Shared Memory: A Look Underneath

Shared Memory Implementation

- Coherence - defines the behavior of reads and writes to the same memory location
  - ensuring that modifications made by a processor propagate to all copies of the data
  - Program order preserved
  - Writes to the same location by different processors serialized
- Synchronization - coordination mechanism
- Consistency - defines the behavior of reads and writes with respect to access to other memory locations
  - defines when and in what order modifications are propagated to other processors

Synchronization

- Basic types
  - Mutual exclusion
    - Primitive: locks
  - Events
    - Global event-based
      - Primitive: Barriers
    - Point-to-point event-based
      - Semaphores (blocking)
      - Condition variables
      - Flags (busy-waiting/spinning)
      - Full-empty bits (hardware implementation; also considered message passing – produce-consumer)
  - Interrupts
Components of a Synchronization Event

- Acquire method (enter critical section, proceed past event)
- Waiting algorithm (busy waiting, blocking)
- Release method (enable others to proceed)

The Critical-Section Problem

- Problem context:
  - n processes all competing to use some shared data
  - Each process has a code segment, called critical section, in which the shared data is accessed.
- Find a solution that satisfies the following:
  1. Mutual Exclusion. No two processes simultaneously in the critical section.
  2. Progress. No process running outside its critical section may block other processes.
  3. Bounded Waiting/Fairness. Given the set of concurrent processes, a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value

Synchronization Using Special Instruction: TSL (test-and-set)

entry_section:

TSL R1, LOCK | copy lock to R1 and set lock to 1
CMP R1, #0 | was lock zero?
JNE entry_section | if it wasn’t zero, lock was set, so loop
RET | return; critical section entered

exit_section:

MOV LOCK, #0 | store 0 into lock
RET | return; out of critical section
Implementing Critical Sections Using Busy Waiting

- In all our solutions today, a process enters a loop until the entry is granted $\Rightarrow$ busy waiting.

- Problems with busy waiting:
  - Waste of CPU time
  - Potential for extra traffic/communication
  - If a process is switched out of CPU during critical section
    - other processes may have to waste a whole CPU quantum
    - may even deadlock with strictly prioritized scheduling (priority inversion problem)

Spinlock Algorithms

- Test&test&set (w, w/o exponential backoff)
- Ticket lock (w, w/o proportional backoff)
- Array based queue locks
- MCS linked-list based queue locks

Implementing Locks Using Test&Set

- On the SPARC ldstub moves an unsigned byte into the destination register and rewrites the same byte in memory to all 1s

  _Lock_acquire:
  
  ldstub [%o0], %o1
  addcc %g0, %o1, %g0
  bne _Lock
  nop

  fin:
  jmpl %r15+8, %g0
  nop

  _Lock_release:
  st %g0, [%o0]
  jmpl %r15+8, %g0
  nop

Using ll/sc for Atomic Exchange

- Swap the contents of R4 with the memory location specified by R1

  try: mov R3, R4 ; mov exchange value
  ll R2, 0(R1) ; load linked
  sc R3, 0(R1) ; store conditional
  beqz R3, try ; branch if store fails
  mov R4, R2 ; put load value in R4
MCS Lock Acquire

```assembly
mcs_lock_acquire:
st %g0, [%o1+4]
mov %o1, %g3
swap [%o0], %g3
cmp %g3, 0
be .LL4
mov 1, %g2
st %g2, [%o1]
st %o1, [%g3+4]

.LL9:
  ld [%o1], %g2
cmp %g2, 0
bne .LL9
nop

.LL4:
  retl
nop
```

MCS Lock Release

```assembly
mcs_lock_release:
  ld [%o1+4], %g2
cmp %g2, 0
bne .LL11
nop
  cas [%o0], %o1, %g2
cmp %g2, %o1
be .LL10
nop

.LL17:
  ld [%o1+4], %g2
cmp %g2, 0
be .LL17
nop

.LL11:
  st %g0, [%g2]

.LL10:
  retl
nop
```

Barrier Algorithms

- Centralized sense-reversing barrier
- Software combining tree
- Tournament barrier
- Dissemination barrier
- Combining tree with improved locality

Performance Goals

- Low latency, short critical path
- Low traffic
- Scalability
- Low storage cost
- Fairness
Properties

- Deadlock – caused by a cycle of resource dependencies
- Livelock – activity without forward progress
- Starvation – extreme form of unfairness where one or more processes do not make forward progress while other do

