

Multiprocessor Issues

Consistency and Coherency

- Memory can change under a cache
 - Writes from other processors to memory
- 1. Consistency:**
 - The order in which changes to memory are seen by different processors
- 2. Cache coherence:**
 - Values in caches all match each other
 - How consistency is implemented with caches

Memory consistency

When several processors are reading and writing memory, what value is read by each processor?

- Defines semantics of order-dependent operations
 - E.g. does mutual exclusion work?
 - How to ensure that it does work?
- There are many memory consistency **models**

Consistency models: terminology

- **Program order**: order in which a program on a processor appears to issue reads and writes
 - Refers only to local reads/writes
 - Even on a uniprocessor \neq order the CPU issues them!
 - Write-back caches, write buffers, out-of-order execution, etc.
- **Visibility order**: order in which all reads and writes are seen by one or more processors
 - Refers to all operations in the machine
 - Depends on memory consistency model

Memory consistency models

- **Strict/Sequential consistency**
 - Writes appear in the order they are made
 - reads return the most recently written value
 - Concurrent operations can occur in any (consistent) order
- **Serializable**
 - All operations appear to occur in some serial order, to all processors
- **Linearizable (strictly serializable)**
 - Serializable and consistent with real time

Other Models

- **Processor ordering**
 - writes from one CPU are seen in order
 - Writes may be seen in different orders by different CPUs
- **Weak/Barrier ordering**: rules for synchronizing accesses (atomic read/modify/write instructions)
 - Synchronising accesses sequentially consistent
 - Synchronising accesses act as a barrier:
 - previous writes completed
 - future read/writes blocked

Important to know your hardware!

- x86: processor ordering
- PowerPC: weak/barrier ordering

Sequential consistency (SC)

1. Operations from a processor appear (to all others) in program order
2. Every processor's visibility order is the same as every other processor's

Requirements:

- Each processor issues memory ops in program order
- RAM totally orders all operations to each location
- Memory operations are atomic

Adve Implementation Rule

Let's assume (wlog) that each process specifies that its own operations happen in some order

- E.g., read A, write B, append C, ...
- If concurrent, system can choose the order

Serializable/sequentially consistent if

1. Operations applied in processor order, and
2. all operations to same memory location are serialized (as if to a single copy)

Sequential consistency example

Results:

- $(u=1, v=1)$:
 - Possible under SC: (a_1, a_2, b_1, b_2)
 - (a_1, a_2) and (b_1, b_2) are both program orders
- $(u=1, v=0)$:
 - Impossible under SC:
 - No interleaving of program orders that generates this result
 - Would require: $a_2 > b_1 > b_2 > a_1$

CPU A		CPU B	
a_1 :	$*p = 1;$	b_1 :	$u = *q;$
a_2 :	$*q = 1;$	b_2 :	$v = *p;$

Sequential consistency example

Results:

- $(u=1, v=1)$:
 - Possible under SC: (a_1, b_1, a_2, b_2)
 - (a_1, a_2) and (b_1, b_2) are both program orders
- $(u=0, v=0)$:
 - Impossible under SC:
 - No interleaving of program orders that generates this result
 - Would require: $a_2 > b_1 > b_2 > a_1$

CPU A		CPU B	
a_1 :	$*p = 1;$	b_1 :	$*q = 1;$
a_2 :	$u = *q;$	b_2 :	$v = *p;$

Sequential consistency

- Advantages:
 - Easy to understand for the programmer
 - Easy to write correct code to
 - Easy to analyze automatically
- Disadvantages:
 - Hard to build a fast implementation
 - Cannot reorder reads/writes
 - even in the compiler
 - even from a single processor!
 - Cannot combine writes to same cache line (write buffer)

Relaxing sequential consistency

CPU A		CPU B	
a ₁ :	*p = 1;	b ₁ :	u = *q;
a ₂ :	*q = 1;	b ₂ :	v = *p;

- Recall program order requirement for SC:
 - Out-of-order execution might reorder (b₂, b₁)
 - Write buffer might reorder (a₁, a₂)
 - a₁ might miss in the cache, and a₂ might hit
 - Compiler might reorder operations in each thread
 - Or optimize out entire reads or writes
- What can be done?

Relaxing sequential consistency

- Many, many different ways to do this!
E.g.:
 - Write-to-read: later reads can bypass earlier writes
 - Write-to-write: later writes can bypass earlier writes
 - Break write atomicity (no single visibility order)
 - Weak ordering: no implicit order guarantees at all
- Explicit synchronization instructions
 - x86: lfence (load fence), sfence (store fence), mfence (memory fence)
 - Alpha: mb (memory barrier), wmb (write memory barrier)

Processor Consistency

- Also PRAM (Pipelined Random Access Memory)
 - Implemented in Pentium Pro, now part of x86 architecture.
- Write-to-read relaxation:
later reads can bypass earlier writes
 - All processors see **writes** from one processor in the order they were issued.
 - Processors can see **different interleavings of writes** from different processors.

Processor (PRAM) Consistency

CPU A		CPU B		CPU C	
a_1 :	$*p = 1;$	b_1 :	$u = *p;$	c_1 :	$v = *q;$
		b_2 :	$*q = 1;$	c_2 :	$w = *p;$

- $(u,v,w) = (1,1,0)$ is possible in PRAM
 - B sees visibility order (a_1, b_2)
 - C sees visibility order (b_2, a_1)

Other consistency models

	Alpha	PA-RISC	Power	X86_32	X86_64	ia64	zSeries
Reads after reads	✓	✓	✓			✓	
Reads after writes	✓	✓	✓			✓	
Writes after reads	✓	✓	✓	✓	✓	✓	✓
Writes after writes	✓	✓	✓			✓	
Dependent reads	✓						
Ifetch after write	✓		✓	✓		✓	✓

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!

Implementing Single Copy

- Cache invalidation
 - Before every write, locate all copies of data and remove them
 - Apply change to single remaining copy
- Lease: permission for some period of time
 - Ex: lease to use cached copy of some data item
 - Wait until lease expires before applying update (plus clock skew)
 - Or ask client to return lease

Terminology

Lease

- Allow client to use cached copy for some period of time (lease)
- Before lease expires, client can renew
- After lease expires, client discards cached copy
- On next use, client fetches the latest version.

Write through

- All writes are sent through to memory

Write back

- Writes applied to local copy
- Sent to memory in background

Implementing SC with a snoopy cache

- Cache “snoops” on reads/writes from other processors
- If a line is valid in local cache:
 - Remote (other processor) write to line
⇒ invalidate local line
- Requires a write-through cache!
 - But coherency mechanism ⇒ sequential consistency
- Line can be valid in many caches, until a write

Directory-Based Cache Coherence

- How do we know which cores have a location cached?
 - Hardware keeps track of all cached copies
 - On a read miss, if held exclusive, fetch latest copy and invalidate that copy
 - On a write miss, invalidate all copies
- Read-modify-write instructions
 - Fetch cache entry exclusive, prevent any other cache from reading the data until instruction completes

Write Through Cache Coherence

- Before applying update at server:
 1. Send message to all clients with copy
 2. Each client invalidates, responds to server
 3. Server waits for all invalidations, then does update
 4. Then returns to client
- Reads can proceed
 - Whenever there is a local copy
 - Or if no write ahead of it in the queue at the server

Questions

- If write is in progress, can server perform reads/writes to other memory locations?
- While waiting for invalidations/lease expiration, is it ok to read (old value) at a client?
- While waiting for invalidations/lease expiration, can server return new value to a new client request?

More Questions

- Why does server need to wait until write is applied before returning to client?
- Why does server need to queue incoming requests while write is in progress?
- How much directory state do we need at the server?

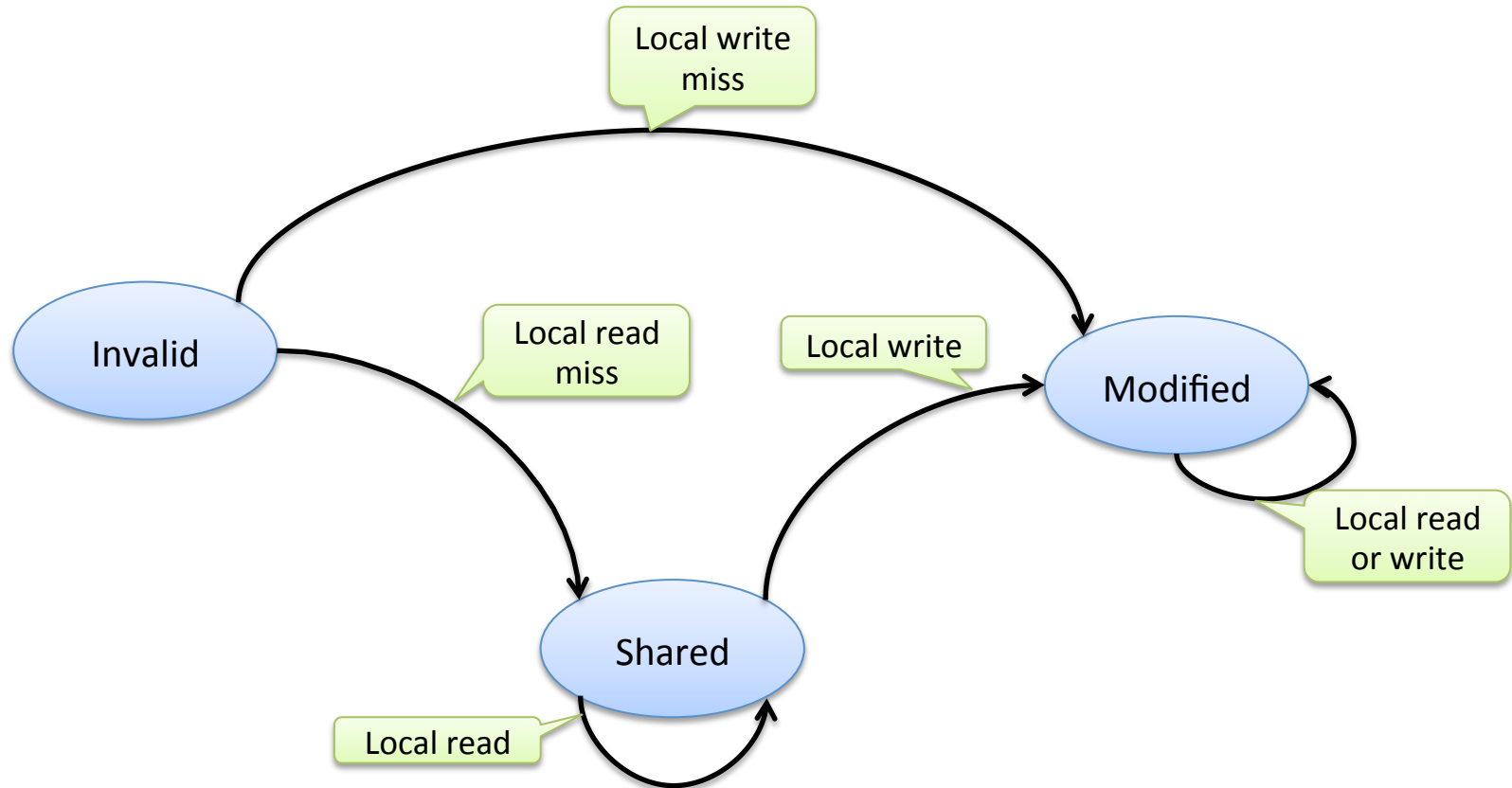
Example

- Two concurrent writes to two concurrent readers. Readers have item cached.
- Writers send change through to server
 - What is the order of operations?
- Server uses callback state to invalidate caches
 - For first write, what about second write?
- Reader has a cache miss and fetches the value from the server.

