spin:
- Load \( R_{head} \), (head)
- Load \( R_{tail} \), (tail)
- if \( R_{head} \neq R_{tail} \) 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_t = R_s;
status ← success;
else status ← fail;

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)

status is an implicit argument
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];`

**Store-conditional (m), R:**
- `if <flag, adr> == <1, m>
  then cancel other procs’ reservation on m;
  M[m] ← R;
  status ← succeed;
else status ← fail;`

**try:** Load-reserve `R_{head}, (head)`
**spin:** 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)
Mutual Exclusion Using Load/Store

- A protocol based on two shared variables \(c_1\) and \(c_2\).
- Initially, both \(c_1\) and \(c_2\) are 0 (not busy)

Process 1

```
...  
c1=1;  
L:   if c2=1 then go to L  
    < critical section>  
c1=0;
```

Process 2

```
...  
c2=1;  
L:   if c1=1 then go to L  
    < critical section>  
c2=0;
```

What is wrong? Deadlock!
Mutual Exclusion: Second Attempt

- To avoid deadlock, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting

**Process 1**

```
L: c1=1;
   if c2=1 then
      { c1=0; go to L}
      < critical section>
   c1=0
```

**Process 2**

```
L: c2=1;
   if c1=1 then
      { c2=0; go to L}
      < critical section>
   c2=0
```

What could go wrong?

- **Livelock!**
- **Starvation!**
A Protocol for Mutual Exclusion

T. Dekker, 1966

- A protocol based on 3 shared variables $c_1$, $c_2$ and $\text{turn}$.
  Initially, both $c_1$ and $c_2$ are 0 (not busy)

- $\text{turn} = i$ ensures that only process $i$ can wait.
  Variables $c_1$ and $c_2$ ensure mutual exclusion.

- Solution for $n$ processes was given by Dijkstra and is quite involved!

\begin{align*}
\text{Process 1} & \quad \text{Process 2} \\
\ldots & \quad \ldots \\
\text{c1}=1; & \quad \text{c2}=1; \\
\text{turn} = 1; & \quad \text{turn} = 2; \\
\text{L: } & \quad \text{L:} \\
\text{if c2}=1 & \quad \text{if c1}=1 \\
\text{& turn}=1 & \quad \text{& turn}=2 \\
\text{then go to L} & \quad \text{then go to L} \\
\text{< critical section> } & \text{< critical section> } \\
\text{c1}=0; & \quad \text{c2}=0;
\end{align*}
## Analysis of Dekker’s Algorithm

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process 1</strong></td>
<td><strong>Process 1</strong></td>
</tr>
<tr>
<td>( c_1 = 1; )</td>
<td>( c_1 = 1; )</td>
</tr>
<tr>
<td>( \text{turn} = 1; )</td>
<td>( \text{turn} = 1; )</td>
</tr>
<tr>
<td><strong>L:</strong> if ( c_2 = 1 ) &amp; ( \text{turn} = 1 )</td>
<td><strong>L:</strong> if ( c_2 = 1 ) &amp; ( \text{turn} = 1 )</td>
</tr>
<tr>
<td>then go to <strong>L</strong></td>
<td>then go to <strong>L</strong></td>
</tr>
<tr>
<td>(&lt; \text{critical section}&gt; )</td>
<td>(&lt; \text{critical section}&gt; )</td>
</tr>
<tr>
<td>( c_1 = 0; )</td>
<td>( c_1 = 0; )</td>
</tr>
</tbody>
</table>

| **Process 2**  | **Process 2**  |
| \( c_2 = 1; \)  | \( c_2 = 1; \)  |
| \( \text{turn} = 2; \)  | \( \text{turn} = 2; \)  |
| **L:** if \( c_1 = 1 \) & \( \text{turn} = 2 \)  | **L:** if \( c_1 = 1 \) & \( \text{turn} = 2 \)  |
| then go to **L**  | then go to **L**  |
| \(< \text{critical section}> \)  | \(< \text{critical section}> \)  |
| \( c_2 = 0; \)  | \( c_2 = 0; \)  |
Implementing Issues of SC

- 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
Implementing Issues of SC

- Implementation of SC is complicated by two issues
  - Caches
    - Caches can prevent the effect of a store from being seen by other processors
Memory Fences

- Processors with relaxed or weak memory models
  - They permit Loads and Stores to different addresses to be reordered
  - They need to provide memory fence instructions to force the serialization of memory accesses
  - Examples of processors with relaxed memory models:
    - PowerPC (WO): Sync, EIEIO
    - Sparc V8 (TSO,PSO): Membar
    - Sparc V9 (RMO):
      - Membar #LoadLoad, Membar #LoadStore
      - Membar #StoreLoad, Membar #StoreStore

- Memory fences are expensive operations, however, one pays the cost of serialization only when it is required
Using Memory Fences

Producer posting item x:
- Load \( R_{tail} \), (tail)
- Store \( (R_{tail}) \), x
- Membar_{SS}
- \( R_{tail} = R_{tail} + 1 \)
- Store tail, \( R_{tail} \)

Consumer:
- Load \( R_{head} \), (head)
- spin:
  - Load \( R_{tail} \), (tail)
  - if \( R_{head} == R_{tail} \) goto spin
- Membar_{LL}
- Load \( R \), (\( R_{head} \))
- \( R_{head} = R_{head} + 1 \)
- Store head, \( R_{head} \)
- process(\( R \))

What does this do?
Data-Race Free Programs

- Synchronization variables (e.g., mutex) are disjoint from data variables
  - Accesses to writable shared data variables are protected in critical regions
    - No data races except for locks (formal definition is elusive)
- In general, it cannot be proven if a program is data-race free.

Process 1
```
... Acquire(mutex);
< critical section>
Release(mutex);
```

Process 2
```
... Acquire(mutex);
< critical section>
Release(mutex);
```
Fences in Data-Race Free Programs

- Relaxed memory model allows reordering of instructions by the compiler or the processor as long as the reordering is not done across a fence.
- The processor also should not speculate or prefetch across fences.

```
Process 1
...
Acquire(mutex);
membar;
< critical section>
membar;
Release(mutex);

Process 2
...
Acquire(mutex);
membar;
< critical section>
membar;
Release(mutex);
```
Next Class

- Cache Coherence - Snoopy Cache Coherence