Parallel Shared Memory Architecture: Coherence, Synchronization & Ordering

Wei Wang

Shared Memory Architecture

- Shared memory architecture is the most common parallel architecture today
  - Multi-core CPUs
  - Multi-processor servers
- Shared memory architectures rely on cache coherence to coordinate accesses to shared data
  - Cache coherence is the hardware policy to ensure data is access in a synchronized fashion, i.e., atomically and consistently
  - Software synchronization primitives are built based on the atomic operations provided by hardware
- Correctly programming shared memory architectures also requires knowledge of their memory ordering

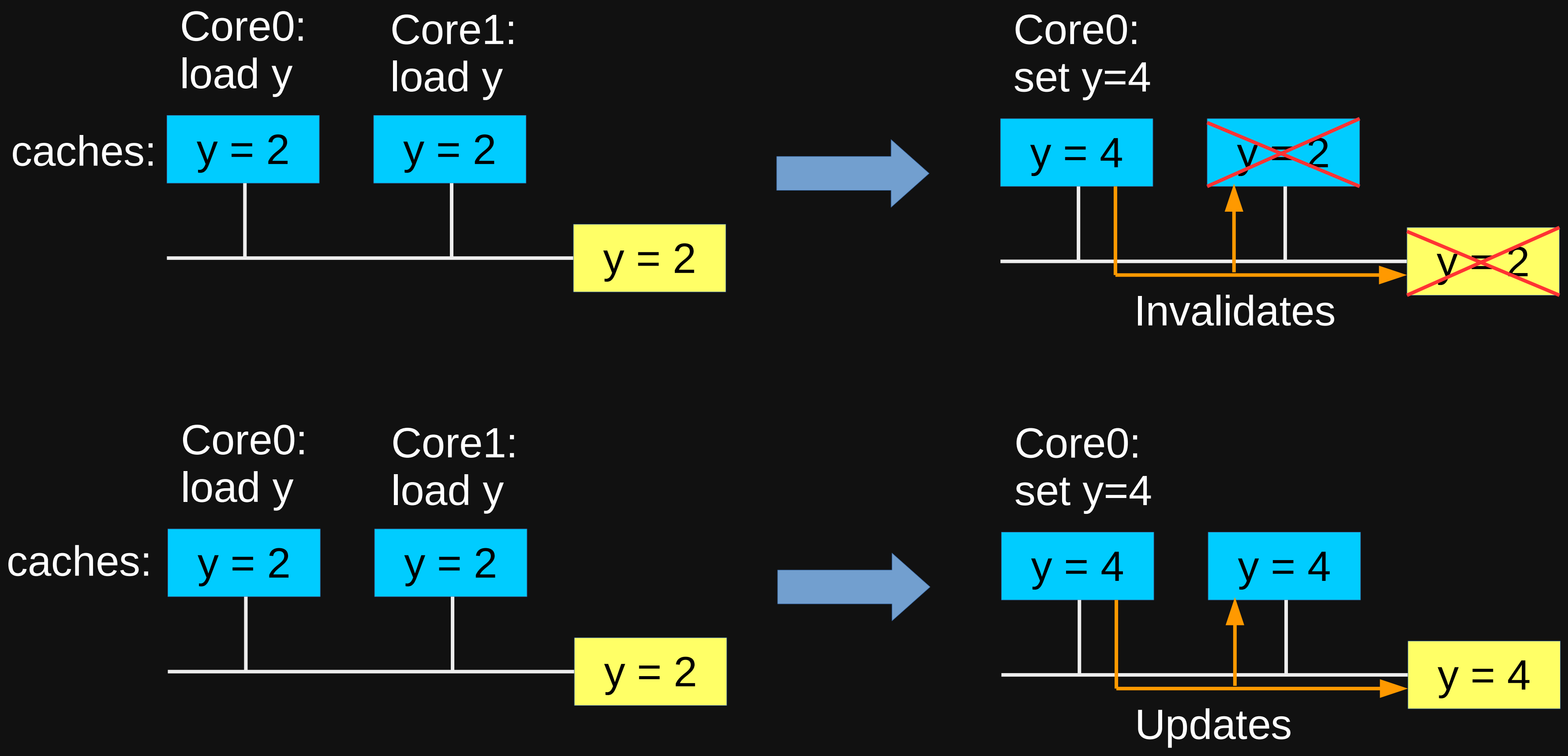
Basic Cache Coherence

Cache Coherence

- Shared memory architectures
  - must coordinate access to data that might have multiple copies
  - In the case of multiple copies of data exist (e.g., each cache has a copy of the same variable), they can easily become inconsistent
- Sequential consistency
  - all data accesses appear to have been executed
    - atomically
    - in some sequential order: consistent with the order of operations in individual threads
    - corollary: each variable must appear to have only a single value at a time
- Modern shared memory architectures satisfy the above requirements through cache coherence protocols

Invalidate V.S. Update Protocols

- Two types of cache coherence protocols



Invalidate V.S. Update Protocols

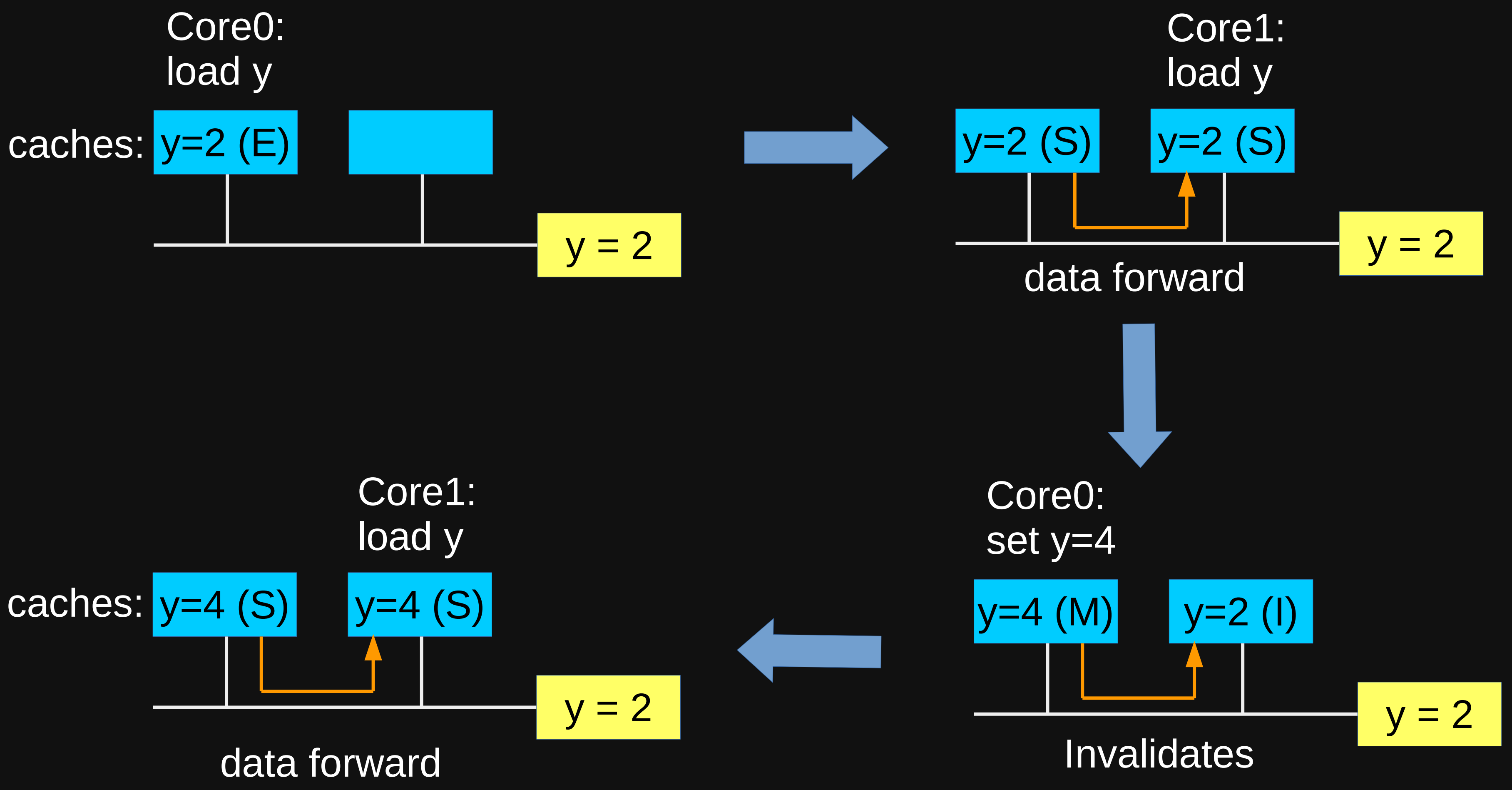
- Cost-benefit trade-off depends upon traffic pattern
  - invalidation is worse when
    - single producer of data and many consumers
  - update is worse when
    - multiple writes by one CPU before data is read by another
    - a cache is filled with data that is not read again
      - e.g., leftovers after thread or process migration
- Modern machines use invalidate protocols as the default

