250P: Computer Systems Architecture

Lecture 14: Synchronization

Anton Burtsev
March, 2019

Coherence and Synchronization

- Topics: synchronization primitives (Sections 5.4-5.5)

Constructing Locks

- Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data

- A lock surrounding the data/code ensures that only one program can be in a critical section at a time

- The hardware must provide some basic primitives that allow us to construct locks with different properties

- Lock algorithms assume an underlying cache coherence mechanism – when a process updates a lock, other processes will eventually see the update

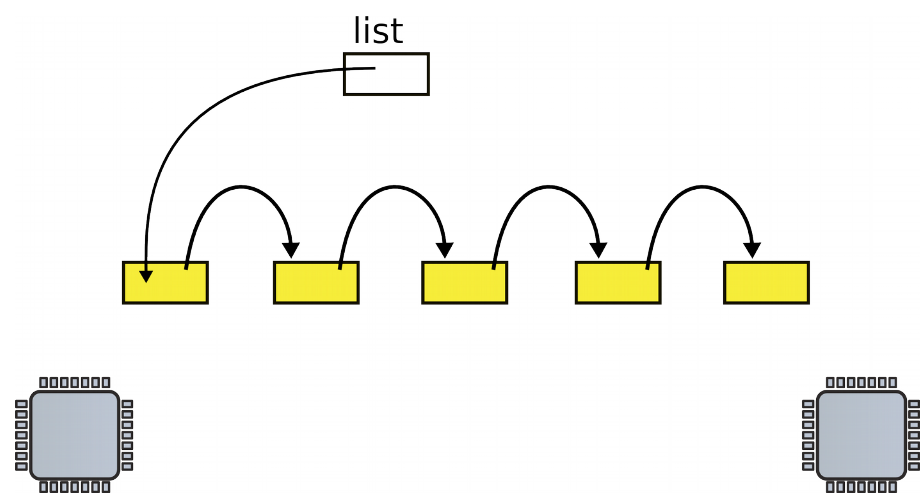
Race conditions

- Example:
  - Disk driver maintains a list of outstanding requests
  - Each process can add requests to the list

```c
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
   struct list *l;

   l = malloc(sizeof *l);
   l->data = data;
   l->next = list;
   list = l;
}
```

List implementation (no locks)

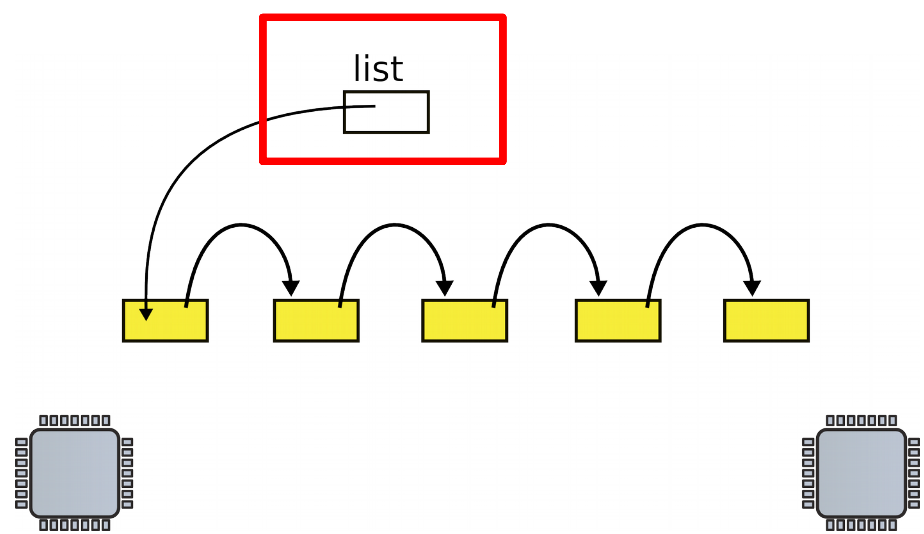
- List
  - One data element
  - Pointer to the next element



List implementation (no locks)

```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
   struct list *l;

   l = malloc(sizeof *l);
   l->data = data;
   l->next = list;
   list = l;
}
```

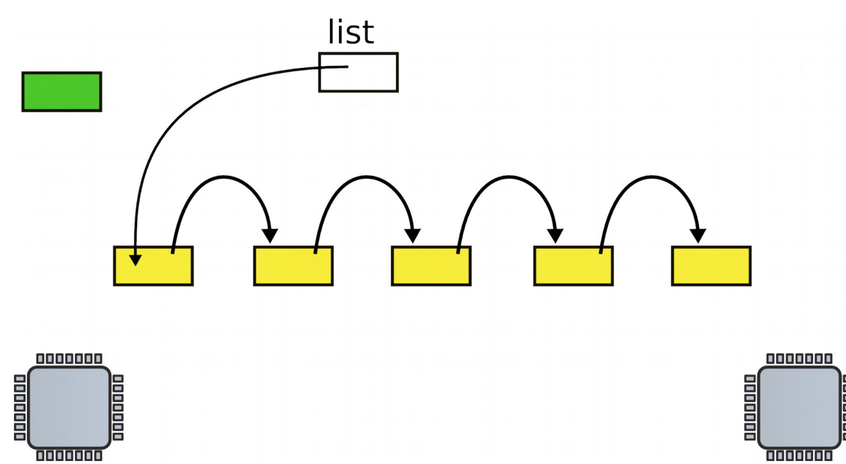
- Global head



List implementation (no locks)

```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

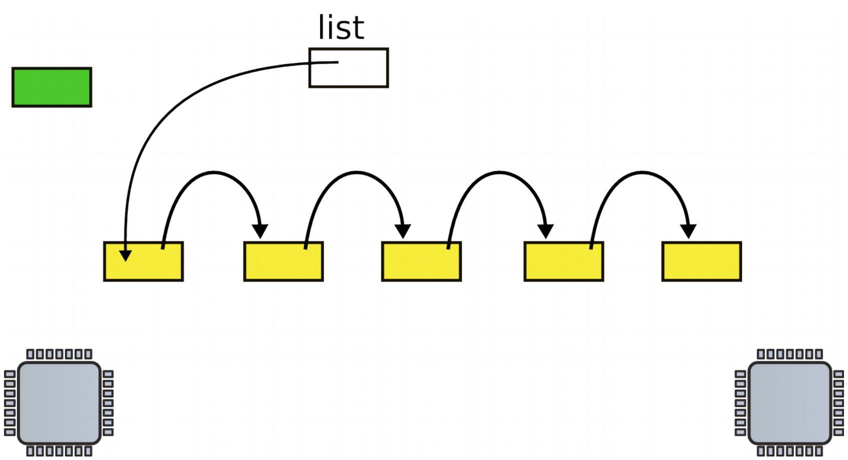
- Insertion
  - Allocate new list element



List implementation (no locks)

```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

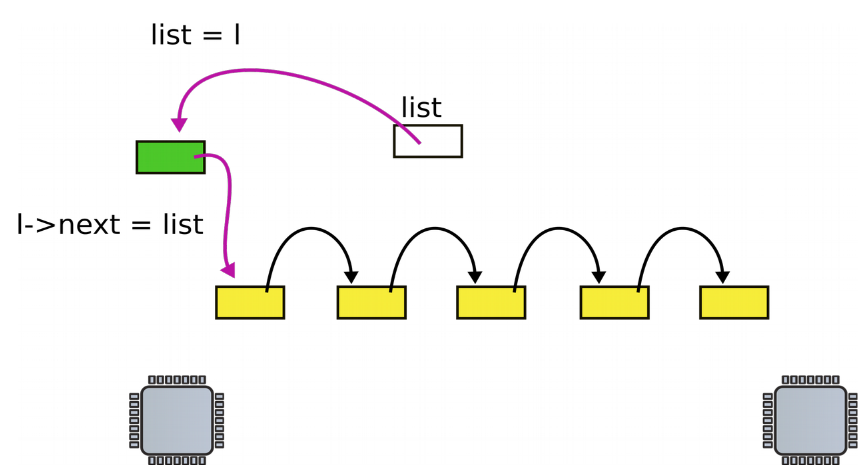
- Insertion
  - Allocate new list element
  - Save data into that element



List implementation (no locks)

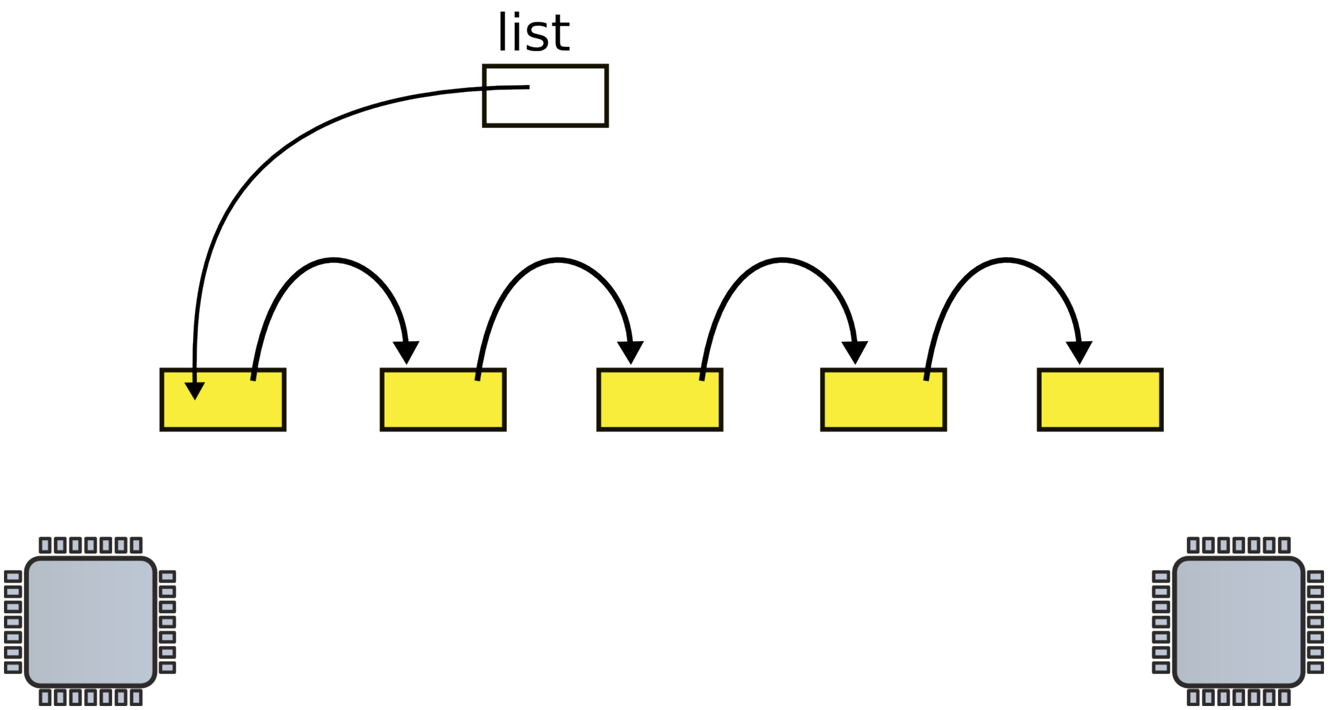
```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
   struct list *l;

   l = malloc(sizeof *l);
   l->data = data;
   l->next = list;
   list = l;
}
```