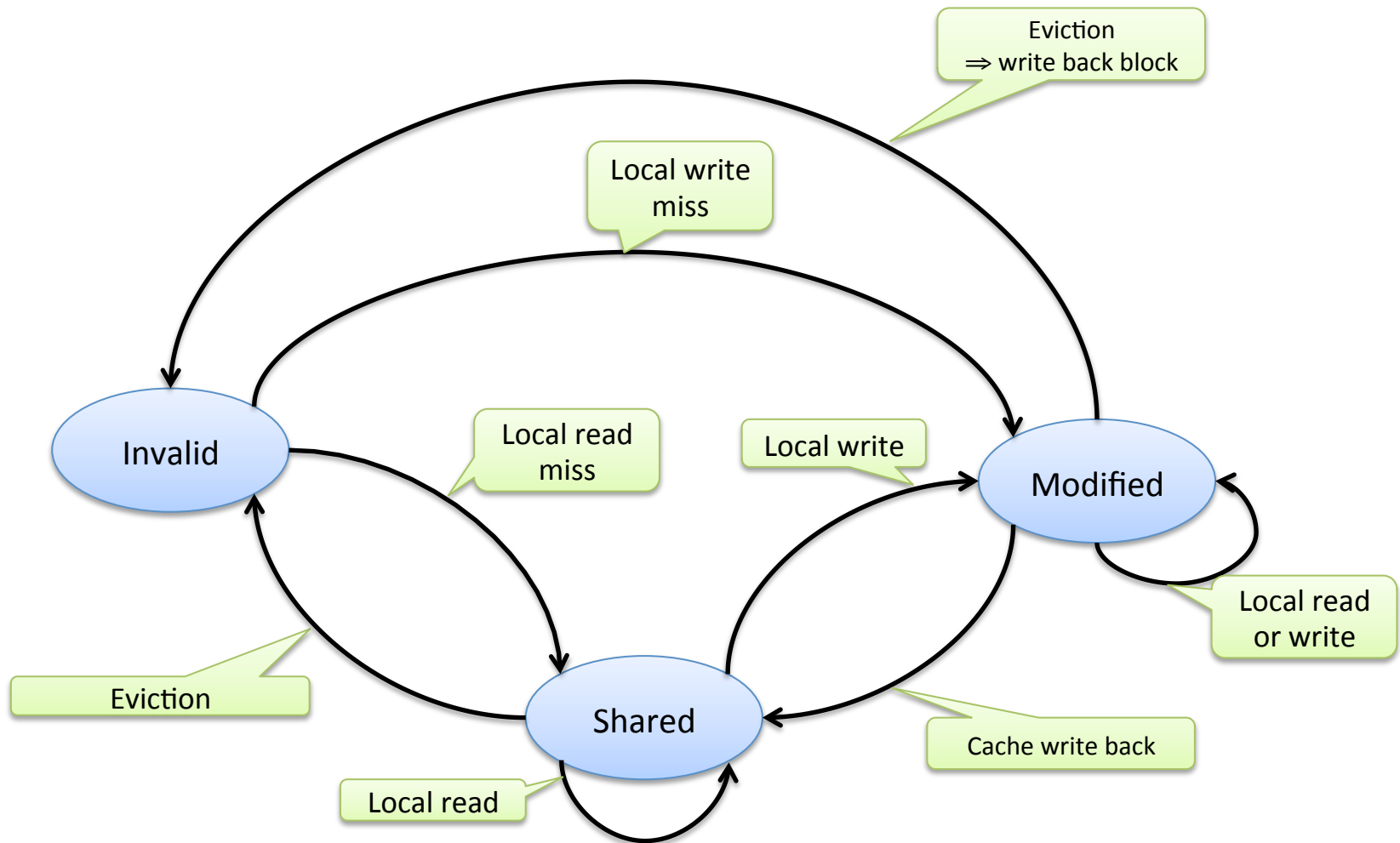
Write Back Cache Coherence

- Server tracks which clients have cached copy
- On write miss, client asks server to:
 1. Send message to all clients with copy
 2. Each client invalidates, responds to server
 3. Server waits for invalidations, then returns to client
 4. Client performs write
- Reads can proceed whenever there is a local copy
- Careful ordering of requests at server
 - Enforce processor order, avoid deadlock

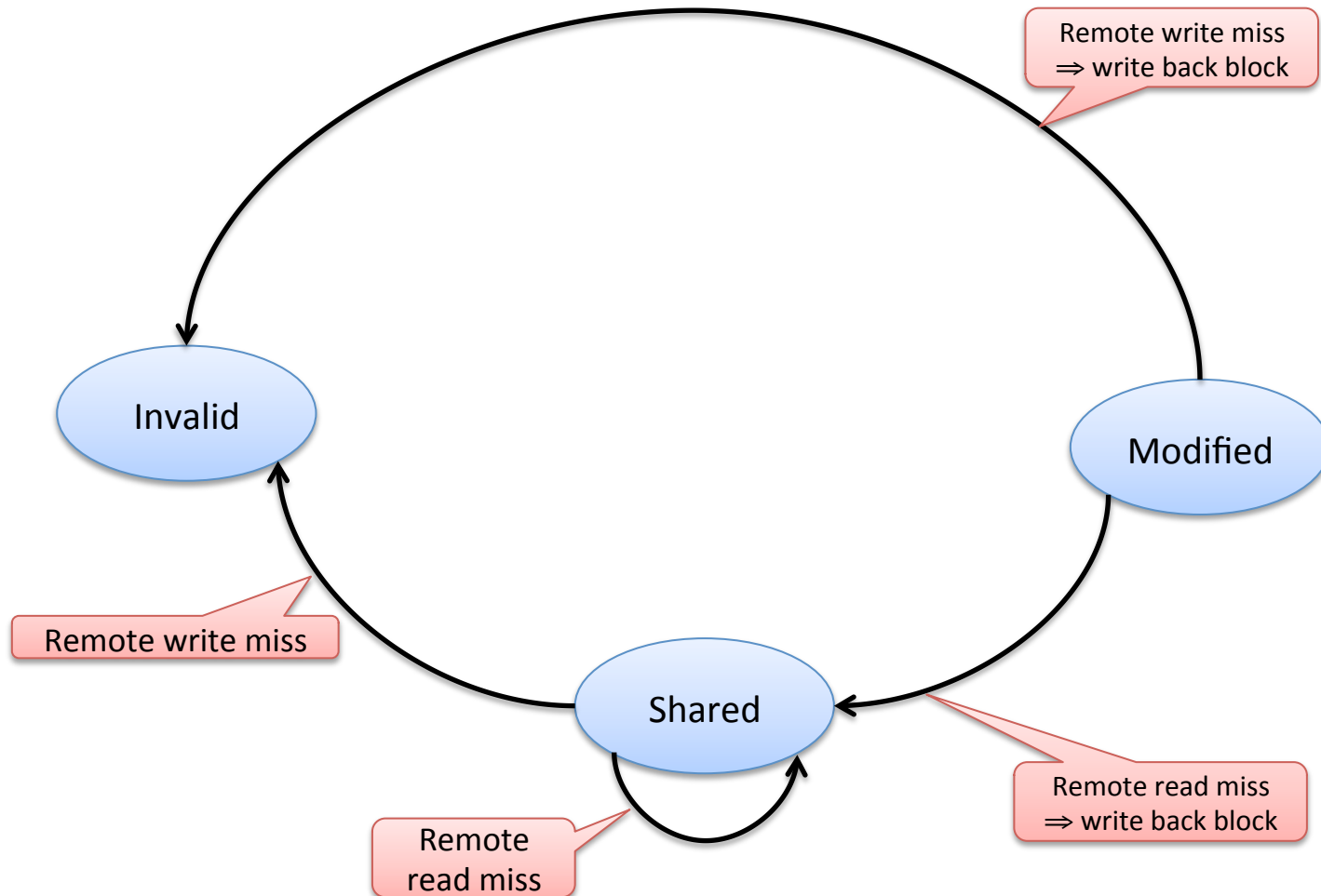
MSI state machine: local (processor) transitions



MSI state machine: local (processor) transitions



MSI state machine: remote (snooped) transitions



MSI issues

Assumes we can distinguish remote processor read and write misses

- In I state, executing a write miss:
 - Need to first **read** line (allocate)
 - If someone else has it in M state, need to wait for flush
- In M state, other core issues a read:
 - Must flush line (required)
 - Invalidate line?
 - But what if you want read sharing? Extra cache miss!
 - Transition to shared?
 - But what if it's actually a remote **write** miss? Extra invalidate!

MESI protocol

- Add a new line state: “exclusive”
 - Modified:** This is the only copy, it’s dirty
 - Exclusive:** This is the only copy, it’s clean
 - Shared:** This is one of several copies, all clean
 - Invalid**
- Add a new bus signal: **RdX**
 - “Read exclusive”
 - Cache can load into either “shared” or “exclusive” states
 - Other caches can see the type of read
- Also: **HIT** signal
 - Signals to a remote processor that its read hit in local cache
- First x86 appearance in the Pentium

MESI invariants

- Allowed combination of states for a line between any pair of caches:

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

- Protocol must preserve these invariants


MSI invariants:

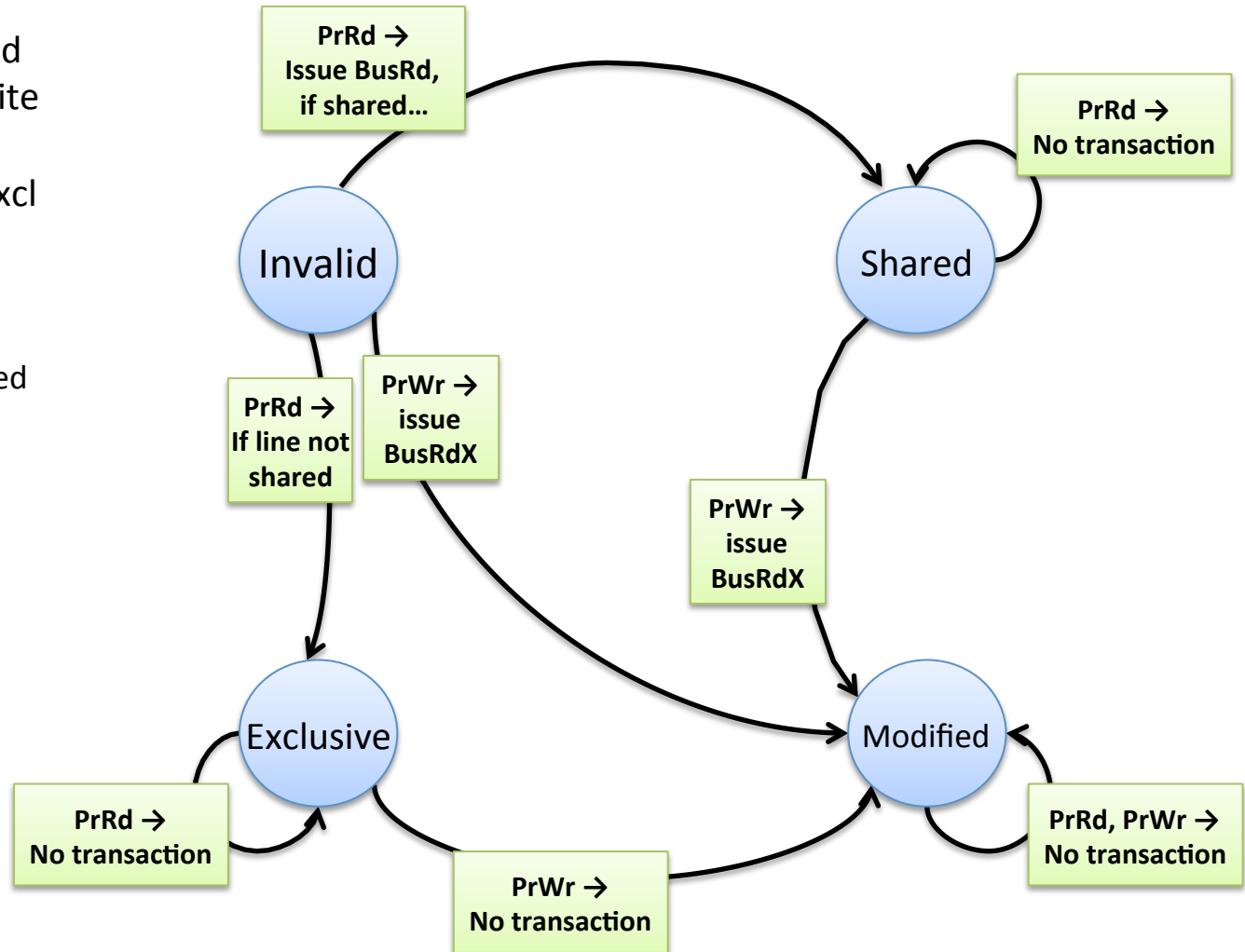
	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓

