Multiprocessors and Locking

COMP9242
2008/S2 Week 12
Part 1
Types of Multiprocessors (MPs)

→ Uniform memory-access (UMA) MP
  • Access to all memory occurs at the same speed for all processors.

→ Non-uniform memory-access (NUMA) MP
  • Access to some parts of memory is faster for some processors than other parts of memory

→ We'll focus on UMA
Types of UMA Multiprocessors

→ Classical *multiprocessor*
  - CPUs with local caches
  - connected by bus
  - fully separated cache hierarchy ⇒ cache coherency issues

→ *Chip Multiprocessor* (CMP)
  - per-core L1 caches
  - shared lower on-chip caches
  - usually called “*multicore*”
  - mild cache coherency issues
    - easily addressed in hardware

→ *Symmetric multithreading* (SMT)
  - replicated functional units, register state
  - interleaved execution of several threads
  - fully shared cache hierarchy
  - no cache coherency issues
Cache Coherency

→ What happens if one CPU writes to (cached) address and another CPU reads from the same address
  • Can be thought of as replication and migration of data between CPUs

→ Ideally, a read produces the result of the last write to the particular memory location
  • Approaches that avoid the issue in software also prevent exploiting replication for parallelism
  • Typically, a hardware solution is used
    - Snooping – typically for bus-based architectures
    - Directory based – typically for non-bus interconnects
Snooping

→ Each cache “broadcasts” transactions on the bus
→ Each cache monitors the bus for transactions that affect its state
→ Typically use “MESI” protocol in bus-based architectures
→ How does snooping work with multiple levels of caches?
  • inclusion property: $L_n \supset L_{n-1}$
  • multi-level snooping
Example Coherence Protocol  MESI

Each cache line is in one of four states

→ **Modified** (M)
  • The line is valid in the cache and in only this cache.
  • The line is modified with respect to system memory—that is, the modified data in the line has not been written back to memory.

→ **Exclusive** (E)
  • The addressed line is in this cache only.
  • The data in this line is consistent with system memory.

→ **Shared** (S)
  • The addressed line is valid in the cache and in at least one other cache.
  • A shared line is always consistent with system memory. That is, the shared state is shared-unmodified; there is no shared-modified state.

→ **Invalid** (I)
  • This state indicates that the addressed line is not resident in the cache and/or any data contained is considered not useful.
MESI Coherence Protocol

**Events**
- RH = Read Hit
- RMS = Read miss, shared
- RME = Read miss, exclusive
- WH = Write hit
- WM = Write miss
- SHR = Snoop hit on read
- SHI = Snoop hit on invalidate
- LRU = LRU replacement

**Bus Transactions**
- Push = Write cache line back to memory
- Invalidate = Broadcast invalidate
- Read = Read cache line from memory
Directory-based Coherence

➔ Each memory block has a home node
➔ Home node keeps directory of cache that have a copy
  • E.g., a bitmap of processors per memory block
✔ Invalidation/update messages can be directed explicitly
✗ Requires more storage to keep directory
  • E.g. each 256 bits or memory requires 32 bits of directory
Interaction with Memory Architecture

Example: critical section

/* counter++ */
load r1, counter
add r1, r1, 1
store r1, counter
/* unlock(mutex) */
store zero, mutex

Relies on all CPUs seeing update of counter before update of mutex

Depends on assumptions about ordering of stores to memory
Memory Models: Strong Ordering

- Loads and stores execute in program order
- Memory accesses of different CPUs are sequentialised
- Traditionally used by many architectures

CPU 0

store r1, adr1
load r2, adr2

CPU 1

store r1, adr2
load r2, adr1

- At least one CPU must load the other's new value
Other Memory Models

→ Modern hardware features can interfere with store order:
  • write buffer (or store buffer or write-behind buffer)
  • instruction reordering (out-of-order completion)
  • superscalar execution
  • pipelining

→ Each CPU keeps its own data consistent, but how about others?
  • multiprocessing
  • DMA
Total Store Ordering

- Stores go to write buffer to hide memory latency
- Loads read from write buffer if possible
- Stores are guaranteed to occur in FIFO order

\[\begin{align*}
\text{CPU 0} & \quad \text{store } r1, \text{adr1} \quad \text{store } r1, \text{adr2} \\
& \quad \text{load } r2, \text{adr2} \quad \text{load } r2, \text{adr1}
\end{align*}\]

- Both CPUs may read old value!
- Need hardware support, e.g.
  - atomic swap
  - test & set
  - load-linked + store-conditional
  - memory barriers
- Stall pipeline and drain (and bypass) write buffer
Partial Store Ordering

- All stores go to write buffer
- Loads read from write buffer if possible
- *Redundant stores are cancelled*

```assembly
load r1, counter
add r1, r1, 1
store r1, counter
barrier
store zero, mutex
```