- Insertion
  - Allocate new list element
  - Save data into that element
  - Insert into the list



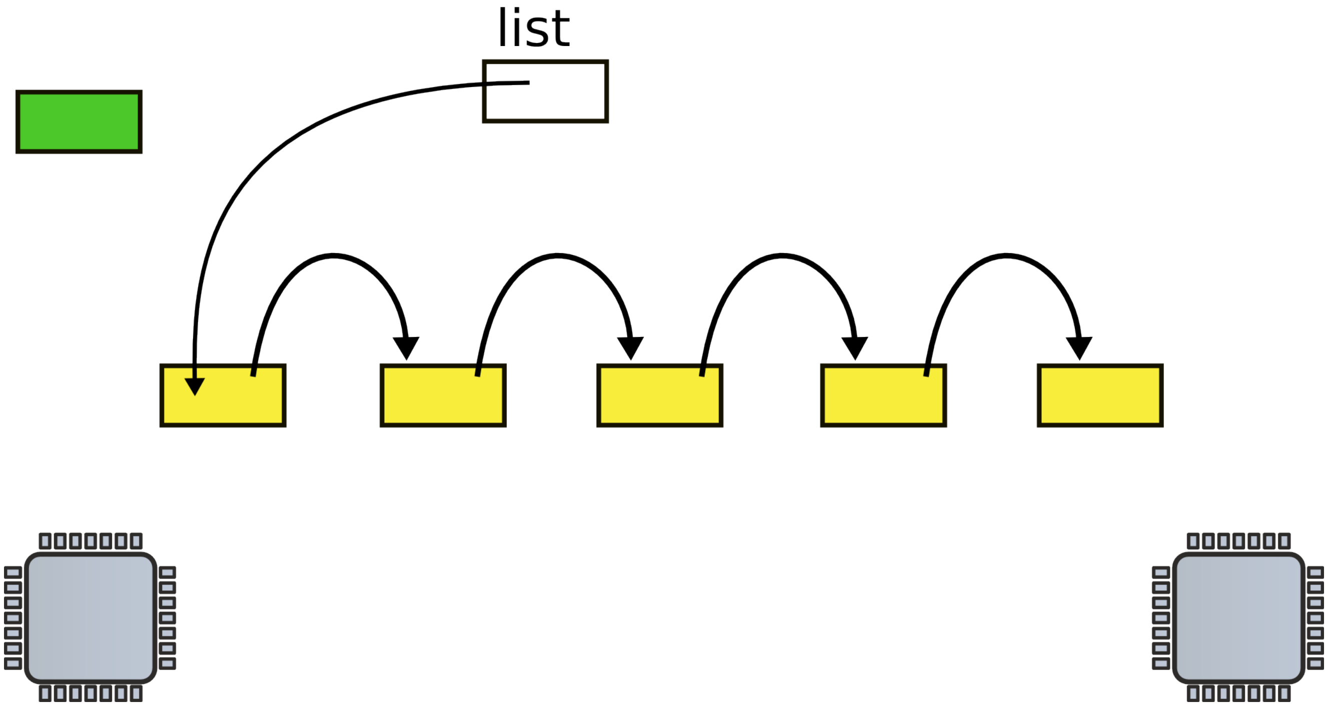
Now what happens when two CPUs access the same list

Request queue (e.g. pending disk requests)