Modern Invalidate Protocol: MESI Protocol

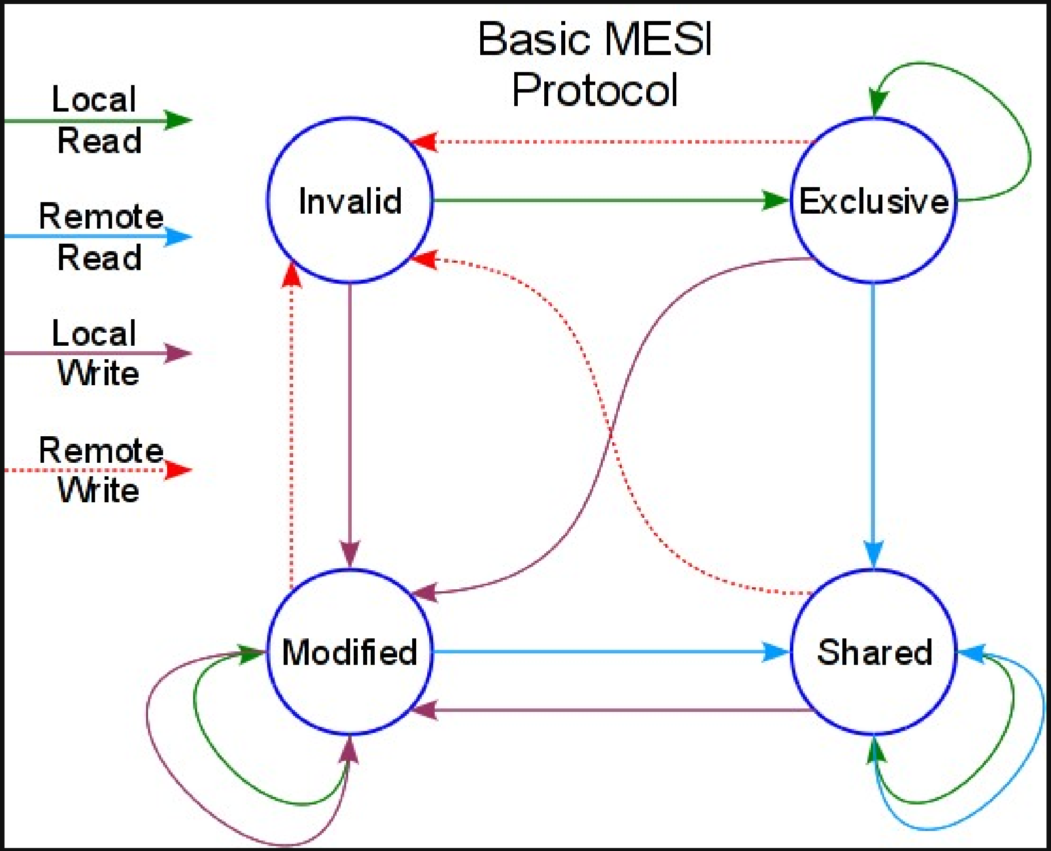
- Each copy of data in cache is associated with a state
- MESI stands for four states:

State	Description
Modified	Only one copy of the data exists, data is modified and different from memory
Exclusive	Only one copy of the data exists, data has the same value as the code in memory
Shared	Multiple copies of the data exist, all copies have the same value
Invalid	The copy of data is invalid and should not be used

MESI Protocol Example



MESI Transition Graph



MESI figure credit: http://sc.tamu.edu/Images/MESI.png
(Michael Thomadakis, Texas A&M)

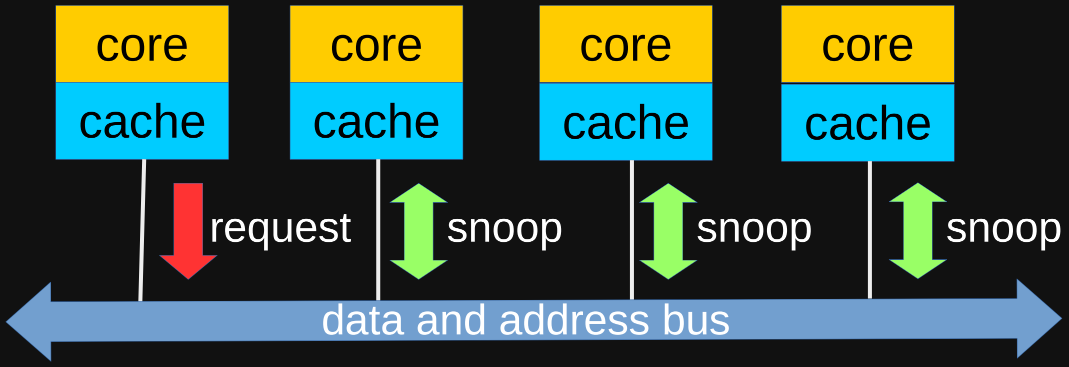
MESI Transition Table

MESI state transition table for a cache line in various states. The first column is the old state of the cache line, the first row is a possible operations, and the cells are the new state of the cache line after the operation. This table is the same as the transition graph in the previous slide.

Original States	Local Read	Remote Read	Local Write	Remote Write
M	M	S	M	I
E	E	S	M	I
S	S	S	M	I
I	E	I	M	I

Snoopy Cache Systems

- How does a cache adjust the states of its data?
  - All caches are connected to a bus
  - Broadcast all invalidate and read requests to the bus
  - Each core (and its cache) snoops the requests and updates the states of its data accordingly



Operation of Snoopy Caches

- Data operation requests
  - Data write
    - Broadcast invalidation, mark data as “Modified (M)”
  - Data load
    - Broadcast read request
    - If data from another cache, mark data as “Shared (S)”
    - If data from memory, mark data as “Exclusive (E)”
- Responses to requests
  - Invalidate request snooped
    - If a copy of data in cache, mark data as “Invalidate (I)”
  - Read request snooped
    - If a copy of data in cache, forward data to requesting cache, mark local data as “Shared (S)”

Intel MESIF Protocol

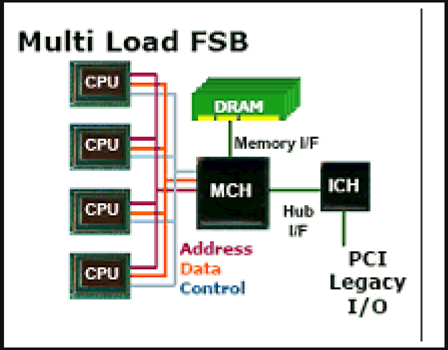
- Besides MESI, Intel cache protocol also includes a Forward state.
- If a copy of data is shared
  - one shared copy of the cache line is in the F state
  - remaining copies of the cache line are in the S state
- Forward (F) state designates a single copy of data from which further copies can be made
  - MESI protocol does not specify which cache should forward data if there are multiple copies of the data in multiple caches
  - cache line in the F state will respond to a request for a copy of the cache line
  - consider how one embodiment of the protocol responds to a read
    - newly created copy is placed in the F state
    - cache line previously in the F state is put in the S or the I state

Cache Coherence on Large-scale Shared Memory Architecture

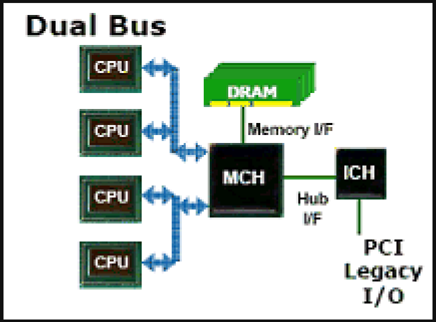
The Limitation of Bus

- Snoopy cache performance is limited by the bandwidth of the bus
  - There is a maximum number of requests can be sent per unit time
- Bus is a bottleneck where there are large number of cache that need to maintain coherent
  - This especially true for multi-processor machines
- The solution is to directly connect caches

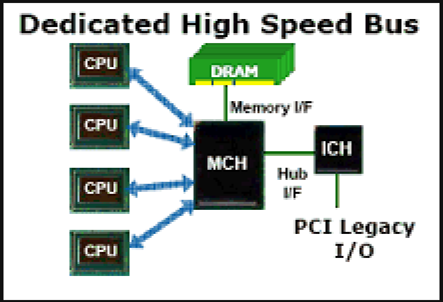
Evolution of Inter-processor/cache connections



Before 2004, Front-side bus (FSB) handles coherence requests. Each CPU/cache handles coherence requests locally



Around 2005, Dual FSB provide higher aggregated bus. But memory controller (MCH) now must assist to handle snoop requests



Eventually, a dedicated bus is assigned to CPU for maximum bandwidth (around 2007). MCH becomes very complex and a bottleneck due to these direction connections.

