cs6465: Advanced Operating System Implementation

Lecture: Synchronization

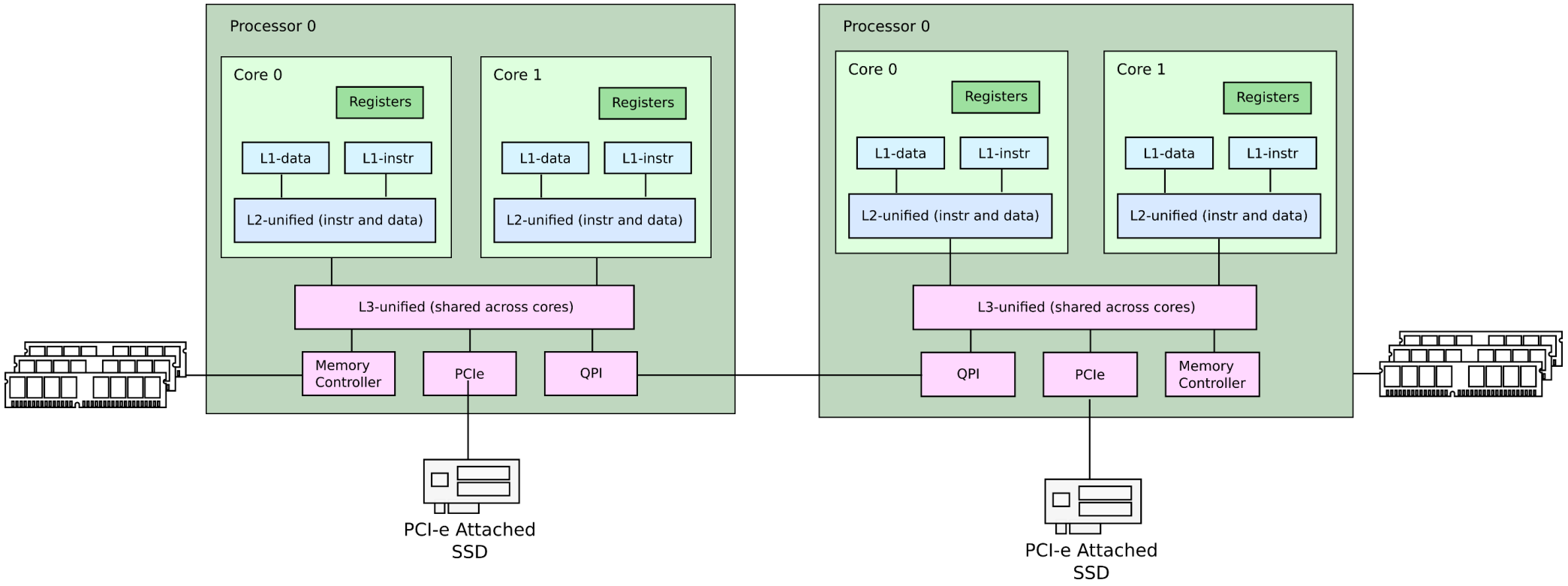
Anton Burtsev

September, 2024

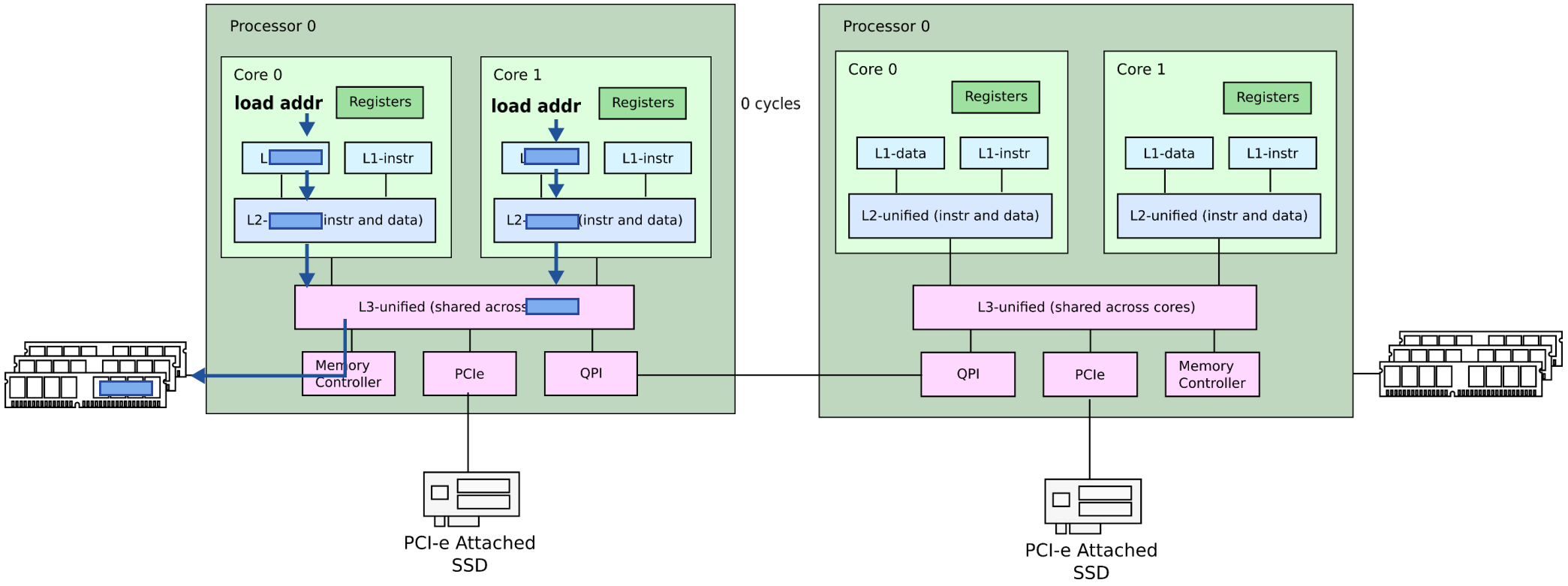
How do CPUs access memory?

Intel Memory Hierarchy

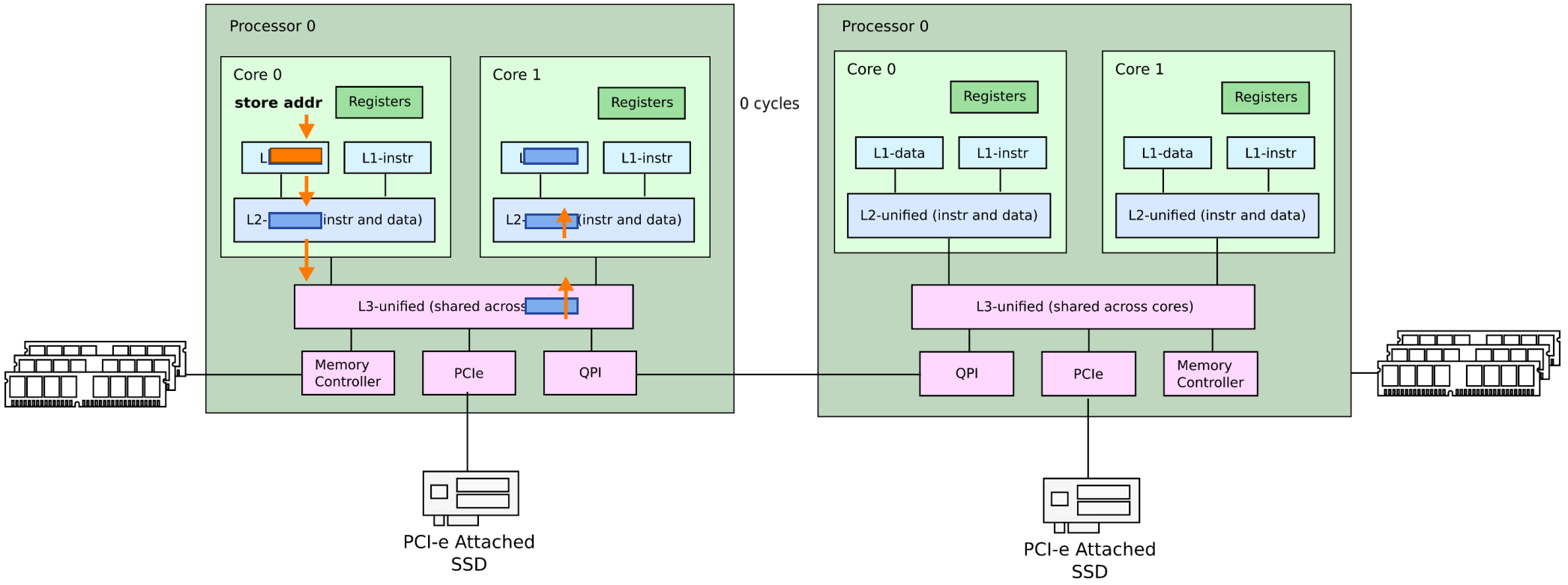
Processors, cores, memory and PCIe



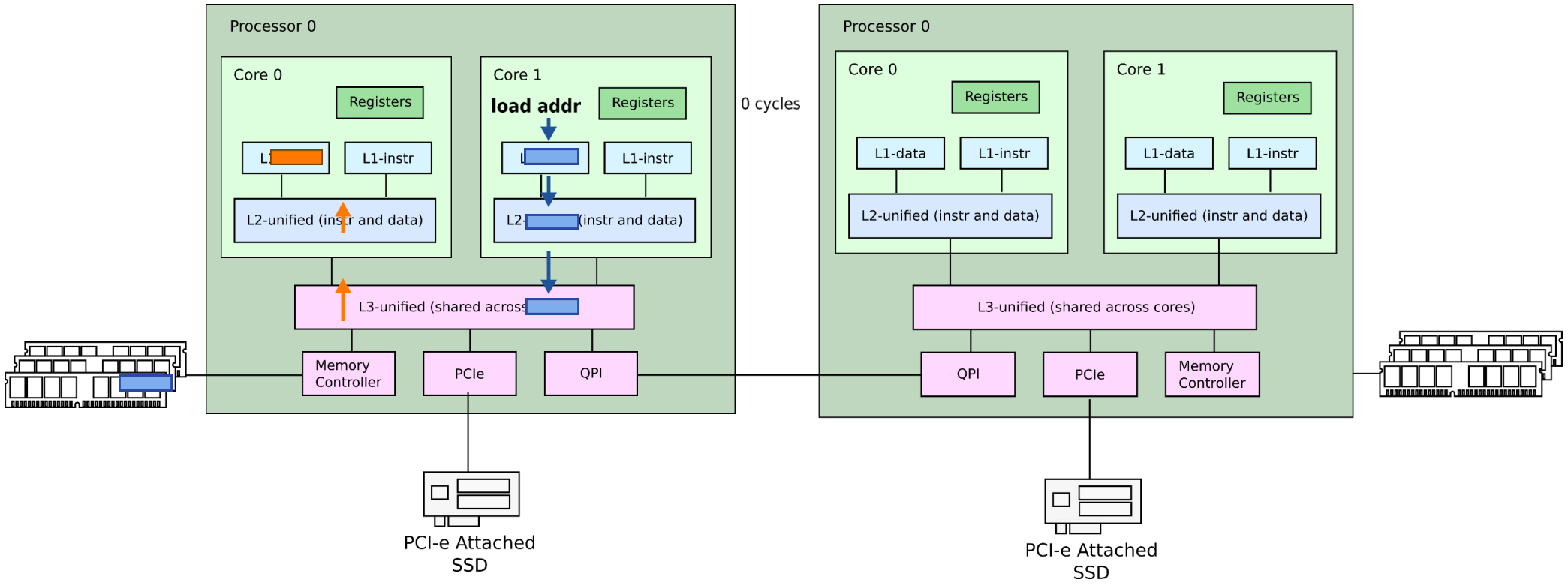
Caches (load)


Processor 0
Core 0
load addr
Registers
L1-instr
L2-
instr and data)
Core 1
load addr
Registers
L1-instr
L2-
(instr and data)
L3-unified (shared across
Memory Controller
PCIe
QPI
0 cycles
Processor 0
Core 0
Registers
L1-data
L1-instr
L2-unified (instr and data)
Core 1
Registers
L1-data
L1-instr
L2-unified (instr and data)
L3-unified (shared across cores)
QPI
PCIe
Memory Controller
PCI-e Attached SSD
PCI-e Attached SSD

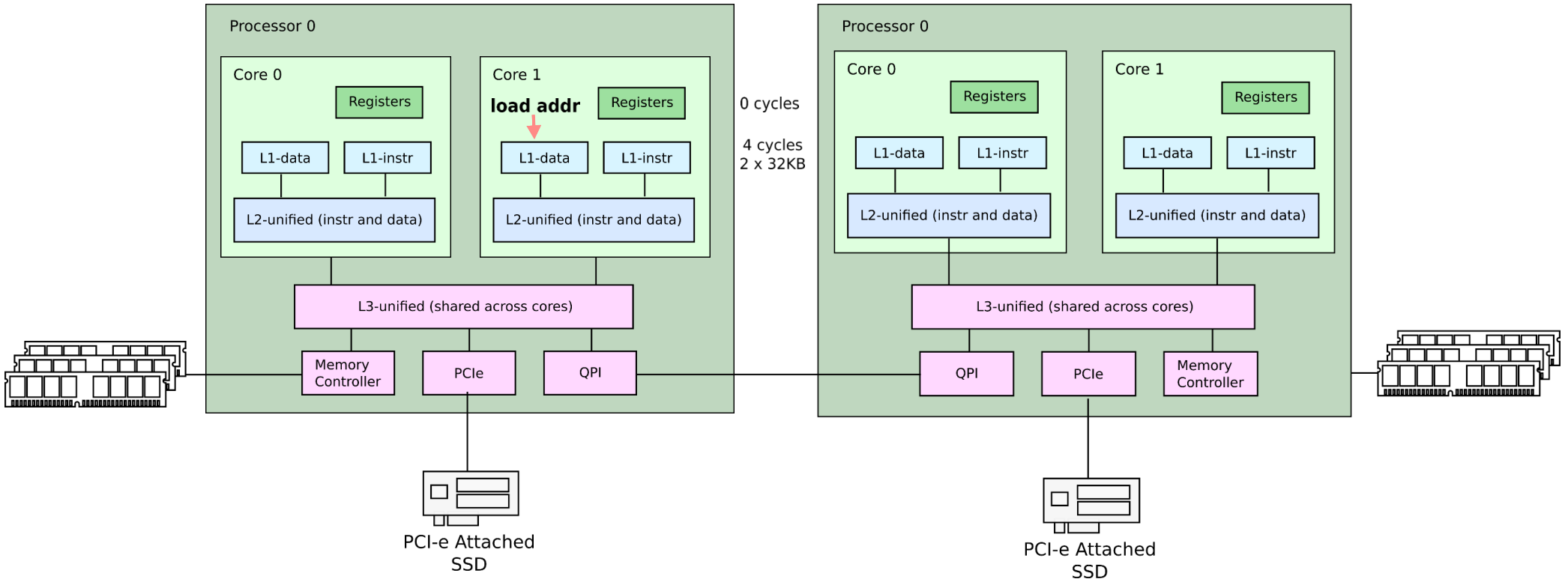
Cache-coherence (store)



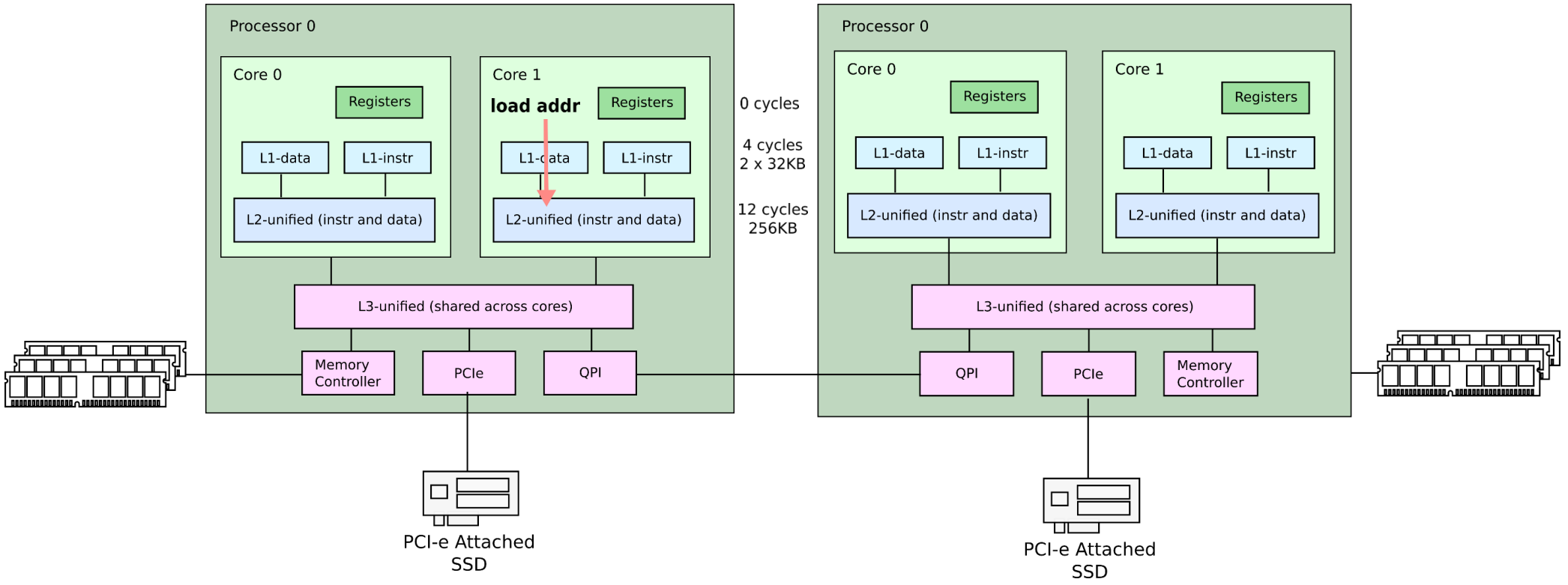
Cache-coherence (load of modified)