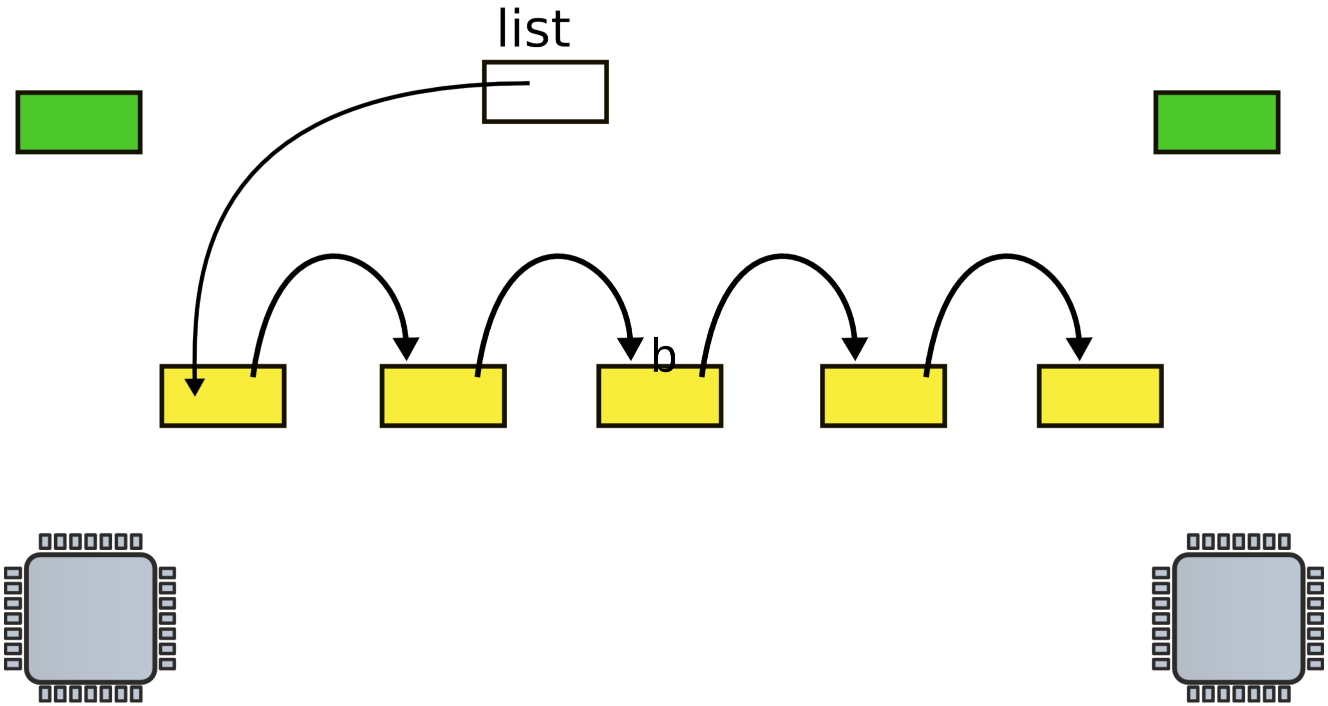


• Linked list, list is pointer to the first element

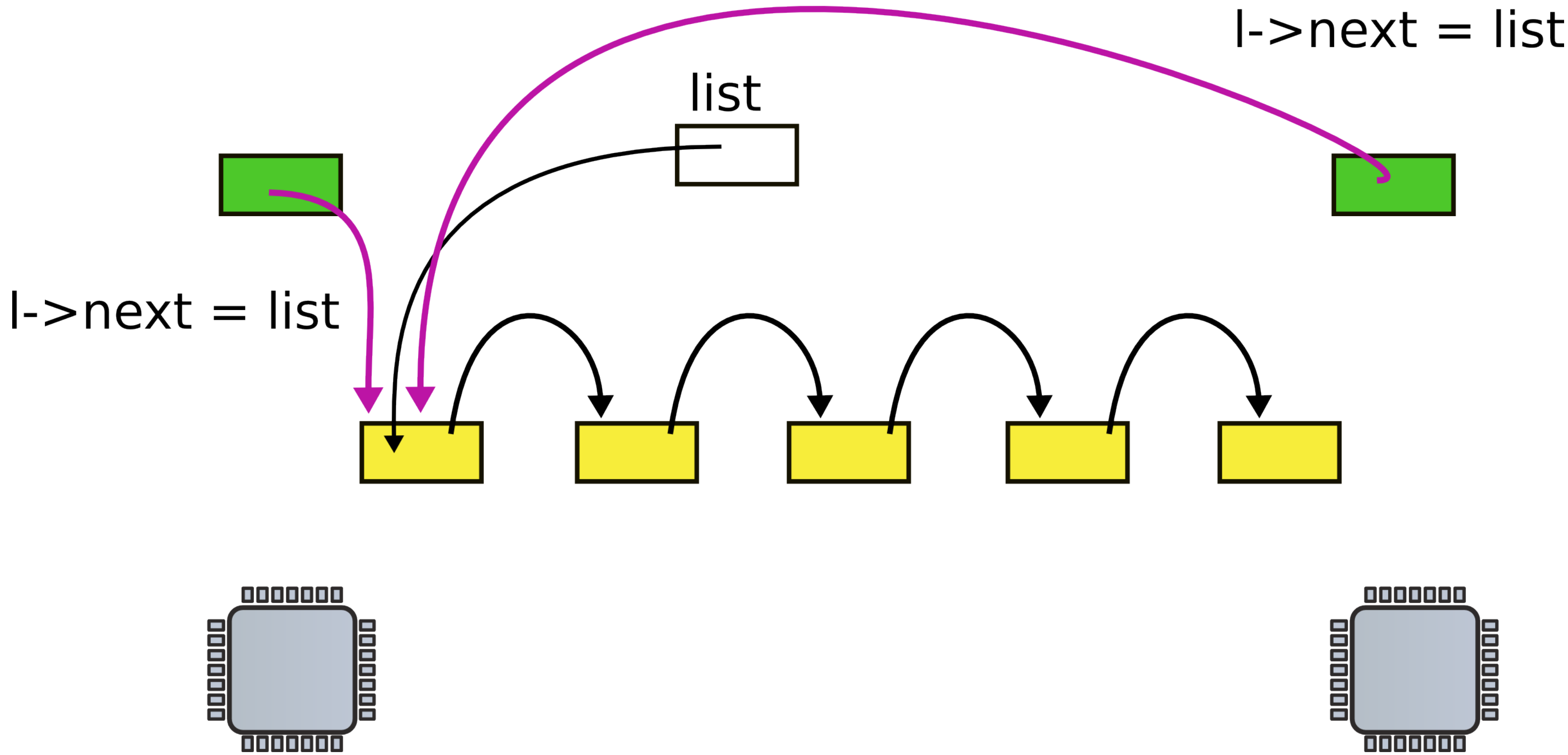
CPU1 allocates new request



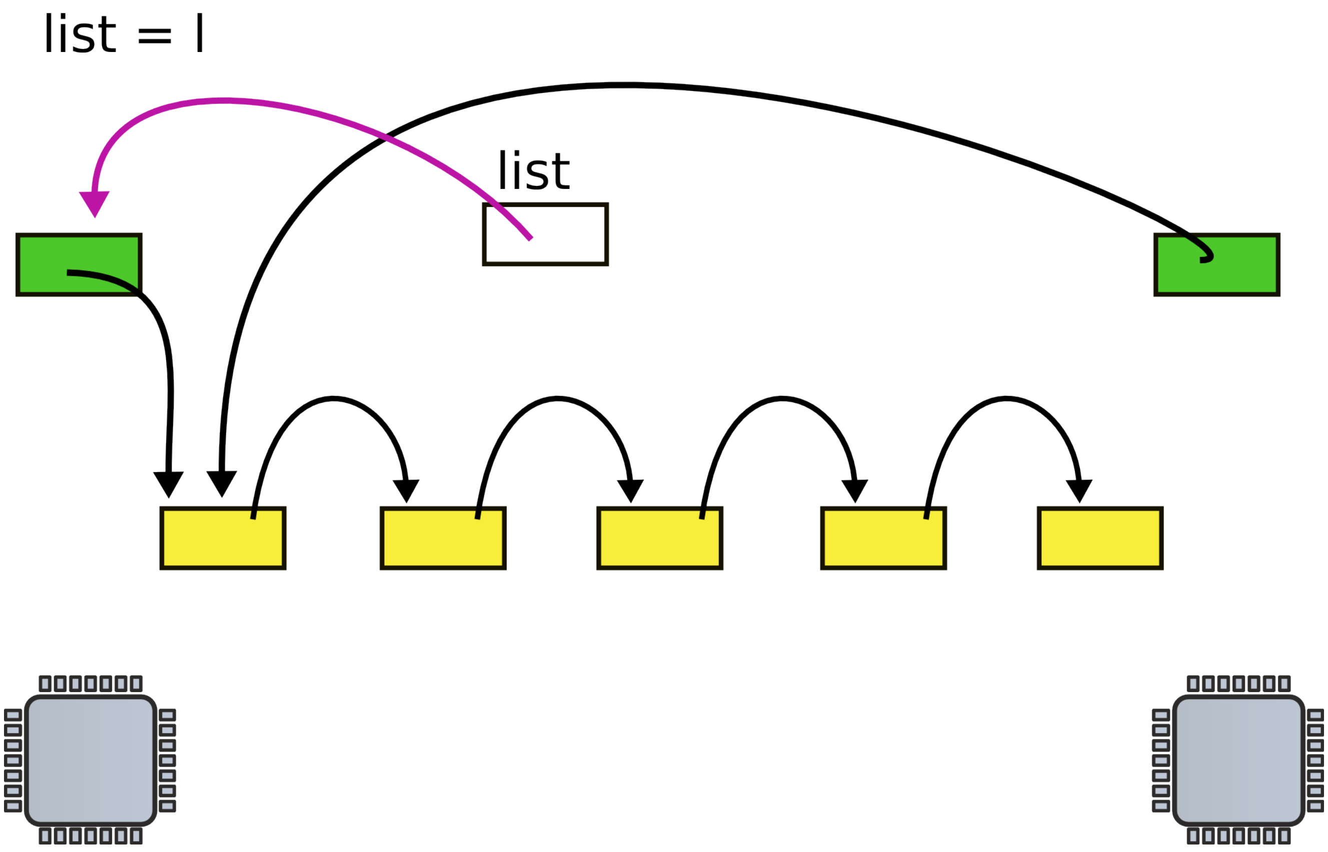
CPU2 allocates new request



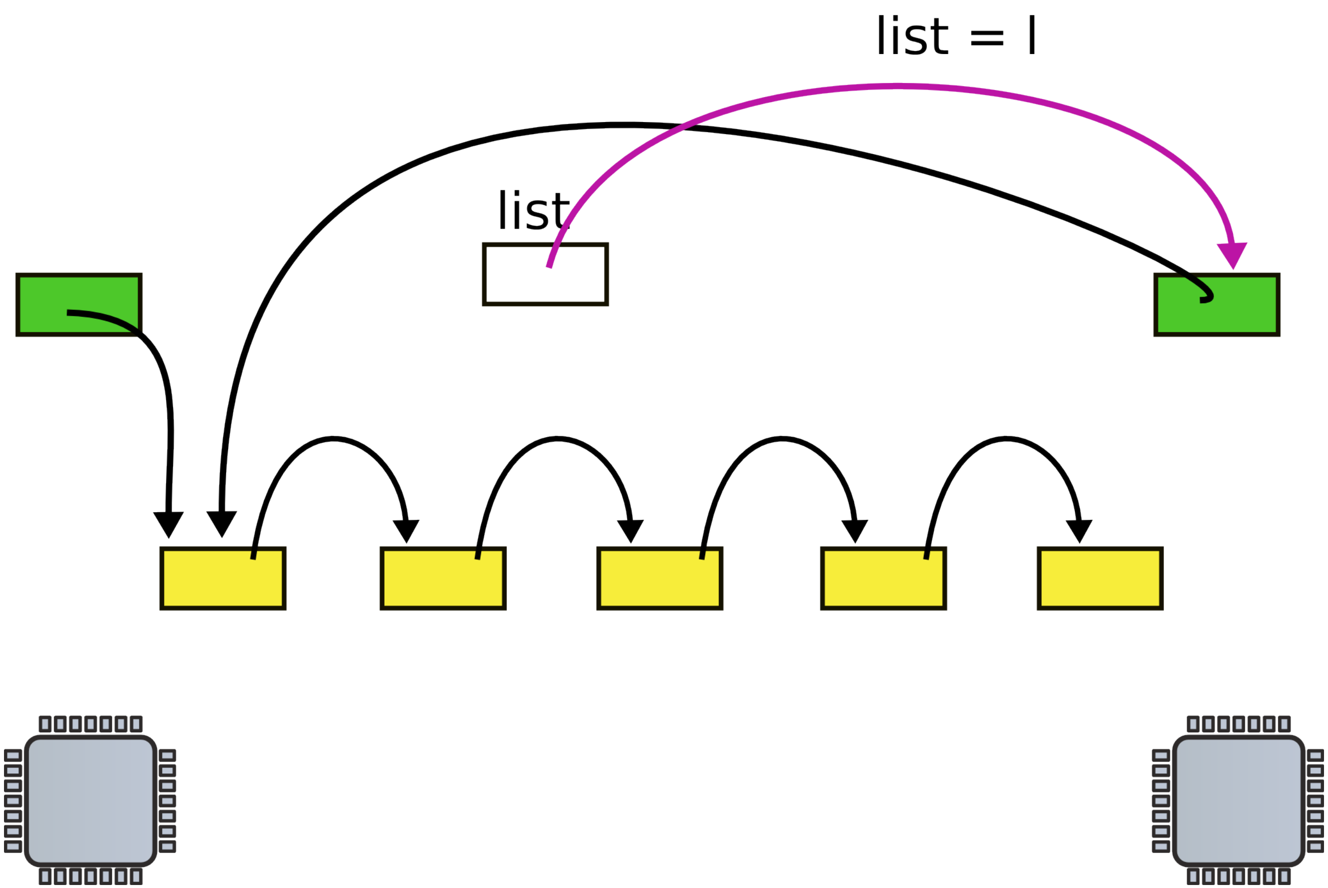
CPUs 1 and 2 update next pointer



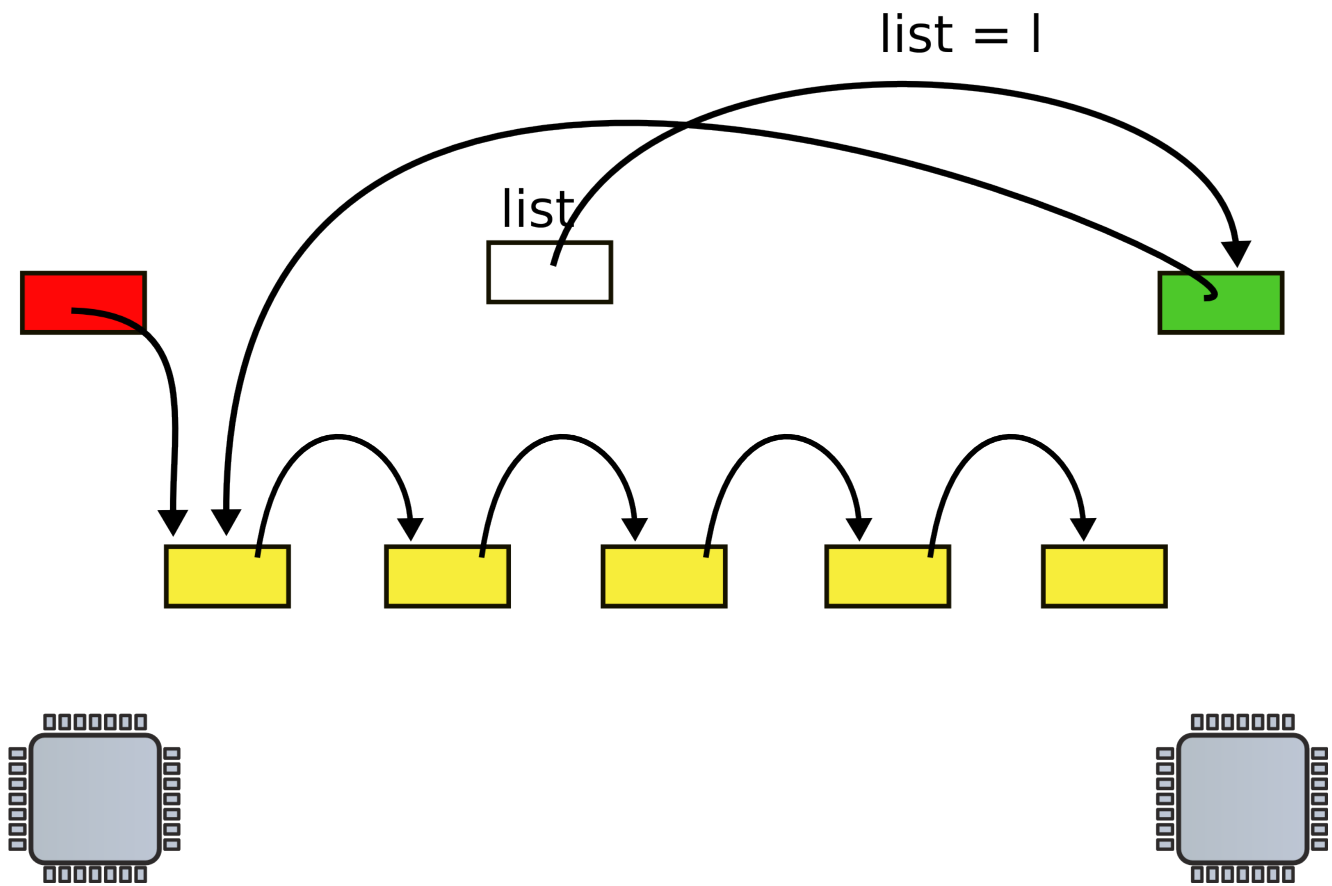
CPU1 updates head pointer



CPU2 updates head pointer



State after the race (red element is lost)



Mutual exclusion

- Only one CPU can update list at a time

List implementation with locks

```
struct list {
  int data;
  struct list *next;
};
struct list *list = 0;
struct lock listlock;
insert(int data)
{
  struct list *l;
  l = malloc(sizeof *l);
  acquire(&listlock);
  l->data = data;
  l->next = list;
  list = l;
  release(&listlock);
}
```

- Critical section

- How can we implement acquire()?