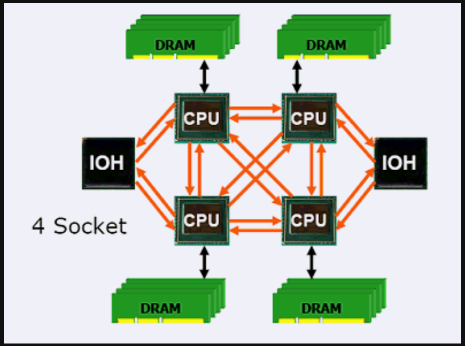


Figure credit: An Introduction to Intel QuickPath Interconnect, Robert A. Maddox, Gurbir Singh, and Robert J. Safranek, Intel April 06, 2009

After 2008, Intel QuickPath Interconnect (QPI) provides direct connections between CPUs. AMD's counterpart is call HyperTransport (HT). Directory-based cache coherence protocol is used here.

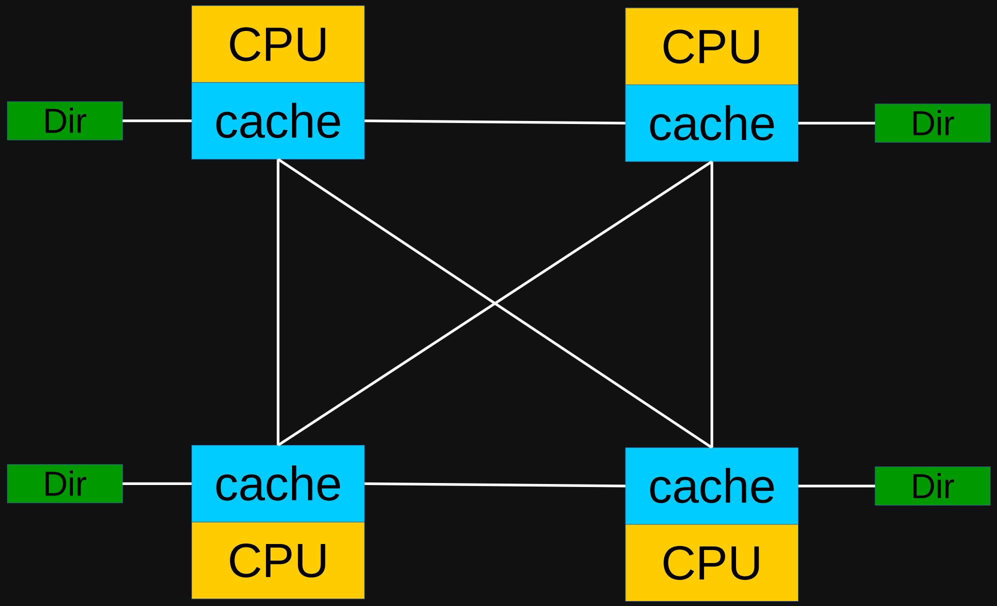
The Cost of Snoopy Coherence Protocols

- Each coherence operation is sent to all cores/processors, even those does not have the data
- Scalability is bad:
  - When there are large number of cores/processors, caches and memory controller are overwhelmed by handling cache requests

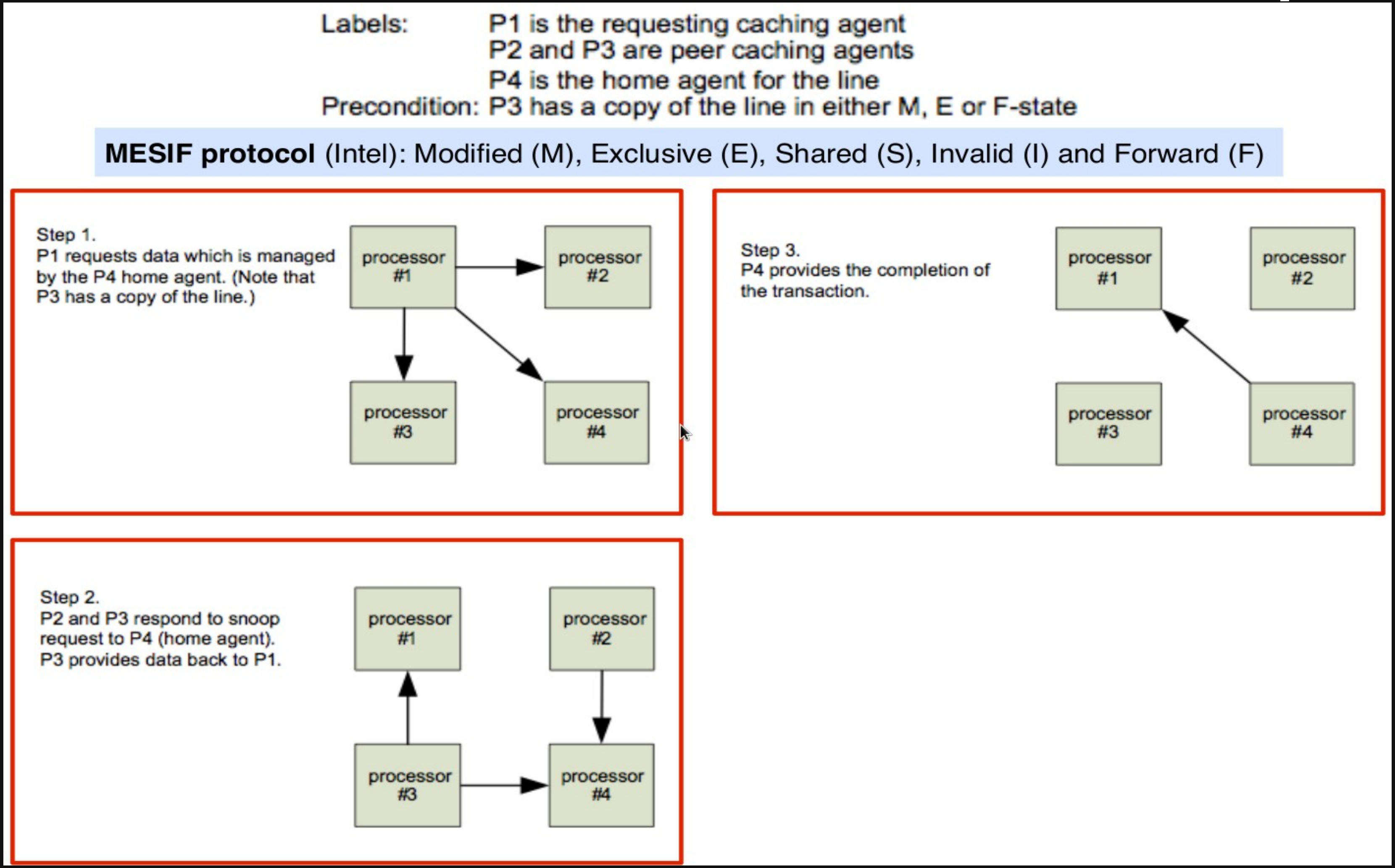
Directory Based Cache Coherence Protocol

- Aiming at providing better scalability than snoopy protocols
- Each cache has a hardware, called “directory”, which tracks the states and coordinate the accesses of some data.
- A piece of data is only tracked by one directory of one CPU
  - The directory/CPU is call the “home agent” or “home processor” for this data

Illustration of Directory Based Cache Coherence Protocol



Operations of Directory-based Protocol: Source Snoop*


Labels: P1 is the requesting caching agent
P2 and P3 are peer caching agents
P4 is the home agent for the line
Precondition: P3 has a copy of the line in either M, E or F-state

MESIF protocol (Intel): Modified (M), Exclusive (E), Shared (S), Invalid (I) and Forward (F)

Step 1.
P1 requests data which is managed by the P4 home agent. (Note that P3 has a copy of the line.)