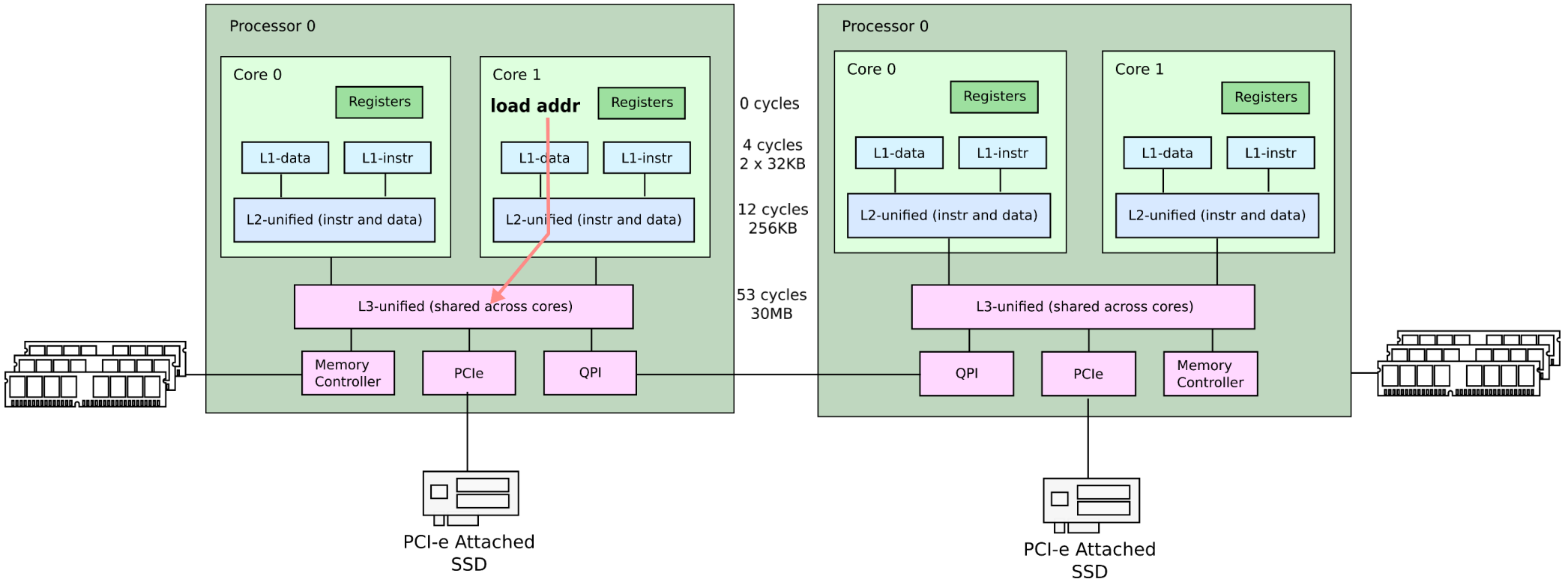
Latencies: load from local L1



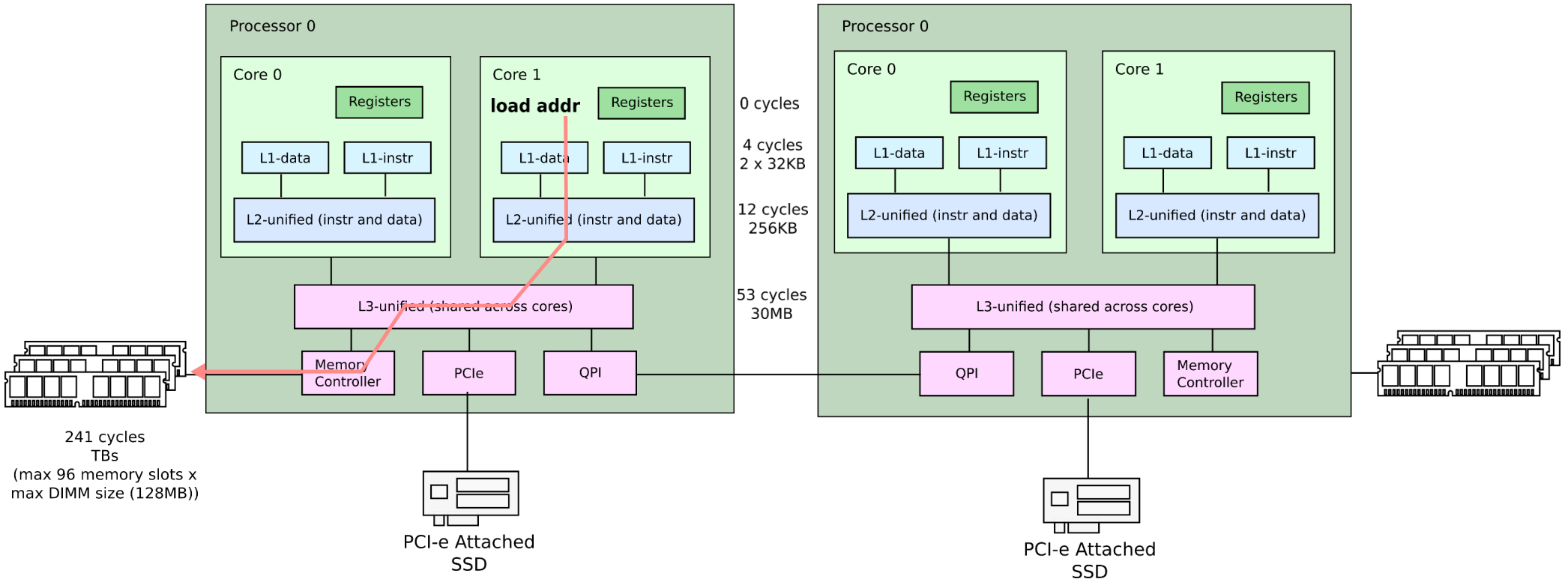
Latencies: load from local L2



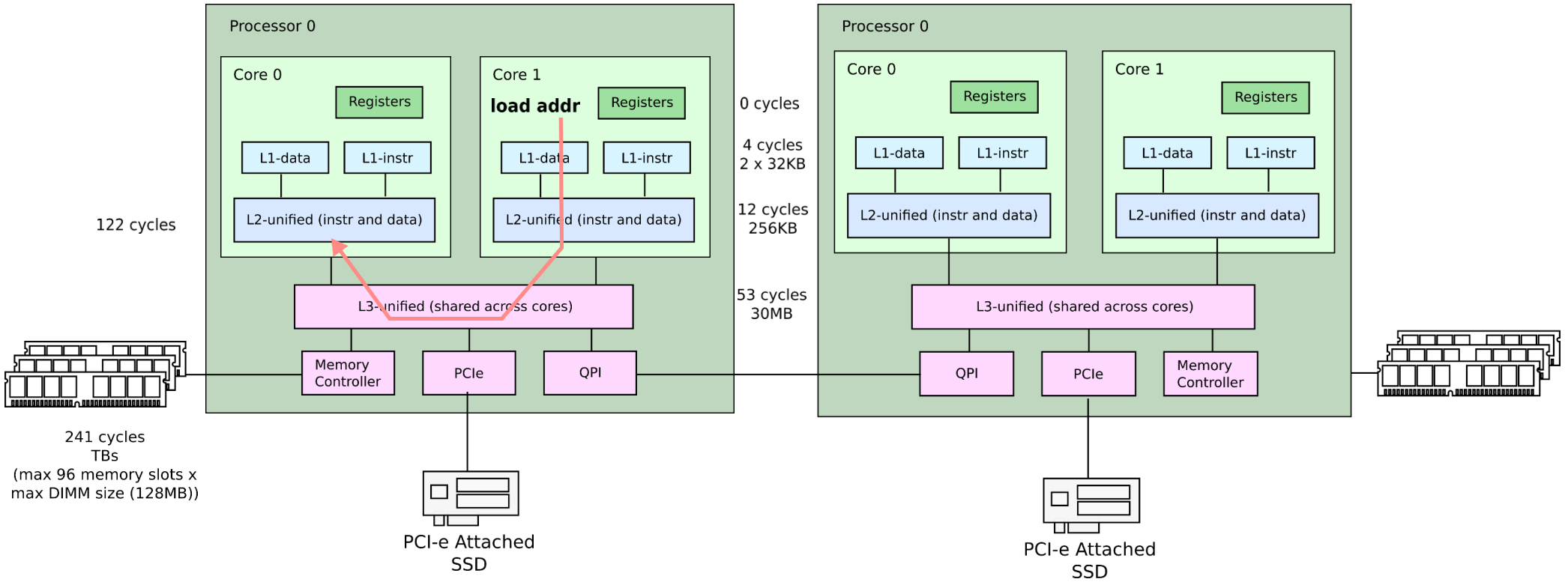
Latencies: load from local L3



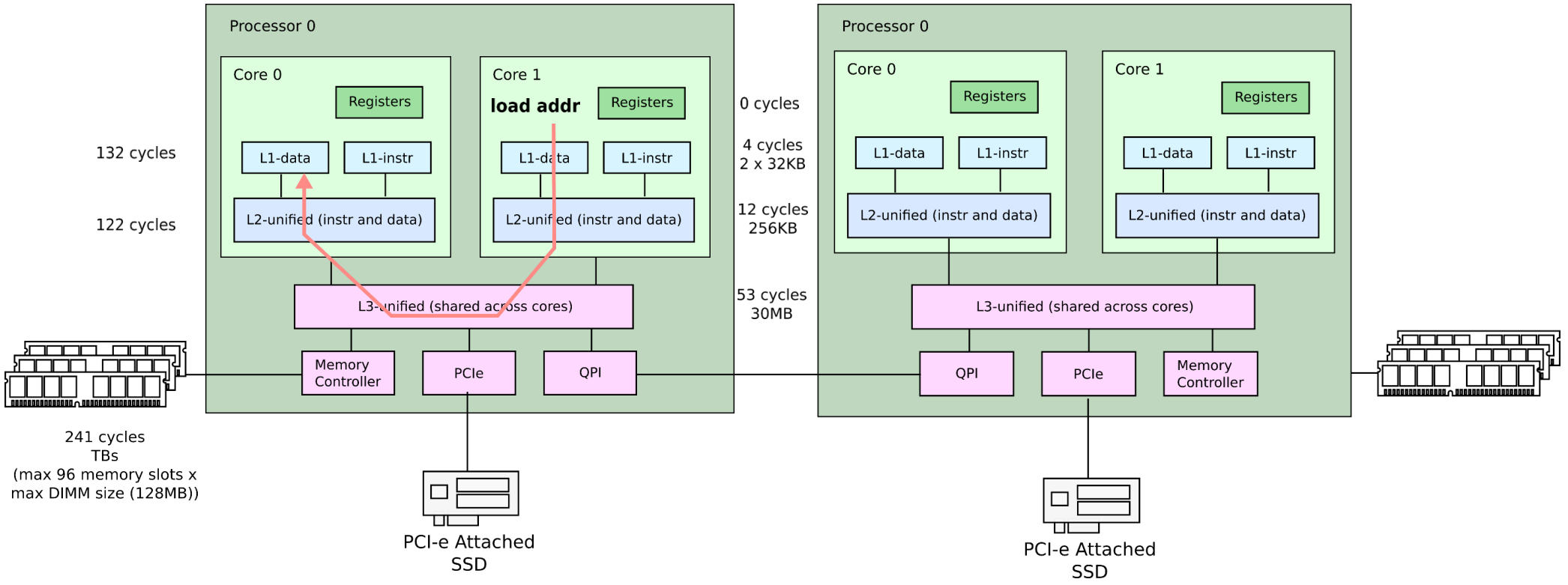
Latencies: load from local memory



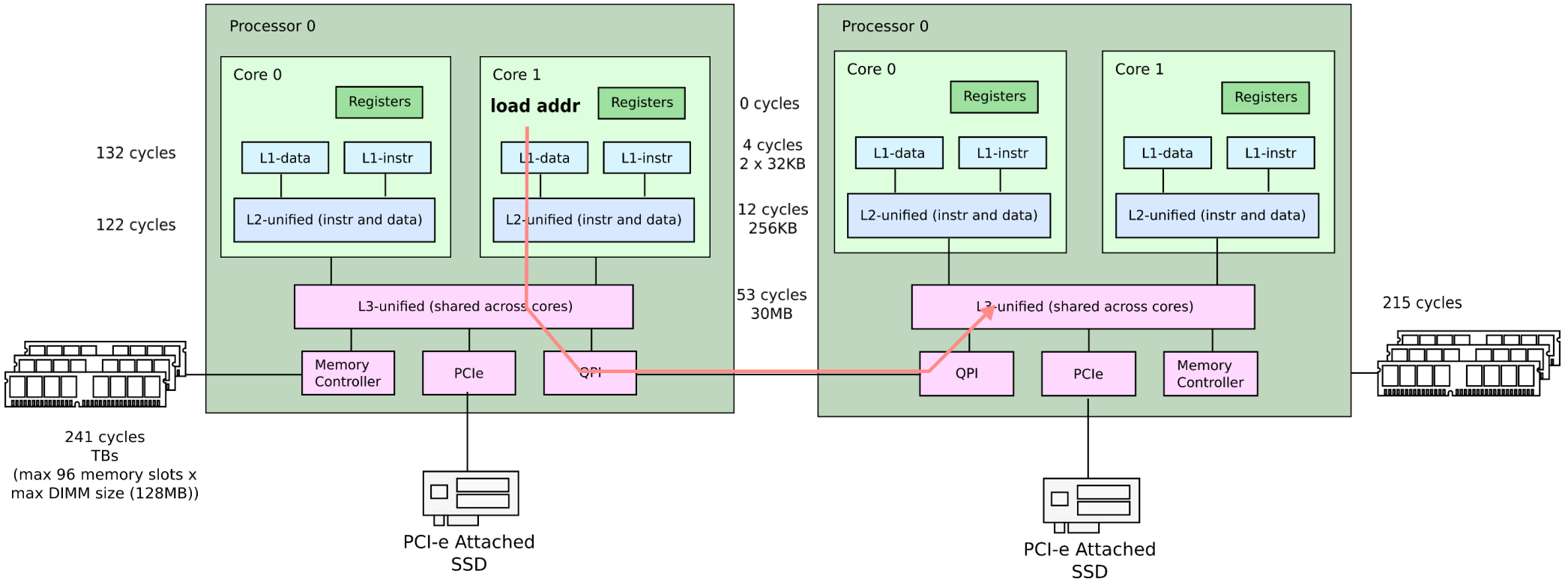
Latencies: load from same die core's L2



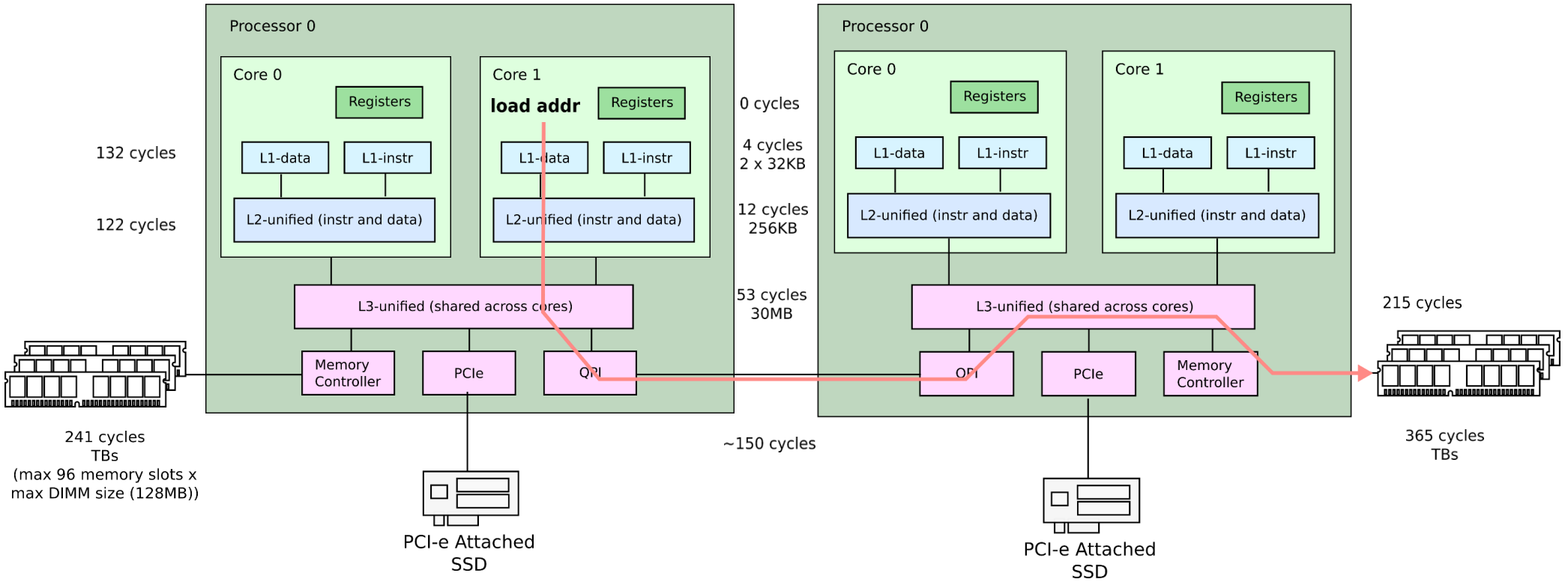
Latencies: load from same die core's L1



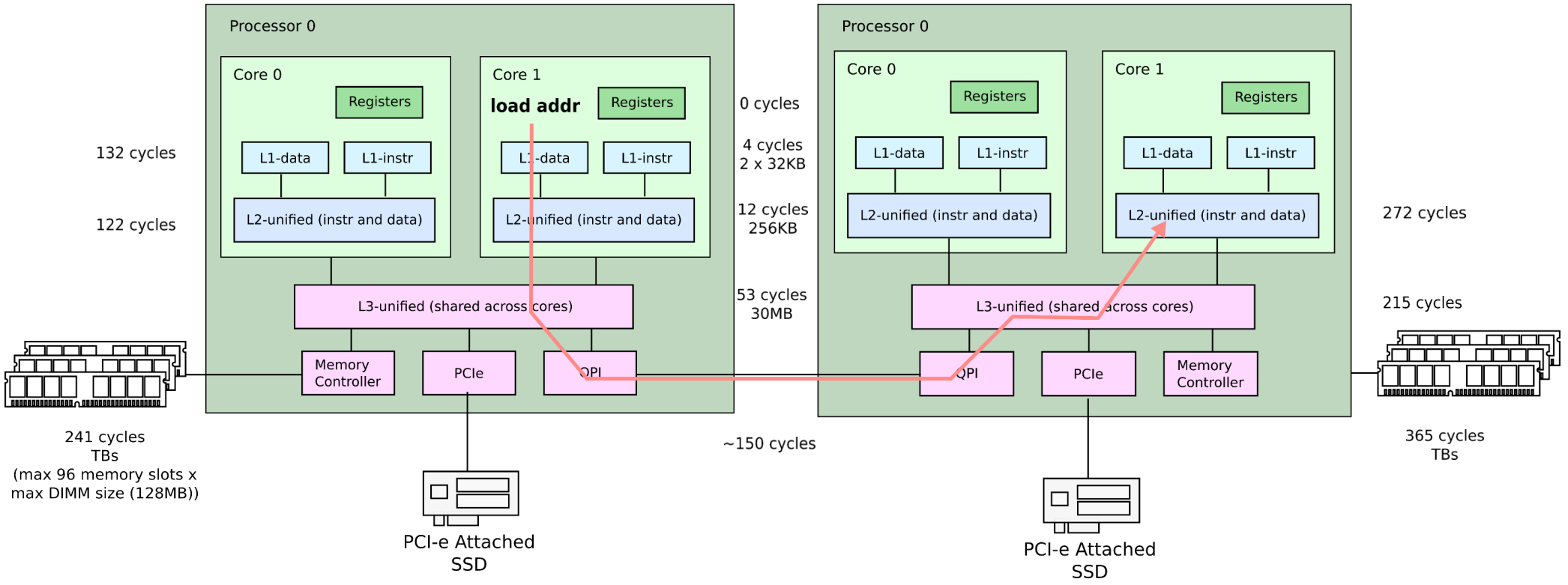
Latencies: load from remote L3



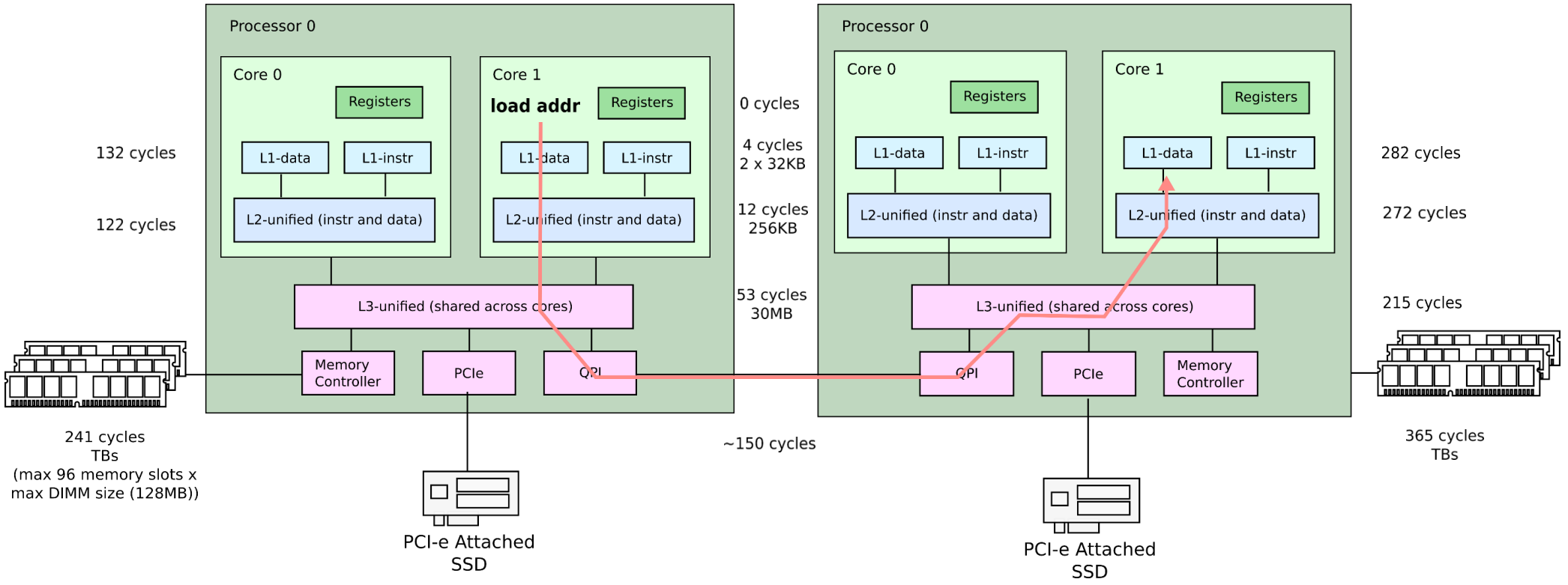
Latencies: load from remote memory



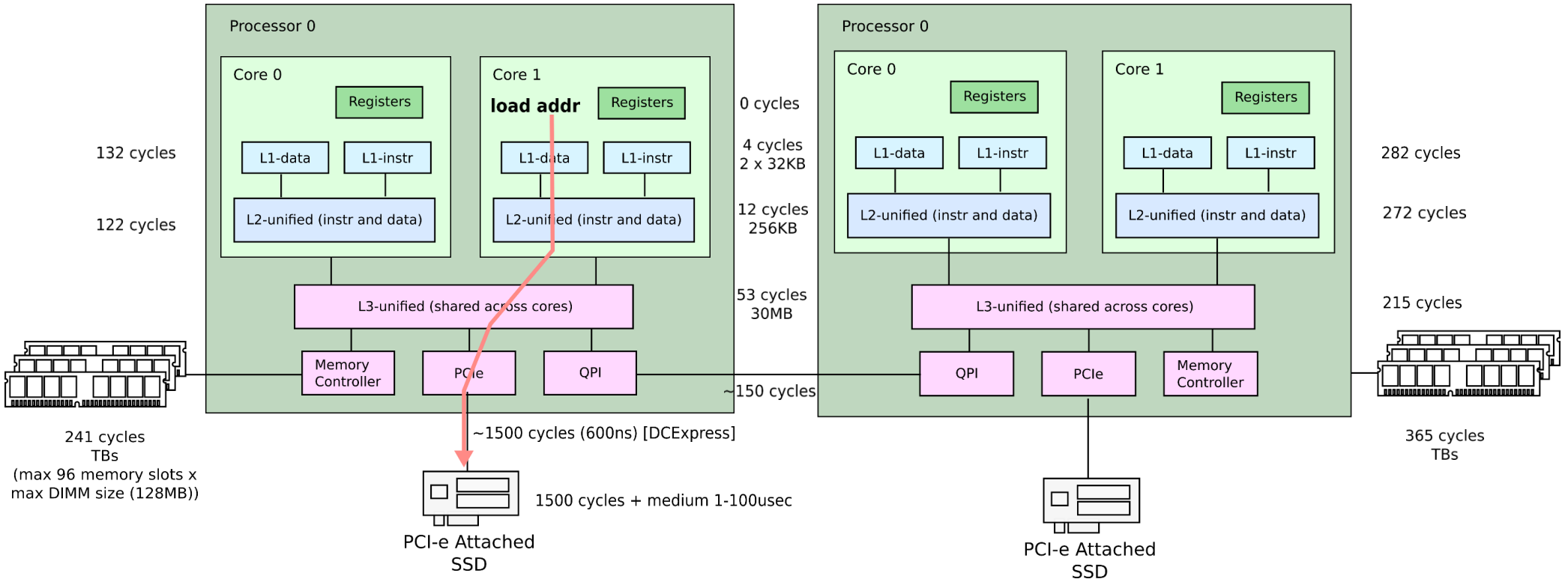
Latencies: load from remote L2



Latencies: load from remote L2



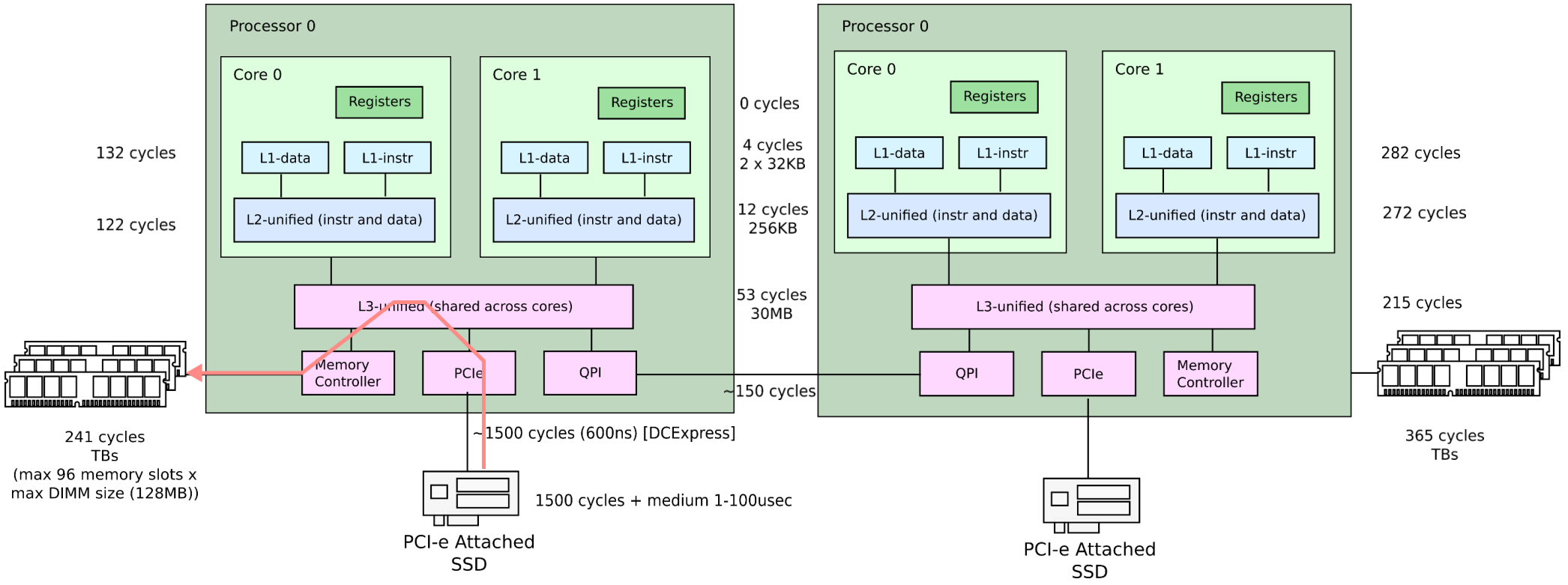
Latencies: PCIe round-trip



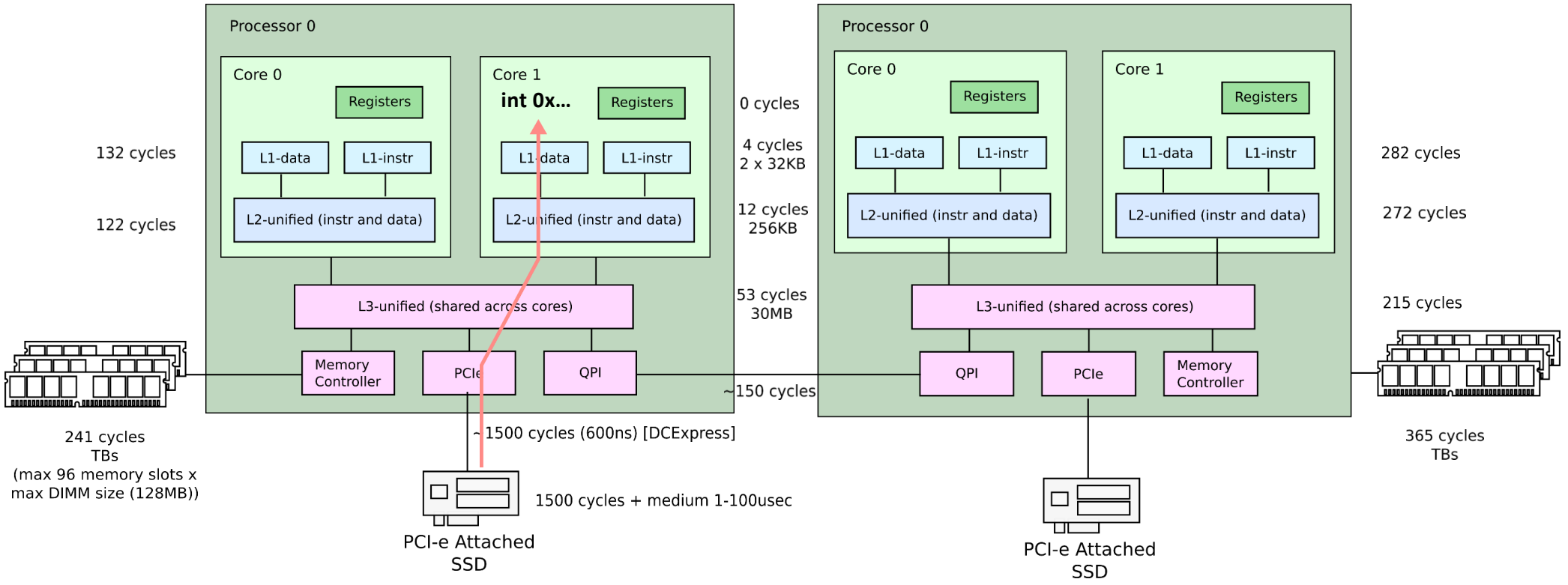
Device I/O

- Essentially just sending data to and from external devices
- Modern devices communicate over PCIe