Spinlock

```
void
acquire(struct spinlock *lk)
{
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}
```

- Spin until lock is 0
- Set it to 1

Still incorrect

```
21 void
22 acquire(struct spinlock *lk)
23 {
24   for(;;) {
25     if(!lk->locked) {
26       lk->locked = 1;
27       break;
28     }
29   }
30 }
```

- Two CPUs can reach line #25 at the same time
  - See not locked, and
  - Acquire the lock
- Lines #25 and #26 need to be atomic
  - I.e. indivisible

Synchronization

- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write

- Atomic exchange: swap contents of register and memory

- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory

- acquire: t&s register, location
  bnz register, acquire
  CS
  release: st location, #0

```
acquire:   t&s   register, location
           bnz   register, acquire
           CS
release:   st    location, #0
```

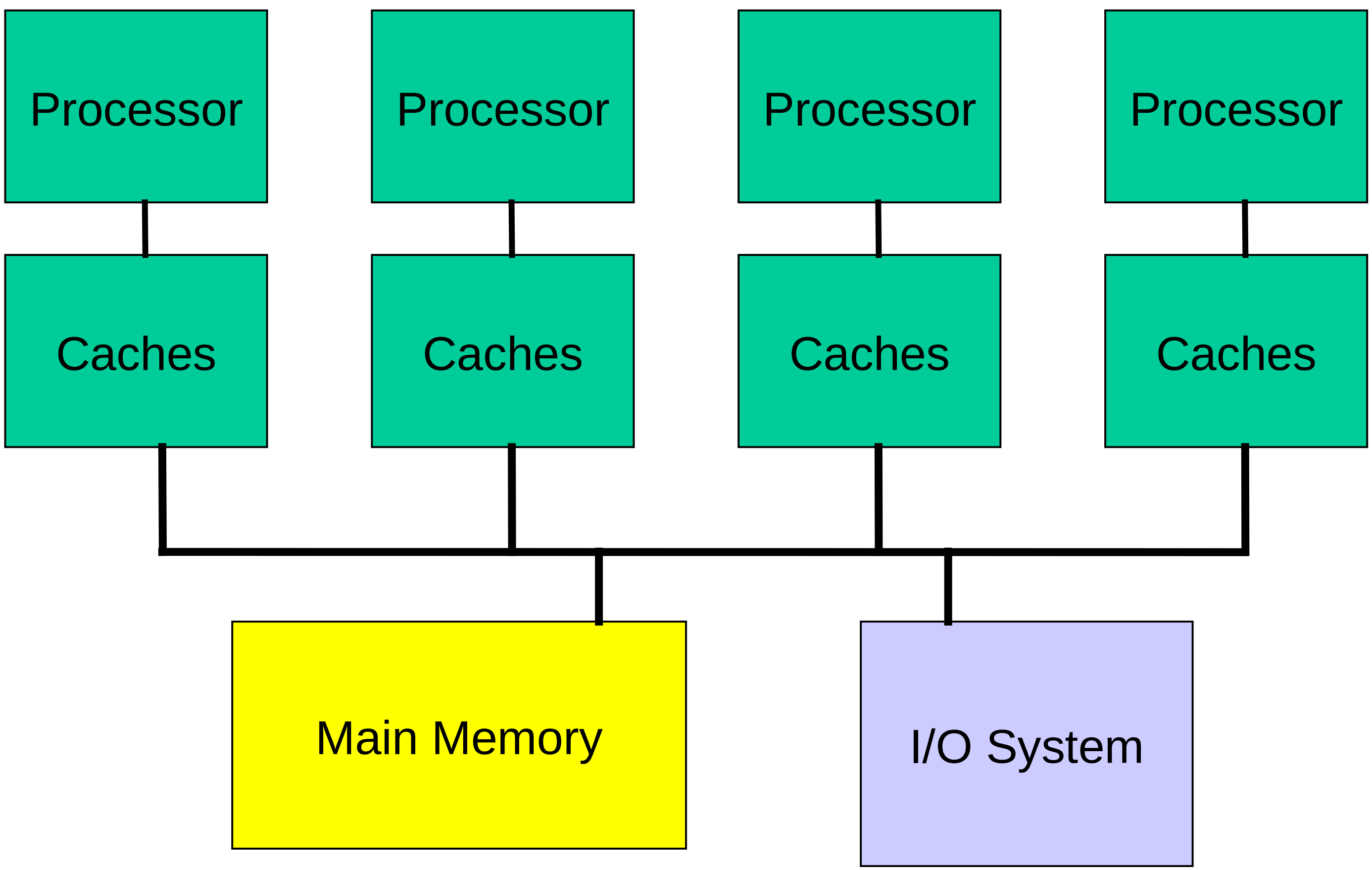
Caching Locks

- Spin lock: to acquire a lock, a process may enter an infinite loop that keeps attempting a read-modify till it succeeds

- If the lock is in memory, there is heavy bus traffic → other processes make little forward progress

- Locks can be cached:
  - cache coherence ensures that a lock update is seen by other processors
  - the process that acquires the lock in exclusive state gets to update the lock first
  - spin on a local copy – the external bus sees little traffic

SMP/UMA/Centralized Memory Multiprocessor



Coherence Traffic for a Lock

- If every process spins on an exchange, every exchange instruction will attempt a write → many invalidates and the locked value keeps changing ownership

- Hence, each process keeps reading the lock value – a read does not generate coherence traffic and every process spins on its locally cached copy

- When the lock owner releases the lock by writing a 0, other copies are invalidated, each spinning process generates a read miss, acquires a new copy, sees the 0, attempts an exchange (requires acquiring the block in exclusive state so the write can happen), first process to acquire the block in exclusive state acquires the lock, others keep spinning

Test-and-Test-and-Set

- lock:

```
lock:   test   register, location
        bnz    register, lock
        t&s    register, location
        bnz    register, lock
        CS
        st     location, #0
```

Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility
- LL: read a value and update a table indicating you have read this address, then perform any amount of computation
- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was "effectively" atomic)
- SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set

Spin Lock with Low Coherence Traffic

```
lockit:  LL      R2, 0(R1)    ; load linked, generates no coherence traffic
         BNEZ    R2, lockit   ; not available, keep spinning
         DADDUI  R2, R0, #1   ; put value 1 in R2
         SC      R2, 0(R1)    ; store-conditional succeeds if no one
                              ; updated the lock since the last LL
         BEQZ    R2, lockit   ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?

Spin Lock with Low Coherence Traffic

```
lockit:  LL      R2, 0(R1)    ; load linked, generates no coherence traffic
         BNEZ    R2, lockit   ; not available, keep spinning
         DADDUI  R2, R0, #1   ; put value 1 in R2
         SC      R2, 0(R1)    ; store-conditional succeeds if no one
                              ; updated the lock since the last LL
         BEQZ    R2, lockit   ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?
  1 write by the releaser + i read-miss requests +
  i responses + 1 write by acquirer + 0 (i-1 failed SCs) +
  i-1 read-miss requests + i-1 responses

Further Reducing Bandwidth Needs

- Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable

-

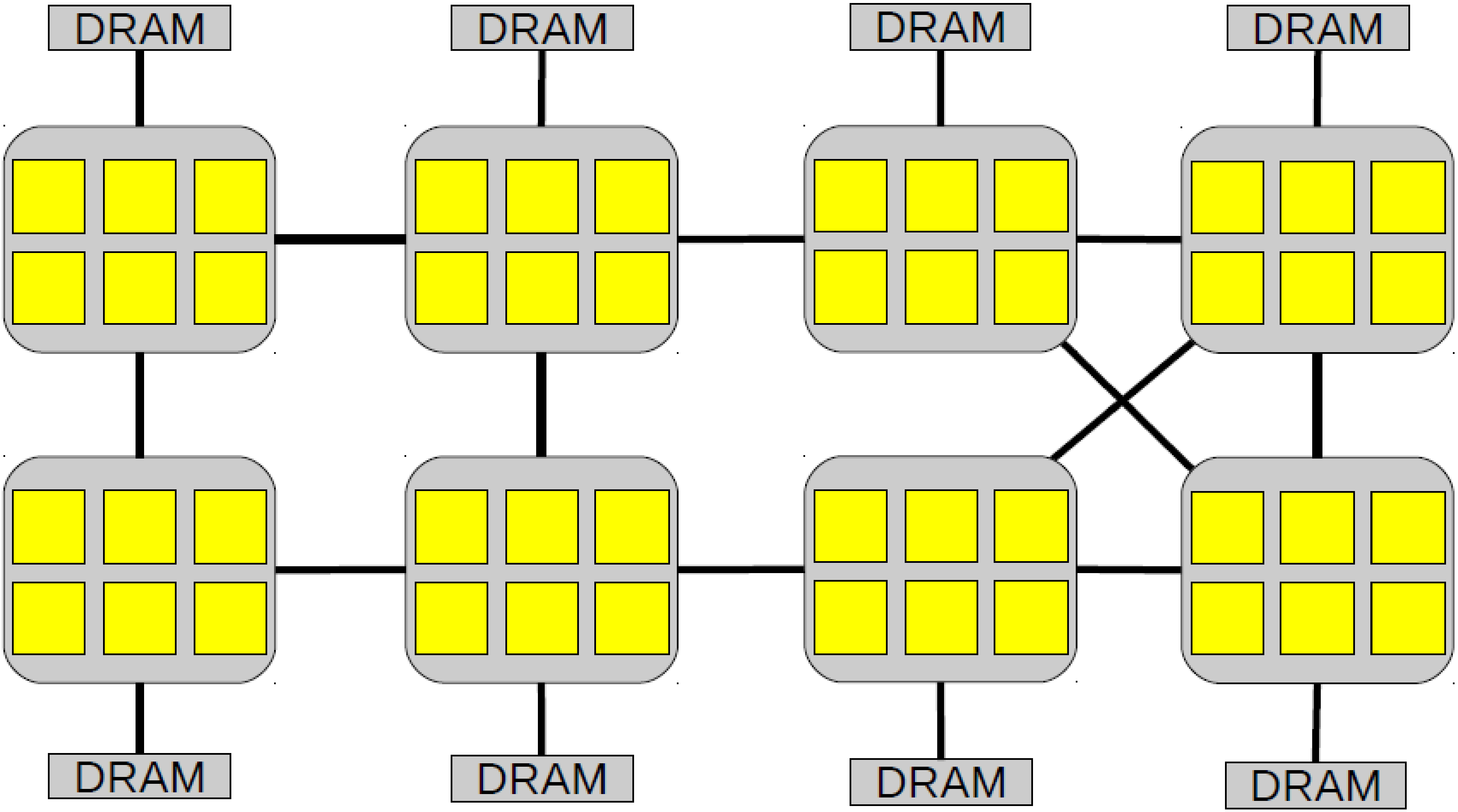
Ticket lock in Linux

```
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 ++;
}
```

