Lecture 25:
Synchronization primitives

Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
Last time: Symmetric multiprocessing (SMP)

- SMP only works because of caches!
- Shared memory rapidly becomes bottleneck
Last time:
Coherency and Consistency

1. Coherency:
   – Values in caches all match each other
   – Processors all see a coherent view of memory

2. Consistency:
   – The order in which changes to memory are seen by different processors
Last time:
Sequential consistency

1. Operations from a processor appear (to all others) in *program order*

2. Every processor’s *visibility order* is the same interleaving of all the program orders.

Requirements:

- Each processor issues memory ops in program order
- RAM totally orders all operations
- Memory operations are atomic
Last time: cache coherence protocols

Terminology:
- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl
- BusWr: bus write

Processor-initiated

Snoop-initiated

MESI Protocol

PrRd → Issue BusRd, if shared...
PrRd → If line not shared
PrRd → No transaction
PrRd, PrWr → No transaction

PrWr → issue BusRdX
PrWr → No transaction

BusRdX → discard
BusRdX → discard
BusRdX → Write back

BusRd → Signal HIT
BusRd → Signal HIT
BusRd → Write back

Shared

Exclusive

Invalid

Modified
Other consistency models

<table>
<thead>
<tr>
<th></th>
<th>Alpha</th>
<th>PA-RISC</th>
<th>Power</th>
<th>X86_32</th>
<th>X86_64</th>
<th>ia64</th>
<th>zSeries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads after reads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reads after writes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Writes after reads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Writes after writes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dependent reads</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ifetch after write</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

*Icache is incoherent: requires explicit Icache flushes for self-modifying code*

*Read of value can be seen before read of address of value!*

*Not shown: SPARC, which supports 3 different memory models*

*Portable languages like Java must define their own memory model, and enforce it!*

---

- **Alpha**: 
- **PA-RISC**: 
- **Power**: 
- **X86_32**: 
- **X86_64**: 
- **ia64**: 
- **zSeries**:
Today

- Barriers and fences
- Test and Set
- Compare and Swap
- Load-Locked / Store Conditional
- Limits of symmetric multiprocessing
- Simultaneous multithreading (SMT)
Barriers and Fences

• General rule:
  
  the weaker the consistency model is, the faster/cheaper it goes in hardware

• We’ve seen that visibility order is:
  
  – Essential for correct functioning of some algorithms
  – Difficult to guarantee with many compilers and memory models

• Solution is to use **barriers** (also called **fences**)
  
  – **Compiler** barriers: prevent compiler reordering statements
  – **Memory** barriers: prevent CPU reordering instructions
Compiler barriers

- Prevent compiler from reordering visible loads and stores
  - May still reorder register access (private)
- Typically part of compiler intrinsics
- **GCC:**
  ```
  __asm__ __volatile__ ("" ::: "memory");
  ```
- **Intel ECC:**
  ```
  __memory_barrier()
  ```
- **Microsoft Visual C & C++:**
  ```
  __ReadWriteBarrier()
  ```
Memory barriers on x86

- **MFENCE** instruction
  - Prevents the CPU reordering any loads or stores past it

Also:
- LFENCE: loads
- SFENCE: stores
Today

- Barriers and fences
- Test and Set
- Compare and Swap
- Load-Locked / Store Conditional
- Limits of symmetric multiprocessing
- Simultaneous multithreading (SMT)
Synchronization

Two ways to synchronize:

1. **Atomic operations** on shared memory
   - e.g.: test-and-set, compare-and-swap
   - Also have ordering constraints specified in the memory model

2. **Interprocessor interrupts** (IPIs)
   - Invoke interrupt handler on remote CPU
   - Very slow (500+ cycles on Intel), often avoided except in OS

• Used for different purposes
  (e.g. locks, vs. asynchronous notification)
Test-And-Set (TAS)

• One of the simplest non-trivial atomic operations
  1. Read a memory location’s value into a register
  2. Store “1” into the location
• Read-modify-write cycle required
  – Memory bus must be “locked” during instruction
• Can also appear as a register
  – Reading returns value; sets to 1
  – Writes of zero reset register

Not to be confused with TAZ:
Using Test-And-Set

• Acquire a mutex with TAS:

```c
void acquire( int *lock) {
    while ( TAS(lock) == 1) 
    ;
}
```

• This is a **spinlock**: keep trying in a tight loop
  – Often fastest if lock is not held for long

• Release is simple:

```c
void release( int *lock) {
    *lock = 0;
}
```
Performance of TAS-based spinlock