MESI state machine

Terminology:

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


 Processor-initiated

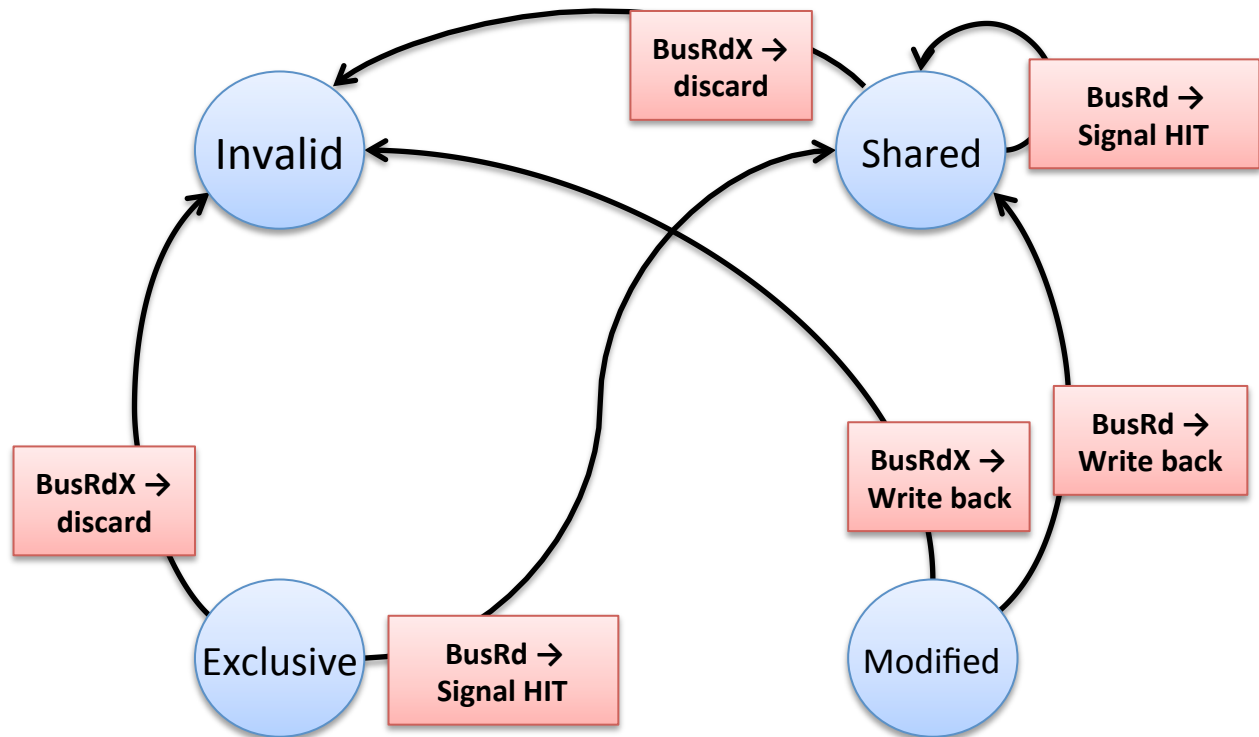


MESI state machine

Terminology:

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

 Snoop-initiated



MESI observations

- Dirty data always written through memory
 - No cache-cache transfers
 - “Invalidation-based” protocol
- Data is always either:
 1. Dirty in one cache
 - ⇒ must be written back before a remote read
 2. Clean
 - ⇒ can be safely fetched from memory

Good if:

latency of memory \ll latency of remote cache

MOESI protocol

Add new “Owner” state: allow line to be modified, but other unmodified copies to exist in other caches.

Modified:

No other cached copies exist, local copy dirty

Owner:

Unmodified copies may exist, local copy is dirty

Exclusive:

No other cached copies exist, local copy clean

Shared:

Other cached copies exist, local copy clean

One other copy might be dirty (state Owner)

Invalid:

Not in cache.

MOESI invariants

- Allowed combination of states for a line between any pair of caches

	M	O	E	S	I
M					✓
O				✓	✓
E					✓
S		✓		✓	✓
I	✓	✓	✓	✓	✓

- MOESI can satisfy a read request in state I from a remote cache in state O, for example.

Good if:

latency of remote cache < latency of main memory

Three Clients Example

Client 1	Client 2	Client 3
<pre>k1 = f(data); done1 = true;</pre>	<pre>while(done1 == false) ; k2 = g(k1); done2 = true;</pre>	<pre>while(done2 == false) ; rslt = h(k1,k2);</pre>

Initially, done1, done2 = false

Intuitive intent:

client3 should execute h() with results from client1 and client2
waiting for client2 implies waiting for client1

Question

Is write back always more efficient than write through?

Distributed Shared Memory

- Can run a parallel program across a network of servers
 - Threads communicate through shared memory, not message passing
- Set virtual memory page protection to trigger fault whenever remote operation needed:
 - read to an invalid page
 - write to an invalid or read-only page

Example

Parallel successive mesh approximation

- Update each element based on neighbors
- Repeat until converged

DSM approach

- Put boundary elements in their own pages
- Automatic exclusive when updated
- Automatic fetch of neighbor's boundary pages

Message passing approach

- Explicitly fetch boundary elements from neighbors

Transactional Memory

Group of operations with four properties:

- Atomic – all or nothing
- Consistent – equivalent to some sequential order
- Isolation – no data races between groups
- Durable – once done, stays done

Transactions appear to occur in some serial order:

- $T_0, T_1 \dots T_i, T_{i+1} \dots$
- Everything T_i depends on completed in some earlier transaction

Transactions Across Shards

Setting: data store partitioned across servers

Individual updates are serializable using cache coherence

What about updates to groups of items?

- Items may be on different shards

Transactional Memory

Use write back cache coherence

- Pull data needed for the transaction into local cache, write ownership
- Perform transaction
- Release data

Synchronization Performance

- A program with lots of concurrent threads can still have poor performance on a multiprocessor:
 - Overhead of creating threads, if not needed
 - Lock contention: only one thread at a time can hold a given lock
 - Shared data protected by a lock may ping back and forth between cores
 - False sharing: communication between cores even for data that is not shared

A Simple Critical Section

```
// A counter protected by a spinlock
Counter::Increment() {
    while (test_and_set(&lock))
        ;
    value++;
    test_and_clear(&lock);
}
```

A Simple Test of Cache Behavior

Array of 1K counters, each protected by a separate spinlock

– Array small enough to fit in cache

- Test 1: one thread loops over array
- Test 2: two threads loop over different arrays
- Test 3: two threads loop over single array
- Test 4: two threads loop over alternate elements in single array

Results (64 core AMD Opteron)

One thread, one array	51 cycles
Two threads, two arrays	52
Two threads, one array	197
Two threads, odd/even	127

Reducing Lock Contention

- Fine-grained locking
 - Partition object into subsets, each protected by its own lock
 - Example: hash table buckets
- Per-processor data structures
 - Partition object so that most/all accesses are made by one processor
 - Example: per-processor heap
- Ownership/Staged architecture
 - Only one thread at a time accesses shared data
 - Example: pipeline of threads

What If Locks are Still Mostly Busy?

- MCS Locks
 - Optimize lock implementation for when lock is contended
- RCU (read-copy-update)
 - Efficient readers/writers lock used in Linux kernel
 - Readers proceed without first acquiring lock
 - Writer ensures that readers are done
- Lock-free data structures

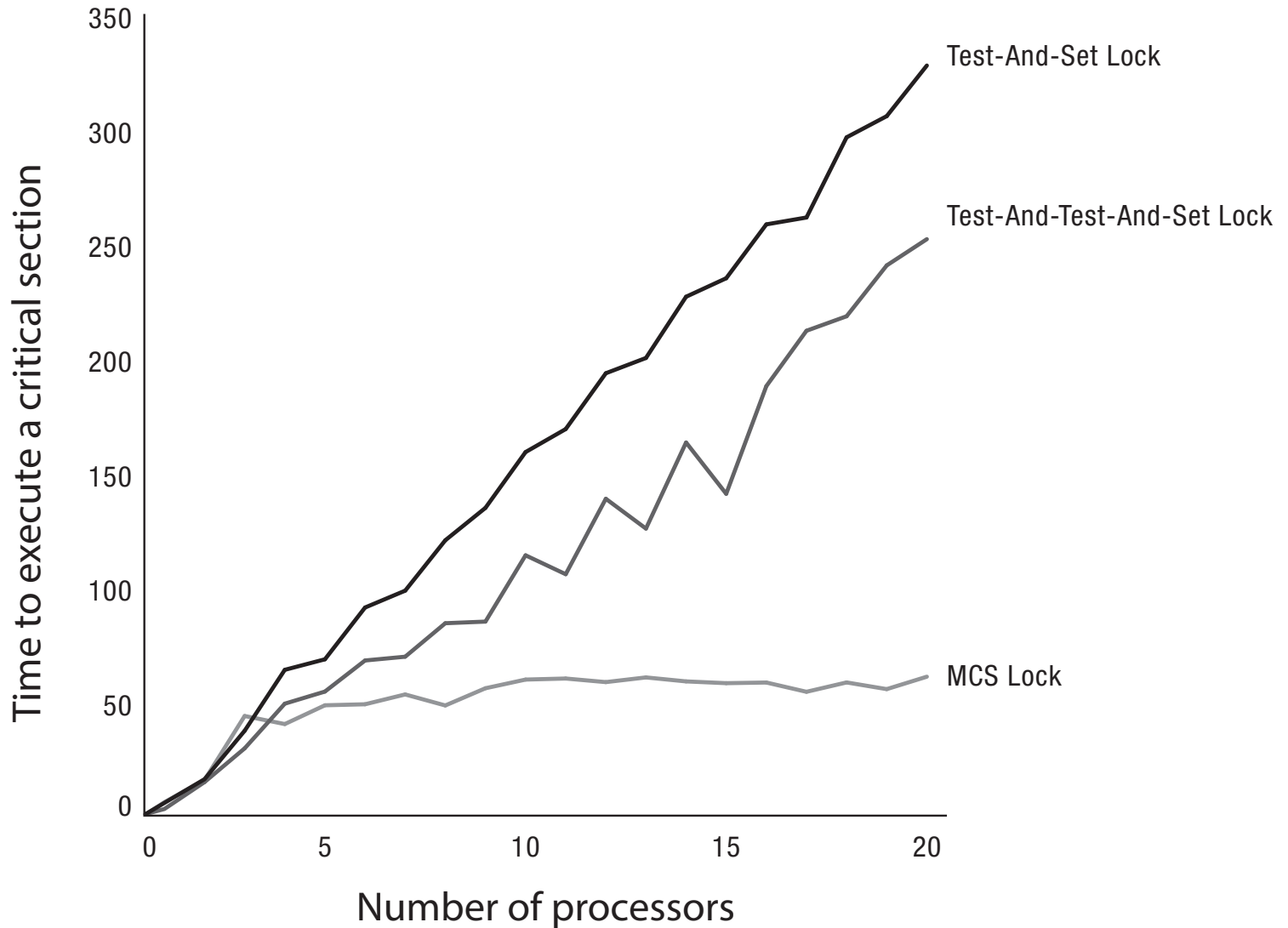
What if many processors call Counter::Increment()?

```
Counter::Increment() {  
    while (test_and_set(&lock))  
        ;  
    value++;  
    lock = FREE;  
    memory_barrier();  
}
```