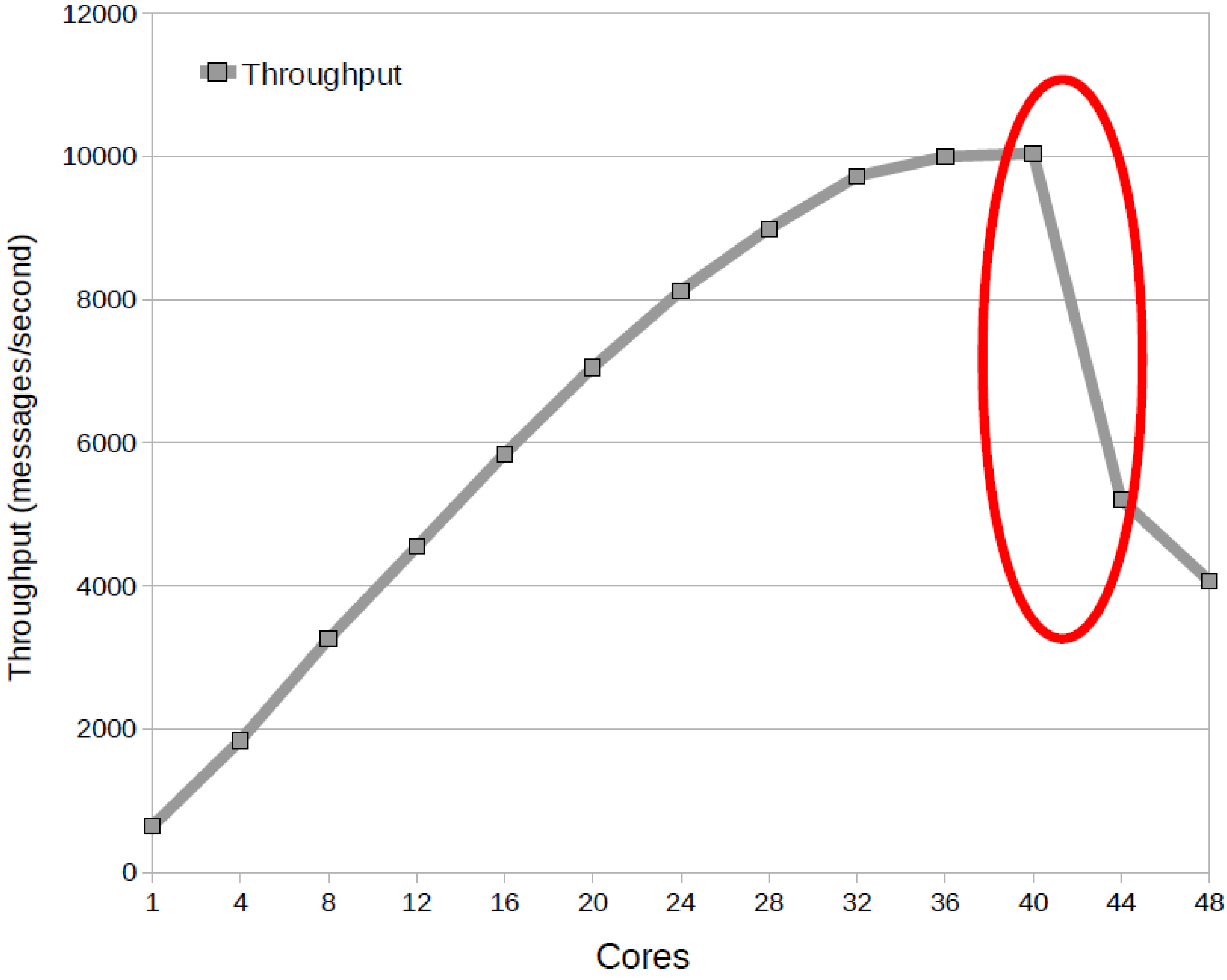
What is really wrong with locks?

- 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

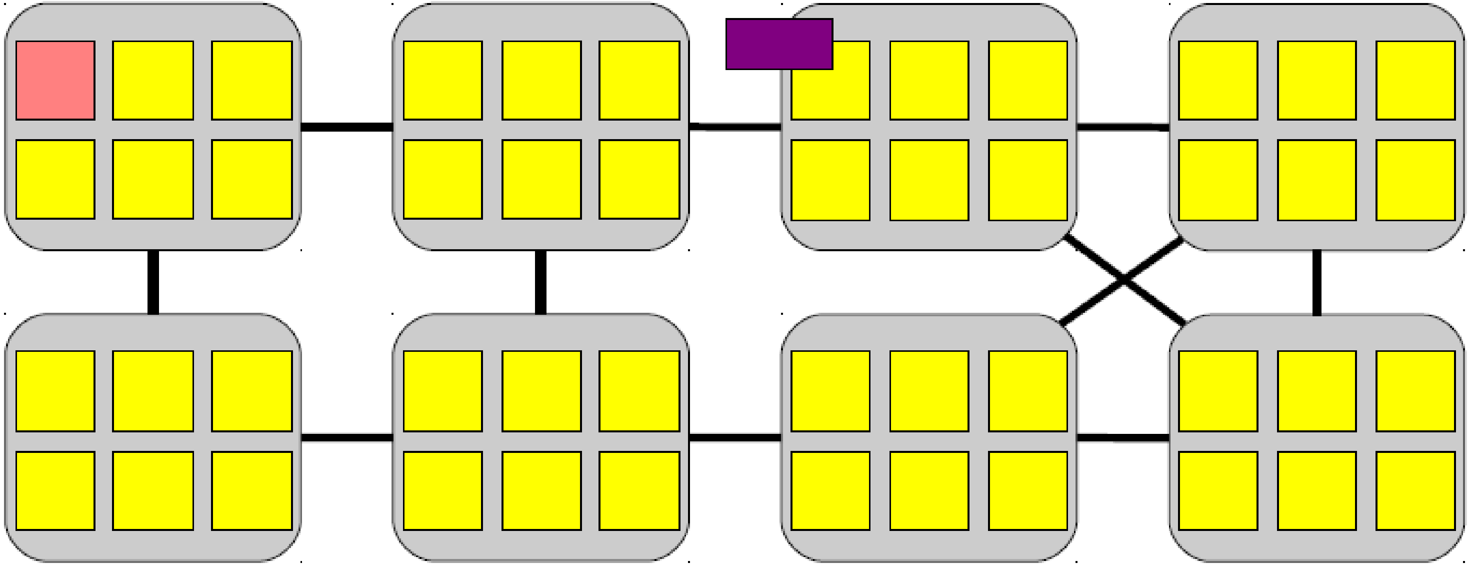
```
struct spinlock_t {
  int current_ticket ;
  int next_ticket ;
}
void spin_lock ( spinlock_t *lock)
{
  int t = atomic_fetch_and_inc (&lock -> next_ticket );
  while (t != lock -> current_ticket )
  ; /* spin */
}
void spin_unlock ( spinlock_t *lock)
{
  lock -> current_ticket ++;
}
```

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;
}
```



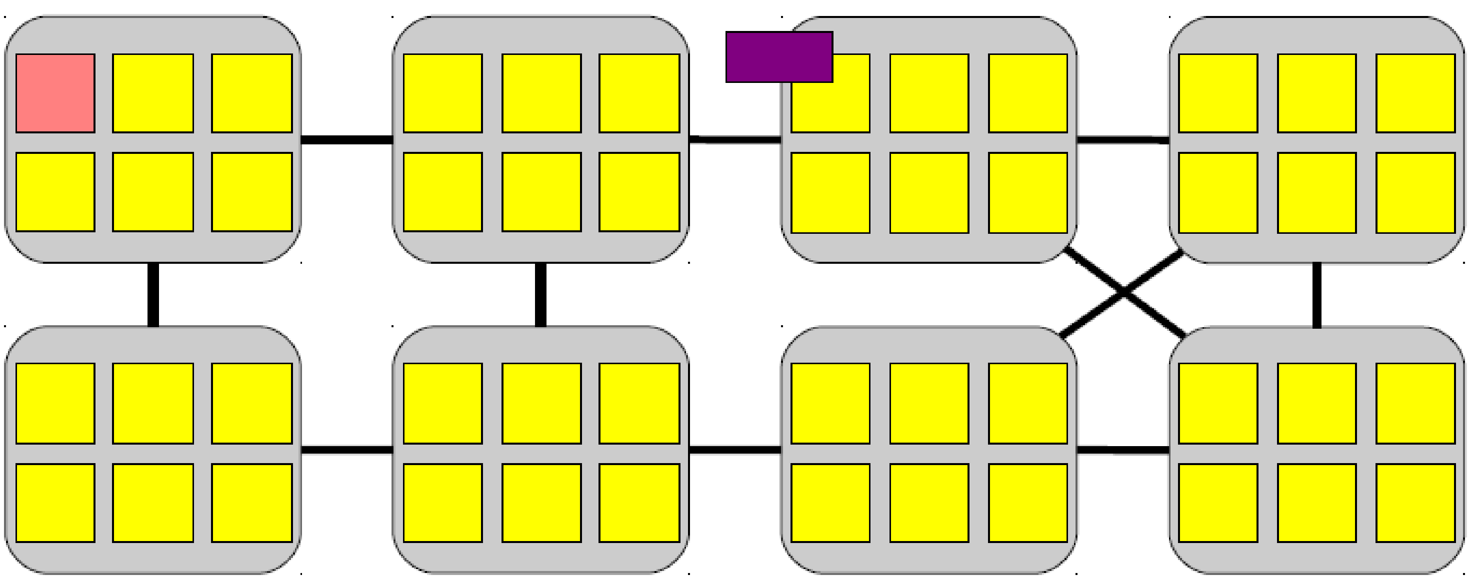
Spin l … entation

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;
}
```