- Store to mutex can overtake store to counter
- Need to use *memory barrier*
- Failure to do so will introduce subtle bugs:
  - changing process state after saving context
  - initiating I/O after setting up parameter buffer
Observation

- Locking primitives require exclusive access to the “lock”
- Care required to avoid excessive bus/interconnect traffic
Focus on locking in the Common Case

- Bus-based UMA, per-CPU write-back caches, snooping coherence protocol.
Kernel Locking

- Several CPUs can be executing kernel code concurrently.
- Need mutual exclusion on shared kernel data.
- Issues:
  - Granularity of locking
  - Lock implementation
Multiprocessing Options

- Expensive communication ➝ distributed data structures

- Fast communication ➝ shared data structures
Multiprocessing Options

• Scheduling domain
  ➔ Hardware thread contexts of an SMT core
  ➔ Multiple cores on a chip with fast cache migration, inter-core interrupts

• Separate domains where communication is slow
  ➔ Multiple cores without shared caches
  ➔ Bus-connected processors
Lock Granularity

- Fine-grained vs coarse-grained?
  - tradeoff is highly dependent on
    - length of system calls
    - number of fine-grained locks required
    - cost of individual locks
    - ...

©2008 Gernot Heiser UNSW, with contributions from Kevin Elphinstone
Mutual Exclusion Techniques

- Disabling interrupts (CLI — STI).
  - Unsuitable for multiprocessor systems.

- Spin locks.
  - Busy-waiting wastes cycles.

- Lock objects.
  - Flag (or a particular state) indicates object is locked.
  - Manipulating lock requires mutual exclusion.
Hardware Provided Locking Primitives

- `int test_and_set(lock *);`
- `int compare_and_swap(int c, int v, lock *);`
- `int exchange(int v, lock *)`
- `int atomic_inc(lock *)`

- `v = load_linked(lock *) / bool store_conditional(int, lock *)`
  - LL/SC can be used to implement all of the above
Spin locks

```c
void lock (volatile lock_t *lk) {
    while (test_and_set(lk)) ;
}
void unlock (volatile lock_t *lk) {
    *lk = 0;
}
→ Busy waits. Good idea?
```
Spin Lock Busy-waits Until Lock Is Released

- Stupid on uniprocessors, as nothing will change while spinning.
  - Should release (yield) CPU immediately.
- Maybe ok on SMPs: locker may execute on other CPU.
  - Minimal overhead (if contention low).
  - Still, should only spin for short time.
- Generally restrict spin locking to:
  - short critical sections,
  - unlikely to be contended by the same CPU.
  - local contention can be prevented
    - by design
    - by turning off interrupts
Spinning versus Switching

- Blocking and switching
  - to another process takes time
    - Save context and restore another
    - Cache contains current process not new
      - Adjusting the cache working set also takes time
    - TLB is similar to cache
  - Switching back when the lock is free encounters the same again
- Spinning wastes CPU time directly

⇒ Trade off
- If lock is held for less time than the overhead of switching to and back
⇒ It’s more efficient to spin
Spinning versus Switching

The general approaches taken are

- Spin forever
- Spin for some period of time, if the lock is not acquired, block and switch
  - The spin time can be
    - Fixed (related to the switch overhead)
    - Dynamic
      » Based on previous observations of the lock acquisition time
Interrupt Disabling

- Assume no local contention by design, is disabling interrupt important?

- Hint: What happens if a lock holder is preempted (e.g., at end of its timeslice)?

- All other processors spin until the lock holder is re-scheduled
Alternative: Conditional Lock

```c
bool cond_lock (volatile lock_t *lk) {
    if (test and set(lk))
        return FALSE; // couldn’t lock
    else
        return TRUE; // acquired lock
}
```

→ Can do useful work if fail to acquire lock.
→ **But** may not have much else to do.
→ Starvation: May never get lock!
More Appropriate Mutex Primitive:

```c
void mutex lock (volatile lock t *lk) {
    while (1) {
        for (int i=0; i<MUTEX N; i++)
            if (!test and set(lk))
                return;
        yield();
    }
}
```

→ Spins for limited time only
  • assumes enough for other CPU to exit critical section
→ Useful if critical section is shorter than N iterations.
→ Starvation possible.
Common Multiprocessor Spin Lock

```c
void mp spinlock (volatile lock t *lk) {
    cli(); // prevent preemption
    while (test and set(lk)) ; // lock
}
void mp unlock (volatile lock t *lk) {
    *lk = 0;
    sti();
}
```

- Only good for short critical sections
- Does not scale for large number of processors
- Relies on bus-arbitrator for fairness
- Not appropriate for user-level
- Used in practice in small SMP systems

[Anderson, 1990]