What if many processors call Counter::Increment?

```
Counter::Increment() {  
    while (lock == BUSY && test_and_set(&lock))  
        ;  
    value++;  
    memory_barrier();  
    lock = FREE;  
}
```

Test (and Test) and Set Performance



Some Approaches

- Insert a delay in the spin loop
 - Helps but acquire is slow when not much contention
- Spin adaptively
 - No delay if few waiting
 - Longer delay if many waiting
 - Guess number of waiters by how long you wait
- MCS
 - Create a linked list of waiters using compareAndSwap
 - Spin on a per-processor location

Atomic CompareAndSwap

CompareAndSwap(location, oldValue, newValue)

- If `*location == oldValue`, set `*location = newValue` and return ok
- If `*location != oldValue`, return error

If two threads CompareAndSwap at the same time:

- One thread “wins”, sets `*location` to `newValue`
- One thread “loses”, sees `*location` has changed

MCS Lock

- Maintain a list of threads waiting for the lock
 - Thread at front of list holds the lock
 - MCSLock::tail is last thread in list
 - Add to tail using CompareAndSwap
- Lock handoff: set next->needToWait = FALSE
 - Next thread spins: while needToWait is TRUE

MCS Lock Implementation

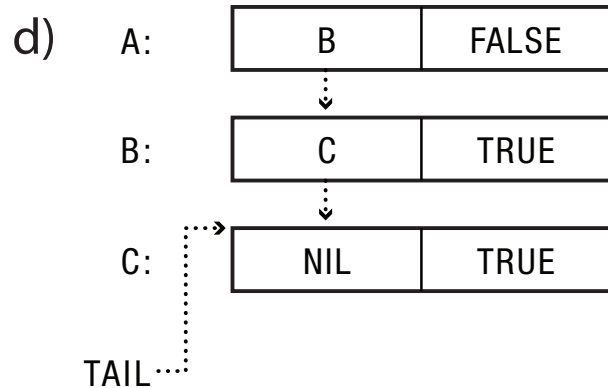
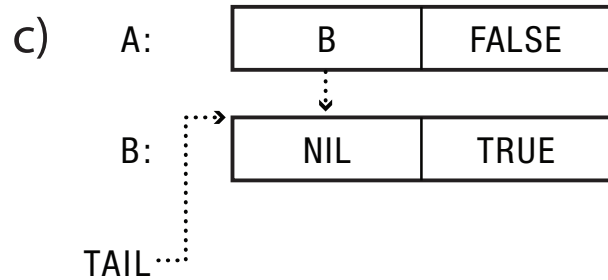
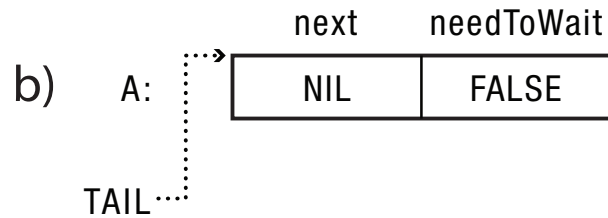
```
MCSLock::acquire() {
    myTCB->next = NULL;
    myTCB->needToWait = FALSE;
    oldTail = tail;
    while (!compareAndSwap(&tail,
                          oldTail, &myTCB)) {
        oldTail = tail;
    }
    if (oldTail != NULL) {
        myTCB->needToWait = TRUE;
        oldTail->next = myTCB;
        memory_barrier();
        while (myTCB->needToWait)
            ;
    }
}
```

```
TCB {
    TCB *next;           // next in line
    bool needToWait;
}
MCSLock {
    Queue *tail = NULL; // end of line
}

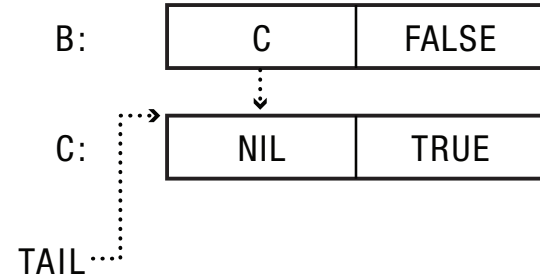
MCSLock::release() {
    if (!compareAndSwap(&tail,
                      myTCB, NULL)) {
        while (myTCB->next == NULL)
            ;
        myTCB->next->needToWait=FALSE;
    }
}
```


MCS In Operation

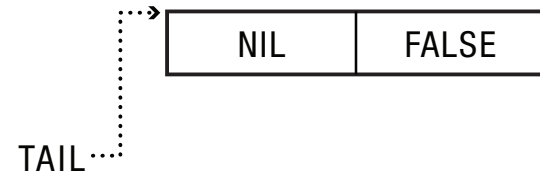
a) TAIL→ NIL



e)



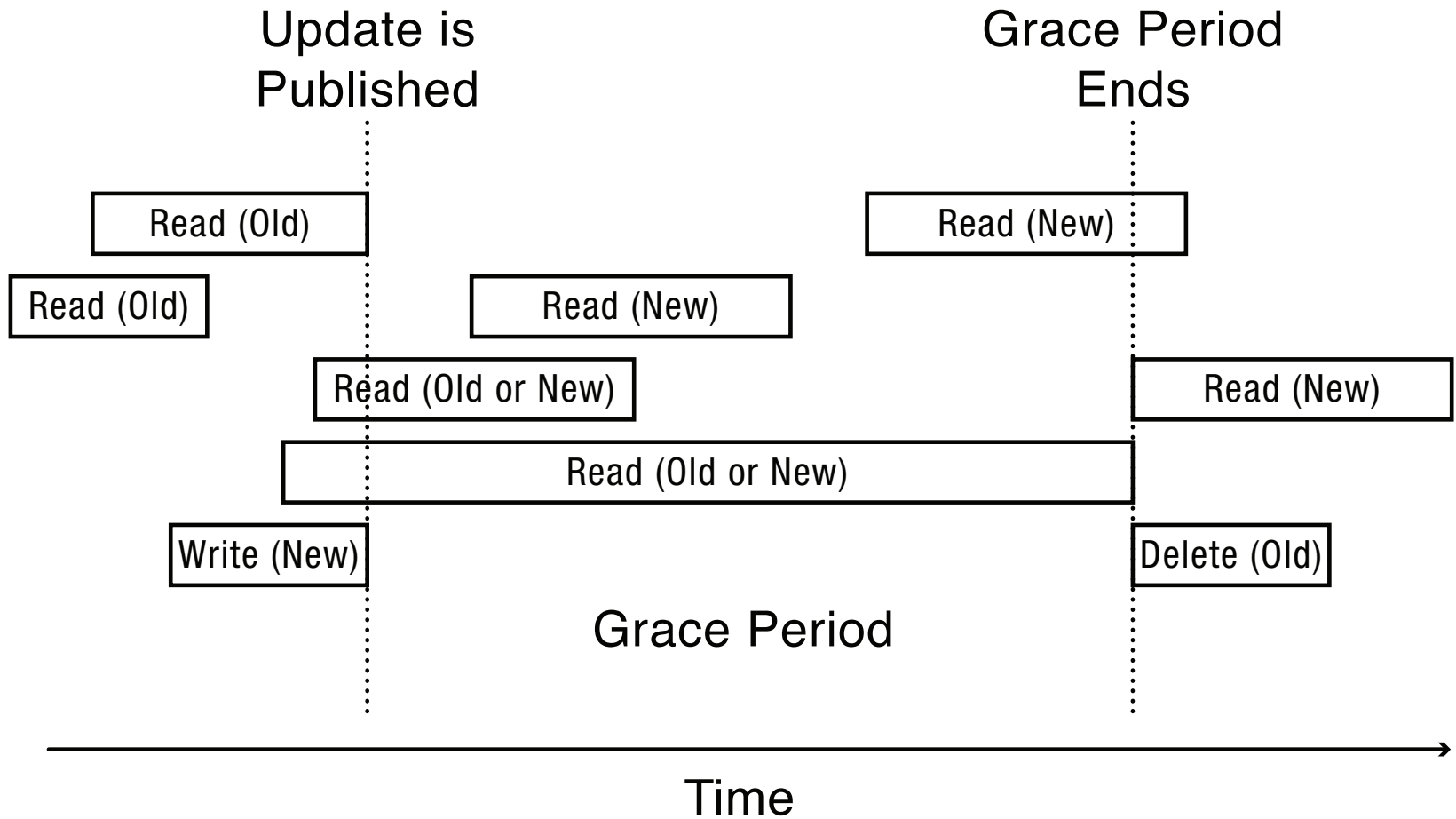
f)



Read-Copy-Update (RCU) Locks

- Goal: very fast reads to shared data
 - Reads proceed without first acquiring a lock
 - OK if write is (very) slow
- Restricted update
 - Writer computes new version of data structure
 - Publishes new version with a single atomic instruction
- Multiple concurrent versions
 - Readers in progress may see old or new version
 - New readers see new version
- Integration with thread scheduler
 - Readers in progress at previous update must complete within grace period
 - Then ok to garbage collect old version

Read-Copy-Update



Read-Copy-Update Implementation

- Readers disable interrupts on entry
 - Guarantees they complete critical section in a timely fashion
 - No read or write lock
- Writer
 - Acquire write lock
 - Compute new data structure
 - Publish new version with atomic instruction
 - Release write lock
 - Wait for time slice on each CPU
 - Only then, garbage collect old version of data structure

Lock-free Data Structures

- Data structures that can be read/modified without acquiring a lock
 - No lock contention!
 - No deadlock!
- General method using compareAndSwap
 - Create copy of data structure
 - Modify copy
 - Swap in new version iff no one else has
 - Restart if pointer has changed

Lock-Free Bounded Buffer

```
tryget() {  
    do {  
        copy = ConsistentCopy(p);  
        if (copy->front == copy->tail)  
            return NULL;  
        else {  
            item = copy->buf[copy->front % MAX];  
            copy->front++;  
        } while (compareAndSwap(&p, p, copy));  
    } while (true);  
    return item;  
}
```

Multiprocessor OSeS

- A multiprocessor OS:
 - Runs on a “tightly-coupled” (usually shared-memory) multiprocessor machine
 - Provides system-wide OS abstractions
- Multiprocessor computers were anticipated by the research community long before they became mainstream
 - Typically restricted to “big iron”
- But few commercial OSeS are designed *from the outset* for multiprocessor hardware