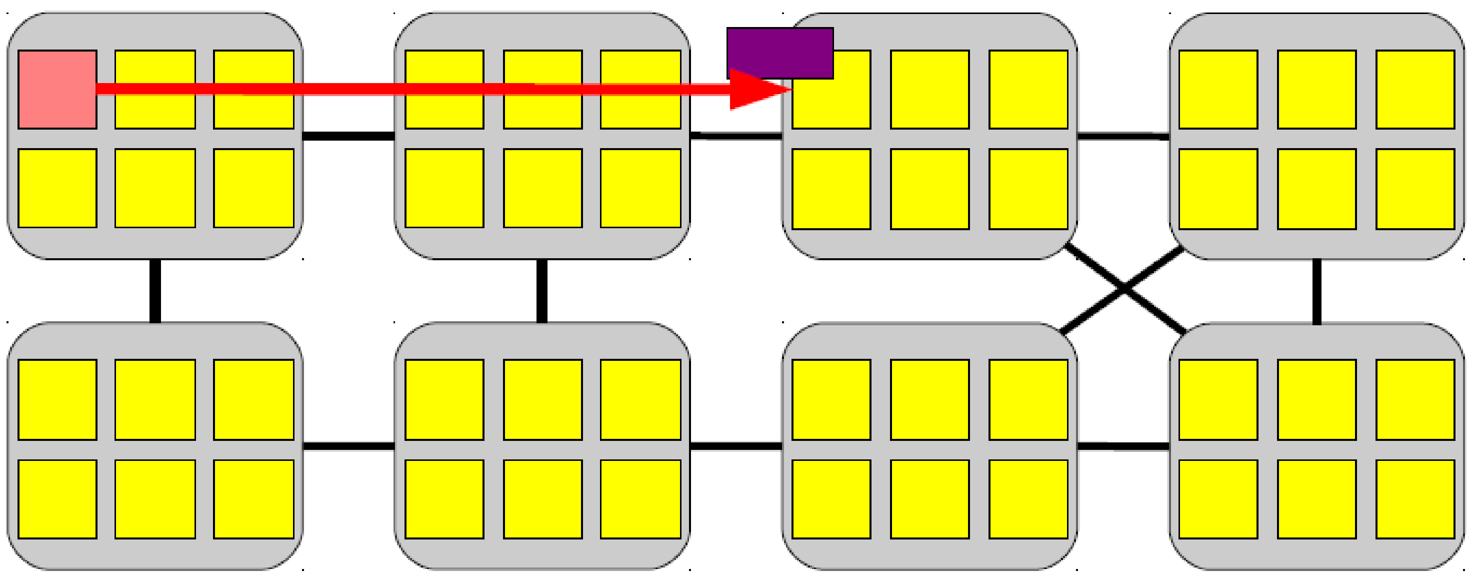
Spin l entation

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;
}
```



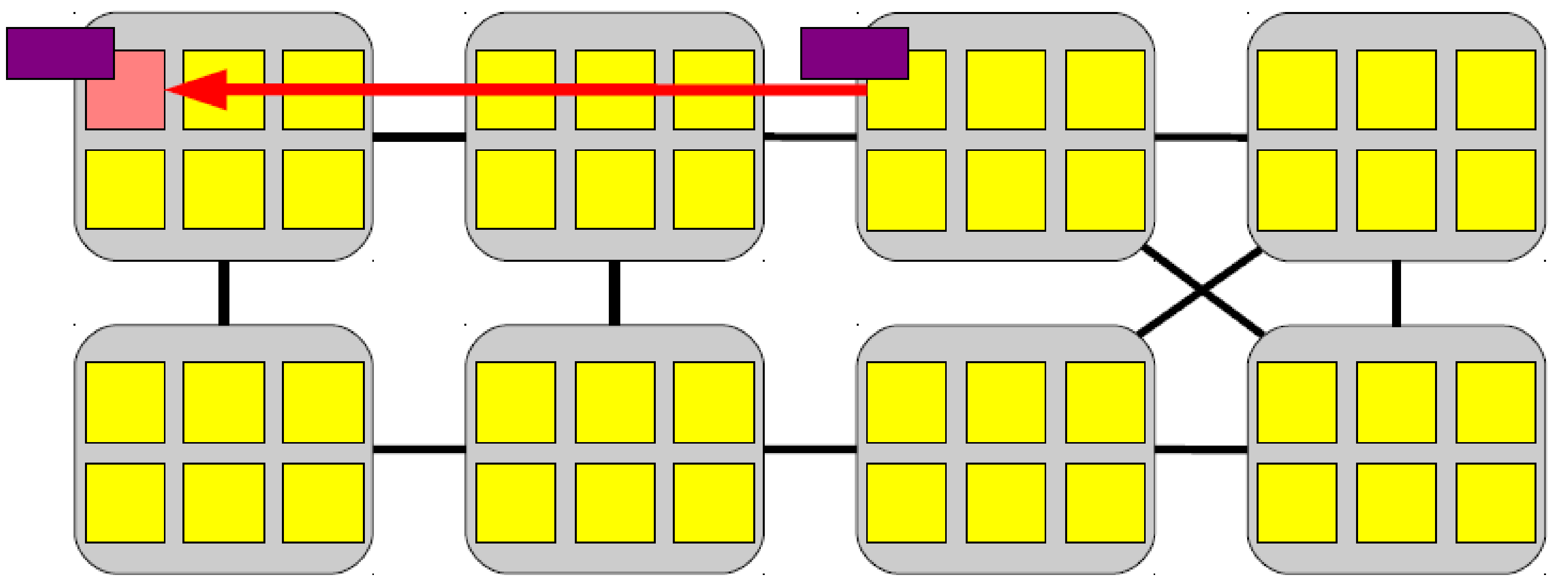
Spin l entation

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;
}
```