Lock-free algorithms

Threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion

- Operations defined on it do not require mutual exclusion over multiple instructions (use atomic primitives)
  - Obstruction-free algorithms
    - One that guarantees that a thread running in isolation will make progress (although livelock is possible)
  - Non-blocking data structures
    - Operations guarantee that some process will complete its operation a finite amount of time, even if other processes halt
  - Wait-free algorithm
    - Operations can guarantee that EVERY non-faulting process will complete its operation in a finite amount of time

Shared Memory: A Look Underneath

Physical Implementation
Shared Memory Implementation

- Coherence - defines the behavior of reads and writes to the same memory location
  - ensuring that modifications made by a processor propagate to all copies of the data
  - Program order preserved
  - Writes to the same location by different processors serialized
- Synchronization - coordination mechanism
- Consistency - defines the behavior of reads and writes with respect to access to other memory locations
  - defines when and in what order modifications are propagated to other processors

Coherence

A multiprocessor memory system is coherent if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the result of the execution and
  - it ensures that modifications made by a processor propagate to all copies of the data
  - program order is preserved for each process in this hypothetical order
  - writes to the same location by different processors are serialized and the value returned by each read is the value written by the last write in the hypothetical order

Snoop-Based Coherence

- Makes use of a shared broadcast medium to serialize events (all transactions visible to all controllers and in the same order)
  - Write update-based protocol
  - Write invalidate-based (e.g., basic MSI, MESI protocols)
- Cache controller uses a finite state machine (FSM) with a handful of stable states to track the status of each cache line
- Consists of a distributed algorithm represented by a collection of cooperating FSMs

A Simple Invalidate-Based Protocol - State Transition Diagram
Correctness Requirements

- Need to avoid
  - Deadlock – caused by a cycle of resource dependencies
  - Livelock – activity without forward progress
  - Starvation – extreme form of unfairness where one or more processes do not make forward progress while other do

Design Challenges

- Cache controller and tag design
- Non-atomic state transitions
- Serialization
- Cache hierarchies
- Split-transaction buses

Multiprocessor Interconnects

- Topology
- Routing algorithm
- Switching strategy (circuit vs. packet)
- Flow control mechanism

Interconnect Topologies

- Fully connected
  - Single large switch
  - Bus
- Linear arrays and rings
- Multi-dimensional meshes and tori
- Trees
- Butterflies
- Hypercube
Switching Strategy

- Circuit-switched: first packet sets up route, subsequent packets follow route without any header processing
- Packet-switched: each packet is independently routed
  - Store-and-forward: each hop receives all packets of a message before forwarding it on
  - Cut-through: each packet forwarded as soon as it is received
  - Virtual cut-through: cut-through routing, but buffer packets when there is contention
  - Wormhole routing: packet spread across multiple hops, in effect holding a circuit open.

Metrics

- Hardware cost – number of wires, pin count, length of wires, physical arrangement
- Topology diameter
  - Length of maximum shortest path between any two nodes in the network
- Latency
  - Overhead+routing_delay+channel_occupancy(bandwidth)+contention_delay
- Bandwidth – local, global, bisection
  - Bisection bandwidth
    - Sum of bandwidths of minimum set of channels/links that, if removed, partitions the network into 2 equal unconnected sets of nodes

Directory-Based Coherence

- Distribute memory, use point-to-point interconnect for scalability
- Need to manage coherence for each memory line – state stored in directory
  - Simple memory-based (e.g., DASH, FLASH, SGI Origin, MIT Alewife, HAL)
  - Cache-based (linked list (e.g., Sequent NUMA-Q, IEEE SCI)

Simple Memory-based Directory Coherence

- Advantage
  - Precise sharing information
- Disadvantage
  - Space/storage proportional to PxM
- Work-around for either width or height
  - Increase cache block size
  - 2-level protocol
  - Limited pointer scheme
  - Directory cache
Cache-Based Directory Coherence

- Home main memory contains a pointer to the first sharer + state bits
- Pointers at each cache line to maintain a doubly-linked list
- Advantage – reduced space overhead
- Disadvantage – serialized invalidates (latency and occupancy)

A Framework for Sharing Patterns

- Predictable vs. unpredictable
- Regular vs. irregular
- Coarse vs. fine-grain (contiguous vs. non-contiguous in the address space)
- Near-neighbor vs. long range in an interconnection topology
- In terms of invalidation patterns
  - Read-only
  - Producer-consumer
  - Broadcast/multicast
  - Migratory
  - Irregular read-write

Memory Consistency Model

- Specifies constraints on the order in which memory operations to different locations must appear to be performed with respect to one another

Sequential Consistency

- "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 this sequence in the order specified by its program." [Lamport 79]
  - In practice, this means that every write must be seen on all processors before any succeeding read or write can be issued
Implications

- Program order
  
  P1  
  A = 1;  
  flag = 1;  
  P2  
  while (flag == 0);  
  print A;

- Write atomicity
  
  P1  
  A = 1;  
  while (A == 0);  
  B = 1;  
  while (B == 0);  
  print A;

Dekker’s Algorithm

P1: A = 0;  
...  
A = 1;  
L1: while (B == 1) {..}  
...  
...  