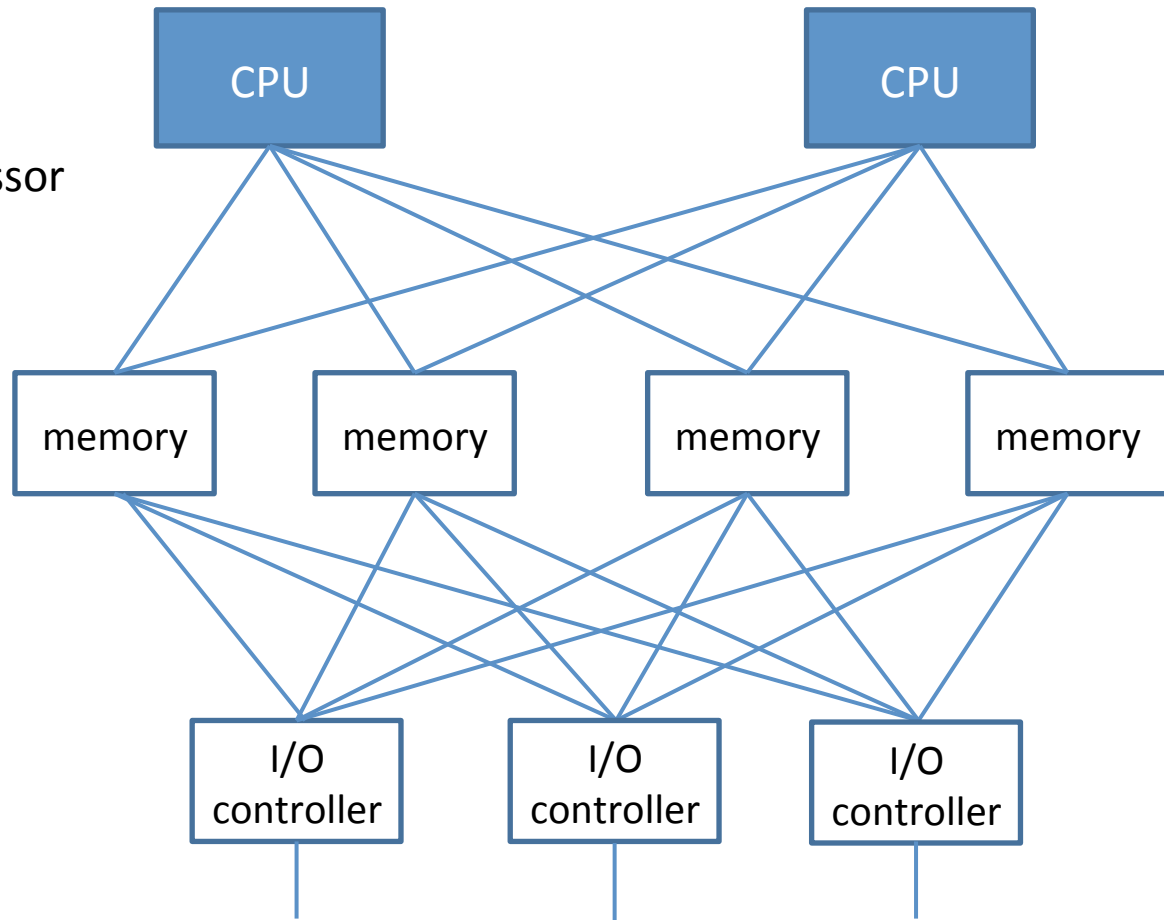
Multics

- Time-sharing operating system for a multiprocessor mainframe
- Joint project between MIT, General Electric, and Bell Labs (until 1969)
- 1965 – mid 1980s
 - Last Multics system decommissioned in 2000
- Goals: reliability, dynamic reconfiguration, etc.
- Very influential

Multics: typical configuration

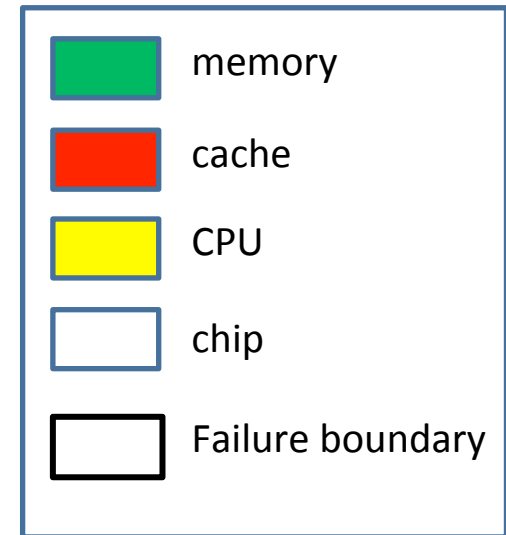
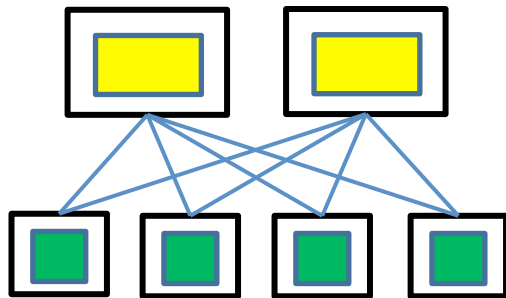
GE645 computer
Symmetric multiprocessor

Communication was by using “mailboxes” in the memory modules and corresponding interrupts (asynchronous).



to remote terminals, magnetic tape, disc, console reader punch etc

Multics on GE645

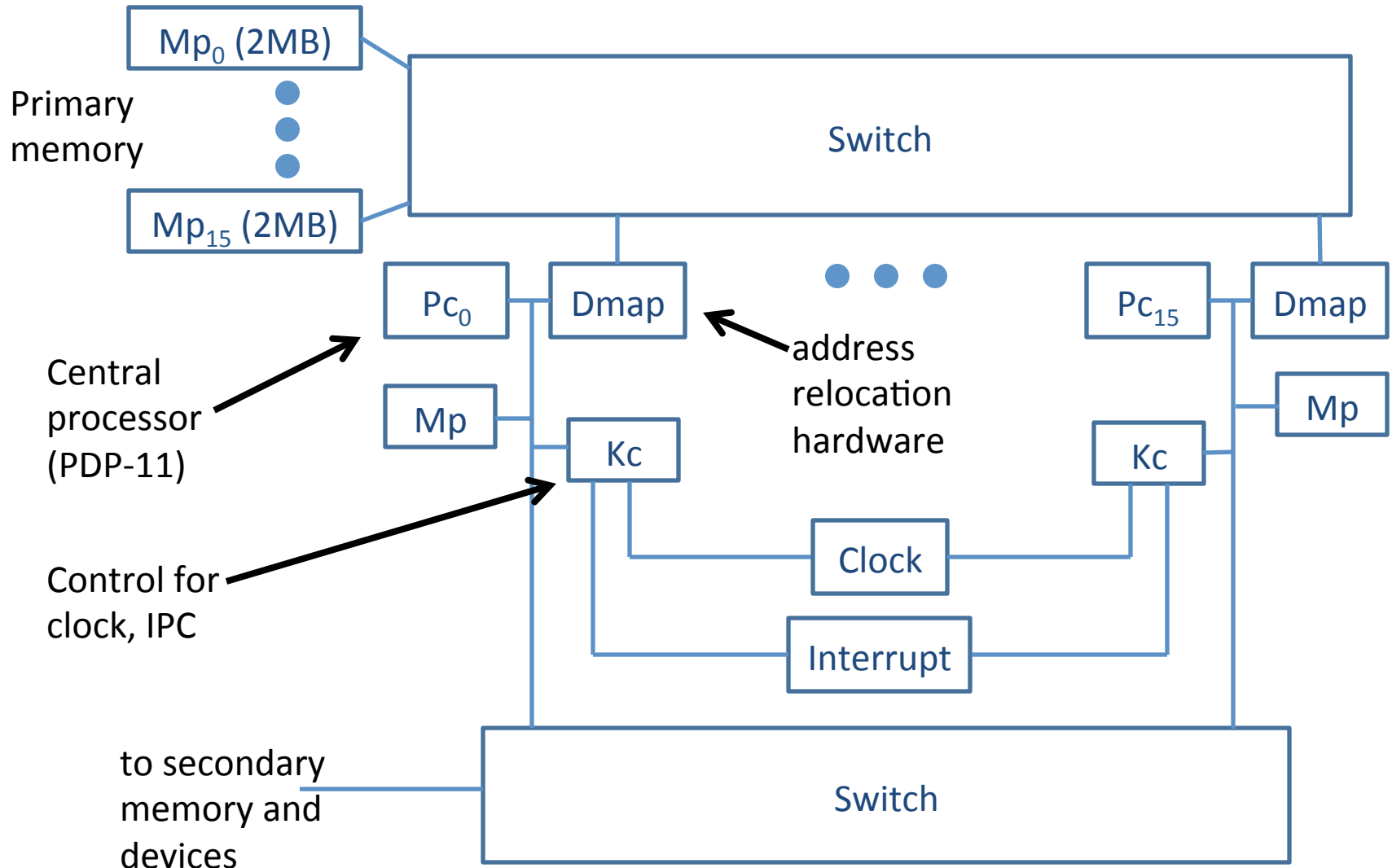


- Reliable interconnect
- No caches
- Single level of shared memory
 - Uniform memory access (UMA)
- Online reconfiguration of the hardware
 - Regularly partitioned into 2 separate systems for testing and development and then recombined

Hydra

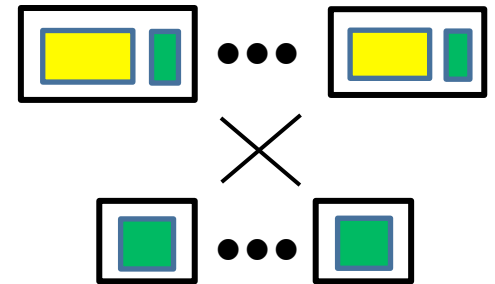
- Early 1970s, CMU
- Multiprocessor operating system for C.mmp (Carnegie-Mellon Multi-Mini-Processor)
 - Up to 16 PDP-11 processors
 - Up to 32MB memory
- Design goals:
 - Effective utilization of hardware resources
 - Base for further research into OSes and runtimes for multiprocessor systems

C.mmp multiprocessor



Hydra (cont)

- Limited hardware
 - No hardware messaging, send IPIs
 - No caches
 - 8k private memory on processors
 - No virtual memory support
- Crossbar switch to access memory banks
 - Uniform memory access ($\sim 1\mu\text{s}$ if no contention)
 - But had to worry about contention
- Not scalable