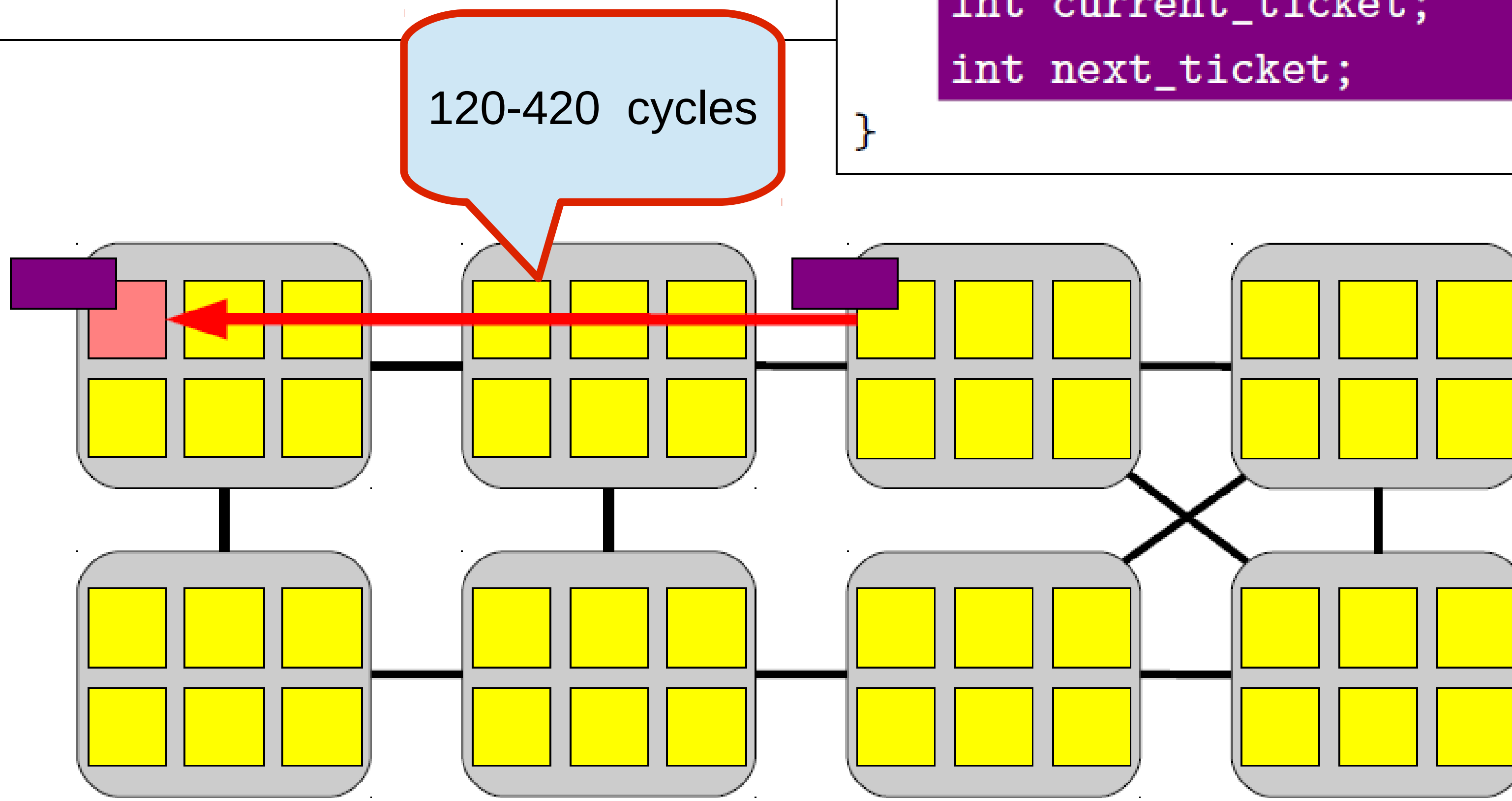


Spin l… entation

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;
}
```



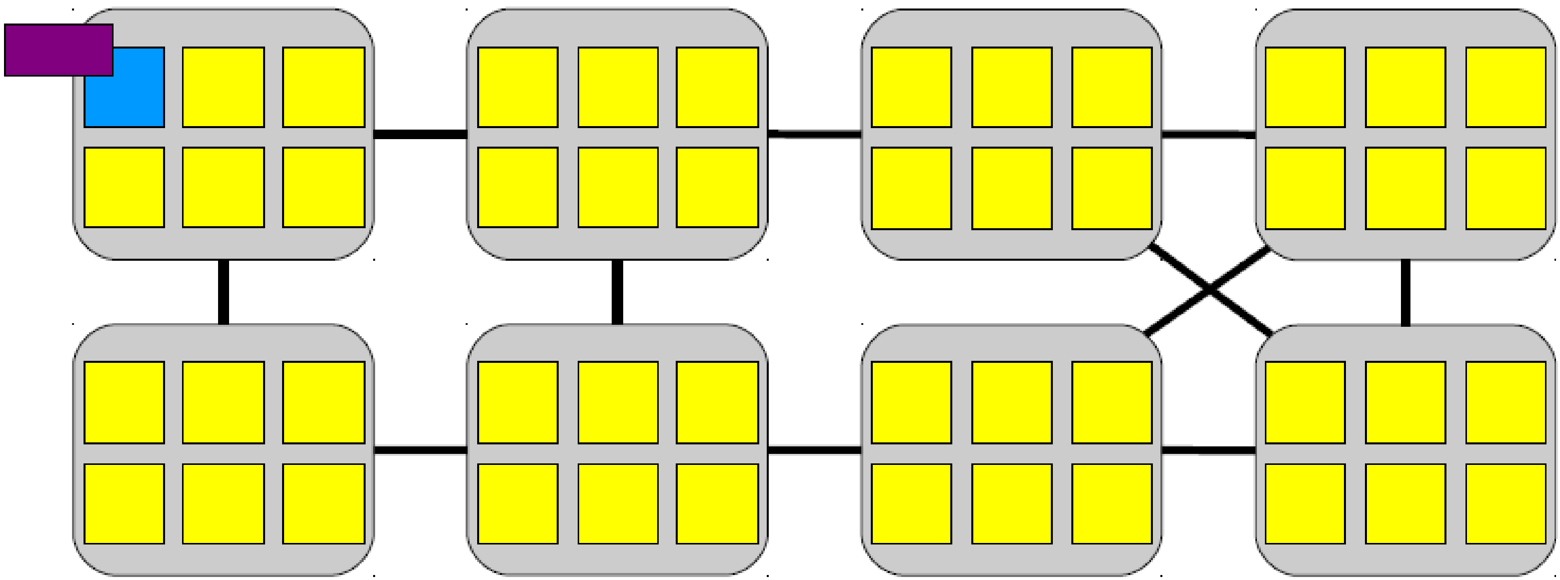
Spin lock impleme

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;
}
```



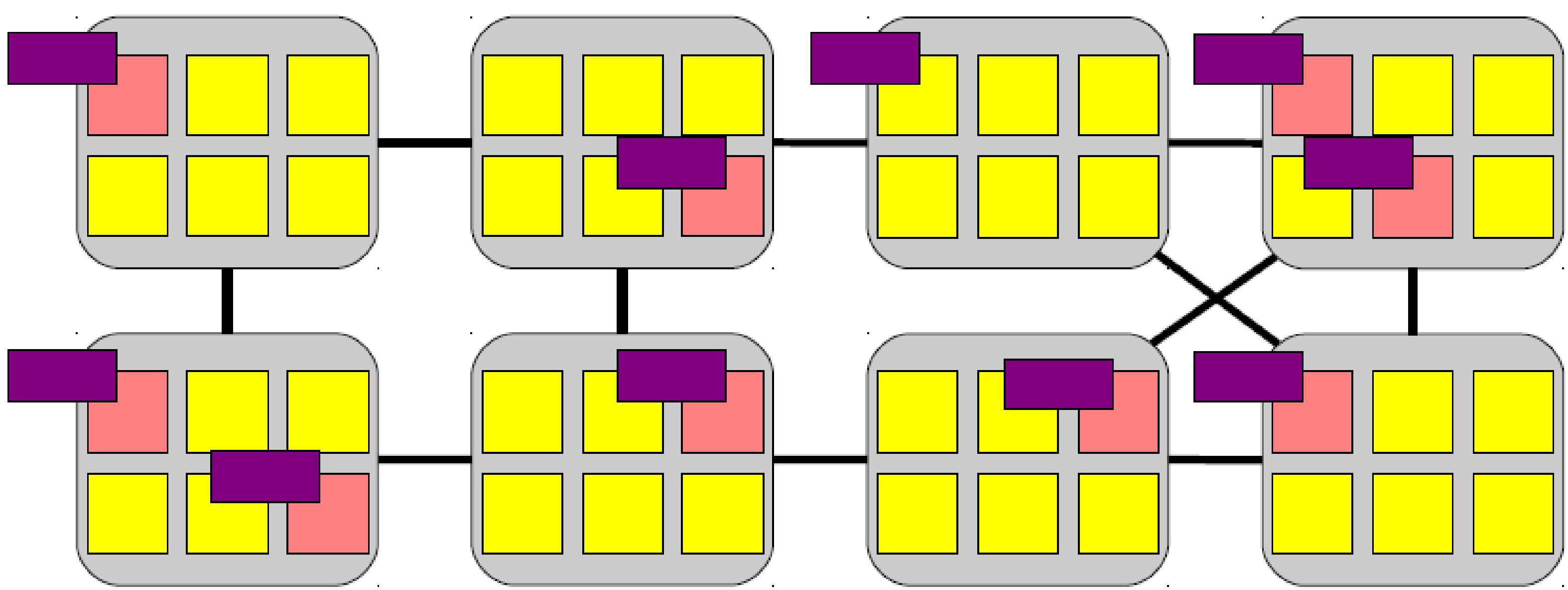
Spin lock implementation

Bunch of cores are spinning

```
void s                         ock)
{
    t = atom  _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;
}
```

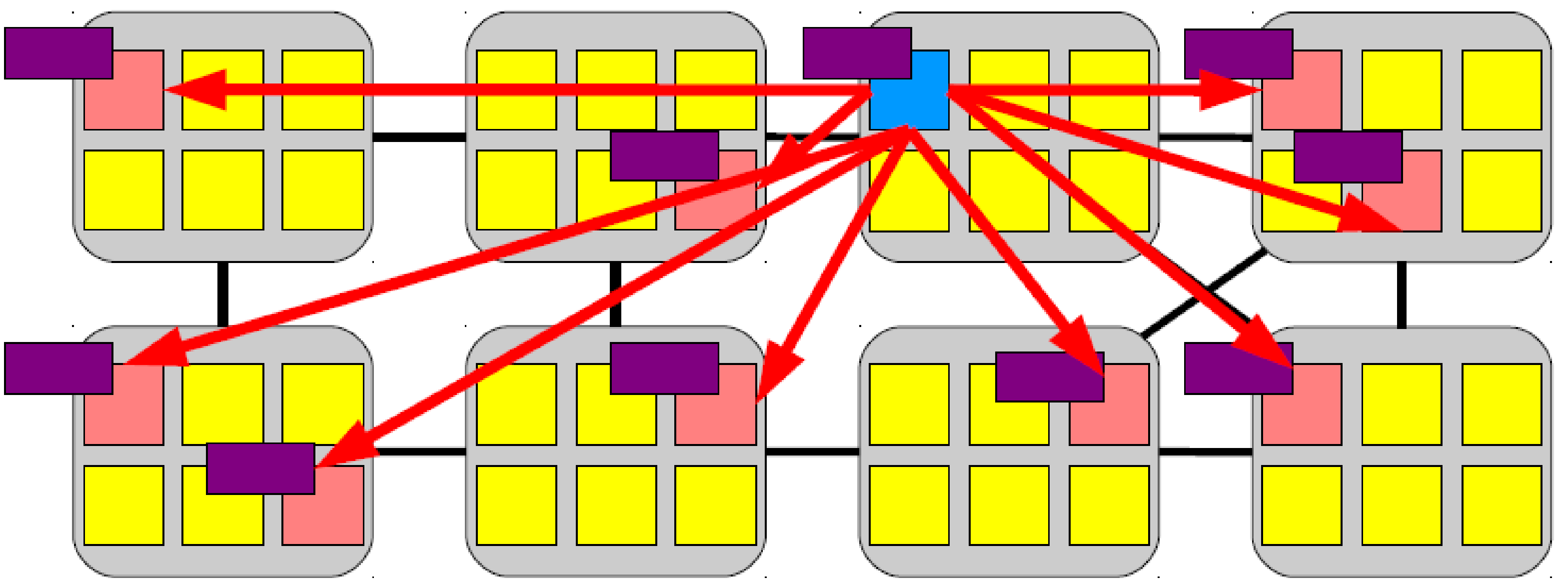


Spin lock implementation

```
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;
}
```

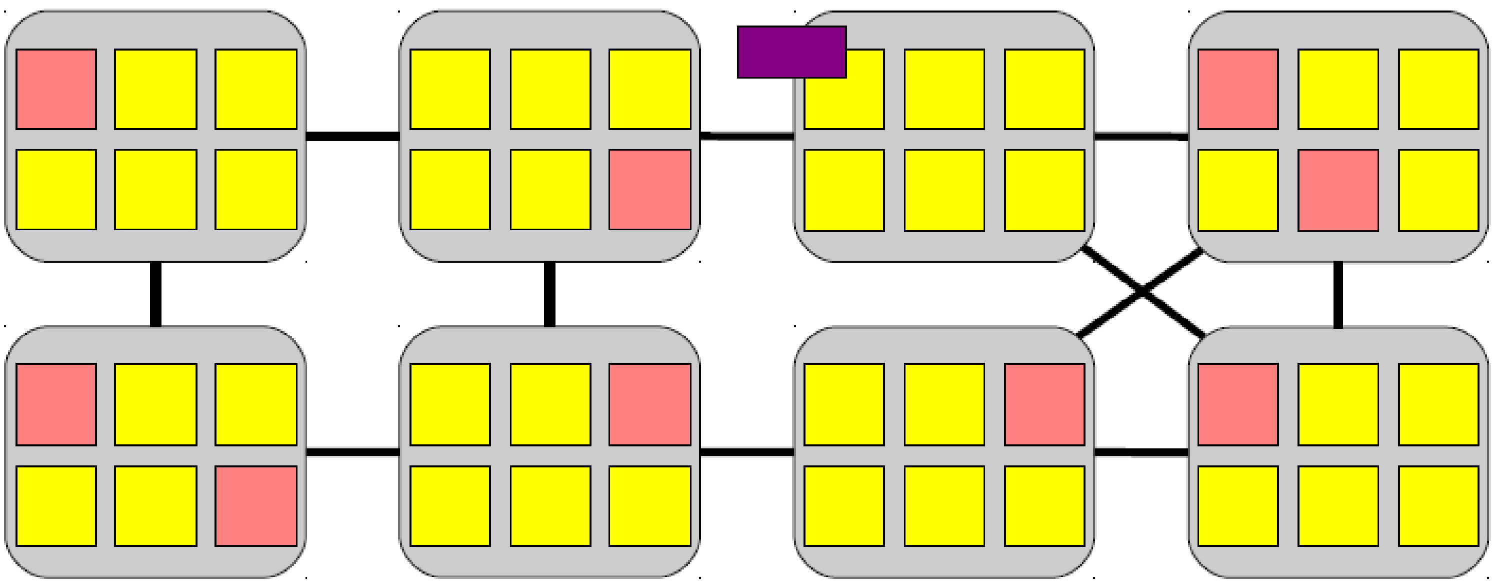


Spin lock implementation

```
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++;

    et;
    next_ticket;
}
```