- How bad can this be?
- It turns out that TAS can be expensive
  - Memory must be locked while a long operation occurs
  - Must do a read, followed by write, while no-one else can access memory.
  - If spinning: slows things down
- Can we make it faster?
Test And Test-And-Set

- Replace most of RMW cycles with simple reads:

  ```c
  void acquire( int *lock) {
    do {
      while (*lock == 1);
    } while ( TAS(lock) == 1);
  }
  ```

- Think about cache traffic:
  - Reads hit in the spinner’s cache
  - Write due to release invalidates cache line
    ⇒ load from main memory, returns 0
    ⇒ triggers further RMW cycle from spinner
  - Highly likely to succeed (unless contention)
Beware of Test And Test-And-Set!

• You may be tempted to use this pattern in your regular code
  – Try whether a value has changed outside a lock
  – If so, then acquire the lock and check again
• Don’t. It doesn’t work. You need to understand:
  – Whether the compiler will reorder reads and writes
  – Whether the processor will reorder reads and writes
  – Whether the memory consistency model will change your code’s semantics.
• In Java, this is almost bound to happen.
Test And Test-And-Set in systems code

• Why does it work in systems code?
  – TAS is a hardware instruction
  – TAS is serializing:
    • Processor won’t reorder instructions past a TAS
    • Compiler won’t (we hope)
  – Or, at least, the compiler and processor are told not to
    • Memory barriers or fences
    • Use of volatile keyword

• Even in an OS – code depends on the processor architecture (⇒ memory consistency model)
Other primitives: Fetch And Add

• Atomically {
  Add to a memory location
  Read the previous value
}

• Allows sequencing of operations
  – “Event counts and sequencers” model
  – Contrast with “mutexes and conditions”
Mutual exclusion with Fetch And Add

- Atomically {
  Add to a memory location
  Read the previous value
}

```c
struct seq_t {
  int sequencer;  // Initially 0
  int event_count  // Initially 0
};

void acquire( seq_t* s ) {
  int ticket = FAA( s->sequencer, 1 );
  while ( s->event_count != ticket )
    ;
}

procedure UnLock( seq_t* s ) {
  FAA( s->event_count, 1 )
}
```
Today

- Barriers and fences
- Test and Set
- Compare and Swap
- Load-Locked / Store Conditional
- Limits of symmetric multiprocessing
- Simultaneous multithreading (SMT)
Compare and Swap

CAS( location, old, new) atomically {

1. Load location into value
2. If value == “old” then store “new” to location
3. Return value
}

Interesting features:

– Theoretically more powerful than TAS, FAA, etc.
– Can implement almost all wait-free data structures
– Requires bus locking, or similar, in the memory system
Use of CAS for mutual exclusion

• CAS is generally used in lock free data structures
  – Essentially, concurrent updates do not require locks

• Such data structures >> 1 word of memory
  – General pattern: “read-copy-update”
  – Readers all read the same datastructure
  – Writers take a copy, modify it, then write back the copy
  – Old version is deleted when all the readers are finished

• CAS used to change pointer to new version
  – As long as nothing else has changed it!
CAS for lock-free update

1. Reader follows global pointer, increments ref count.
2. Writer also reads data structure, increments ref count.
CAS for lock-free update

1. Reader follows global pointer, increments ref count.
2. Writer also reads data structure, increments ref count.
3. Writer copies data; decrements ref count; updates data
4. Writer uses CAS to swap global pointer to update copy
CAS for lock-free update

5. Reader finishes; decrements ref count; now 0 ⇒ frees old data

6. Writer finishes; decrements ref count; now 1.
The ABA problem

• CAS has a problem:
  – Reports when a single location is different
  – Does not report when it is written (with the same value)

• Leads to the “ABA” problem:
  1. CPU A reads value as x
  2. CPU B writes y to value
  3. CPU B writes x to value
  4. CPU A reads value as x \(\Rightarrow\) concludes nothing has changed

• Many problems:
  – E.g., what if the value is a software stack pointer?
Solving the ABA problem

• Basic problem:
  – Value used for CAS comparison has not changed
  – But the data has
  – CAS doesn’t say whether a write has occurred, only if a value has changed.

• Solution:
  – Ensure the value always changes!
  – Split value into:
    • Original value
    • Monotonically increasing counter
  – CAS both halves in a single instruction
Double Compare-And-Swap

CAS2 or DCAS

CAS2( loc1, old1, new1, loc2, old2, new2) atomically:

1. Compares two memory locations with different values
2. If both match, each is updated with a different new value
3. If not, existing values in the locations are loaded

- Rarely implemented: MC680x0
- **Was** claimed to be more useful than purely CAS
- Shown recently to be **not** so useful!
  - Everything can be done fast with CAS, if you’re slightly clever
What about x86?