P2: B = 0;  
...  
B = 1;  
L2: while (A == 1) {...}  
...  
...

Can B = 0 at P1 and A = 0 at P2 at the corresponding if statements?

Write Buffers [Bypassing Capability]

- Reads bypass writes, reads are blocking

Overlapping Write Operations

- Writes may bypass other writes in write buffer
Non-blocking Reads

- Reads are allowed to bypass reads and writes

Drawbacks of Sequential Consistency

- SC imposes a performance penalty
- SC restricts any compiler optimization that can result in reordering memory operations
  - Code motion, register allocation, common sub-expression elimination, loop blocking, software pipelining
- SC restricts hardware generated memory re-orderings because of program-order and write-atomicity requirements
  - Write Buffers, OOO instruction issue, pipelining of memory operations, lock-up free caches, non-atomic memory operations

Memory Model Relaxations

- Possible relaxations
  - Write → Read
  - Write → Write
  - Read → Read, Write
  - Read other’s write early
  - Read own write early

- All Models provide some Safety net
- All models maintain uni-processor data and control dependencies
- Write serialization is maintained by all the models except PC, RCpc, PowerPC (for most practical purposes where all processors observe all write operations in the same order (write serialization), is indistinguishable from a system where all writes are executed atomically)

Categorization of Relaxed Models

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>W→R Order</th>
<th>W→W Order</th>
<th>R→RW Order</th>
<th>Read Others Write Early</th>
<th>Read Own Write Early</th>
<th>Safety Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 370</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sealed</td>
</tr>
<tr>
<td>TSO</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>RMW</td>
</tr>
<tr>
<td>PC</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>RMW</td>
</tr>
<tr>
<td>PSO</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>synch</td>
</tr>
<tr>
<td>RCpc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, sync, RMW</td>
</tr>
<tr>
<td>RCpc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, sync, RMW</td>
</tr>
<tr>
<td>Alpha</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MB, RMW</td>
</tr>
<tr>
<td>RMO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>various MEMBARs</td>
</tr>
<tr>
<td>PowerPC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
Relaxing All Program Orders

• Read or a Write operation may be reordered w.r.t. following read or write to a different location
  – Weak Ordering Model
    • Release Consistency Model (RCsc / RCpc)
    • Digital Alpha, Sparc V9 RMO, IBM Power PC
• Except Alpha, the above models allow reordering of two reads to the same location.
• RCpc and PowerPC allow a read to return the value of another processors write early.

Weak Ordering

• Classifies instructions into “Data” and “Sync”
• Reordering memory operations between sync operations.
• Hardware Implementation using WO counter: issue sync operation counter must be zero
• No operations are issued until previous sync operation completes
• Synchronization accesses are sequentially consistent with respect to one another.

Weak Ordering (Cont’d)

• Open up opportunities for buffering of reordered write operations between two synchronization points.

<table>
<thead>
<tr>
<th>P1:</th>
<th>W(x)1</th>
<th>W(x)2</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2:</td>
<td>R(x)0</td>
<td>R(x)2</td>
<td>R(x)2</td>
</tr>
<tr>
<td>P3:</td>
<td>R(x)1</td>
<td>R(x)2</td>
<td></td>
</tr>
</tbody>
</table>

OK

Release Consistency

• Extends WO and makes distinction among sync and non-sync operations
• RCsc maintains sequential consistency among special operations
• RCpc maintains processor consistency among special operations
RC Example

Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.

Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.

<table>
<thead>
<tr>
<th>acquire A = 1;</th>
<th>RC</th>
<th>release</th>
</tr>
</thead>
<tbody>
<tr>
<td>While (A==0); acquire B = 1;</td>
<td>While (B==0); Print A;</td>
<td></td>
</tr>
</tbody>
</table>

Programmer Centric View

- System Centric view is accompanied by higher level of complexity for programmers.
- Varied semantics for different models complicates the task of porting programs across systems.

