Unit 12: Memory Consistency Models

Includes slides originally developed by Prof. Amir Roth

Example #1

initialization

```
int x = 0;
int y = 0;
```

thread 1

```
y = 1;
x = 1;
```

thread 2

```
int t1 = x;
int t2 = y;
print(t1,t2)
```

- What outcomes are allowed?

Example #2 (Somewhat More Real)

initialization

```
int A = 0;
int flag = 0;
```

thread 1

```
A = 1;
flag = 1;
```

thread 2

```
while (flag == 0) {
  // do nothing
}
print(A);
```

- What outcomes are allowed?

Example #3

initialization

```
int x = 0;
int y = 0;
```

thread 1	thread 2
`y = 1;` `print(x);`	`x = 1;` `print(y);`

- What outcomes are allowed?

Example #4 (Double Checked Locking)

initialization

```
int* ptr = NULL;
int val = 0;
```

thread 1

```
if (ptr == NULL) {
  acquire(lock);
  if (ptr == NULL) {
    val = 1;
    ptr = &val;
  }
  release(lock);
}
print(*ptr);
```

thread 2

```
if (ptr == NULL) {
  acquire(lock);
  if (ptr == NULL) {
    val = 1;
    ptr = &val;
  }
  release(lock);
}
print(*ptr);
```

- What outcomes are allowed?

Example #4 (Double Checked Locking)

initialization

```
int* ptr = NULL;
int val = 0;
```

thread 1

```
if (ptr == NULL) {
  acquire(lock);
  if (ptr == NULL) {
    val = 1;
    ptr = &val;
  }
  release(lock);
}
print(*ptr);
```

thread 2

```
if (ptr == NULL) {
  acquire(lock);
  if (ptr == NULL) {
    val = 1;
    ptr = &val;
  }
  release(lock);
}
print(*ptr);
```

- What outcomes are allowed?

Example #5

initialization

```
int x = 0;
int y = 0;
```

thread 1

```
if (x != 0) {
  y = 1;
}
```

thread 2

```
y = 2;
print(y);
```

- What outcomes are allowed?

Example #6

initialization

```
int x = 0;
int y = 0;
```

thread 1

```
y = ((x != 0) ? 1 : y)
```

thread 2

```
y = 2;
print(y);
```

- What outcomes are allowed?

What is Going On?

- Memory reordering
- In the compiler
  - Compiler is generally allowed to re-order memory operations to different addresses
  - Many other compiler optimizations also cause problems
- In the hardware
  - To tolerate write latency
    - Processes don't wait for writes to complete
    - And why should they? No reason on a uniprocessors
  - To simplify out-of-order execution

Memory Consistency

- **Memory coherence**
  - Creates globally uniform (consistent) view…
  - Of **a single memory location** (in other words: cache line)
  - Not enough
    - Cache lines A and B can be individually consistent…
    - But inconsistent with respect to each other

- **Memory consistency**
  - Creates globally uniform (consistent) view…
  - Of **all memory locations relative to each other**

- Who cares? Programmers
  - Globally inconsistent memory creates mystifying behavior

Coherence vs. Consistency

```
                A=0   flag=0
Processor 0           Processor 1
A=1;                  while (!flag); // spin
flag=1;               print A;
```

- **Intuition says**: P1 prints A=1
- **Coherence says**: absolutely nothing
  - P1 can see P0's write of `flag` before write of `A`!!! How?
    - **P0 has a coalescing store buffer that reorders writes**
    - **Or out-of-order execution**
    - **Or compiler re-orders instructions**
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **Real systems** act in this strange manner
  - What is allowed is defined as part of the ISA of the processor

Store Buffers & Consistency

```
                A=0   flag=0
Processor 0           Processor 1
A=1;                  while (!flag); // spin
flag=1;               print A;
```

- Consider the following execution:
  - Processor 0's write to A, misses the cache. Put in store buffer
  - Processor 0 keeps going
  - Processor 0 write "1" to flag hits, completes
  - Processor 1 reads flag… sees the value "1"
  - Processor 1 exits loop
  - Processor 1 prints "0" for A
- Ramification: store buffers can cause "strange" behavior
  - How strange depends on lots of things

Hardware Memory Consistency Models

- **Sequential consistency (SC)** (MIPS, PA-RISC)
  - **Formal definition of memory view programmers expect**
  - Processors see their own loads and stores in program order
    + Provided naturally, even with out-of-order execution
  - But also: processors see others' loads and stores in program order
  - And finally: all processors see same global load/store ordering
    – Last two conditions not naturally enforced by coherence
  - Corresponds to some sequential interleaving of uniprocessor orders
  - **Indistinguishable from multi-programmed uni-processor**
- **Processor consistency (PC)** (x86, SPARC)
  - Allows a in-order store buffer
    - Stores can be deferred, but must be put into the cache in order
- **Release consistency (RC)** (ARM, Itanium, PowerPC)
  - Allows an un-ordered store buffer
    - Stores can be put into cache in any order. Loads re-ordered, too.

Reordering Loads and Stores

initialization

```
int x = 0;
int y = 0;
```

thread 1

```
y = 1;
x = 1;
```

thread 2

```
int t1 = x;
int t2 = y;
print(t1,t2)
```

- Allowed by some processors today
  - PowerPC, ARM, Itanium

Delaying Stores (Past Loads)

initialization

```
int x = 0;
int y = 0;
```

thread 1	thread 2
`y = 1;` `print(x);`	`x = 1;` `print(y);`

- Allowed by most hardware today
  - PowerPC, ARM, Itanium
  - Plus: SPARC TSO, Intel/AMD x86

Restoring Order

- Sometimes we need ordering (mostly we don't)
  - Prime example: ordering between "lock" and data
- How? insert **Fences (memory barriers)**
  - Special instructions, part of ISA
- Example
  - Ensure that loads/stores don't cross lock acquire/release operation

```
acquire
fence
critical section
fence
release
```

- How do fences work?
  - They stall execution until write buffers are empty
  - Makes lock acquisition and release slow(er)
- **Use synchronization library, don't write your own**