  - Well there are other popular buses, e.g., USB, SATA (disks), etc.
  - Conceptually they are similar
- Devices can
  - Read memory
  - Send interrupts to the CPU

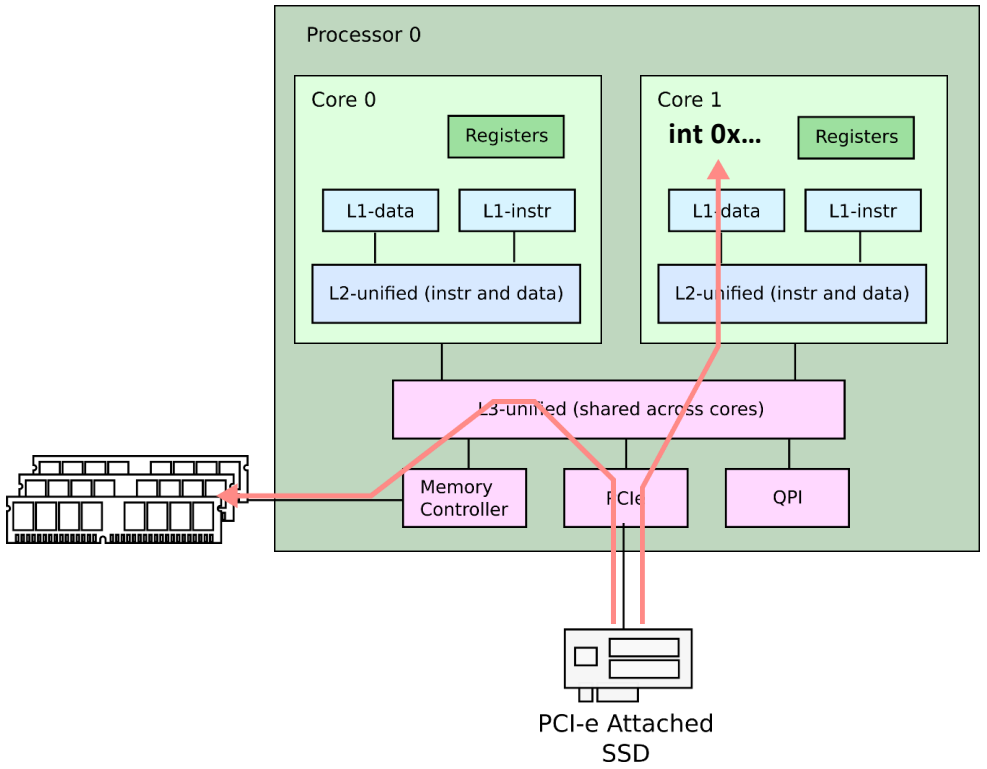
Direct memory access



Interrupts

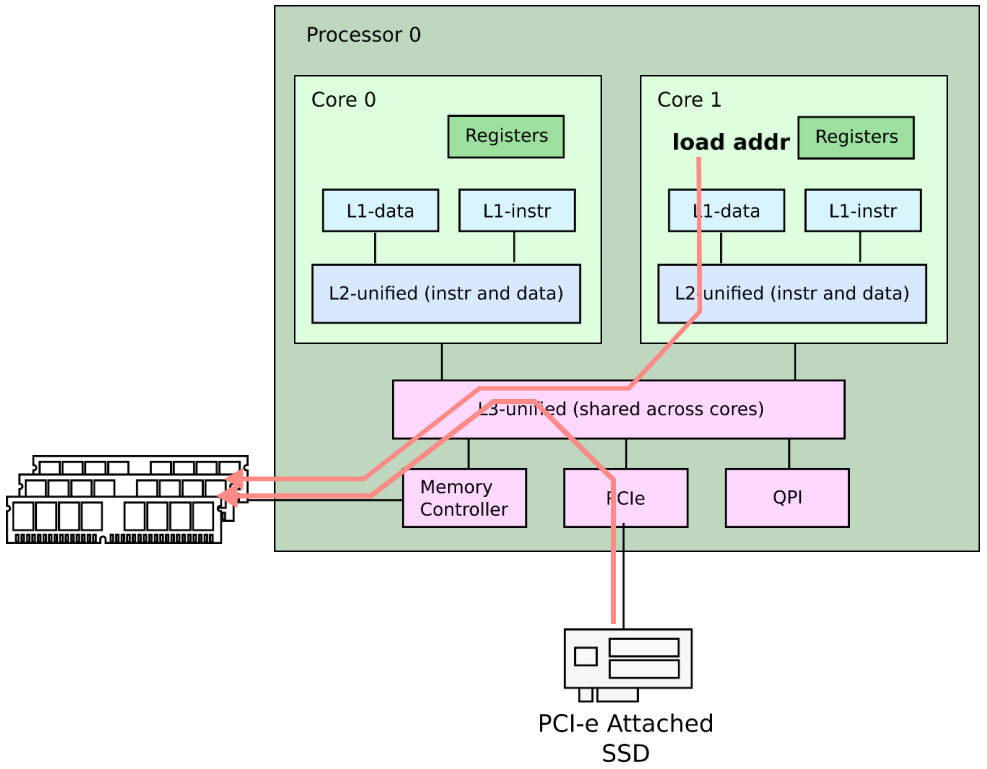

Processor 0
Core 0
Registers
L1-data
L1-instr
L2-unified (instr and data)
Core 1
int 0x...
Registers
L1-data
L1-instr
L2-unified (instr and data)
L3-unified (shared across cores)
Memory Controller
PCIe
QPI
Processor 0
Core 0
Registers
L1-data
L1-instr
L2-unified (instr and data)
Core 1
Registers
L1-data
L1-instr
L2-unified (instr and data)
L3-unified (shared across cores)
QPI
PCIe
Memory Controller
0 cycles
4 cycles
2 x 32KB
12 cycles
256KB
53 cycles
30MB
~150 cycles
132 cycles
122 cycles
241 cycles
TBs
(max 96 memory slots x
max DIMM size (128MB))
~1500 cycles (600ns) [DCExpress]
1500 cycles + medium 1-100usec
PCI-e Attached
SSD
282 cycles
272 cycles
215 cycles
365 cycles
TBs
PCI-e Attached
SSD