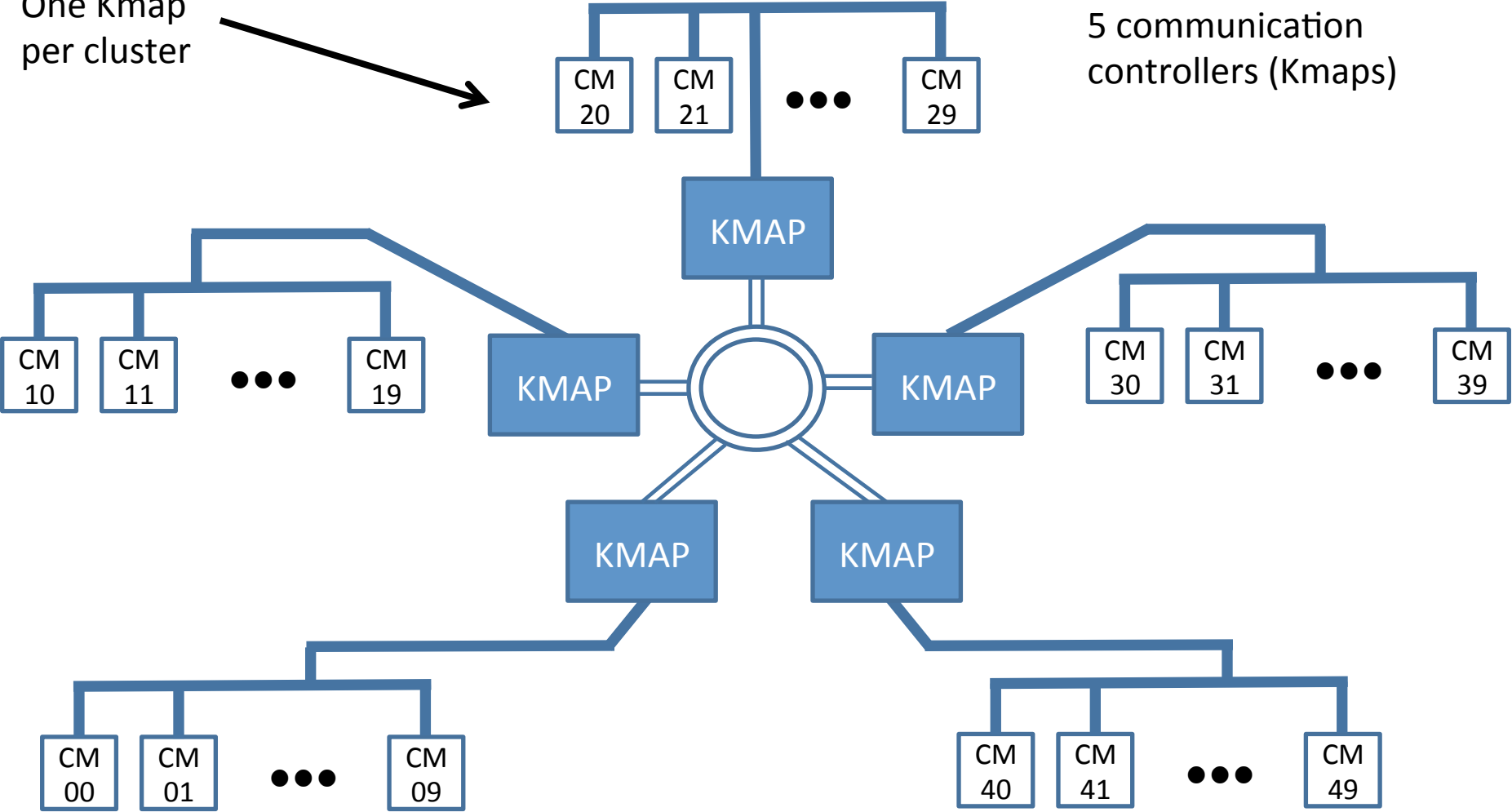
Cm*

- Late 1970s, CMU
- Improved scalability over C.mmp
 - 50 processors, 3MB shared memory
 - Each processor is a DEC LSI-11 processor with bus, local memory and peripherals
 - Set of clusters (up to 14 processors per cluster) connected by a bus
 - Memory can be accessed locally, within the cluster and at another cluster (NUMA)
 - No cache
- 2 Oses developed: StarOS and Medusa

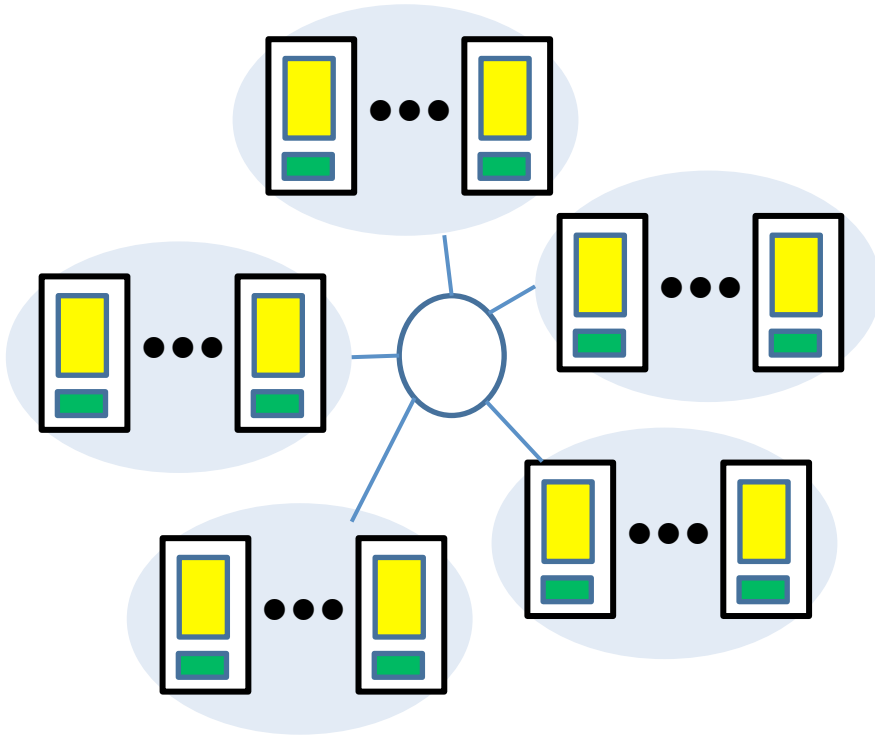
Cm*

One Kmap
per cluster

50 compute
modules (CMs)
5 communication
controllers (Kmaps)



Cm*



- NUMA
- Reliable message-passing
- No caches
- Contention and latency big issues when accessing remote memory
 - Sharing is expensive
 - Concurrent processes run better if independent

Medusa

- OS for Cm*, 1977-1980
- Goal: reflect the underlying distributed architecture of the hardware
- Single copy of the OS impractical
 - Huge difference in local vs non-local memory access times
 - 3.5us local vs 24us cross-cluster
- Complete replication of the OS impractical
 - Small local memories (64 or 128KB)
 - Typical OS size 40-60KB

Medusa (cont)

- Replicated kernel on each processor
 - Interrupts, context switching
- Other OS functions divided into disjoint utilities
 - Utility code always executed on local processor
 - Utility functions invoked (asynchronously) by sending messages on pipes
- Utilities:
 - Memory manager
 - File system
 - Task force manager
 - All processes are task forces, consisting of multiple activities that are co-scheduled across multiple processors
 - Exception reporter
 - Debugger/tracer

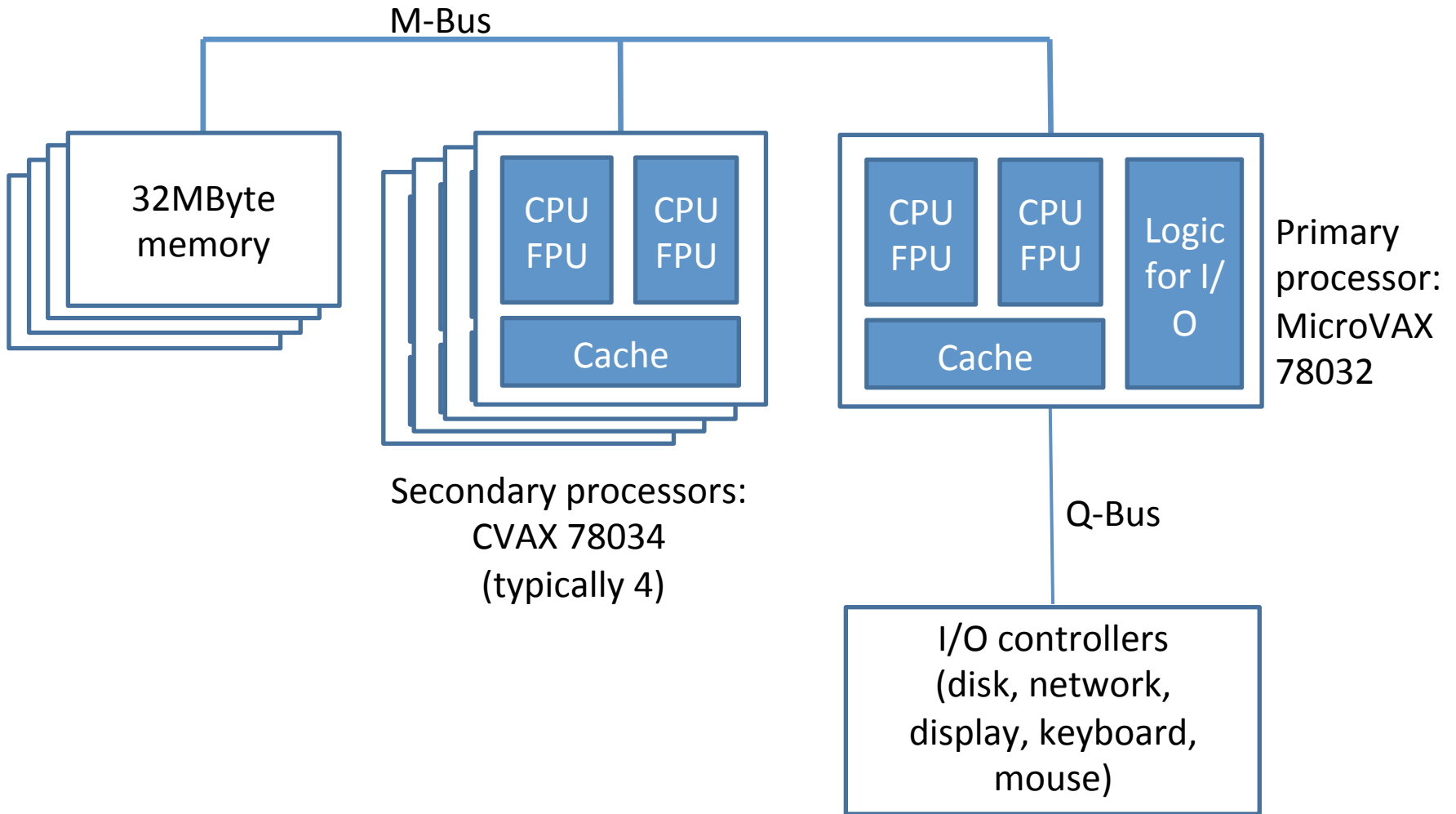
Medusa (cont)

- Had to be careful about deadlock, eg file open:
 - File manager must request storage for file control block from memory manager
 - If swapping between primary and secondary memory is required, then memory manager must request I/O transfer from file system
 - Deadlock
- Used coscheduling of activities in a task force to avoid livelock

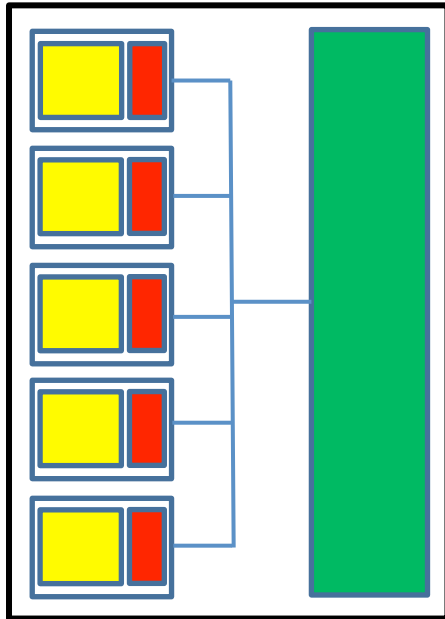
Firefly

- Shared-memory, multiprocessor, personal workstation
 - Developed at DEC SRC, 1985-1987
- Requirements:
 - Research platform (powerful, multiprocessor)
 - Built in a short amount of time (off-the-shelf components as much as possible)
 - Suitable for an office (not too large, loud or power-hungry)
 - Ease of programming (hardware cache coherence)

Firefly (version 2)



Firefly



- SMP
- Reliable interconnect
- Hardware support for cache coherence
- Bus contention an important issue
 - Analysis found that adding processors improved performance up to about 9 processors

