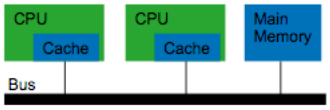
Multiprocessor Synchronization

- Multiprocessor Systems
- Memory Consistency
- Simple Synchronization Primitives: Implementation Issues

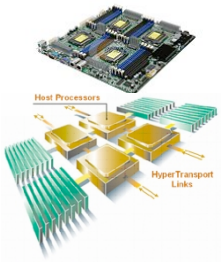
(Much material in this section has been freely borrowed from Gernot Heiser at UNSW and from Kevin Elphinstone)

MP Memory Architectures

- Uniform Memory-Access (UMA)
  - Access to all memory locations incurs same latency.

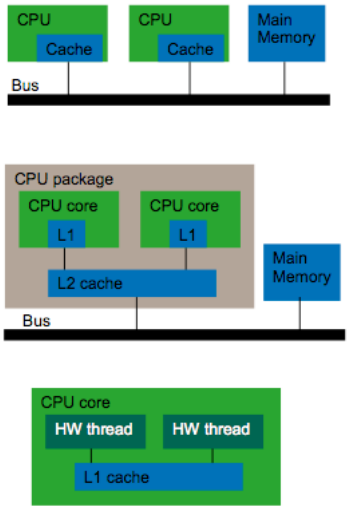


- Non-Uniform Memory-Access (NUMA)
  - Memory access latency differs across memory locations for each processor.
    - e.g. Connection Machine, AMD HyperTransport, Intel Itanium MPs

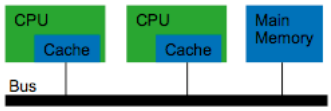


UMA Multiprocessors: Types

- **"Classical" Multiprocessor**
  - CPUs with local caches
  - typically connected by bus
  - fully separated cache hierarchy -> cache coherency problems
- **Chip Multiprocessor (multicore)**
  - per-core L1 caches
  - shared lower on-chip caches
  - cache coherency addressed in HW
- **Simultaneous Multithreading**
  - interleaved execution of several threads
  - fully shared cache hierarchy
  - no cache coherency problems

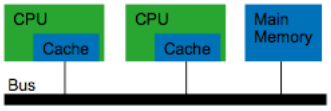


Cache Coherency



- What happens if one CPU writes to (cached) address and another CPU reads from the same address?
  - similar to replication and migration of data between CPUs.
- Ideally, a read produces the result of the last write to the same memory location. ("Strict Memory Consistency")
  - Approaches that avoid the issue in software also reduce replication for parallelism
  - Typically, a hardware solution is used
    - snooping – for bus-based architectures
    - directory-based – for non bus-interconnects

Snooping



- Each cache broadcasts transactions on the bus.
- Each cache monitors the bus for transactions that affect its state.
- Conflicts are typically resolved using the **MISE** protocol.
- Snooping can be easily extended to multi-level caches.

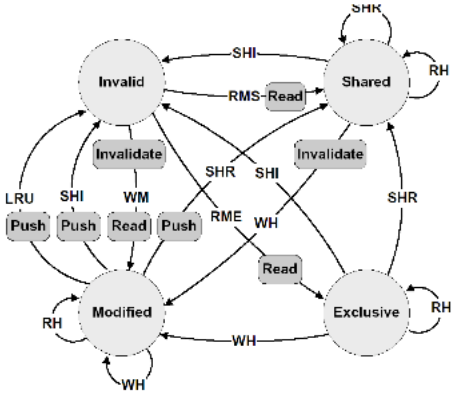
The MESI Cache Coherency Protocol

Each cache line is in one of 4 states:

- **M**odified:
  - 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.
- **E**xclusive:
  - The addressed line is in this cache only.
  - The data in this line is consistent with system memory.
- **S**hared:
  - 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.
- **I**nvalid:
  - 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*

Memory Consistency: Example

- 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

Example of Strong Ordering: Sequential Ordering

- Strict Consistency is impossible to implement.
- **Sequential Consistency**:
  - Loads and stores execute in program order
  - Memory accesses of different CPUs are "sequentialised"; i.e., any valid interleaving is acceptable, but all processes must see the same sequence of memory references.
- Traditionally used by many architectures

CPU 0	CPU 1
`store r1, adr1`	`store r1, adr2`
`load r2, adr2`	`load r2, adr1`

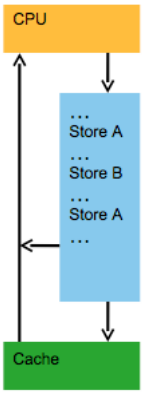
- In this example, at least one CPU must load the other's new value.