Device I/O



- Write incoming data in memory, e.g.,
  - Network packets
  - Disk requests, etc.
- Then raise an interrupt to notify the CPU
  - CPU starts executing interrupt handler
  - Then reads incoming packets form memory

Device I/O (polling mode)



- Alternatively the CPU has to check for incoming data in memory periodically
  - Or poll
- Rationale
  - Interrupts are expensive

References

- Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. http://ieeexplore.ieee.org/abstract/document/7349629

- Intel SGX Explained https://eprint.iacr.org/2016/086.pdf

- DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express https://www.usenix.org/system/files/conference/fast14/fast14-paper_vucinic.pdf

Synchronization

Compare and swap: xchg

- Swap a word in memory with a new value
- Return old value

Correct implementation

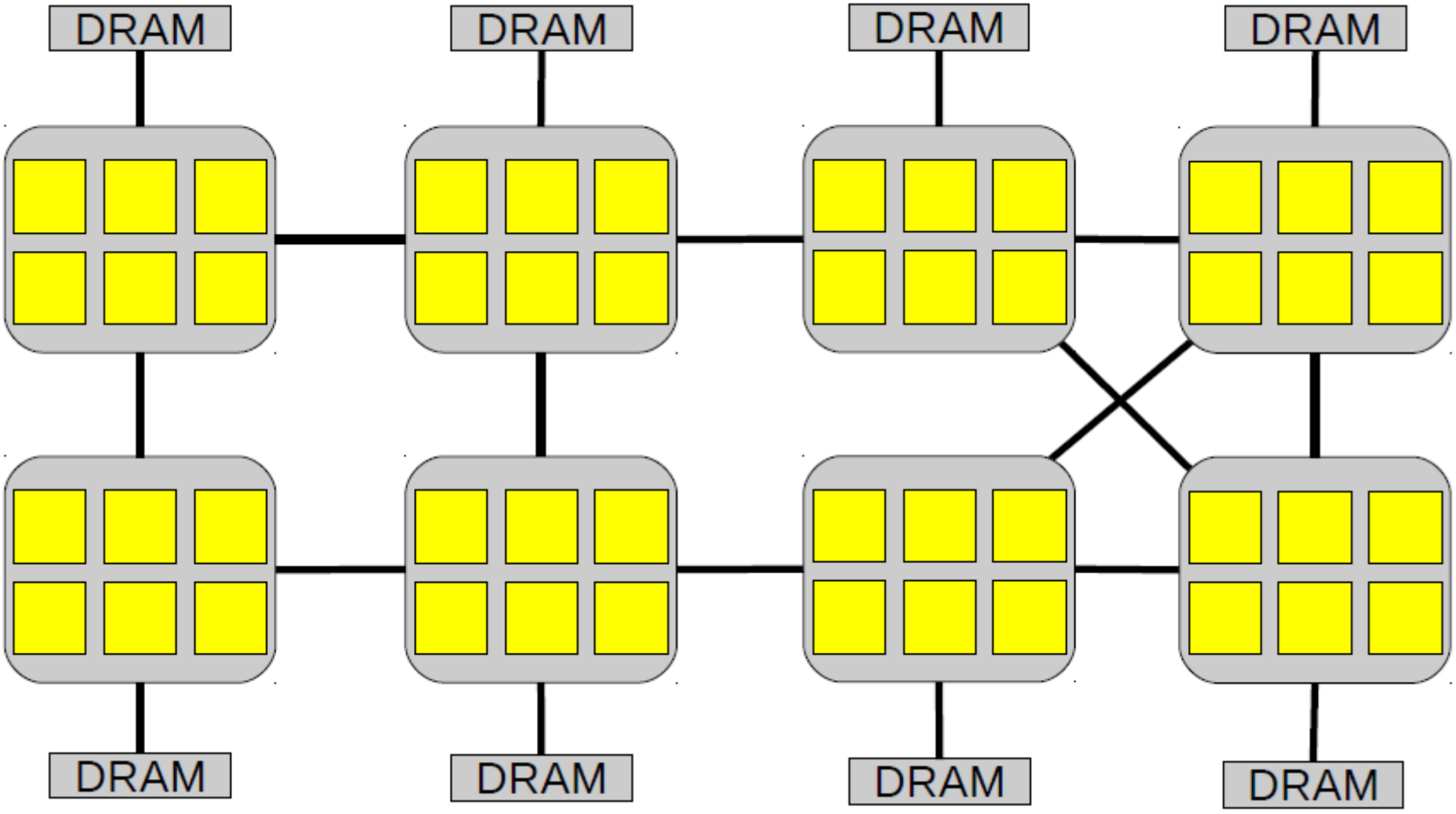
```
1573 void
1574 acquire(struct spinlock *lk)
1575 {
...
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
...
1592 }
```

xchg instruction

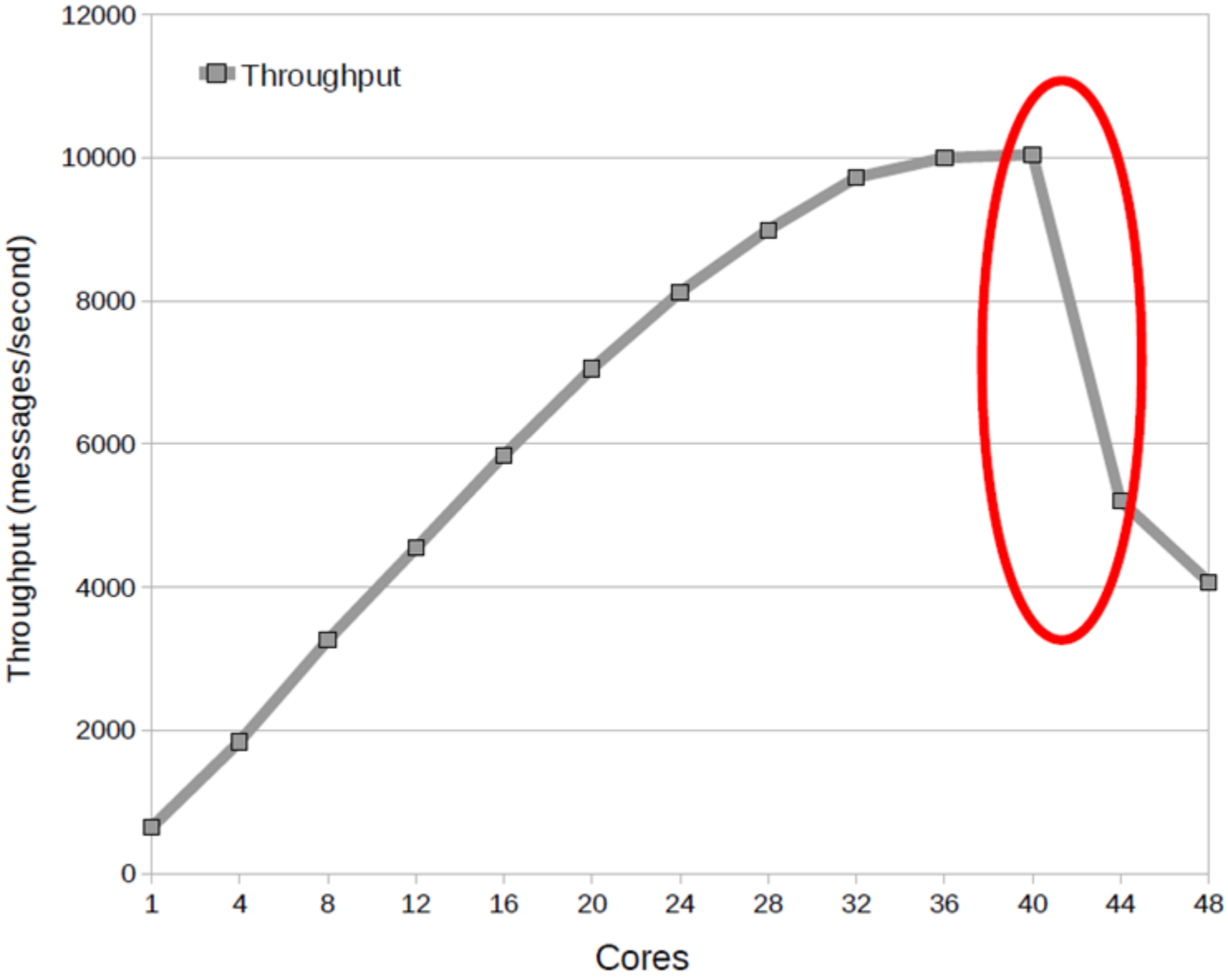
```
0568 static inline uint
0569 xchg(volatile uint *addr, uint newval)
0570 {
0571   uint result;
0572
0573   // The + in "+m" denotes a read−modify−write
       operand.
0574   asm volatile("lock; xchgl %0, %1" :
0575                "+m" (*addr), "=a" (result) :
0576                "1" (newval) :
0577                "cc");
0578   return result;
0579 }
```

Scalability

48-core AMD server



Exim collapse



Oprofile results

40 cores:
10000 msg/sec

samples	%	app name	symbol name
2616	7.3522	vmlinux	radix_tree_lookup_slot
2329	6.5456	vmlinux	unmap_vmas
2197	6.1746	vmlinux	filemap_fault
1488	4.1820	vmlinux	__do_fault
1348	3.7885	vmlinux	copy_page_c
1182	3.3220	vmlinux	unlock_page
966	2.7149	vmlinux	page_fault

48 cores:
4000 msg/sec

samples	%	app name	symbol name
13515	34.8657	vmlinux	lookup_mnt
2002	5.1647	vmlinux	radix_tree_lookup_slot
1661	4.2850	vmlinux	filemap_fault
1497	3.8619	vmlinux	unmap_vmas
1026	2.6469	vmlinux	__do_fault
914	2.3579	vmlinux	atomic_dec
896	2.3115	vmlinux	unlock_page

Exim collapse

- sys_open eventually calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        return mnt;
}
```

Exim collapse

- sys_open eventually calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        return mnt;
}
```

Critical section is short. Why does it cause a scalability bottleneck?

- spin_lock and spin_unlock use many more cycles than the critical section

Ticket lock in Linux

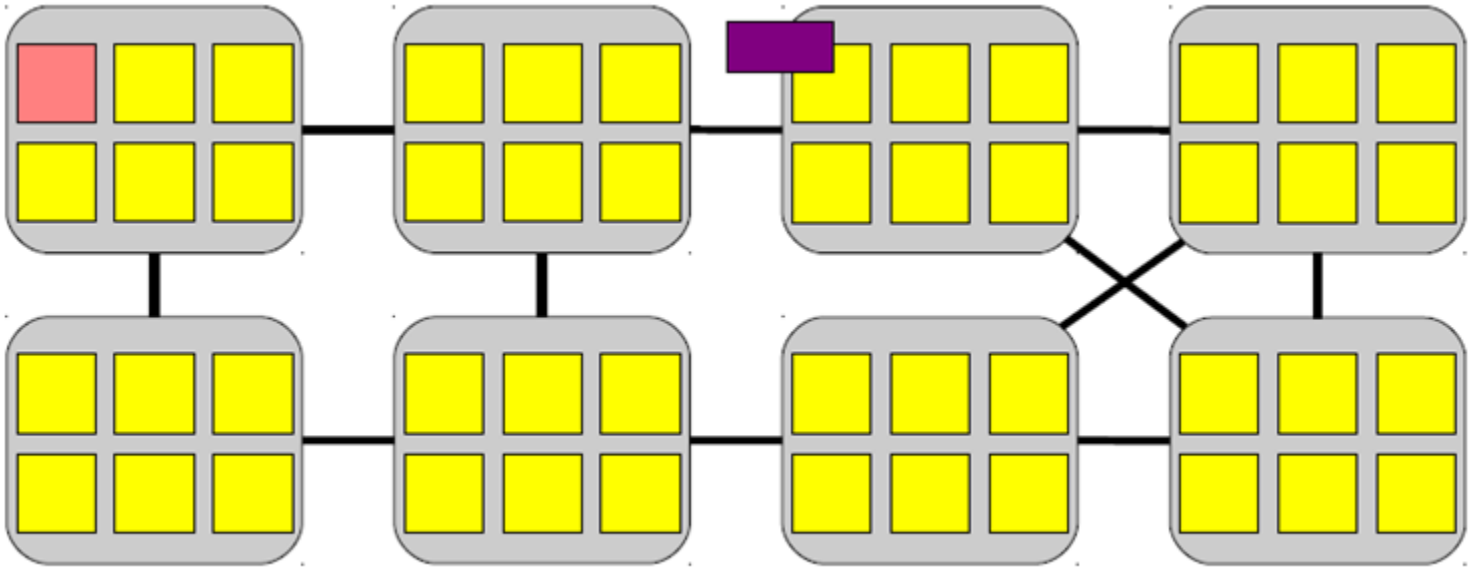
```
struct spinlock_t {
  int current_ticket ;
  int next_ticket ;
}
void spin_lock ( spinlock_t *lock)
{
  int t = atomic_fetch_and_inc (&lock -> next_ticket );
  while (t != lock -> current_ticket )
  ; /* spin */
}
void spin_unlock ( spinlock_t *lock)
{
  lock -> current_ticket ++;
}
```