Step 2.
P2 and P3 respond to snoop request to P4 (home agent).
P3 provides data back to P1.

Step 3.
P4 provides the completion of the transaction.

* While the name has "snoop", this is directory based protocol. Names in computer science are usually confusing.

Figure credit: An Introduction to Intel QuickPath Interconnect, Robert A. Maddox, Gurbir Singh, and Robert J. Safranek, Intel April 06, 2009

Operations of Directory-based Protocol: Home Snoop

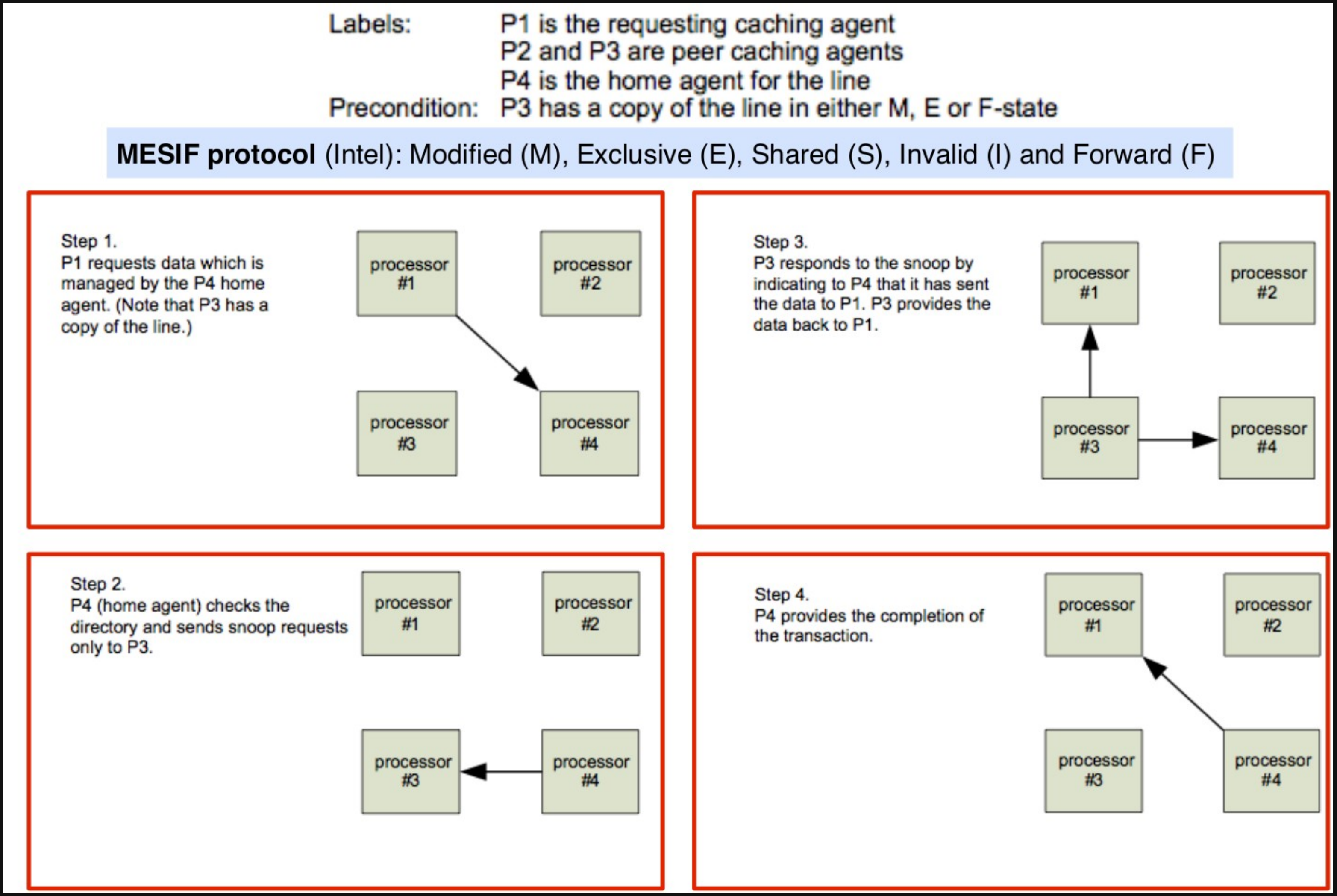
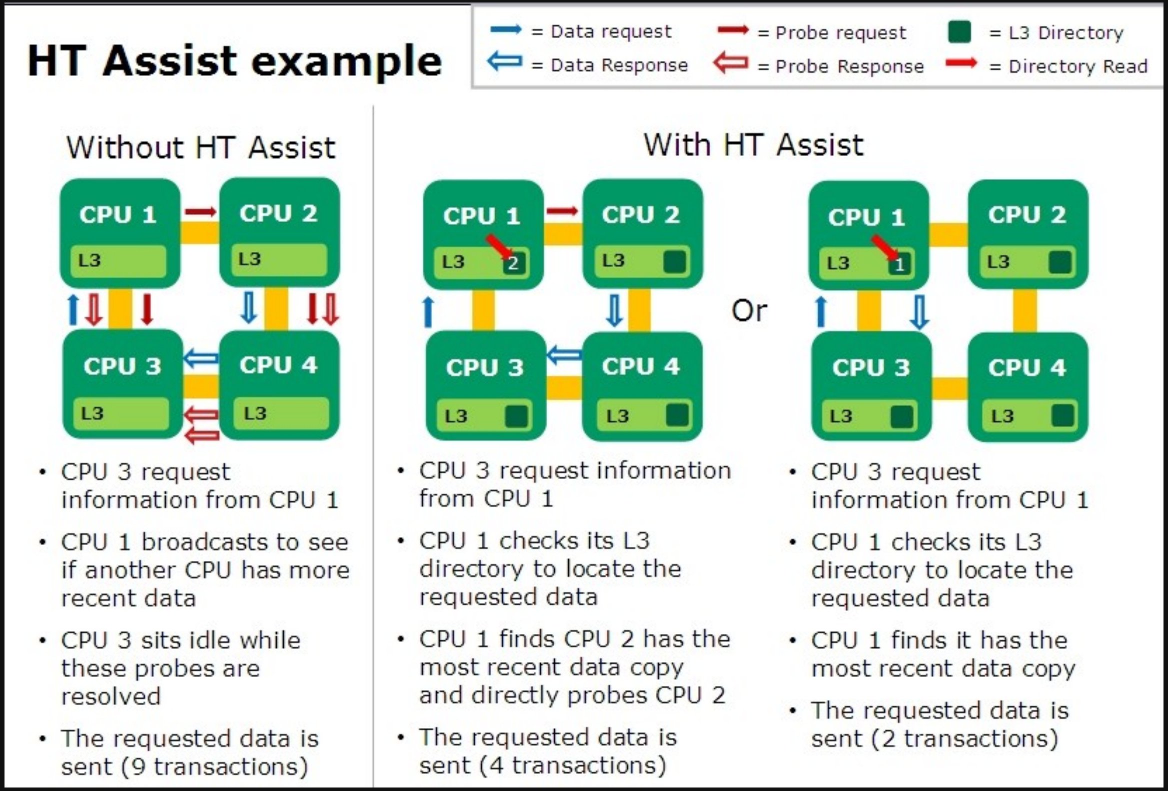

Labels:
P1 is the requesting caching agent
P2 and P3 are peer caching agents
P4 is the home agent for the line
Precondition:
P3 has a copy of the line in either M, E or F-state
MESIF protocol (Intel): Modified (M), Exclusive (E), Shared (S), Invalid (I) and Forward (F)
Step 1.
P1 requests data which is managed by the P4 home agent. (Note that P3 has a copy of the line.)
Step 2.
P4 (home agent) checks the directory and sends snoop requests only to P3.
Step 3.
P3 responds to the snoop by indicating to P4 that it has sent the data to P1. P3 provides the data back to P1.
Step 4.
P4 provides the completion of the transaction.
processor #1
processor #2
processor #3
processor #4