Motivates for higher level of abstraction for programmers
- Provide informal rules for correct results defined by SC i.e. Consistency Model is defined in terms of program level information provided by the programmer.
  - DRF0 is one such approach which explores the information that is required to allow optimization similar to Weak Ordering.
  - PL (Properly Labeled) approach for defining RCsc optimizations.

Alpha, RMO and PowerPC

- Alpha employ RCsc model with Memory Barrier and Write Memory Barrier (WMB) fence instructions.
- Sparc V9 (RMO) employ RCsc model with MemBar instruction to specify any combination of RtoR, RtoW, WtoR, WtoW ordering.
  - No need for RMW to preserve WtoR ordering
  - Write atomicity is maintained
- PowerPC employ RCpc
  - SYNC instruction similar to MB instruction except for RtoR order.
  - RMW required to make writes atomic and preserve RtoR order.

The Data-race-free-0 Model

- Weak Ordering classifies instruction into “Data” and “Sync”
- Key Goal is to formally distinguish operations as data or Synchronization on the bases of races.

An operation forms a race with another operation if,
- They access the same location && atleast one operation is a write && there are no intervening operations between the two operations.

P1
A = 23;
while (Flag != 1) {;}
B = 37;
... = B;
Flag = 1;
... = A;

Flag = Synchronization, Data = A, B
Programming With DRF-0

- Write operation assuming SC.
- For every memory operation specified in the program do:

```
START
no

don't know or
don't care

Language Support:
- Synchronization with special constructs
- Support to distinguish individual accesses
```

Distinguishing Memory Operations

- At the Programming Language Level
  - Special synchronizaiton operation (library call)
  - High-level paradigms
  - Data or synchronization attribute with code or data
- At the hardware level
  - Address regions
  - Special instructions

Comparison of Sync/Swym and Niagara

- Sync/Swym are Sparc V9 architecture based systems which defines three different memory models: TSO, PSO, RMO (Relaxed Memory Ordering) model.
- Programs written for RMO will work in PSO and TSO as well. Programs written for PSO will work in TSO. MEMBAR inst. induce ordering in the inst. stream of a single processor.
  - Portability issues: Programs which use single-writer/multiple reader locks for all shared accesses are portable across all models.
  - Programs that use write locks to protect write accesses but read without locking will be portable across all memory models, only if writes to shared data are separated by MEMBAR #StoreStore instructions, and if reading the data is followed by a MEMBAR #Load-Load instruction.
- Niagara has 2 flavors of Stores: TSO and RMO

Power4 Memory Model

- All orders relaxed
- For certain instructions which require completion serialization
  Groups so marked will not be issued until that group is the next to complete, i.e., all prior groups have successfully completed
- Additionally, instructions that read a non-renamed register cannot be executed until we are sure all writes to that register have completed
- Load Hit Store: Read Own Write Early
- Store Hit Load: Write -> Read Program Order is Relaxed
- Load Hit Load: Read -> Read Program Order is Relaxed
  To guard against Read Others Write Early, if a younger load obtains old data then the older store must obtain new data. This requirement is called sequential load consistency

Safety nets: sync, lwsync, isync, eieio
Consistency Model of Intel P4

Named as a Processor Ordering with Following rules built in:
- Reads can be carried out speculatively and in any order
- Reads can pass buffered writes, but the processor is self-consistent
- Writes to memory are always carried out in program order, with the exception of writes executed with CLFLUSH instruction and streaming stores (writes)
- Writes can be buffered
- Writes are not performed speculatively; they are only performed for instructions that have actually been retired
- Data from buffered writes can be forwarded to waiting reads within the processor
- Reads or writes cannot pass (be carried out ahead of) I/O instructions, locked instructions, or serializing instructions
- Reads cannot pass LFENCE and MFENCE instructions
- Writes cannot pass SFENCE and MFENCE instructions

Consistency Model Classification

- Models vary along the following dimensions
  - Local order - order of a processor's accesses as seen locally
  - Global order - order of a single processor's accesses as seen by each of the other processors
  - Interleaved order - order of interleaving of different processor's accesses on other processors

Summary

- Defined Sequential Consistency
- Optimizations to SC
- Relaxed Memory Models
  - IBM 370, TSO, PC
  - PSO and SPARC V9
  - WO
  - RC (RCsc and RCpc)
- Provided a Programmer Centric view for identifying different operations in a program.
- Discussed consistency models of Sparc-V9, Power4 and Intel P4.