Spin lock implementation

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



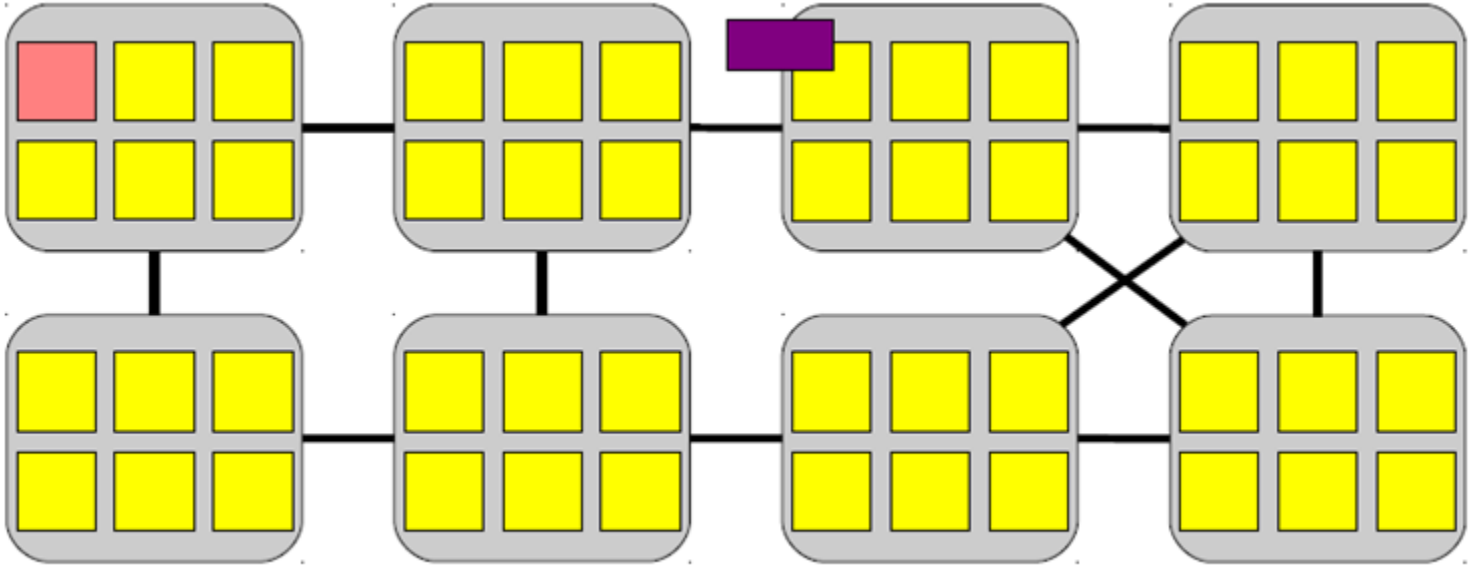
Spin l entation

Allocate a ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



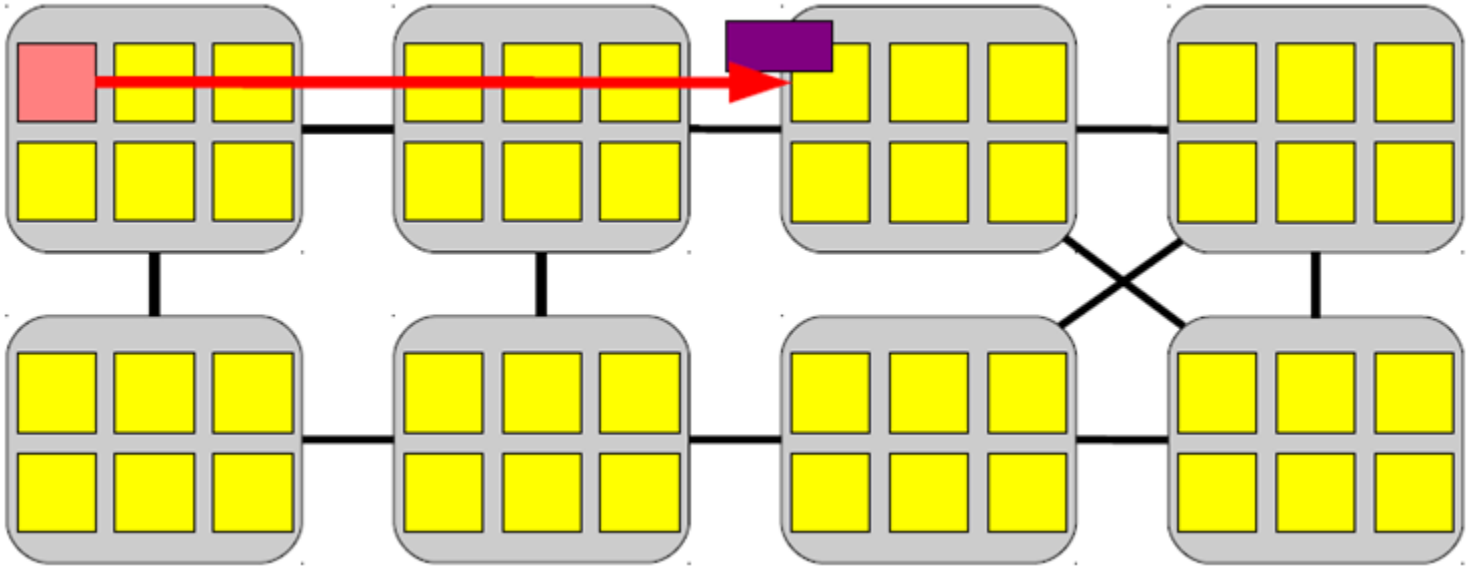
Spin l entation

Allocate a ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



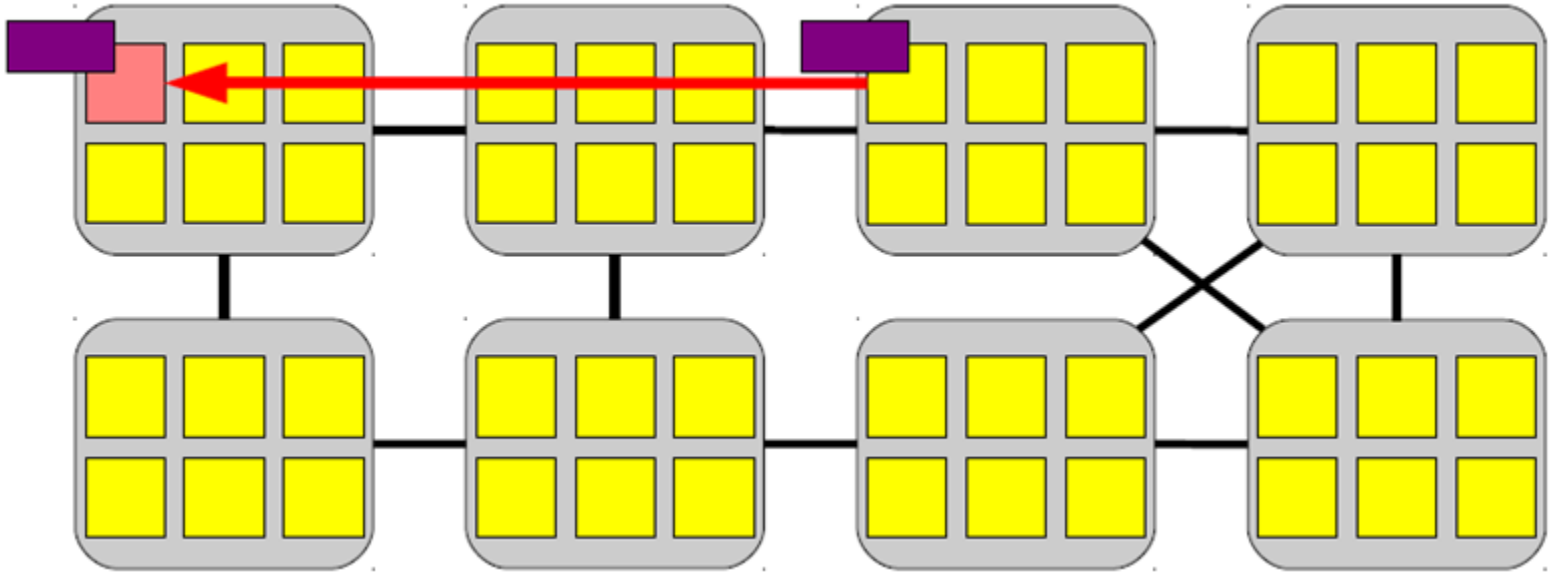
Spin l entation

Allocate a ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



Spin l entation

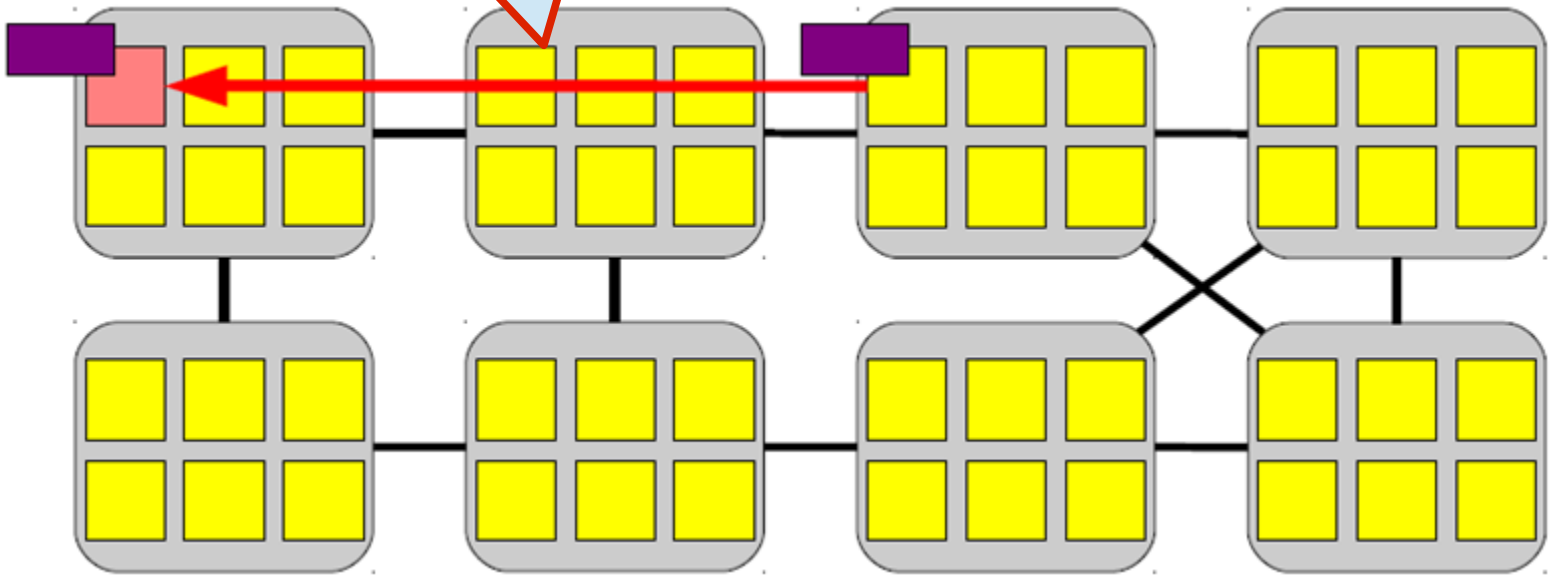
Allocate a ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

120-420 cycles



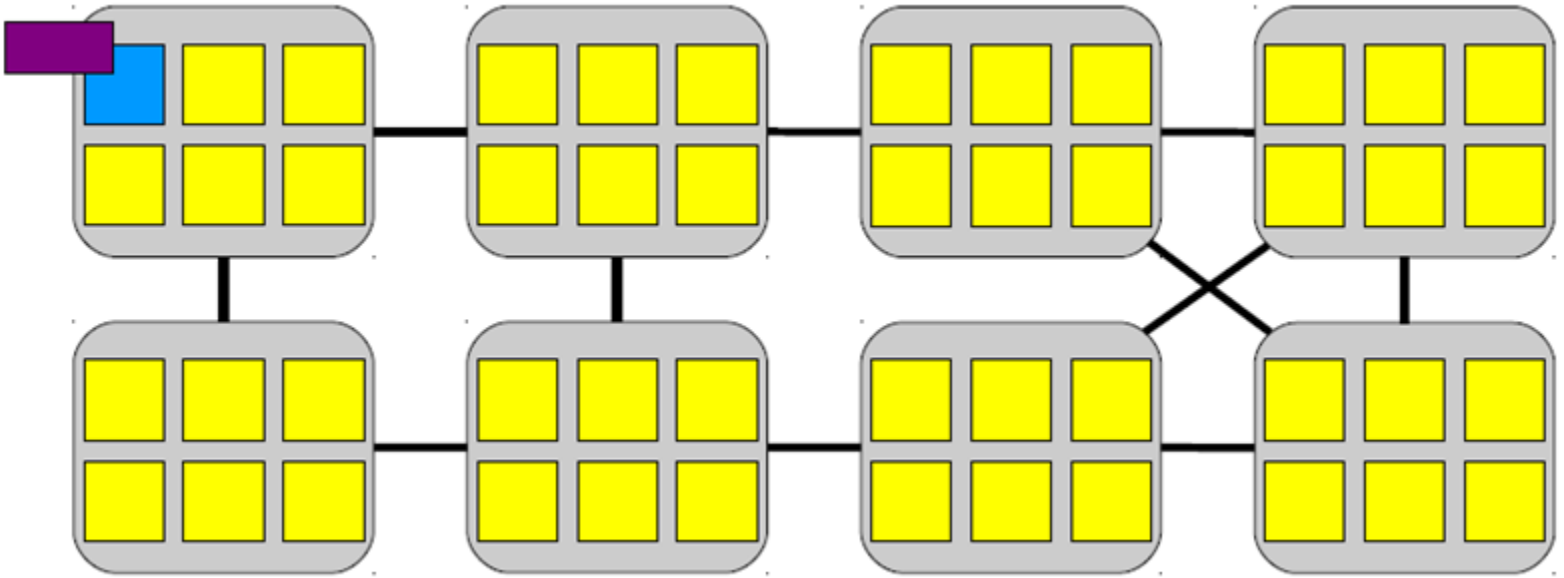
Spin lock impleme

Update the ticket value

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



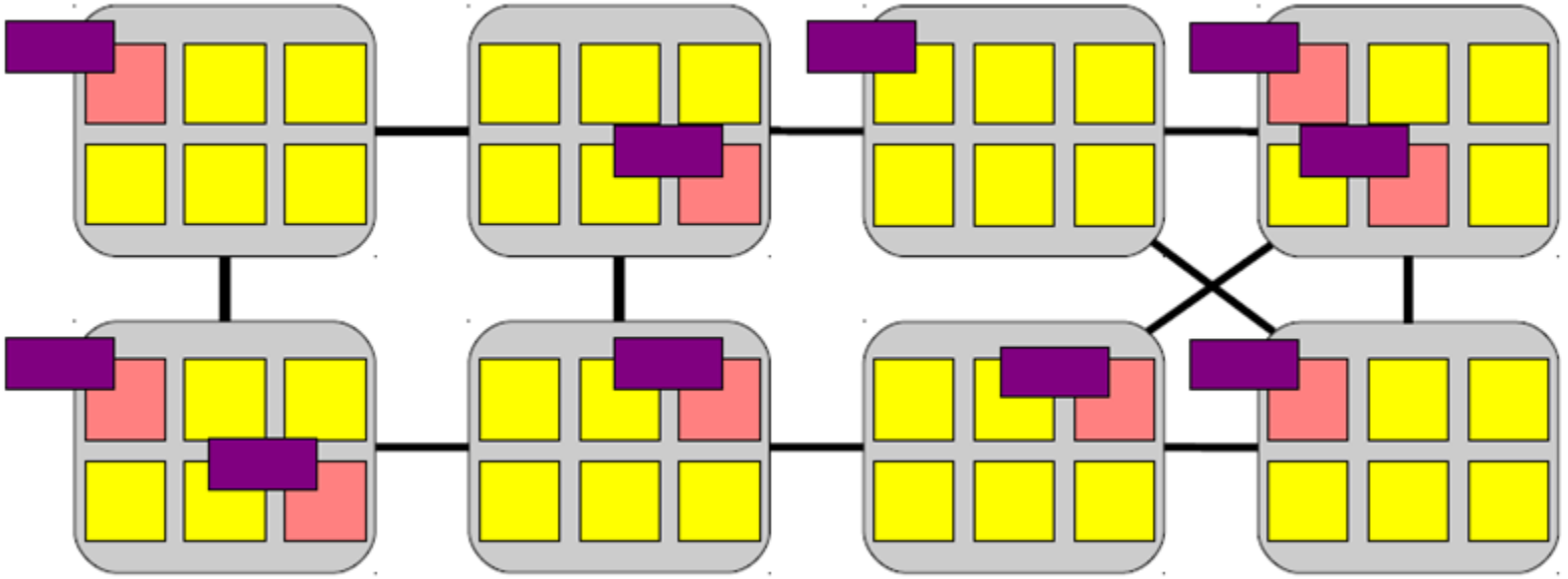
ck implementation

Bunch of cores are spinning

```
void spin_     (spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



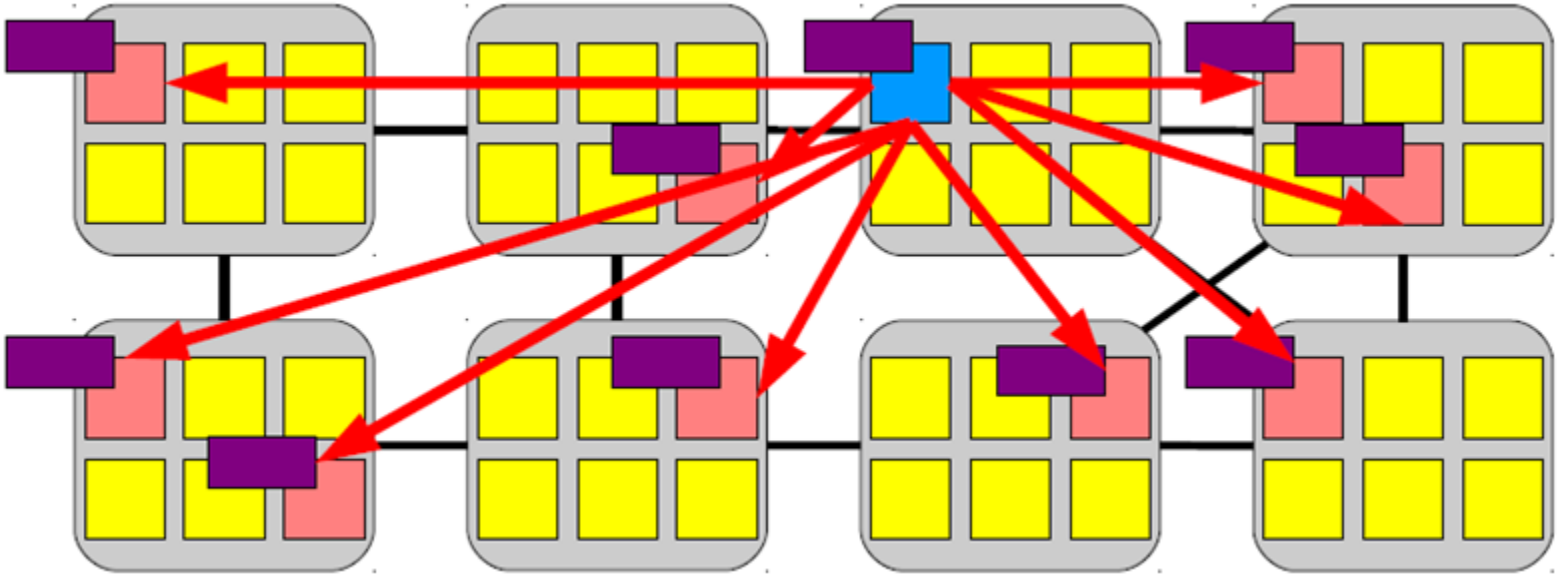
Spin lock implementation

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
{
cket;
nt next_ticket;
}
```

Broadcast message
(invalidate the value)