Figure credit: An Introduction to Intel QuickPath Interconnect, Robert A. Maddox, Gurbir Singh, and Robert J. Safranek, Intel April 06, 2009

AMD HT Assist (Probe filter)

- Similar to Intel Home snooping (AMD probably is the first user (inventor?) of this technology)



Without HT Assist:

- CPU 3 request information from CPU 1
- CPU 1 broadcasts to see if another CPU has more recent data
- CPU 3 sits idle while these probes are resolved
- The requested data is sent (9 transactions)

With HT Assist:

- CPU 3 request information from CPU 1
- CPU 1 checks its L3 directory to locate the requested data
- CPU 1 finds CPU 2 has the most recent data copy and directly probes CPU 2
- The requested data is sent (4 transactions)

Or:

- CPU 3 request information from CPU 1
- CPU 1 checks its L3 directory to locate the requested data
- CPU 1 finds it has the most recent data copy
- The requested data is sent (2 transactions)

Figure credit: http://www.qdpma.com/systemarchitecture/SystemArchitecture_Opteron.html

Source Snoop VS Home Snoop

- Source snoop offers the lowest latency (fewer steps) for small multiprocessor configurations.
  - Technically, source snoop is a combination of directory-based and snoopy cache coherence protocols
- Home snooping/HT Assist/Probe filter offers the best performance in systems with a high number of agents.
  - Home snooping requires knowing the home agent. This can be computed based on data address.
  - Home snooping requires home agent knows the current holder of data. This requires additionally hardware to record current data holders.
    - On AMD Magny-cours processor, part of L3 cache is used to record this information, reducing cache size from 6MB to 5MB.
- Another common disadvantage of directory-based protocol is it requires additional hardware for the directory.

Hardware Synchronization Instructions

Synchronization

- Coordinate sharing among threads
  - Support mutually exclusive access to shared data, e.g., mutex, lock and semaphores
  - Ensure threads advance through computation phases together, e.g., barriers
- Properly implementing all synchronization functions requires hardware support, i.e., support for atomic operations.
  - Modern architecture support atomic operations through coherence protocols
  - Older architectures may support atomic operations in other memory hardware, such as DRAM

Atomic Operation with Cache Coherence Protocol

- The sequence of operations when two cores want to atomically increment an integer *i* (one possible implementation)

Steps	Core0's Cache State	Core1's Cache State	Core0's Operation	Core1's Operation
0 (initial)	i=0 (S)	i=0 (S)		
1	i=0 (S)	i=0 (S)	Acquire bus	Acquire bus (success)
2	i=0 (I)	i=0 (I)		Invalidate all copies of *i*
3	i=0 (I)	i=0 (E)		Re-read *i* as exclusive
4	i=0 (I)	i=1 (M)		Increment *i*
5	i=0 (I)	i=1 (M)	Acquire bus (success)	Release bus
6	i=0 (I)	i=1 (I)	Invalidate all copies of *i*	*i* written back to memory
7	i=1 (E)	i=1 (I)	Re-read *i* as exclusive	
8	i=2 (M)	i=1 (I)	Increment *i*	
9	i=2 (M)	i=1 (I)	Release bus	

* Note that, technically any operations that use the bus require acquiring the bus first.

Hardware (HW) Synchronization Primitives/Instructions

- Besides an atomic increment, hardware provides quite a few atomic operations to support the implementation of high-level software synchronization primitives
- These atomic operations are provide to software as instructions

Common Atomic Instructions

- *As a common practice, the following instructions are described as C functions. But they are only instructions.
- test_and_set(void **M)*
  - Write 1 to the memory at location *M*
  - Returns the old value of the memory at *M*
- swap(void **M*, *Val)*
  - Write *Val* to the memory at location *M*
  - Returns the old value of the memory at *M*
- fetch_and_Φ(void **M*, *Val)*
  - Φ can be add, or, xor, sub …
  - Let the old value of memory location *M* be *old_val*
  - Replace the value at *M* with (*old_val* Φ *Val*)
  - Return *old_val*
- compare_and_swap(void *M, target_val, val)
  - Let the old value of memory location *M* be *old_val*
  - If (old_val == target_val) then write *Val* to the memory at location *&M*
  - Return True/1 if store was performed

Implement a Spinlock with Atomic Instructions

- a spinlock is a lock which causes its caller waits in a loop ("spin") while repeatedly checking if the lock is available.
- Four implementations each uses on atomic instruction

```
function Lock(int *lock)
{
 while (test_and_set(lock) == 1);
}
```

```
function Lock(int *lock)
{
 while (swap(lock, 1) == 1);
}
```

```
function Lock(int *lock)
{
 while (fetch_and_and(lock, 1) == 1);
}
```

```
function Lock(int *lock)
{
 while (compare_and_swap(lock, 0, 1) == false);
}
```

Questions On Spinlock Implementation

- How to release a lock?

Questions On Spinlock Implementation

- How to release a lock?
  - Just write a “0” to the lock. As long as the write is less than 64-bit wide on modern 64-bit machines, the write is always atomic.

Real Architecture Support for Atomic Instructions

- x86 and x86-64
  - Many instructions can be made atomic with prefix “LOCK”
    - e.g., the exchange-and-add instruction XADD becomes atomic fetch_and_add when used as “LOCK XADD”
    - e.g., the compare-and-exchange instruction CMPXCHG becomes atomic compare_and_exchange when used as “LOCK CMPXCHG”
    - e.g., the exchange instruction XCHG becomes atomic swap when used as “LOCK XCHG”
    - No test_and_set on x86, which can be easily simulated with “LOCK XCHG”
- ARM
  - Has two instructions to construct atomic code sequence
    - LDREX: load memory into register
    - STREX: store register value into memory
    - The use of LDREX starts an atomic operation sequence which ends with STREX
  - Common atomic operations can be simulated with these instructions

Compiler Support for Atomic Operations

- GCC provides a group of atomic intrinsics
  - __sync_fetch_and_(add/sub/or/and/xor/nand)
  - __sync_(add/sub/or/and/xor/nand)_and_fetch
  - __sync_bool_compare_and_swap
  - __sync_val_compare_and_swap
  - __sync_lock_test_and_set
  - Atomic intrinsics are automatically converted to corresponding atomic instructions by GCC

Software Synchronization with Hardware Atomic Operations

Approaches: Spinning vs. Blocking

- Blocking
  - what: suspend execution until a resource is available
  - advantage: frees up a processor for useful work
    - important when # threads > # cores
  - disadvantage: longer latency (context switch at a minimum)
  - examples: pthread_mutex_lock/unlock/trylock
- Spinning
  - what: repeatedly test a condition until it becomes true
  - advantage: low latency
  - disadvantage: ties up a processor core
    - may displace useful computation
  - examples: pthread_spin_lock/unlock/trylock

Approaches: Spinning vs. Blocking cont’d

- Rule of thumb
  - use spinning in a dedicated environment if # threads <= # cores
  - use blocking in shared environment or if # threads > # cores
- Blocking synchronizations are usually provided by the Operating System (OS) today
- Internally, OS uses spinlock because hardware does not directly support blocking synchronizations

Implementing a Spinlock

- Examples are on slide 29

Implementing a Blocking Lock

- Linux provides two functions:
  - WAIT (addr, val)
    - Checks if the value stored at the address addr is val, and if it is puts the current thread to sleep.
    - OS guarantee this operation is atomic
  - WAKE (addr)*
    - Wakes up one thread waiting on the address addr.

```
function Lock(int *lock)
{
  while(swap(lock,1)==1);
    WAIT(lock, 1);
}
```

```
function UnLock(int *lock)
{
  *lock = 0;
  WAKE(lock);
}
```

*Note that this is a simplified version of kernel WAKE function. The real Lock/UnLock implementation needs to be slightly changed based on the actual WAKE function. A complete implementation can be found at https://locklessinc.com/articles/mutex_cv_futex/

Combing Spinlock and Blocking Lock

- Even when # threads > # cores, spinlock may still be more efficient than blocking lock if
  - Every thread releases the lock in a few cycles after acquiring it
  - Because it may takes longer to make the WAIT system call then just waiting for that few cycles
- A better locking solution is to combine spinlock and blocking lock:
  - First, spin for a few cycles and wait for the lock
  - If lock is still not released, invoke WAIT and do blocking wait

Implement Barrier with Atomic Operations

- A barrier data structure will always have at least two members
  - count: the number of threads have reached barrier
  - total: the maximum number of threads that use this barrier
- The operations of a barrier wait:
  - Update the the counter to indicate a new thread has arrived, i.e., count++
  - If there are still threads that haven’t reached the barrier (i.e., count < total), wait for these threads (i.e., loop until count == total)
  - If all threads have arrived (i.e., count == total), reset counter to zero (i.e., count == 0)

A Naive Implementation of Barrier

```
struct barrier{
    int count;
    int total;
};
```

```
void barrier_wait(struct barrier *bar)
{
   /* increment the counter */
   atomic_inc(bar→count);
   /* wait if some threads are missing */
   while(bar→count < bar→total){};
   /* reset the counter*/
   bar→count = 0;
}
```

A Naive Implementation of Barrier cont’d

```
struct barrier{
    int count;
    int total;
};
```

```
void barrier_wait(struct barrier *bar)
{
    /* increment the counter */
    atomic_inc(bar→count);
    /* wait if some threads are missing */
    while(bar→count < bar→total){};
    /* reset the counter*/
    bar→count = 0;
}
```

To many threads reseting the counter. If one thread calls the barrier again before another thread reseting it, the value of count will be totally wrong.