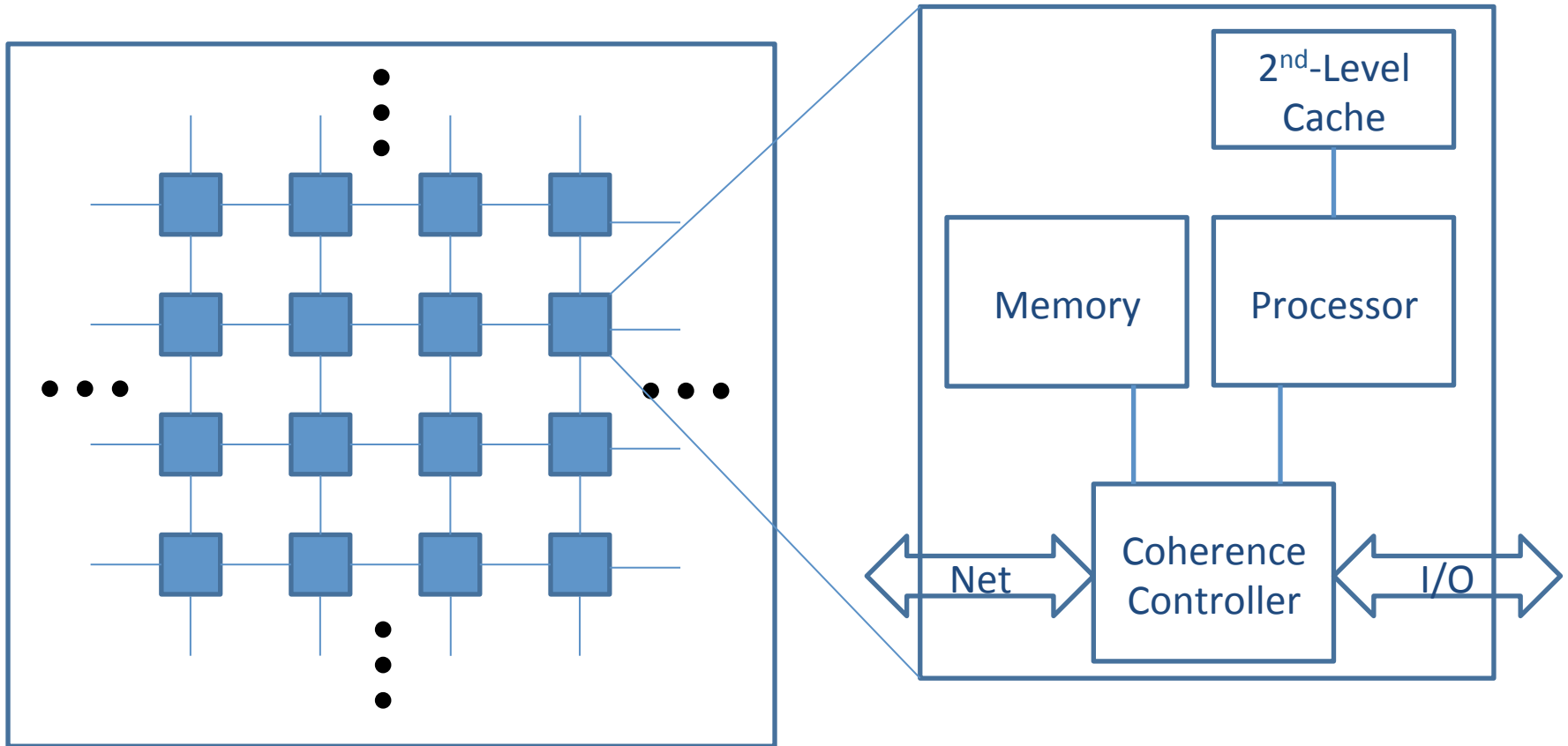
Topaz

- Software system for the Firefly
- Multiple threads of control in a shared address space
- Binary emulation of Ultrix system call interface
- Uniform IPC communication mechanism
 - Same machine and between machines
- System kernel called the Nub
 - Virtual memory
 - Scheduler
 - Device drivers
- Rest of the OS ran in user-mode
- All software multithreaded
 - Executed simultaneously on multiple processors

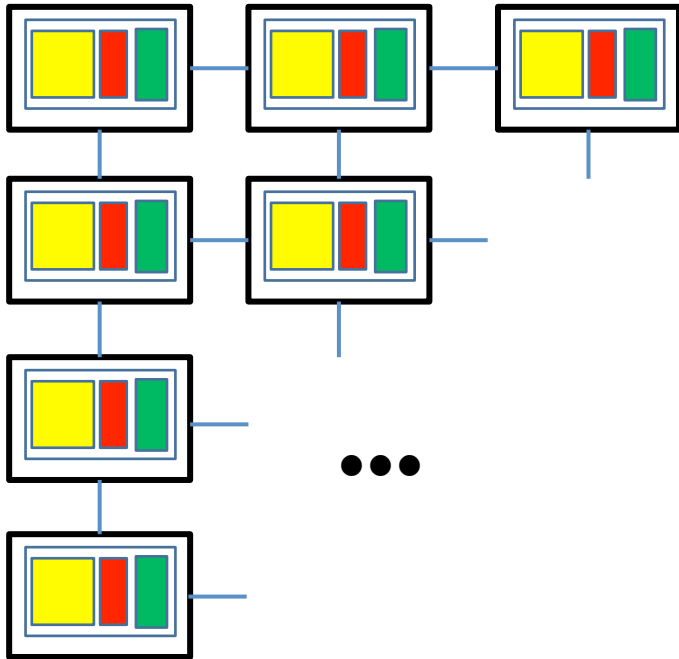
Hive

- Stanford, early 1990s
- Targeted at the Stanford FLASH multiprocessor
 - Large-scale ccNUMA
- Main goal was fault containment
 - Contain hardware and software failure to the smallest possible set of resources
- Second goal was scalability through limited sharing of kernel resources

Stanford FLASH architecture



Stanford FLASH



- Reliable message-passing
 - Nodes can fail independently
- Designed to scale to 1000's of nodes
- Non-Uniform Memory Access
 - Latency increases with distance
- Hardware cache coherence

Hive (cont)

- Each “cell” (ie kernel) independently manages a small group of processors, plus memory and I/O devices
 - Controls a portion of the global address space
- Cells communicate mostly by IPC
 - But for performance can read and write each other’s memory directly
- Resource management in user-space (by Wax)
 - Global allocation policies for memory and processors
 - Threads on different cells synchronize via shared memory

Hive: failure detection and fault containment

- Failure detection mechanisms
 - RPC timeouts
 - Keep-alive increments on shared memory locations
 - Consistency checks on reading remote cell data structures
 - Hardware errors, eg bus errors
- Fault containment
 - Hardware firewall (an ACL per page of memory) prevents wild writes
 - Preemptive discard of all pages belonging to a failed process
 - Aggressive failure detection
 - Distributed agreement algorithm confirms cell has failed and reboot it

Disco

- Context: ca. 1995, large ccNUMA multiprocessors appearing
- Problem: scaling OSes to run efficiently on these was hard
 - Extensive modification of OS required
 - Complexity of OS makes this expensive
- Idea: implement a **scalable VMM**, run multiple OS instances
- VMM has most of the features of a scalable OS, e.g.:
 - NUMA-aware allocator
 - Page replication, remapping, etc.
- VMM substantially simpler/cheaper to implement
- Run multiple (smaller) OS images, for different applications

Disco Contributions

- First project to revive an old idea: virtualization
 - New way to work around shortcomings of commodity Oses
- Another interesting idea:
programming a single machine as a distributed system
 - Example: parallel make, two configurations:
 1. Run an 8-CPU IRIX instance
 2. Run 8 IRIX VMs on Disco, one with an NFS server
 - Speedup for case 2, despite VM and vNIC overheads

K42

- OS for cache-coherent NUMA systems
- IBM Research, 1997–2006ish
- Successor of Tornado and Hurricane systems (University of Toronto)
- Supports Linux API/ABI
- Aims: high locality, scalability
- Heavily object-oriented
 - Resources managed by set of object instances



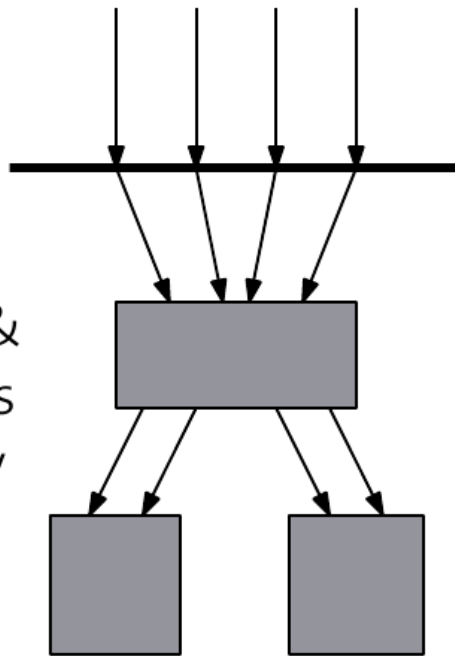
Why use OO in an OS?