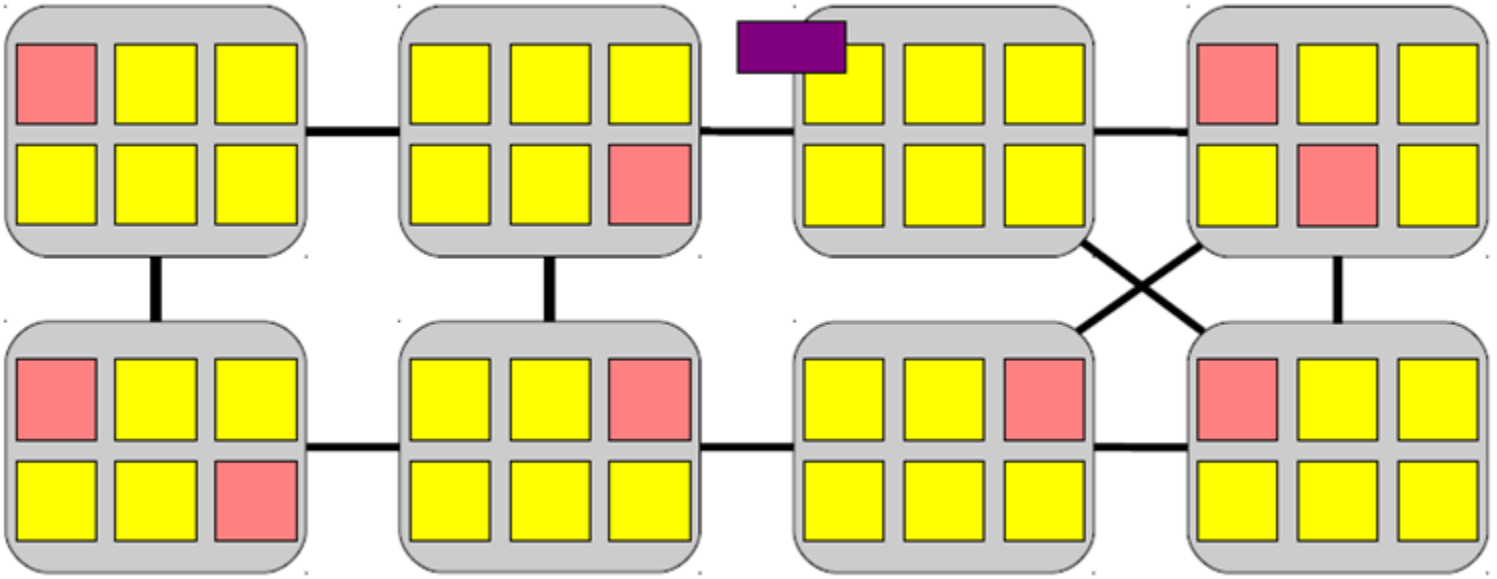


Spin lock implementation

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

Cores don't have the value of current_ticket

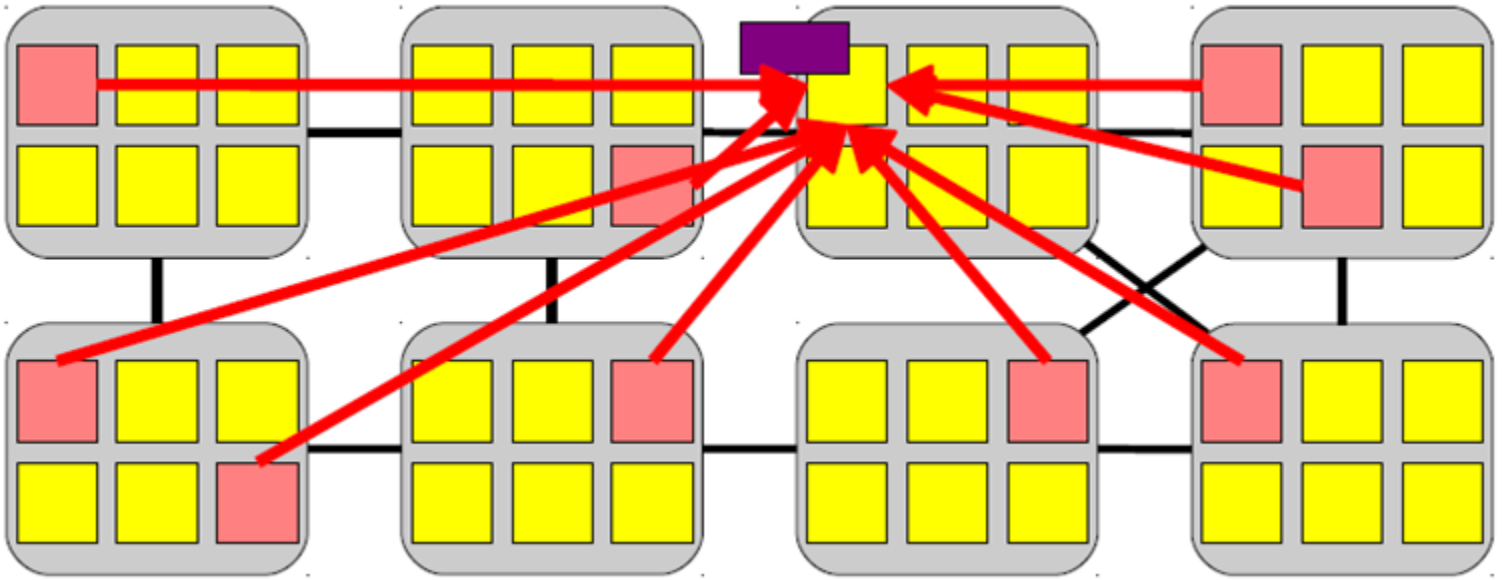


Spin lock implementation

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

Re-read the value

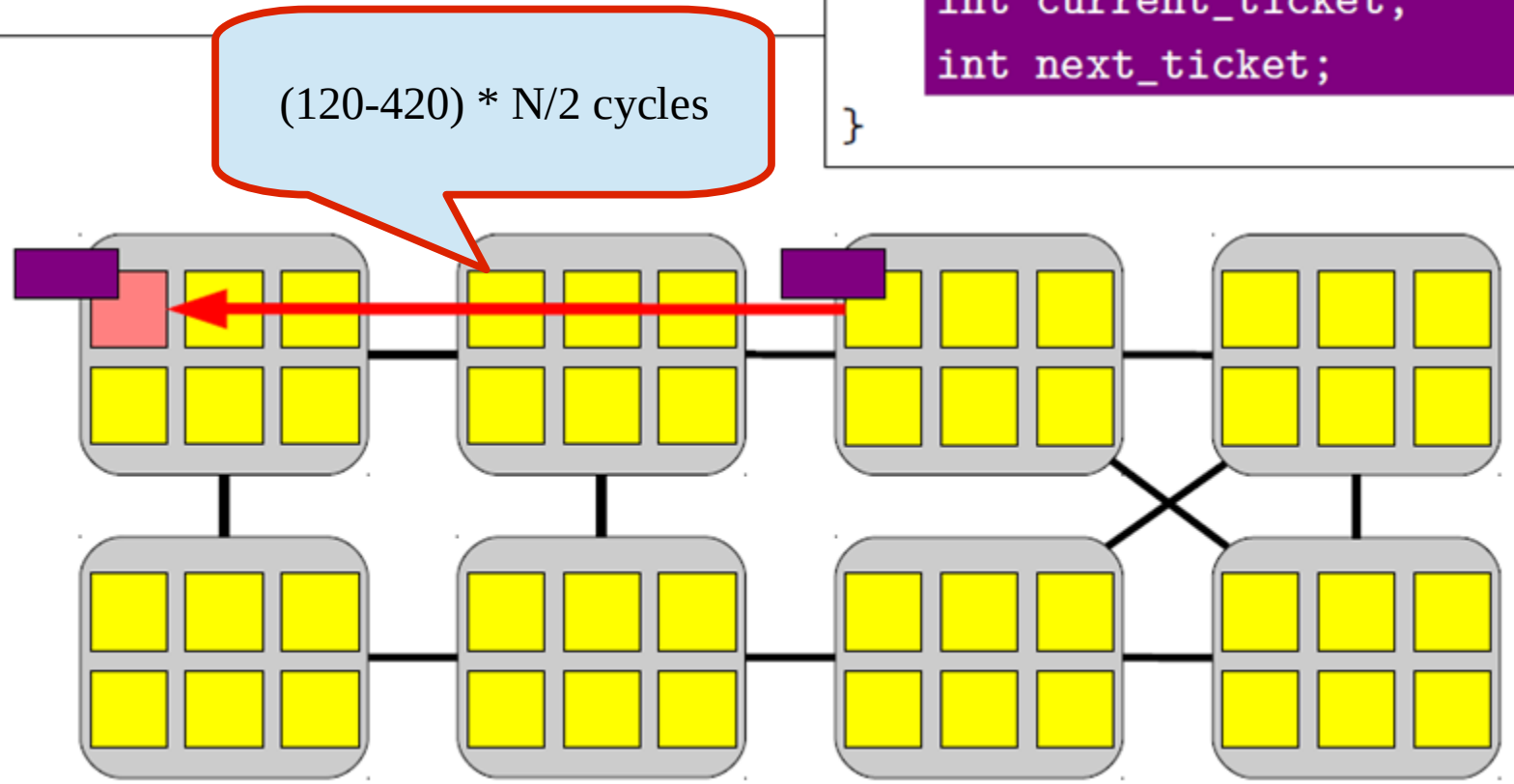


Spin lock implementation

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

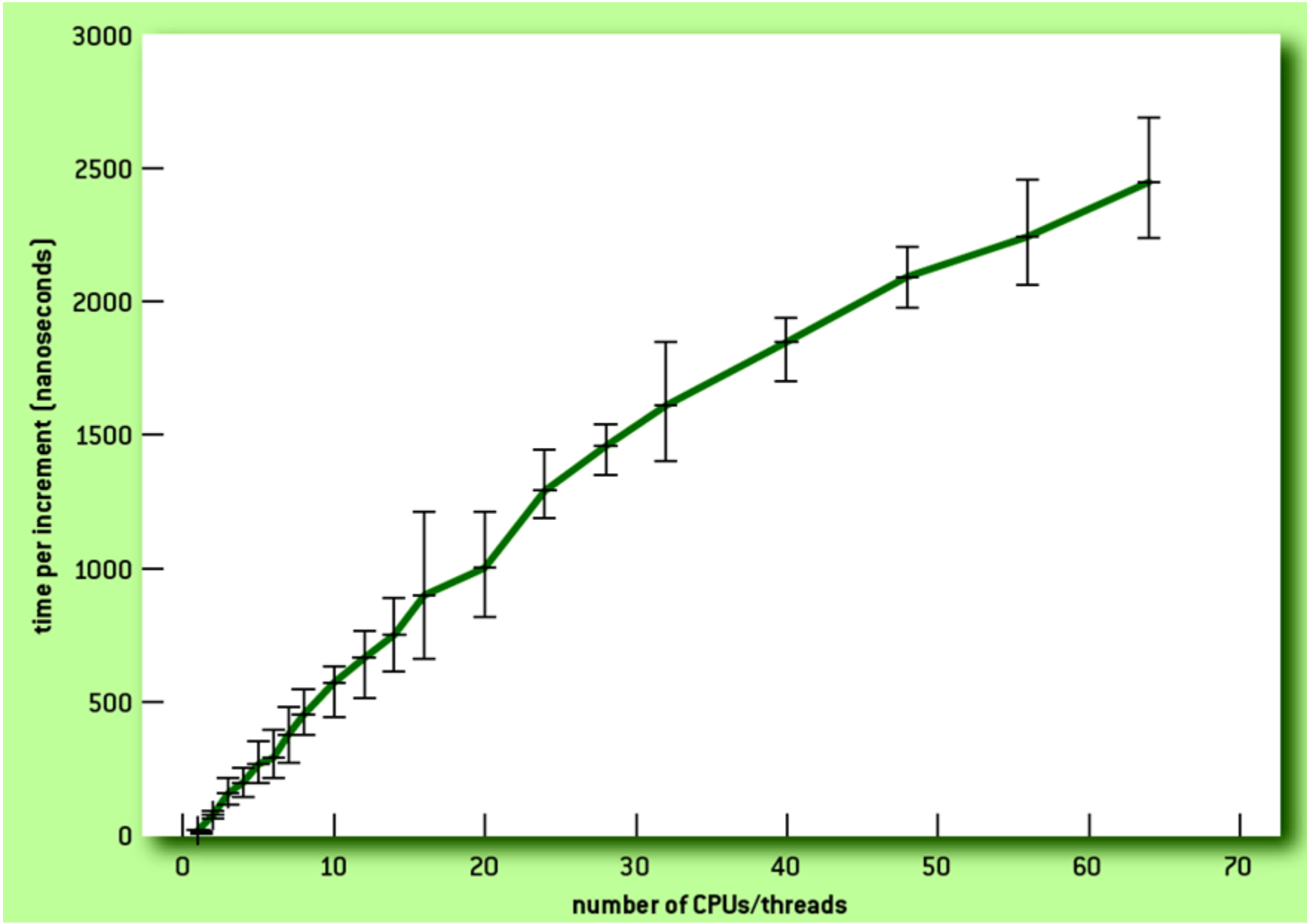
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



- In most architectures, the cache-coherence reads are serialized (either by a shared bus or at the cache line's home or directory node)
- Thus completing them all takes time proportional to the number of cores
- The core that is next in line for the lock can expect to receive its copy of the cache line midway through this process.
- N/2

Atomic synchronization primitives do not scale well

Atomic increment on 64 cores



What can we do about it?

Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        if ((mnt = hash_get(percore_mnts[cpu()], path)))
                return mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        hash_put(percore_mnts[cpu()], path, mnt);
        return mnt;
}
```

Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        if ((mnt = hash_get(percore_mnts[cpu()], path)))
                return mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        hash_put(percore_mnts[cpu()], path, mnt);
        return mnt;
}
```

Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        if ((mnt = hash_get(percore_mnts[cpu()], path)))
                return mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        hash_put(percore_mnts[cpu()], path, mnt);
        return mnt;
}
```

- Fast path: local hash lookup

Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        if ((mnt = hash_get(percore_mnts[cpu()], path)))
                return mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        hash_put(percore_mnts[cpu()], path, mnt);
        return mnt;
}
```

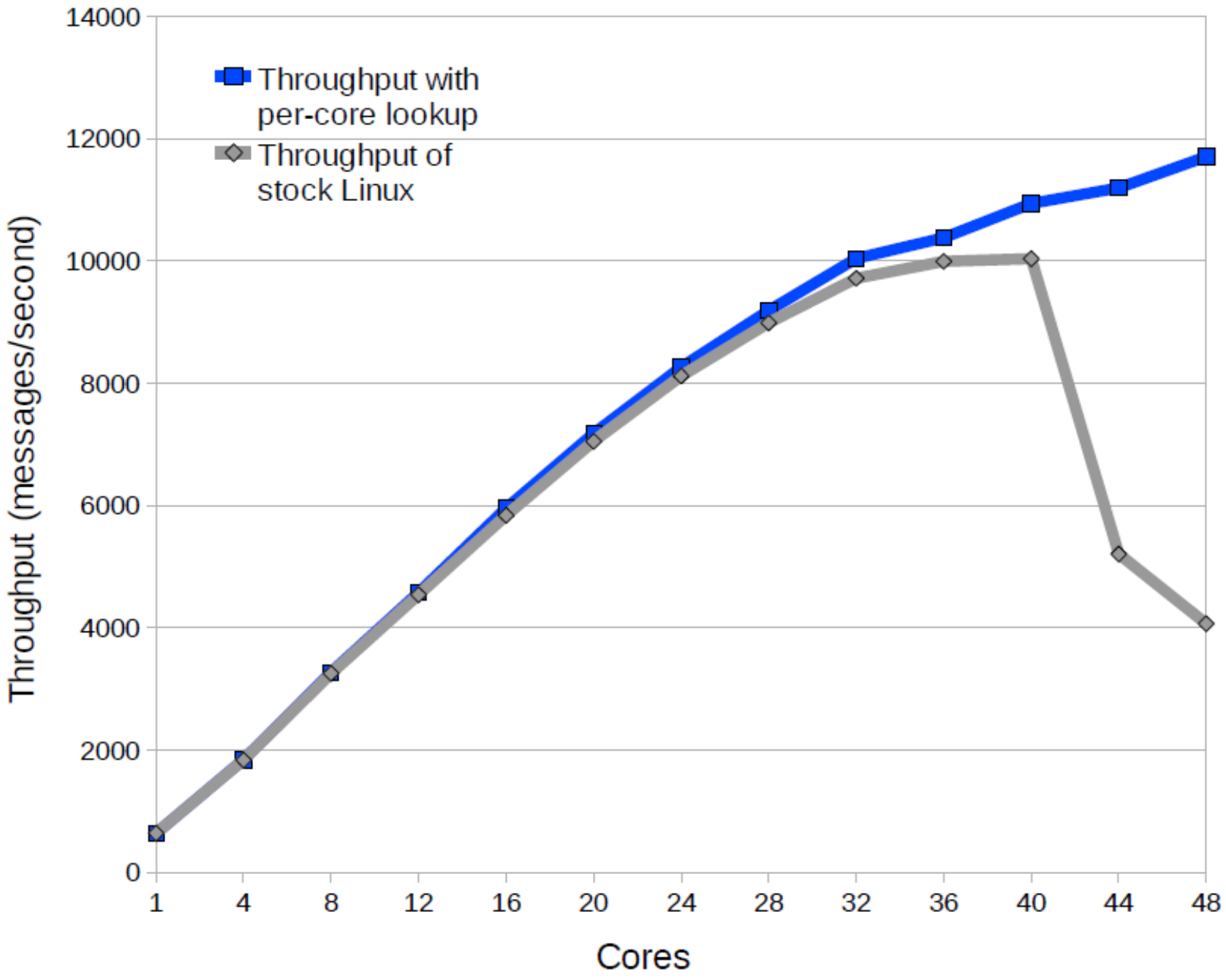
- Slow path: lookup global mount table, then update local, per-core copy

Solution #1: per-core mount tables

- Observation: mount table is rarely modified

```
struct vfsmount *lookup_mnt(struct path *path)
{
        struct vfsmount *mnt;
        if ((mnt = hash_get(percore_mnts[cpu()], path)))
                return mnt;
        spin_lock(&vfsmount_lock);
        mnt = hash_get(mnts, path);
        spin_unlock(&vfsmount_lock);
        hash_put(percore_mnts[cpu()], path, mnt);
        return mnt;
}
```

- On update: invalidate local hash table copies


Throughput with per-core lookup
Throughput of stock Linux
Throughput (messages/second)
14000
12000
10000
8000
6000
4000
2000
0
1
4
8
12
16
20
24
28
32
36
40
44
48
Cores

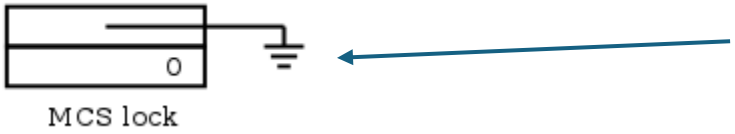
Solution #2:

Is it possible to build scalable spinlocks?

MCS lock
(Mellor-Crummey and M. L. Scott)

- High-level idea: maintain a queue of waiters on the lock
  - Wait on different cache lines
  - Instead of waiting on the same cache line

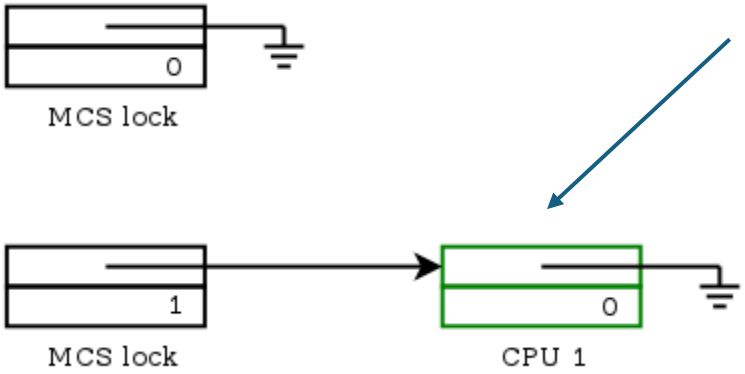
MCS lock



Queue head (global)
mcslock_t

```
typedef struct {
        struct qnode *v;
} mcslock_t;
```

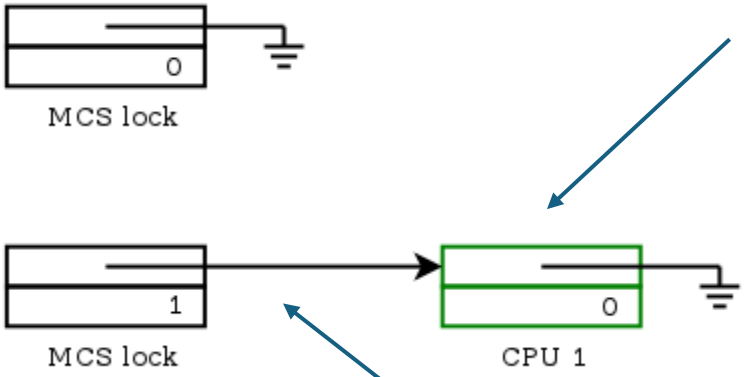
MCS lock



Per-CPU queue node

```
struct qnode {
        volatile void *next;
        volatile char locked;
};
```