A Better Implementation of Barrier

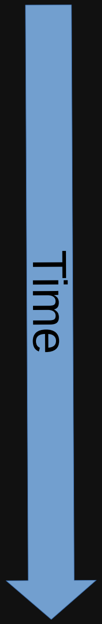
```
struct barrier{
    int count;
    int total;
};
```

```
void barrier_wait(struct barrier *bar)
{
   /* increment the counter */
   atomic_inc(bar→count);
   /* wait if some threads are missing */
   while(bar→count < bar→total){};
   /* reset the counter*/
   compare_and_swap(bar→count, bar→total, 0);
}
```

Only one thread will reset the counter.
Because compare_and_swap means:
  if(bar→count == bar→total){bar_count=0;}

A Better Implementation of Barrier cont’d

- However, there still is a problem with the previous implementation



```
Thread1:                                   Thread2:
call barrier_wait:                         ↓ Halt
 atomic_inc(count);//count=1               ↓ Halt
 compare if(count < total)//true           call barrier_wait:
 ↓ Halt                                     atomic_inc(count);//count=2
 ↓ Halt                                     compare if(count < total)//false
 ↓ Halt                                     comp&swap(count,total,0)//count=0
 ↓ Halt                                    call barrier_wait:
 ↓ Halt                                     atomic_inc(count);//count=1
 compare if(count < total)//true            compare if(count < total)//true
 compare if(count < total)//true            compare if(count < total)//true
 compare if(count < total)//true            compare if(count < total)//true
 compare if(count < total)//true            compare if(count < total)//true
 Now Thread1 misses one barrier wakeup, and both threads stuck here
```

The Pool Barrier Implementation

- The problem with previous implementation is that a thread may miss the event {count == total}.
  - The solution is to let thread wait on some other flag instead of “count”
- Add a new variable to the barrier
  - seq: a sequence number, roughly indicates the number of times that all threads have reach the barrier
  - Let threads wait on this sequence number

The Pool Barrier Implementation

```
struct barrier{
    int count;
    int total;
    int seq;
};
```

```
void barrier_wait(struct barrier *bar)
{
   /* get current sequence number */
   int cur_seq = bar->seq;
   /* increment the counter */
   old_count = fetch_and_add(bar→count,1);
   if(old_count< (bar->total-1))
      /* wait if some threads are missing */
      while(cur_seq == bar→seq){};
   else{ /*(bar→count == bar->total)*/
      /* reset the counter and increment seq*/
      bar→seq++;
      bar→count = 0;
   }
}
```

The Pool Barrier Implementation cont’d

```
struct barrier{
    int count;
    int total;
    int seq;
};
```

```
void barrier_wait(struct barrier *bar)
{
   /* get current sequence number */
   int cur_seq = bar->seq;
   /* increment the counter */
   old_count = fetch_and_add(bar→count,1);
   if(old_count< (bar->total-1))
       /* wait if some threads are missing */
       while(cur_seq == bar→seq){};
   else{ /*(bar→count == bar->total)*/
       /* reset the counter and increment seq*/
       bar→seq++;
       bar→count = 0;
   }
}
```

If a thread wakes up because of seq is increased, but it calls barrier_wait again before “count” is reset, deadlock still occurs. In short, these two statements has to be atomic

The Pool Barrier Implementation cont’d

```
struct barrier{
    union{
        struct
        {
            int seq;
            int count;
        };
        unsigned long long reset;
    };
    int total;
};
```

A small C trick: use “union” to create new variable “reset” that occupies same memory as seq and count;

```
void barrier_wait(struct barrier *bar)
{
   /* get current sequence number */
    int cur_seq = bar->seq;
    /* increment the counter */
    old_count = fetch_and_add(bar→count,1);
    if(old_count< (bar->total-1))
        /* wait if some threads are missing */
        while(cur_seq == bar→seq){};
    else{ /*(bar→count == bar->total)*/
        /* reset the counter and increment seq at the same time*/
        b→reset = b→seq + 1;
   }
}
```

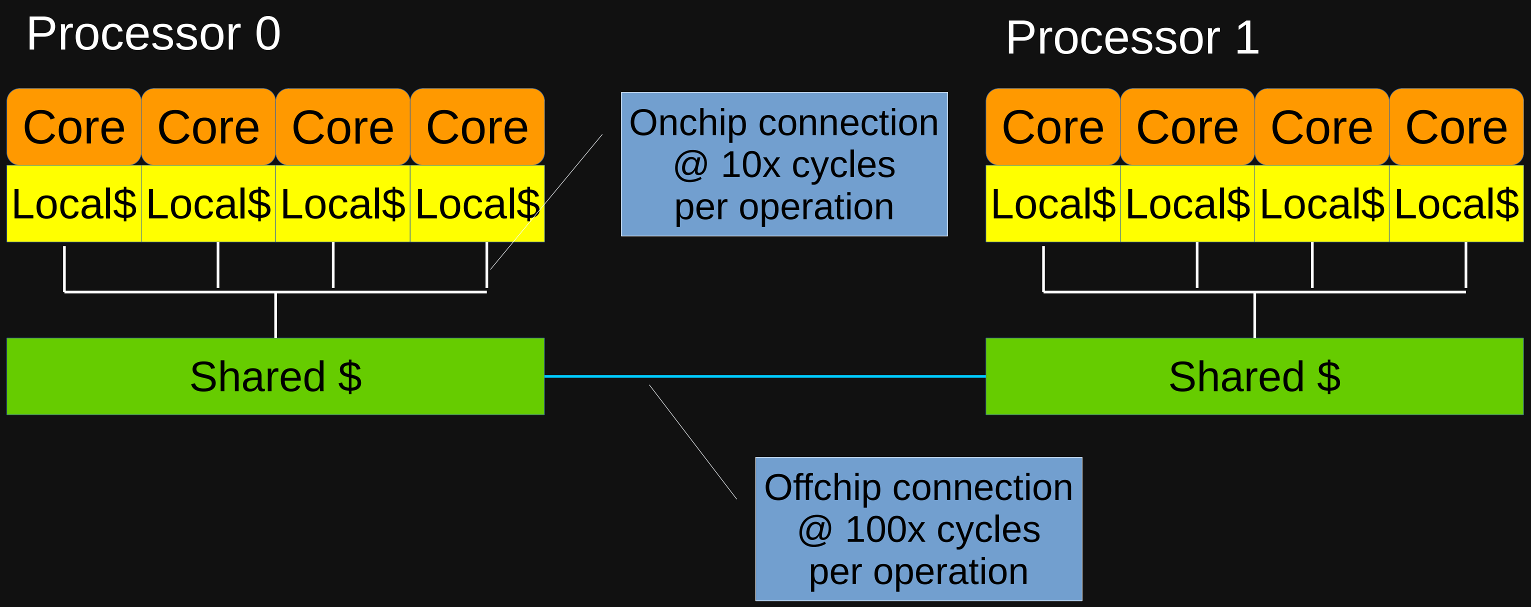
Synchronization on Large-scale Shared Memory Architecture

The Performance Bottleneck for Any Synchronization

- Cache coherence protocol is not free
  - Every coherence operation takes some time to finish and it adds to execution time
- Synchronization eventually relies on a shared object for communication
  - E.g., a lock variable or a barrier counter
  - Ensuring the coherence of a shared object adds to execution time