Bewildering array of options!

- **XCHG**: atomic exchange of register and memory location
  \(\Leftrightarrow\) equivalent to Test-And-Set
- **LOCK** prefix (e.g. **LOCK** ADD, **LOCK** BTS, ...)
  - Executes instruction with bus locked
- **LOCK XADD**: Atomic Fetch-and-add
- **CMPXCHG**: 32-bit compare and swap
- **CMPXCHG8B**: 64-bit compare and swap
  - Not the same as CAS2!
- **CMPXCHG16B**: 128-bit compare and swap (x86_64 only!)

Useful for solving ABA problem
Today

• Barriers and fences
• Test and Set
• Compare and Swap
• Load-Locked / Store Conditional
• Limits of symmetric multiprocessing
• Simultaneous multithreading (SMT)
Load Locked / Store Conditional

• CAS requires bus locking for read-modify-write
  – Complicates memory bus logic
  – May be a memory bottleneck
  – Very CISC-y – not a good match for load-store RISC

• Alternative: Load-locked / store conditional
  – Also called “load-linked”, or “load and reserve”
  – Implemented in MIPS, PowerPC, ARM, Alpha,…
  – Theoretically as powerful as CAS

• Provides lock-free atomic read-modify-write
  – Factored into two instructions
LL/SC usage

Atomic read-modify-write with multiple instructions:

1. Load-locked <location> → register
2. Modify value in <register>
3. Store-conditional <register> → location
4. Test for SC success; retry if not

- Will probably succeed if no updates to location occur in the meantime.
- Note: does not suffer from the ABA problem!
Implementation of LL/SC

• Remember one address, re-use cache coherence:
  – Processor remembers physical address of locked load
  – Snoops on cache invalidates for that cache line
  – Store fails if an invalidate has been seen

• Limitations:
  – Cache-line granularity: danger of false sharing
  – Context switches, interrupts, etc. produce false positives
  – Referred to by theorists as “weak LL/SC”
Further reading

• Definitive work on shared-memory synchronization and lock-free or wait-free algorithms
Today

- Barriers and fences
- Test and Set
- Compare and Swap
- Load-Locked / Store Conditional
- Limits of symmetric multiprocessing
- Simultaneous multithreading (SMT)
Performance limits of SMP

- Cache-coherent SMP still has memory as a bottleneck
  - All accesses to main memory stall the processor
    - Limit scalability: remember Amdahl’s law?
  - MOESI allows reads to be serviced from another cache
  - But the cache itself can also be slow
Performance limits of SMP

- Cache-coherent SMP still has memory as a bottleneck
  - All accesses to main memory stall the processor
    - Limit scalability: remember Amdahl’s law?
    - MOESI allows reads to be serviced from another cache
    - But the cache itself can also be slow
- Memory stalls halt the processor
  - And other processors accessing memory
  - And the more processors, the more often this will happen
Today

• Barriers and fences
• Test and Set
• Compare and Swap
• Load-Locked / Store Conditional
• Limits of symmetric multiprocessing
• Simultaneous multithreading (SMT)
Simultaneous Multithreading

• Can we do anything useful when the processor is waiting for memory (or another cache)?
  – Most functional units (ALU, etc.) are idle
  – Many instructions don’t require the memory unit
  – But ILP is limited: can’t execute many other instructions in the thread because of data dependencies.

• So what about instructions in other threads?
  – Multiple fetch/decode units, registers
  – Reuse superscalar functional units
Conventional Superscalar
SMT (or Hyperthreading)

• Label instructions in hardware with thread id
  – Logical extension of superscalar techniques
  – Now have multiple independent instruction streams
• Fine-grained multithreading:
  – Select from threads on a per-instruction basis
• Coarse-grained multithreading:
  – Switch between threads on a memory stall
• Multithreaded CPU appears to OS as multiple CPUs!
Is it worth it?
Well...

• Can be slower than a single thread!
  – Might get no advantage from memory accesses
  – Multiple threads compete for cache
• Advantage is limited (10-20% is typical)
  – BUT: it’s cheap (in transistors)
• Depends on workloads...
  – Not necessarily good for scientific computing
  – Good for lots of memory-intensive requests,
    • e.g. web servers!
    • Target application domain for Sun Niagara, Rock, etc.
Tomorrow

• Non-Uniform Memory Access
• Multicore processors
• Performance implications