MCS lock

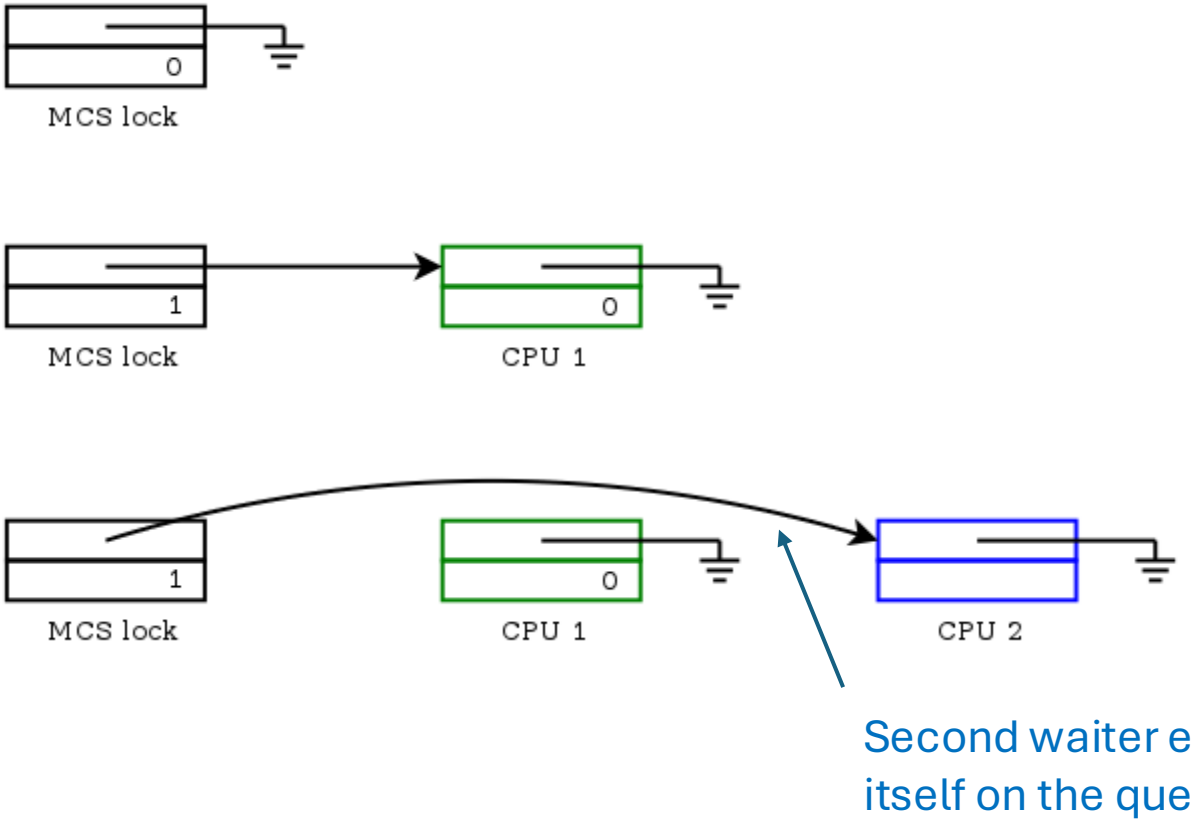


Per-CPU queue node

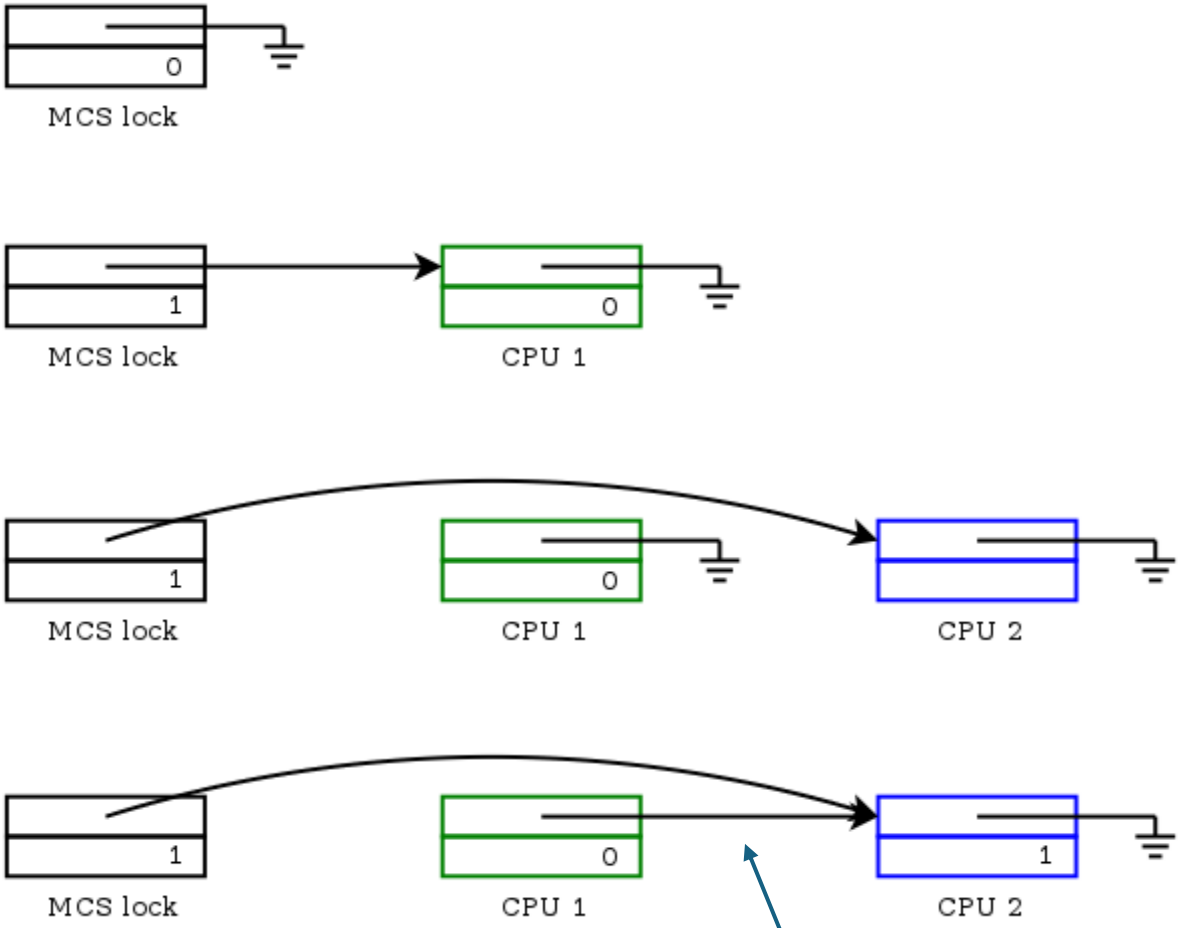
```
struct qnode {
        volatile void *next;
        volatile char locked;
};
```

First waiter enqueues itself on the queue

MCS lock

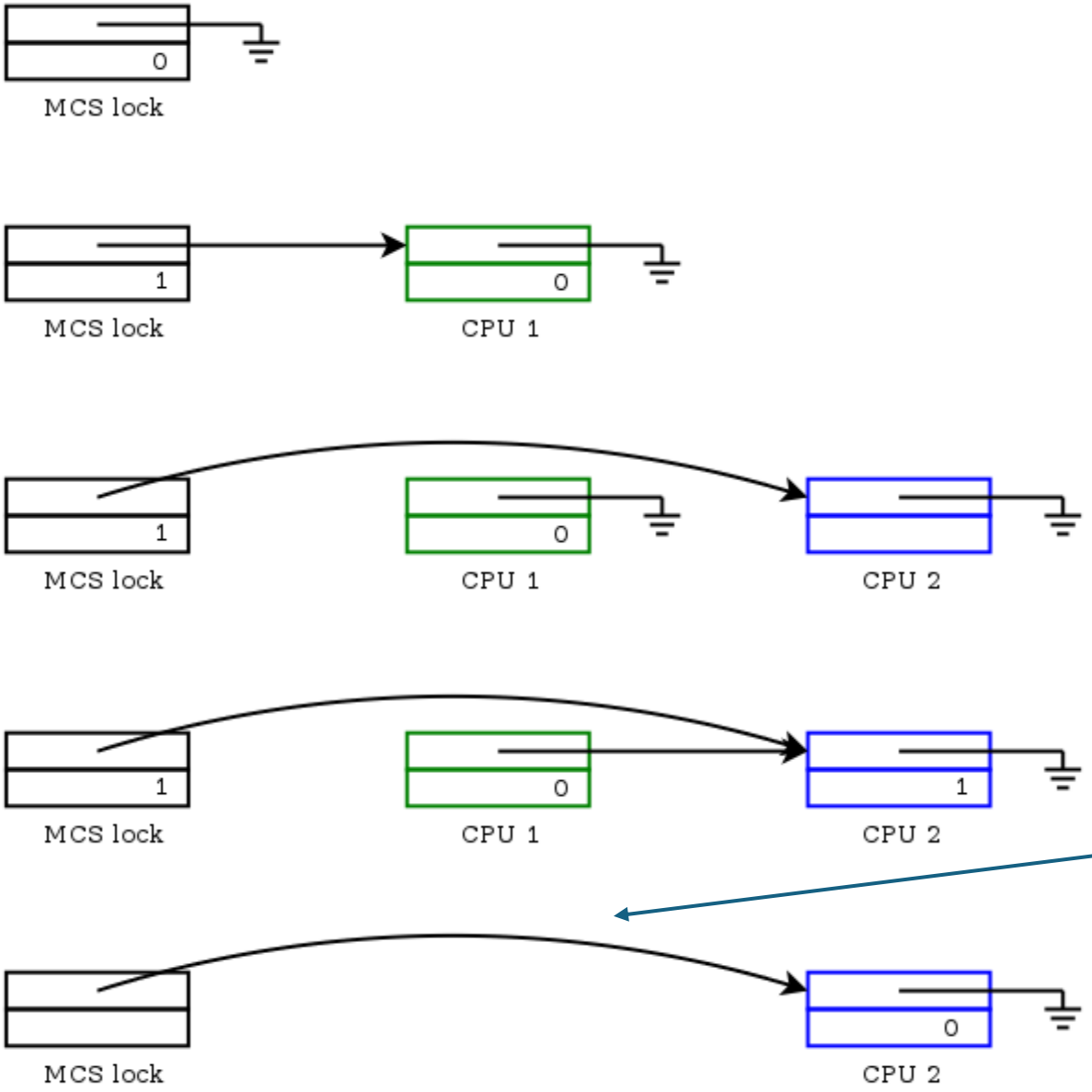

0
MCS lock
1
MCS lock
0
CPU 1
1
MCS lock
0
CPU 1
CPU 2
Second waiter enqueues itself on the queue

MCS lock



The second waiter also enqueues itself on the first waiter, to let the first know that it has to wake up the second waiter

MCS lock



When the first waiter is done, it removes itself from the queue and passes the lock to the second waiter

MCS lock (Mellor-Crummey and M. L. Scott)

```
struct qnode {
        volatile void *next;
        volatile char locked;
};
typedef struct {
        struct qnode *v;
} mcslock_t;
arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
        struct qnode *predecessor;
        mynode->next = NULL;
        predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
        if (predecessor) {
                mynode->locked = 1;
                barrier();
                predecessor->next = mynode;
                while (mynode->locked) ;
        }
}
```

Enqueue yourself on the global head

MCS lock (Mellor-Crummey and M. L. Scott)

```
struct qnode {
        volatile void *next;
        volatile char locked;
};
typedef struct {
        struct qnode *v;
} mcslock_t;
arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
        struct qnode *predecessor;
        mynode->next = NULL;
        predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
        if (predecessor) {
                mynode->locked = 1;
                barrier();
                predecessor->next = mynode;
                while (mynode->locked) ;
        }
}
```


If queue was not empty, set yourself to ”locked” and enqueue yourself on the previous node

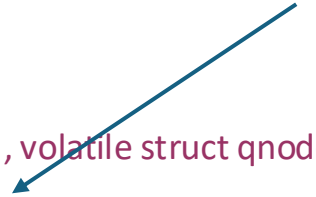
MCS lock (Mellor-Crummey and M. L. Scott)

```
struct qnode {
        volatile void *next;
        volatile char locked;
};
typedef struct {
        struct qnode *v;
} mcslock_t;
arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
        struct qnode *predecessor;
        mynode->next = NULL;
        predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
        if (predecessor) {
                mynode->locked = 1;
                barrier();
                predecessor->next = mynode;
                while (mynode->locked) ;
        }
}
```

Wait to be unlocked by previous lock holder

unlock

Check if someone is waiting after me



```
arch_mcs_unlock(mcslock_t *l, volatile struct qnode *mynode) {
    if (!mynode->next) {
        if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
            return;
        while (!mynode->next) ;
    }
    ((struct qnode *)mynode->next)->locked = 0;
}
```

unlock

```
arch_mcs_unlock(mcslock_t *l, volatile struct qnode *mynode) {
    if (!mynode->next) {
        if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
            return;
        while (!mynode->next) ;
    }
    ((struct qnode *)mynode->next)->locked = 0;
}
```

Yes (someone is waiting). Pass lock to them. No need to do anything else, they already removed us from the queue

unlock

```
arch_mcs_unlock(mcslock_t *l, volatile struct qnode *mynode) {
    if (!mynode->next) {
        if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
            return;
        while (!mynode->next) ;
    }
    ((struct qnode *)mynode->next)->locked = 0;
}
```

No (no one is waiting). Try to remove ourself from the global queue

unlock

```
arch_mcs_unlock(mcslock_t *l, volatile struct qnode *mynode) {
    if (!mynode->next) {
        if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
            return;
        while (!mynode->next) ;
    }
    ((struct qnode *)mynode->next)->locked = 0;
}
```

No one raced with us (cmpxchg() returned our old pointer. Safe to return.

unlock

```
arch_mcs_unlock(mcslock_t *l, volatile struct qnode *mynode) {
    if (!mynode->next) {
        if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
            return;
        while (!mynode->next) ;
    }
    ((struct qnode *)mynode->next)->locked = 0;
}
```

cmpxchg() returned some other pointer.

Means there is a race and someone put themselves into the queue.

Let's wait for them to update our pointer and them it will be safe to pass the lock to them.

Why does this scale?

Ticket spinlock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

- Remember O(N) re-fetch messages after invalidation broadcast

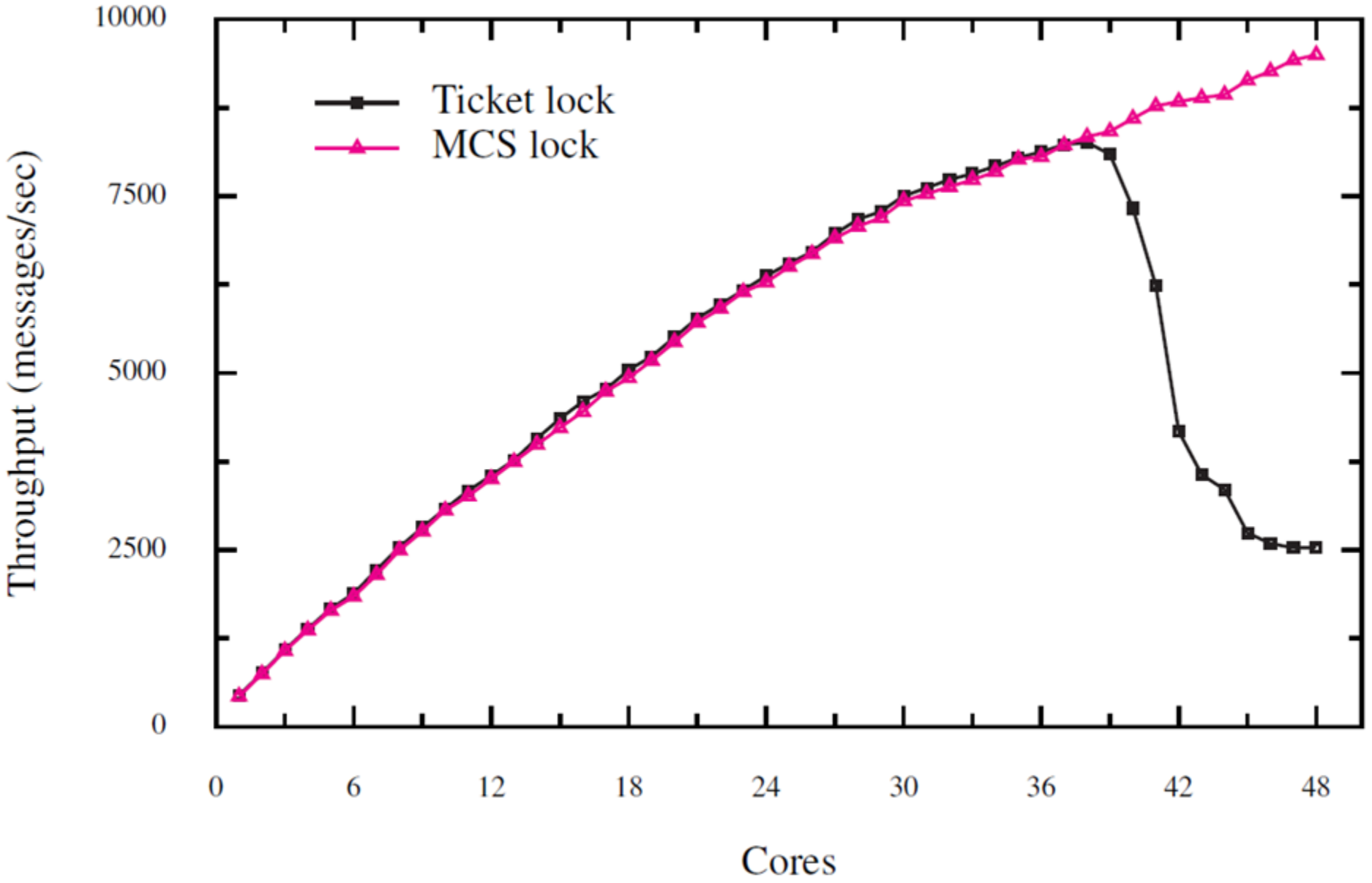
```
arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
        struct qnode *predecessor;
        mynode->next = NULL;
        predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
        if (predecessor) {
                mynode->locked = 1;
                barrier();
                predecessor->next = mynode;
                while (mynode->locked) ;
        }
}
arch_mcs_unlock(mcslock_t *l, volatile struct qnode *mynode) {
        if (!mynode->next) {
                if (cmpxchg((long *)&l->v, (long)mynode, 0) == (long)mynode)
                        return;
                while (!mynode->next) ;
        }
        ((struct qnode *)mynode->next)->locked = 0;
}
```

One re-fetch message after invalidation

Cache line isolation

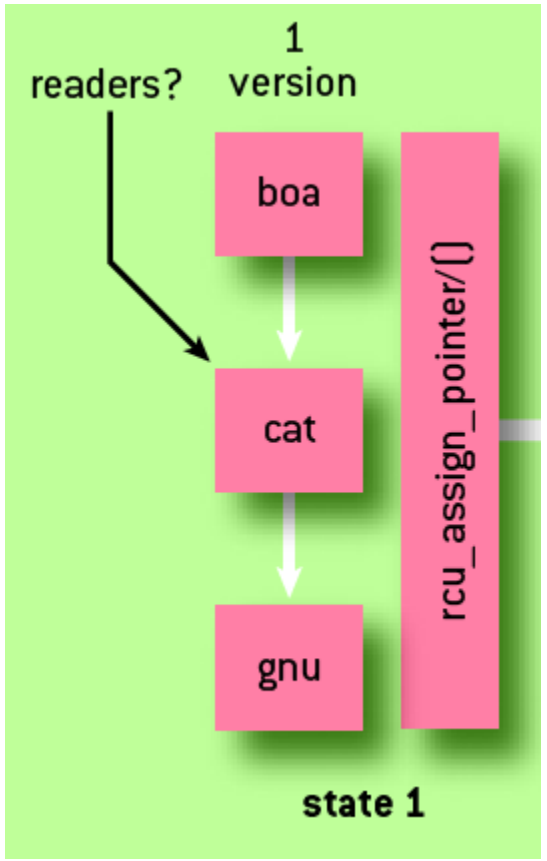
```
struct qnode {
        volatile void *next;
        volatile char locked;
        char __pad[0] __attribute__((aligned(64)));
};

typedef struct {
        struct qnode *v __attribute__((aligned(64)));
} mcslock_t;
```