Coherence Cost on Large-Scale Shared Memory Machine

- The cost to ensure shared objects coherent is extremely high on large-scale shared memory machine
  - Because, inter-processor connections are very slow



Processor 0
Processor 1
Core
Local$
Shared $
Onchip connection @ 10x cycles per operation
Offchip connection @ 100x cycles per operation

Optimizing Synchronization on Large-scale Shared Mem Machine

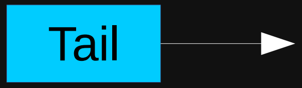
- The common and general idea is to disperse a shared object
  - instead of using one shared object, create multiple shared objects.
  - Each shared object is local to one core or processor
  - Each core/processor mostly only use its local shared objects

MCS Lock

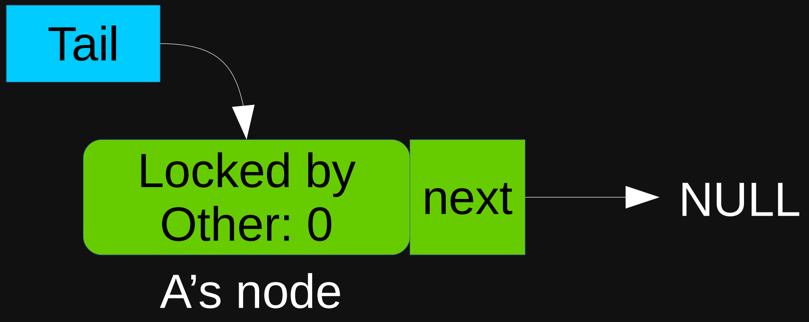
- By John Mellor-crummey and Michael Scott
  - Hence MCS
- The lock is similar to spin-lock
- The idea is to create a lock object for each thread/core, and each thread/core only spinning on it own copy of lock object
  - Thus, eliminate spinning on remote shared object and eliminate coherence operations
- Lock objects are managed with a shared-list

MCS Lock: Step 1

- Initially, the list of lock objects is empty

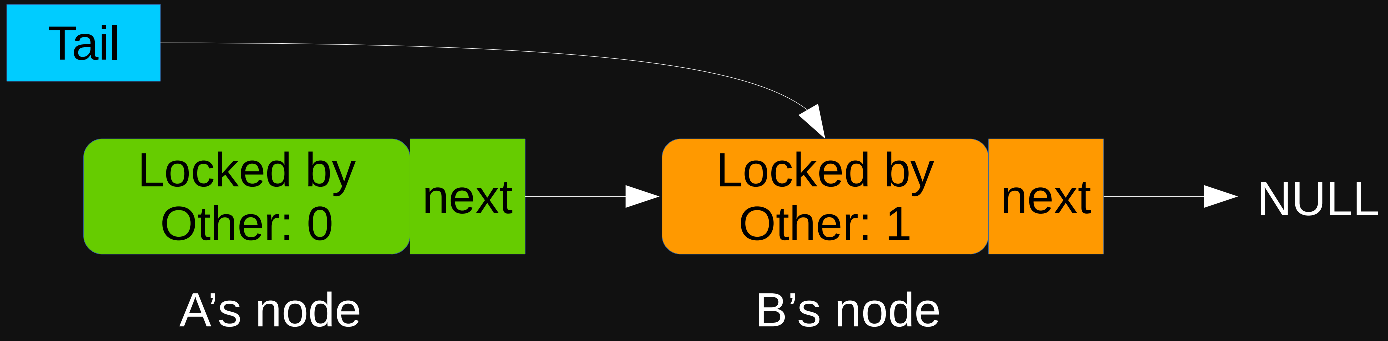


- Thread A tries to acquire the lock and is successful
  - A create a node in the list with a flag – “locked by other (threads)” – and next pointer



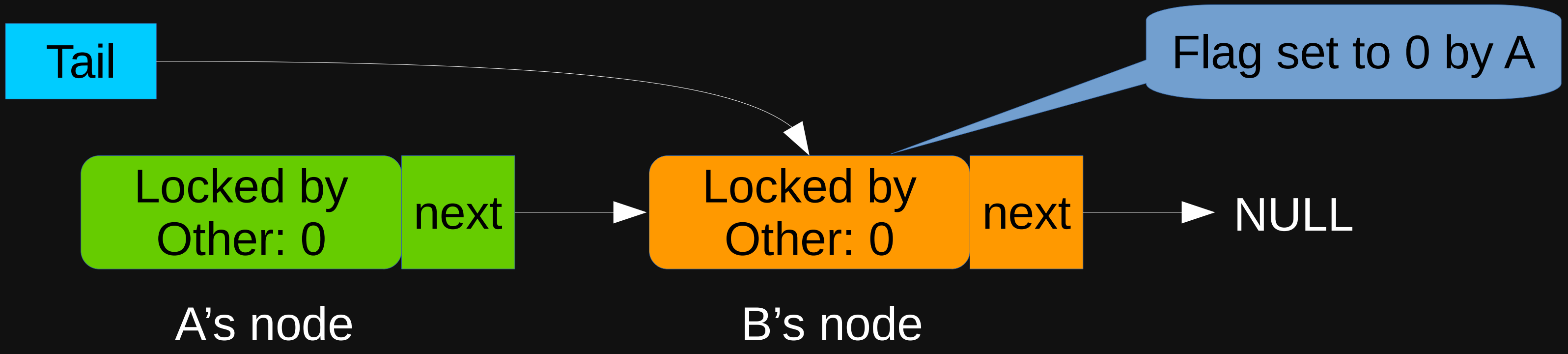
MCS Lock: Step 2

- Thread B tries to lock the lock
  - It creates a node for itself
  - If founds the linked list has node(s), so it assumes the lock is locked
    - with an atomic swap to the tail pointer to 1) set the tail pointer to B’s node and check if the tail pointer was NULL before the swap
  - B marks its “locked by other” to True and set A’s next pointer pointing to B’s node
  - B starts to spin on its own “locked by other” flag



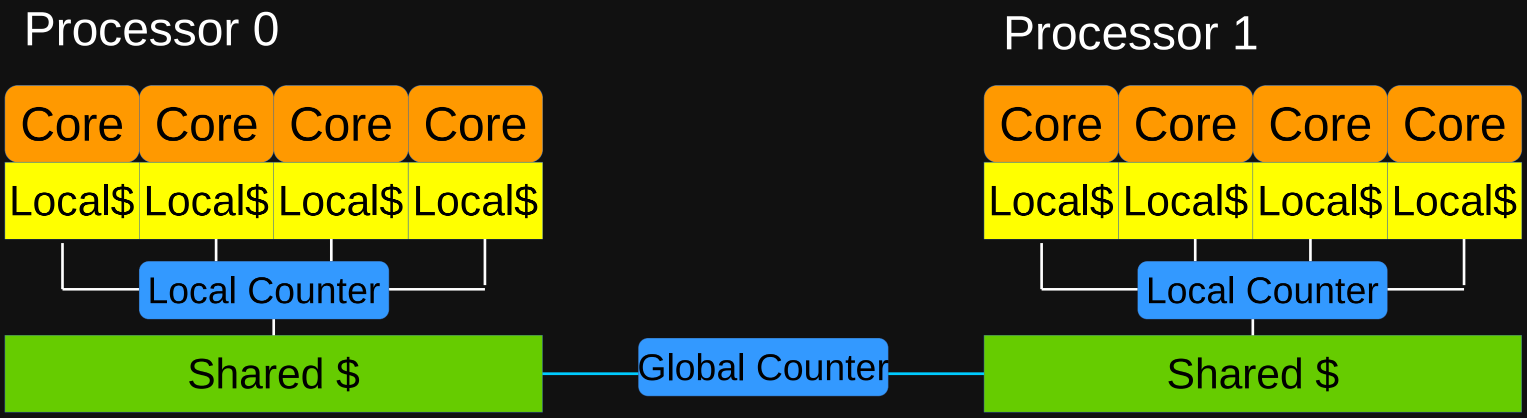
MCS Lock: Step 3

- A releases the lock
  - A check to see if its next node is NULL
  - If it is NULL, set tail to NULL
    - An atomic compare_and_swap is used: if (tail == A'node) {set tail to NULL}
    - If atomic compare_and_swap fails, wait until A's next node is not NULL
  - If A's next node is not NULL, set its next node's "lock by other" flags to false
    - In this example, B's flags is set to false and B can think it acquires the lock