Sequential Consistency (cont)

- Sequential consistency is programmer-friendly, but expensive.
- Lipton & Sandbert (1988) show that improving the read performance makes write performance worse, and vice versa.
- Modern HW features interfere with sequential consistency; e.g.:
  - write buffers to memory (aka store buffer, write-behind buffer, store pipeline)
  - instruction reordering by optimizing compilers
  - superscalar execution
  - pipelining

Weaker Consistency Models: Total Store Order

- **Total Store Ordering** (TSO) guarantees that the sequence in which `store`, `FLUSH`, and `atomic load-store` instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor.
- Both x86 and SPARC processors support TSO.
- A later `load` can bypass an earlier `store` operation.
- i.e., local `load` operations are permitted to obtain values from the write buffer before they have been committed to memory.



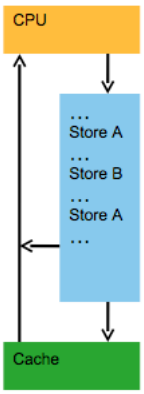
Total Store Order (cont)

- Example:

CPU 0 CPU 1

```
store   r1, adr1        store   r1, adr2
load    r2, adr2        load    r2, adr1
```

- Both CPUs may read old value!
- Need hardware support to force global ordering of privileged instructions, such as:
  - atomic swap
  - test & set
  - load-linked + store-conditional
  - memory barriers
- For such instructions, stall pipeline and flush write buffer.



It gets weirder: Partial Store Ordering

- **Partial Store Ordering** (PSO) does **not** guarantee that the sequence in which `store`, `FLUSH`, and `atomic load-store` instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor.
- The processor can reorder the stores so that the sequence of stores to memory is not the same as the sequence of stores issued by the CPU.
- SPARC processors support PSO; x86 processors do not.
- Ordering of stores is enforced by **memory barrier** (instruction `STBAR` for Sparc) : If two stores are separated by memory barrier in the issuing order of a processor, or if the instructions reference the same location, the memory order of the two instructions is the same as the issuing order.

Partial Store Order (cont)

- Example:

```
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** to separate issuing order.
- Otherwise, we have a race condition.

Mutual Exclusion on Multiprocessor Systems

- Mutual Exclusion Techniques
  - Disabling interrupts
    - 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.
- Software-only mutual-exclusion solutions (e.g. Peterson) do not work with TSO.
  - Need HW locking primitives (recall: they flush the write buffer)

Mutual Exclusion: Spin Locks

- Spin locks rely on busy waiting!
- Bad idea on uniprocessors
  - Nothing will change while we spin!
  - Why not yield immediately?!

```
void lock (volatile lock_t *lk) {
    while (test_and_set(lk)) ;
}

void unlock (volatile lock_t *lk) {
    *lk = 0;
}
```

- Maybe not so bad on multiprocessors
  - The locker may be running on another CPU
  - Busy waiting still unnecessary.
- Restrict spin locks to short critical sections, with little chance of CPU contention.

Why bother with Spin Locks?

- Blocking and switching to other process does not come for free.
  - Context switch time
  - Penalty on cache and TLB
  - Switch-back when lock released generates another context switch
- Spinning burns CPU time directly.
- This is an opportunity for a trade-off.
  - Example: spin for some time; if lock not acquired, block and switch.

Conditional Locks

- Have the application handle the trade-off:

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

- Pros:
  - Application can select to do something else while waiting.
- Cons:
  - There may be not much else to do…
  - Possibility of starvation.

Mutex with "Bounded Spinning"

- Spin for limited time only:

```
void mutex_lock (volatile lock t *lk) {
    while (1) {
        for (int i=0; i<MUTEX_N; i++)
            if (!test_and_set(lk))
                return;
        yield();
    }
}
```

- Starvation still possible.

Common Multiprocessor Spin Lock

- As used in practice in small MP systems (Anderson, 1990)

```
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();
}
```

- Good for short critical sections.
- Does not scale for large number of CPUs.
- Relies on bus-arbitration for fairness.
- No appropriate for user-level. (why?)