Exim: MCS vs ticket lock



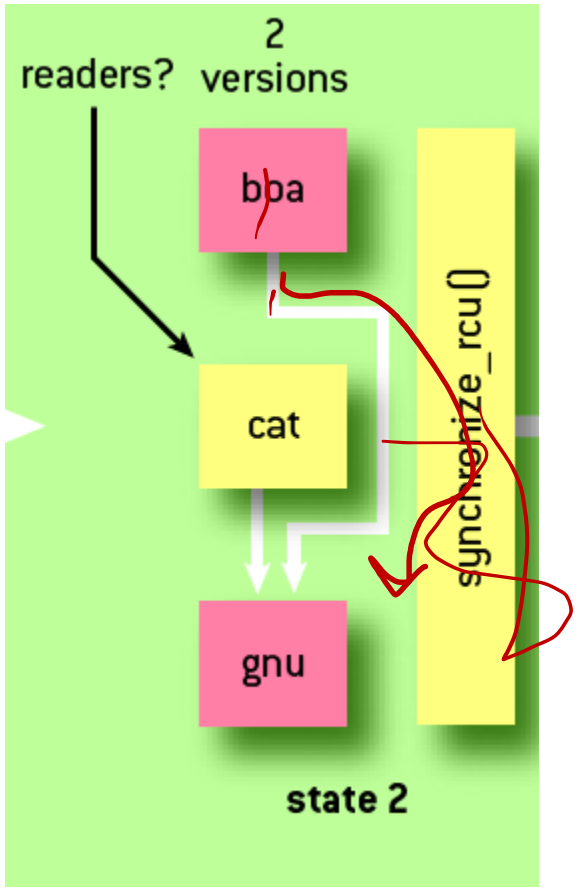
RCU: Read Copy Update

Read copy update



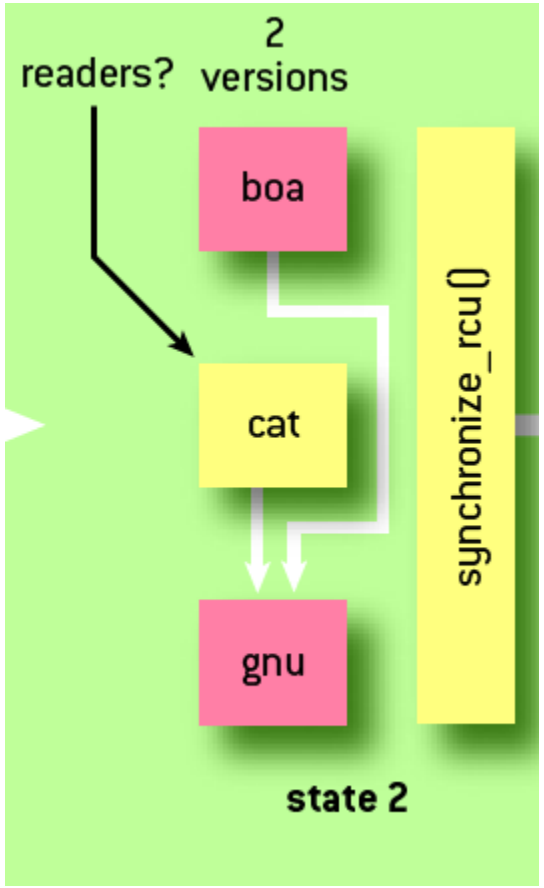
- Goal: remove “cat” from the list
- There might be some readers of “cat”
- Idea: control the pointer dereference
- Make it atomic

Read copy update (2)



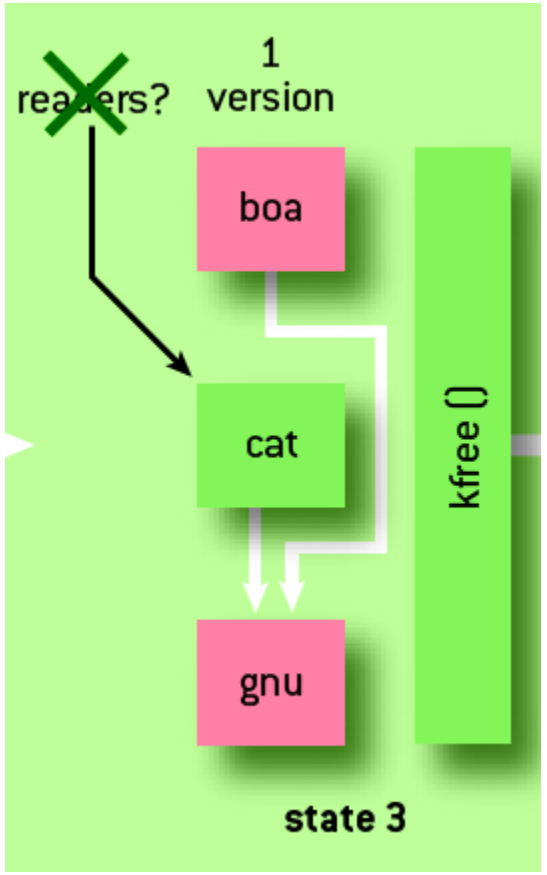
- Remove “cat”
- Update the “boa” pointer
- All subsequent readers will get “gnu” as boa->next

Read copy update (2)



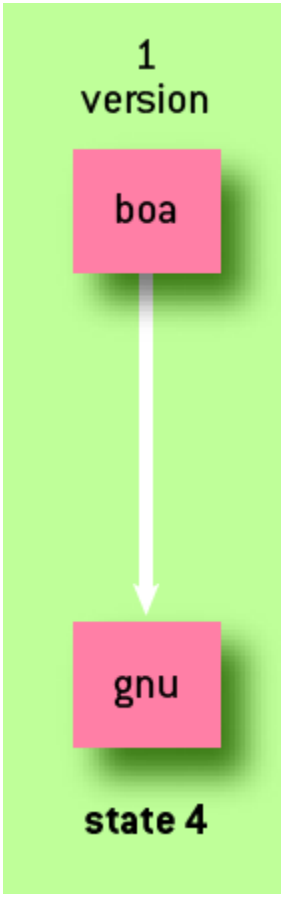
- Wait for all readers to finish
- synchronize_rcu()

Read copy update (3)



- Readers finished
- Safe to deallocate “cat”

Read copy update (4)



- New state of the list

How can we build this?

- Disable preemption while using the RCU data
- rcu_lock(), rcu_unlock()
- Wait for all RCU readers to finish
- Schedule something on each CPU
- If you managed to run on a CPU
  - You preempted a thread on that CPU
  - Thus this thread exited the RCU lock/unlock section

```
void rcu_read_lock()
{
  preempt_disable[cpu_id()]++;
}

void rcu_read_unlock()
{
  preempt_disable[cpu_id()]--;
}

void synchronize_rcu(void)
{
  for_each_cpu(int cpu)
    run_on(cpu);
}
```

RCU implementation

What does it mean to run on a CPU?

- In xv6 scheduler() function goes through a list of all processes
- If we keep a mask of allowable CPUs for each process
- On each CPU the scheduler() function will pick processes with a proper mask
- run_on(cpu)
- sets the mask for the current process
- Invokes scheduler()
  - Calls yield(), which in turn calls swtch()

In practice...

- Linux just waits for all CPUs to pass through a context switch
- Instead of scheduling the updater on each CPU

```
struct vfsmount *lookup_mnt(struct path *path)
{
  struct vfsmount *local_mnts;
  struct vfsmount *mnt;

  rcu_read_lock();
  local_mnts = rcu_dereference(mnts);
  mnt = lookup_mnt(local_mnts, path);
  rcu_read_unlock();

  return mnt;
}
```

RCU example: lookup_mnt()

Why do we need rcu_dereference()?

```
struct vfsmount *lookup_mnt(struct path *path)
{
  ...
  rcu_read_lock();
  local_mnts = rcu_dereference(mnts);
  mnt = lookup_mnt(local_mnts, path);
  rcu_read_unlock();
  ...
}
```

Memory barriers

```
#define __rcu_assign_pointer(p, v, space) \
        do { \
                smp_wmb(); \
                (p) = (typeof(*v) __force space *)(v); \
        } while (0)
```

- Some CPUs may re-order writes
  - I.e. the reader might read a new pointer but …

RCU API

- rcu_read_lock() - begin an RCU critical section
- rcu_read_unlock() - complete an RCU critical section
- synchronize_rcu() - wait for existing RCU critical sections to complete
- call_rcu(callback, arguments...) - call the callback when existing RCU critical sections complete
- rcu_dereference(pointer) - signal the intent to deference a pointer in an RCU critical section
- rcu_assign_pointer(pointer_addr, pointer) - assign a value to a pointer that is read in RCU critical sections

Using RCU

- Simplest use
  - synchronize_rcu () – wait for all readers to complete
  - i.e., all code inside rcu_read_lock()/rcu_read_unlock()
- Sometimes you don’t want to wait for completion
  - Instead use a callback
  - call_rcu() – similar to synchronize_rcu() but asynchronous
  - Wait for all RCU sections to complete and then execute a callback

Example 1: Linked lists

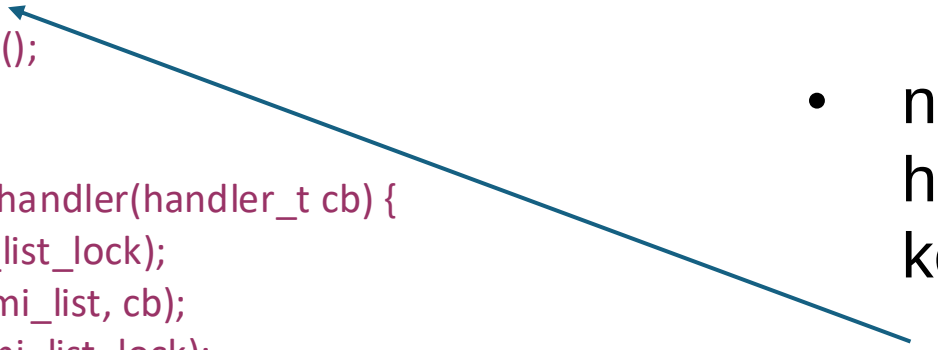
```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;
void handle_nmi() {
   rcu_read_lock();
   rcu_list_for_each(&nmi_list, handler_t cb)
      cb();
   rcu_read_unlock();
}

void register_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_add(&nmi_list, cb);
   spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_remove(cb);
   spin_unlock(&nmi_list_lock);
   synchronize_rcu();
}
```

NMI handlers in Linux

- nmi_list – list of NMI handlers in the Linux kernel
  - Call each callback
- Normally requires a spin lock to protect against concurrent updates

NMI handlers in Linux

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;
void handle_nmi() {
   rcu_read_lock();
   rcu_list_for_each(&nmi_list, handler_t cb)
      cb();
   rcu_read_unlock();
}

void register_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_add(&nmi_list, cb);
   spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_remove(cb);
   spin_unlock(&nmi_list_lock);
   synchronize_rcu();
}
```



- nmi_list – list of NMI handlers in the Linux kernel
  - Call each callback
- Normally requires a spin lock to protect against concurrent updates

NMI handlers in Linux

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;
void handle_nmi() {
   rcu_read_lock();
   rcu_list_for_each(&nmi_list, handler_t cb)
      cb();
   rcu_read_unlock();
}

void register_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_add(&nmi_list, cb);
   spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_remove(cb);
   spin_unlock(&nmi_list_lock);
   synchronize_rcu();
}
```

- rcu_list_for_each() calls rcu_dereference() for each next pointer

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;
void handle_nmi() {
   rcu_read_lock();
   rcu_list_for_each(&nmi_list, handler_t cb)
      cb();
   rcu_read_unlock();
}

void register_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_add(&nmi_list, cb);
   spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_remove(cb);
   spin_unlock(&nmi_list_lock);
   synchronize_rcu();
}
```

NMI handlers in Linux

- Updates are still synchronized with regular synchronization primitives
  - Most of the time: spinlocks

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;
void handle_nmi() {
   rcu_read_lock();
   rcu_list_for_each(&nmi_list, handler_t cb)
      cb();
   rcu_read_unlock();
}

void register_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_add(&nmi_list, cb);
   spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_remove(cb);
   spin_unlock(&nmi_list_lock);
   synchronize_rcu();
}
```

NMI handlers in Linux

- Removal from the list waits for all RCU sections to complete

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;
void handle_nmi() {
   rcu_read_lock();
   rcu_list_for_each(&nmi_list, handler_t cb)
      cb();
   rcu_read_unlock();
}

void register_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_add(&nmi_list, cb);
   spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_remove(cb);
   spin_unlock(&nmi_list_lock);
   synchronize_rcu();
}
```

NMI handlers in Linux

- What did we get?

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;
void handle_nmi() {
   rcu_read_lock();
   rcu_list_for_each(&nmi_list, handler_t cb)
      cb();
   rcu_read_unlock();
}

void register_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_add(&nmi_list, cb);
   spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb) {
   spin_lock(&nmi_list_lock);
   rcu_list_remove(cb);
   spin_unlock(&nmi_list_lock);
   synchronize_rcu();
}
```

NMI handlers in Linux

- What did we get?
  - Concurrent read access to the list without locks

Example 2: Lifetimes

```
void udp_sendmsg(sock_t *sock, msg_t *msg)
{
   ip_options_t *opts;
   char packet[];

   copy_msg(packet, msg);
   rcu_read_lock();
   opts = rcu_dereference(sock->opts);
   if (opts != NULL)
      copy_opts(packet, opts);
   rcu_read_unlock();
   queue_packet(packet);
}

void setsockopt(sock_t *sock, int opt,
            void *arg) {

   if (opt == IP_OPTIONS) {
      ip_options_t *old = sock->opts;
      ip_options_t *new = arg;
      rcu_assign_pointer(&sock->opts, new);
      if (old != NULL)
         call_rcu(kfree, old);
      return;
   }
   /* Handle other opt values */
}
```

Reference Counting

- Ensure that sock->opts are alive while they are being copied
- Normally we would increment a reference counter to prevent concurrent deallocation
- RCU allows us to implement a short critical section without any locks
- What's the drawback?

```
void udp_sendmsg(sock_t *sock, msg_t *msg)
{
   ip_options_t *opts;
   char packet[];

   copy_msg(packet, msg);
   rcu_read_lock();
   opts = rcu_dereference(sock->opts);
   if (opts != NULL)
      copy_opts(packet, opts);
   rcu_read_unlock();
   queue_packet(packet);
}

void setsockopt(sock_t *sock, int opt,
            void *arg) {

   if (opt == IP_OPTIONS) {
      ip_options_t *old = sock->opts;
      ip_options_t *new = arg;
      rcu_assign_pointer(&sock->opts, new);
      if (old != NULL)
         call_rcu(kfree, old);
      return;
   }
   /* Handle other opt values */
}
```

Reference Counting

- Ensure that sock->opts are alive while they are being copied
- Normally we would increment a reference counter to prevent concurrent deallocation
- RCU allows us to implement a short critical section without any locks
- What's the drawback?
  - You cannot hold it for too long or asynchronously

More scalability techniques

Sloppy counters

- Linux relies on reference counting to manage runtime of objects on the heap
- Some objects are accessed frequently (need to acquire and release the counter), but are deallocated infrequently
- MOSBENCH benchmarks encountered bottlenecks in
  - Directory entry objects (dentry), mounted file system objects (vfsmounts), network routing table entries (dst_entrys)

Sloppy counters

- Maintain one global and a set of per-core local counters
- When local-counter is 0 allocate N references by incrementing the global counter by N
- Then locally on each core use these N references

Lock-free algorithms

- Atomic increment and decrement instructions
- xchg() – unconditional atomic exchange (returns the old value)
- Compare-and-swap
  - T cmpxchg(T *ptr, T old, T new);
  - cmpxchg() loads the value pointed to by *ptr and, if it is equal to old, it stores new in its place
  - Otherwise, no store happens
  - The value that was loaded is then returned, regardless of whether it matched old or not.

Example: lock-free singly linked list

```
struct node {
    struct node *next;
    ...
};

struct node *head;

void list_add(struct node *head, struct node *node) {
    node->next = head;
    head = node;
}
```

Example: lock-free singly linked list

```
struct node {
    struct node *next;
    ...
};
struct node *head;


void list_add(struct node *node) {
    struct node *old;
    do {
        old = head;
        node->next = old;
        old = cmpxchg(&head, old, node);
    } while (old != node->next);
}
```

Example: hash table

- Can we build a lock-free hash table?

Example: hash table

ALGORITHM 1: Pseudocode for the `insertOrUpdate` operation

Input: Key k, Data Element d, Update Function $up : Key \times Val \times Val \rightarrow Val$
default: `atomicUpdate(·) = CAS(`$current$, $up(k, current.data, d)$`)`
Output: Boolean `true` when a new key was inserted, `false` if an update occurred

```
1  i = h(k);
2  while true do
3      i = i % c;
4      current = table[i];
5      if current.key == empty_key then                    // Key is not present yet . . .
6          if table[i].CAS(current, ⟨k, d⟩) then return true;
7          else continue;                                  // retry at same position
8      else if current.key == k then                       // Same key already present . . .
9          if table[i].atomicUpdate(current, d, up) then return false;
10         else continue;                                  // retry at same position
11     i++;
```

Conclusion

- What RCU is good for?
- Read-heavy workload
- Updates are rare
  - synchronize_rcu is slow
- System call example:
  - You acquire a lock every time you execute a system call
  - But really the table might never change
- What if you need fast updates?
  - Lock-free data structures, careful design

Table update (well, removal)

```
void retract_table()
{
  syscall_t *local_table;

  spin_lock(&table_lock);
  local_table = table;
  rcu_assign_pointer(&table, NULL);
  spin_unlock(&table_lock);

  synchronize_rcu();
  kfree(local_table);
}
```

Recap: read copy update