Distributed Barrier

- Each processor has a local counter to counter how many threads on this processor has reached the barrier
- Only when all threads on this processor has reached the barrier, the global barrier counter is updated.
  - By using local barriers, we reduce the number of global counter updates, thus reducing the # of coherence operations



Separate Wait Queues

- For blocking lock, OS is involved
- In addition to the shared lock variable, OS also maintain a wait queue for each lock
- The wait queue is also a shared object and can become performance bottleneck
- OS can implement a queue local to each processor/core to mitigate the performance cost of updating/en-queuing/dequeuing a global queue

Memory Models and Weak Ordering

What is a Memory Model

- A contract between a program and any hardware and software that reorders operations in a program execution
- In the context of parallelism, a memory model governs interactions between threads and shared memory
  - atomicity, ordering, visibility
- Weak memory models: any load/store operation can be reordered with another, as long as the reordering doesn’t affect single thread execution
  - i.e., as long as reordering does not affect data dependency
  - read/write, read/read, write/read, write/write
- Why weak memory models? performance!
  - reordering of accesses by compiler, e.g., register allocation
  - reordering by hardware: don’t wait for operations to globally complete before continuing

A Weak Order Example

- In producer/consumer model, producer updates:
  - A flag the indicates there are new products
  - Data that represents the products

```
Application code:
    Update product_data
    Update flag
```

$\Rightarrow$

```
Execution order:
    Update flag
    Update product_data
```

The Problem with Weak Memory Order

- In the producer/consumer example:

```
Execution order:
    Update flag
    Update product_data
```

Consumer sees the flag updated and assumes data is ready, which is not true.

- For certain applications, the order of memory updates is also important, which is broken by the weak memory order.

Enforcing Memory Order

- To ensure memory order correctness, hardware provides memory barrier or memory fence
  - A memory barrier is a type of barrier instruction that causes a CPU enforce an ordering constraint on memory operations issued before and after the barrier instruction.
  - This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.
- Compilers may also provide memory fence intrinsics
  - These intrinsics may or may not depend on hardware instructions

Fixing Producer/Consumer Model with Memory Fence

- Add a memory fence instruction after updating product data

```
Application code:
    Update product_data
    memory fence instruction
    Update flag
```

```
Execution order:
    Update product_data
            ¦
            ¦ CPU wait for
            ¦ updates to
            ¦ finish
            ▼
    Update flag
```

Memory Fence Implementations

- x86 and x86-64
  - MFENCE: the memory fence instruction
  - LFENCE: Load fence; memory fence for load instructions only
  - SFENCE: Store fence; memory fence for store instructions only
- GCC
  - __sync_synchronize

Lock-free Programming and Transactional Memory

The Problem with Locks

- Dead lock
- Priority Inversion
  - Low-priority thread holding a lock can prevents a high-priority thread from executing, resulting in a mid-priority thread to run
- Convoying/Thundering herd
  - Case 1 (convoying): Many threads contending for a lock; one thread holding the lock uses up its time slice and gets context switched out; other threads wake up only to fail to acquired the lock; time is wasted on waking up the waiting threads
  - Case 2 (thundering herd): Many waiting threads wake up; only one thread succeeded to acquire the lock; the rest threads wake up only to fail to acquired the lock; time is wasted on waking up the waiting threads
- Async-Signal Safety
  - Holding a lock while interrupted by a signal and tries to acquire the same lock in the single handler
- Kill-tolerant availability
  - How to safely kill a thread holding a lock?
- Pre-emption tolerance
  - What if a thread holding a lock being pre-empted?
- Performance and scalability

Lock-free Programming

- Coordinated and safe access to shared data without the use of synchronization primitives
  - This definition is kind of vague and inaccurate...
- Still requires some hardware support (e.g., atomic instructions and memory barriers)
  - Essentially programming with only atomic instructions/barriers
- Also known as lockless or non-blocking programming
- Some additional readings:
  - http://preshing.com/20120612/an-introduction-to-lock-free-programming/
  - https://devblogs.microsoft.com/oldnewthing/20141127-00/?p=43523

A Simple Lock-free Stack

- Basic element type for the stack

```
struct node{
    node *next;
    int data;
}

struct node *head; //top of the stack
```

- Simple push with compare&swap

```
void push(int v)
{
    struct node *n = new struct node(t);
    do{
        n→next = head;
    }while(!compare&swap(&head, n→next, n));
}
```

```
if(head == n->next)
    head = n
```

A Simple Lock-free Stack

- A simple pop with compare&swap

```
int pop()
{
    struct node *n = head;
    while(n){
        if(compare&swap(&head, n, n→next))
            return n→data;
        n = head;
    }

    return not_a_vale;
}
```

```
if(head == n)
    head = n->next
```

The ABA Problem

- What happens is pop() is interrupted between getting n→next and compare&swap?

```
int pop()
{
    struct node *n = head;
    while(n){
        if(compare&swap(&head, n, n→next))
            return n→data;
        else
            n = head;
    }

    return not_a_vale;
}
```

There may be some problem
if after n→next is
read and before compare&swap,
some one removes and reuses n

The ABA Problem cont’d

```
Thread1:                                  Thread2:
call pop():                              ↓ Halt
 n = A  //read A from head               ↓ Halt
 read A→next                             call pop():
 ↓ Halt                                    pops A
 ↓ Halt                                  dealloc A
 ↓ Halt                                  alloc node C, OS reuses A’s memory
 ↓ Halt                                  call pop():
 ↓ Halt                                    pops B
 ↓ Halt                                  call push(C)
 compare&swap with A→next succeed        ↓ Halt
 Although compare&swap will success, but the new head points to
 A→next instead of C->next
```

The solution to ABA Problem

- Use some bits of the next pointer to keep a tag
  - Not all 64 bits in a pointer is required, we can use some bits as a tag
  - Or use a compare&swap that is larger than 64bits, and add tag in the compare&swap
- Defer memory reclamation

Other Examples of Lock-free Programming

- The MCS lock implementation we have seen is lock-free
- Technically speaking, spinning locks implementation is lock-free
- Linux’s Read-copy-update (RCU)
  - Many kernel objects are shared and constantly updated, using locks to protect them are too expensive.
  - Instead of locks, using basic atomic instructions for object updates
  - Deleting objects are tricky without locks, since writes to complex objects cannot be done with simple atomic instructions
  - To delete an object
    - Remove pointers to the old object so that no new threads can read the old object
    - Wait for all threads are done reading the old object (doable in a non-preemptive kernel – Linux prohibits context switch for kernel threads holding lock)
    - Delete the original object

Designing Lock-free Algorithms

- The reason we use locks
  - To protect accesses to shared data
  - Shared data may be much larger than any atomic instructions can handle
  - We need a way to use small atomic memory updates to protect much larger data
    - Lock is a straight-forward and kind of a brute-force way of doing it
- Lock-free algorithms
  - Similar to using locks, convert the protection of large shared data to a small atomic memory update
  - Instead of a lock variable, find a small part of the shared data or some other flags as the target for atomic update

Common Use Case of Lock-free Programming

- Designing original lock-free algorithms is hard
- People usually use existing lock-free data structures
  - Link-list, stack, queue etc.
- Not all synchronizations can be replaced with lock-free algorithms
- Lock-free algorithms usually provides better scalability and performance than using locks

Transactional Memory

- Similar to database transactions:
  - A database transaction is a sequence of operations performed as a single logical unit of work atomically and consistently.
- Transactional memory allows a sequence of memory operations performed atomically and consistently.

```
with transaction():
        a -= b;
        b += c;   } three ops guaranteed to be atomic and consist
        C = 0;
```

- Transactional memory requires hardware support
  - Intel officially support transactional memory from Skylake (2015).

Implementation Transactional Memory

- Similar to database transactions, an undo log is created at the beginning of execution of a transaction If one memory operation fails, the memory is restored using the undo log.
  - The implementation is very hard. Intel Haswell was supposed to be the first Intel CPU supporting TM, but it was later found that the implementation is buggy.

The Benefits of Transactional Memory

- Eliminate locks
  - Completely free from deadlock
  - Simplify programming
- Better performance
  - Fully hardware-supported, no need to rely on slow OS system calls to maintain correct concurrency
  - Fast on low-contended environment due to the lack of locks