Traditional System

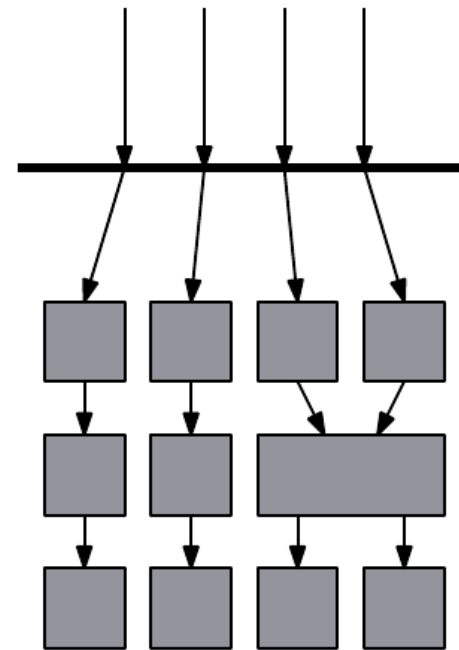
OO Decomposed System

User-level requests

System paths & data structures used to satisfy requests



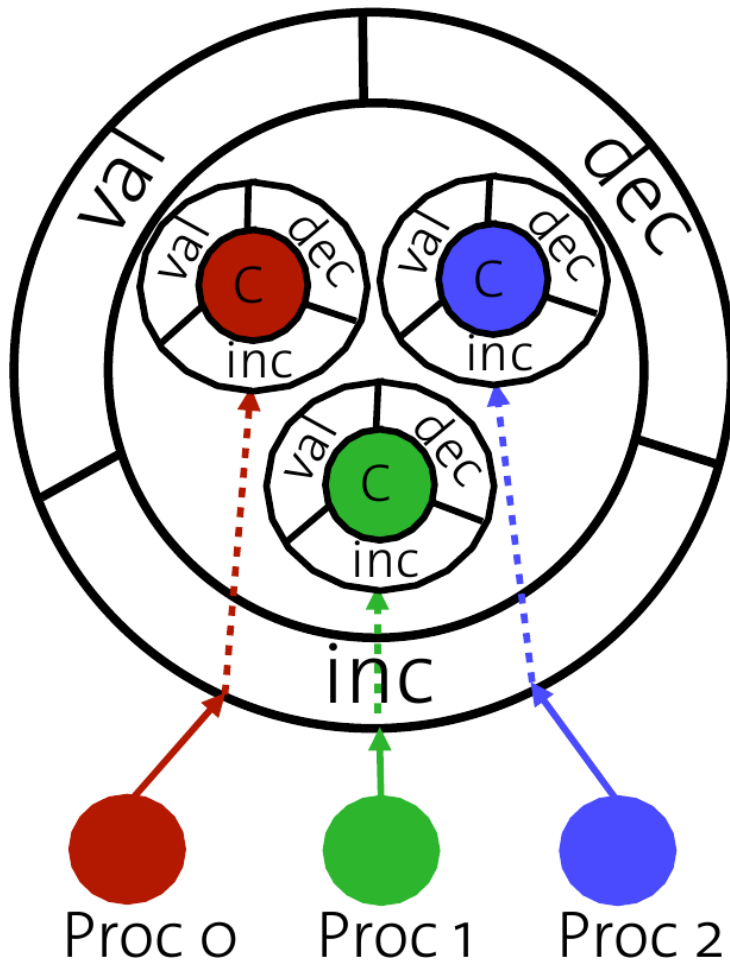
- much sharing



- much **less** sharing
- better performance

Clustered Objects

Example: shared counter

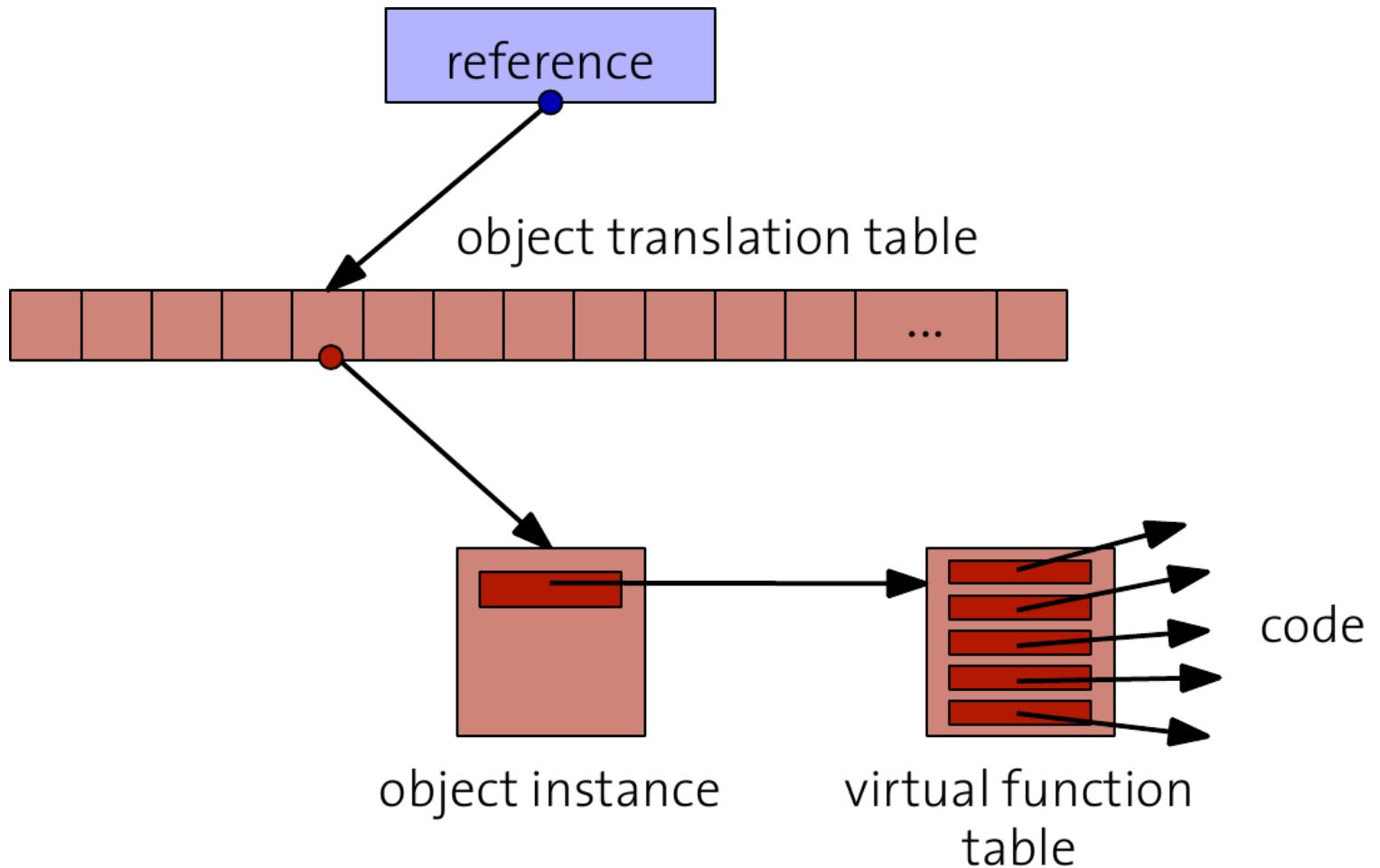


- Object internally decomposed into processor-local representatives
 - Same reference on any processor
 - Object system routes invocation to local representative
- ⇒ Choice of sharing and locking strategy local to each object
- In example, *inc* and *dec* are local; only *val* needs to communicate

Clustered objects

Implementation using processor-local

object translation table:



Challenges with clustered objects

- Degree of clustering (number of replicas, partitioned vs replicated) depends on how the object is used
- State maintained by the object replicas must be kept consistent
- Determining global state can be expensive
 - Eg choosing the next highest priority thread for scheduling when priorities are distributed across many user-level scheduler objects

K42 Principles/Lessons

- Focus on **locality** in addition to concurrency, to achieve scalability
- Distributed component model enables consistent construction of locality-tuned objects
- Support distribution within each object:
 - eases complexity
 - abstraction permits controlled/manageable introduction of localized data structures

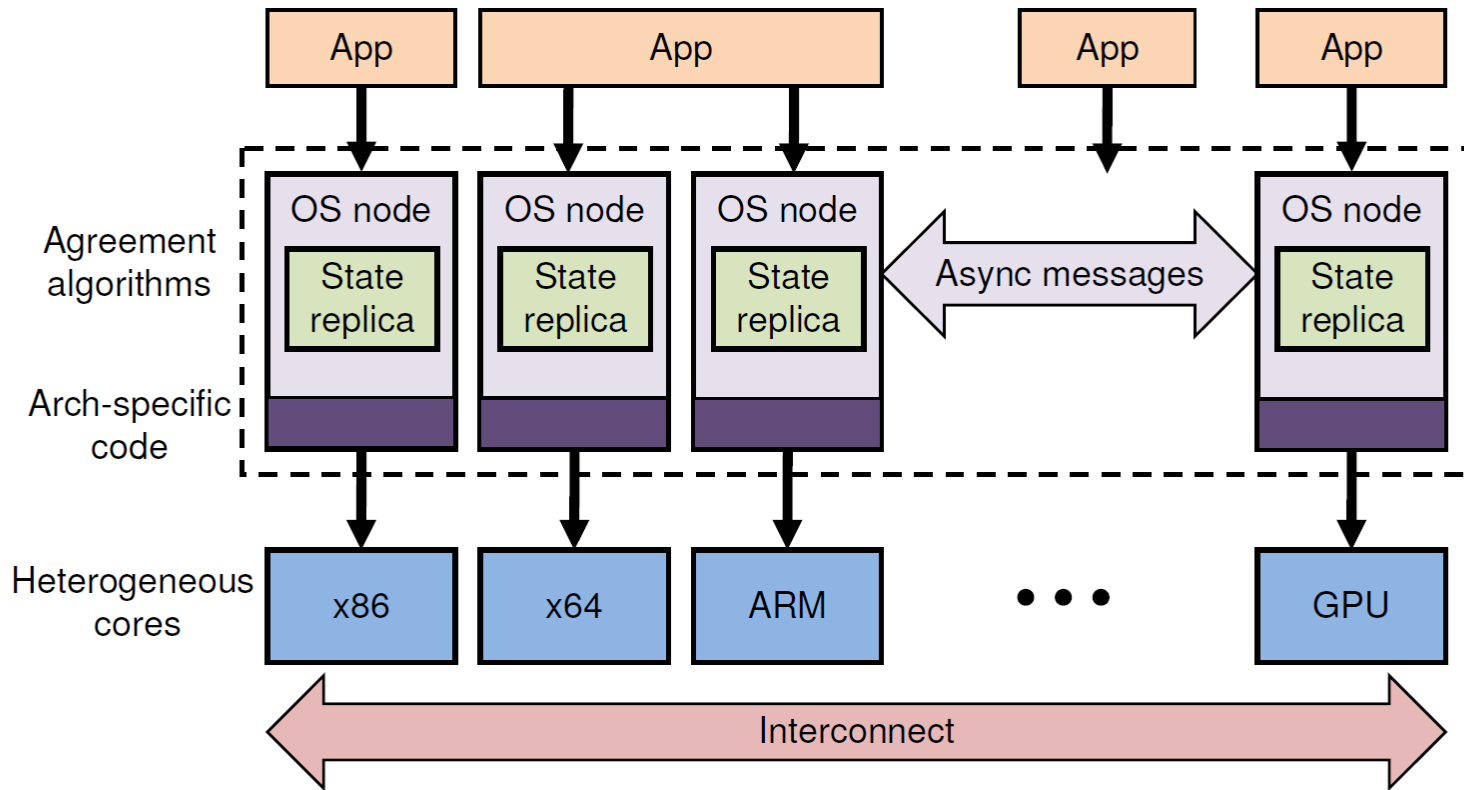
Barrelfish:

The OS as Distributed System

- 2007-today, ETH Zurich
- OS for “multicore” systems
- Goals: Scalability, agility, heterogeneity
 - OS can be reconfigured for each new machine
- No shared state
- Message passing
- Software consistency mechanisms



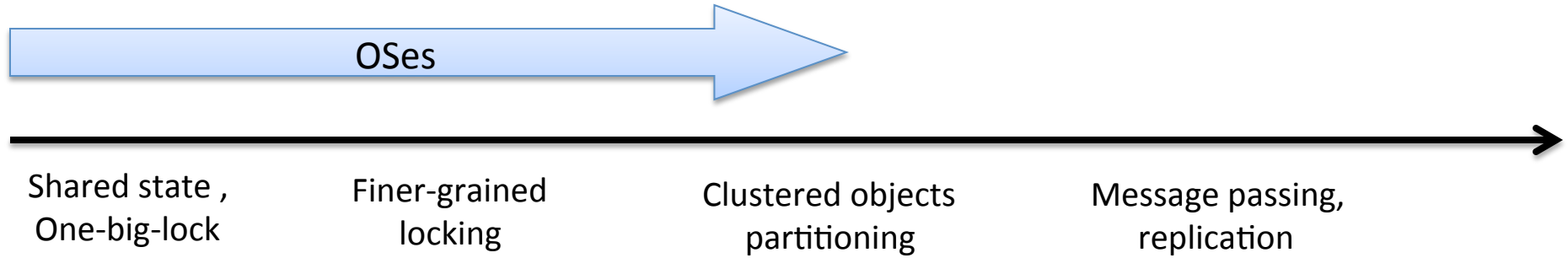
Barrelfish Architecture



Other recent multicore OSes

- Tessellation
 - Berkeley, 2009
 - Space-time partitioning for performance isolation
- fos (factored OS)
 - MIT, 2009
 - Space instead of time sharing, no distinction between cluster and on-chip
- Akaros
 - Berkeley, 2011
 - “peer through layers of virtualization”

Clear trend....



- Finer-grained locking of shared memory
- Replication as an optimization of shared memory

Further reading

- Multics: www.multicians.org
- “C.mmp: a multi-mini-processor”, W. Wulf and C.G. Bell, Fall Joint Computer Conference, Dec 1972
- “HYDRA: The kernel of a multiprocessor operating system”, W. Wulf et al, Comm. ACM, 17(6) , June 1974
- “Overview of the Hydra Operating System Development”, W. Wulf et al, 5th SOSP, Nov 1975
- “Policy/Mechanism Separation in Hydra”, R. Levin et al, 5th SOSP, Nov 1975
- “Medusa: An Experiment in Distributed Operating System Structure”, John K. Ousterhout et al, CACM, 23(2), Feb 1980
- “Firefly: a multiprocessor workstation”, Chuck Thacker and Lawrence Stewart, Computer Architecture News, 15(5), 1987
- “The duality of memory and communication in the implementation of a multiprocessor operating system”, Michael Young et al, 11th SOSP, Nov 1987 [*Mach*]
- Mach: <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html>
- “The Stanford FLASH Multiprocessor”, J Kuskin et al, ISCA, 1994
- “Hive: Fault Containment for Shared-Memory Multiprocessors”, J.Chapin et al, 15th SOSP, Dec 1995
- “K42: Building a Complete Operating System”, 1st EuroSys, April, 2006
- “Tornado: Maximising Locality and Concurrency in a Shared Memory Multiprocessor Operating System”, Gamsa et al, OSDI, Feb 1999
- K42: http://domino.research.ibm.com/comm/research_projects.nsf/pages/k42.index.html