Cores don't have the value of current_ticket



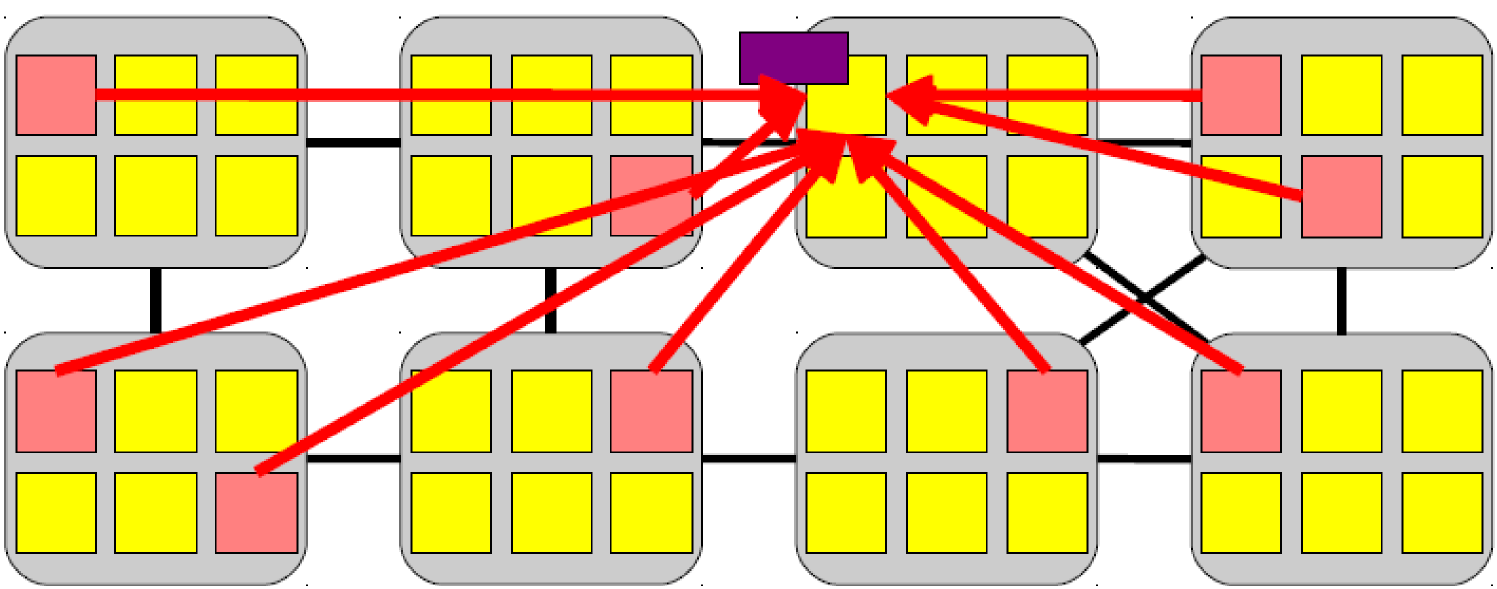
Spin lock implementation

```
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++;
}
```

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

Re-read the value

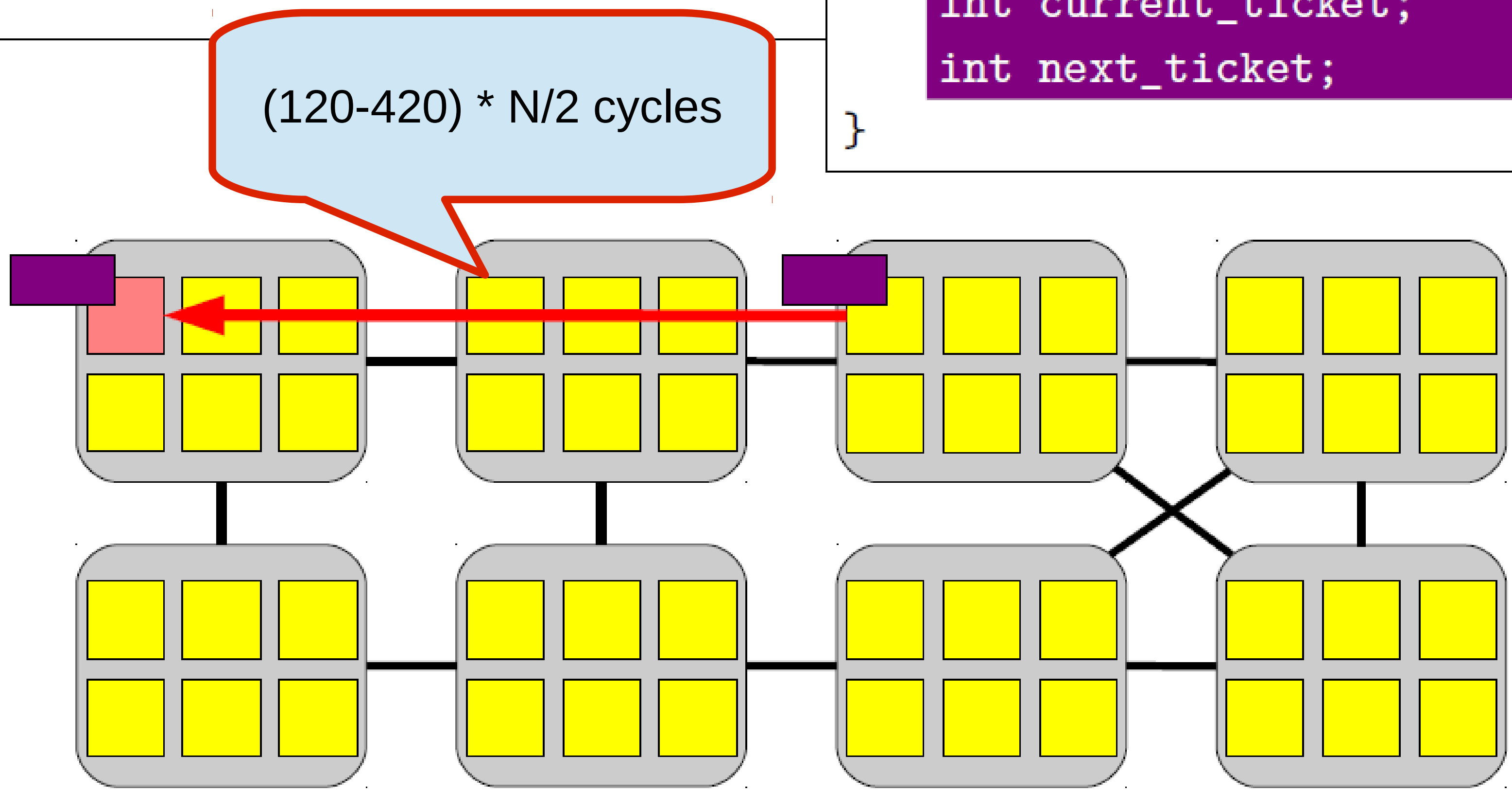


Spin lock implementation

```
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++;
}
```

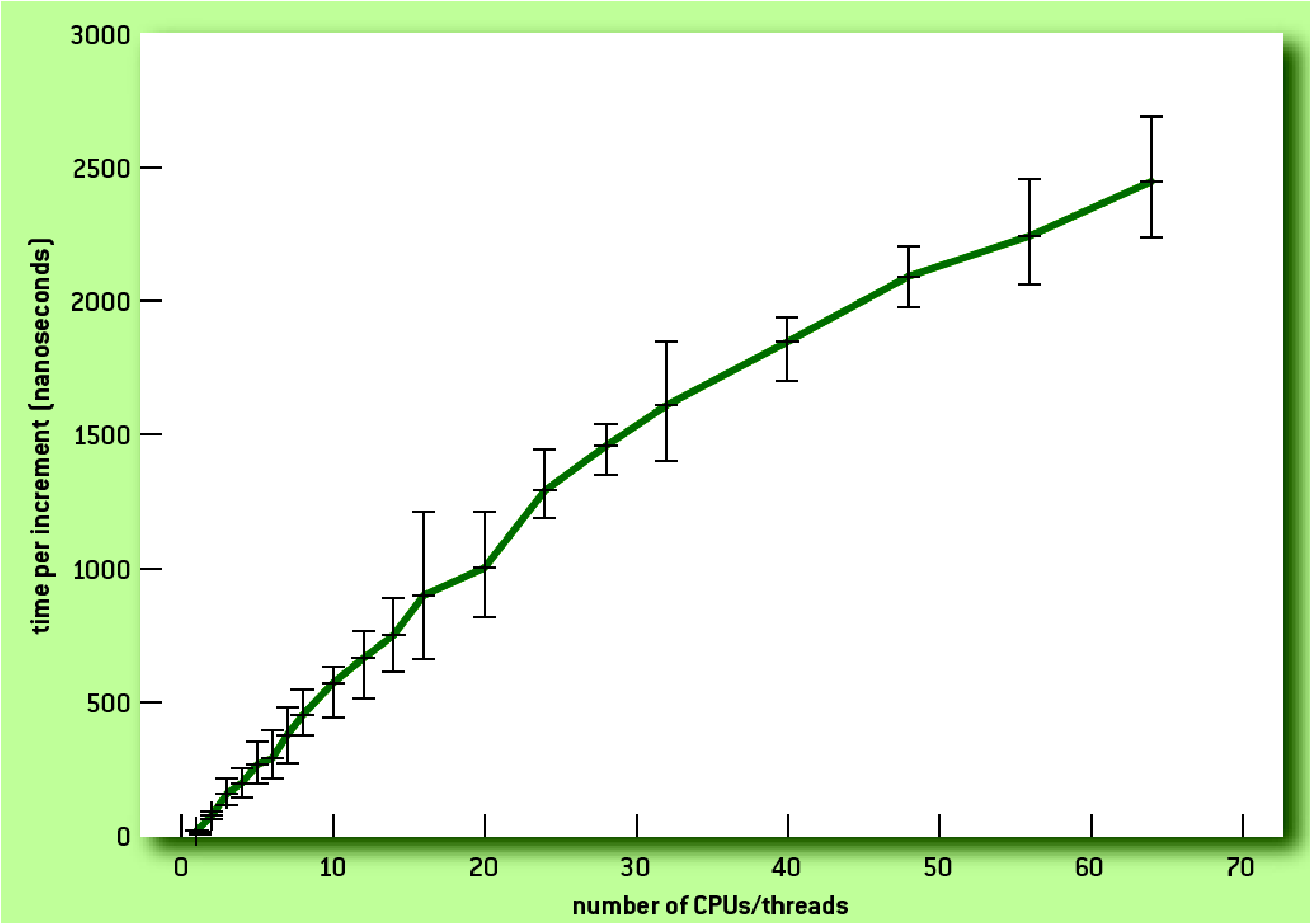
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



- 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



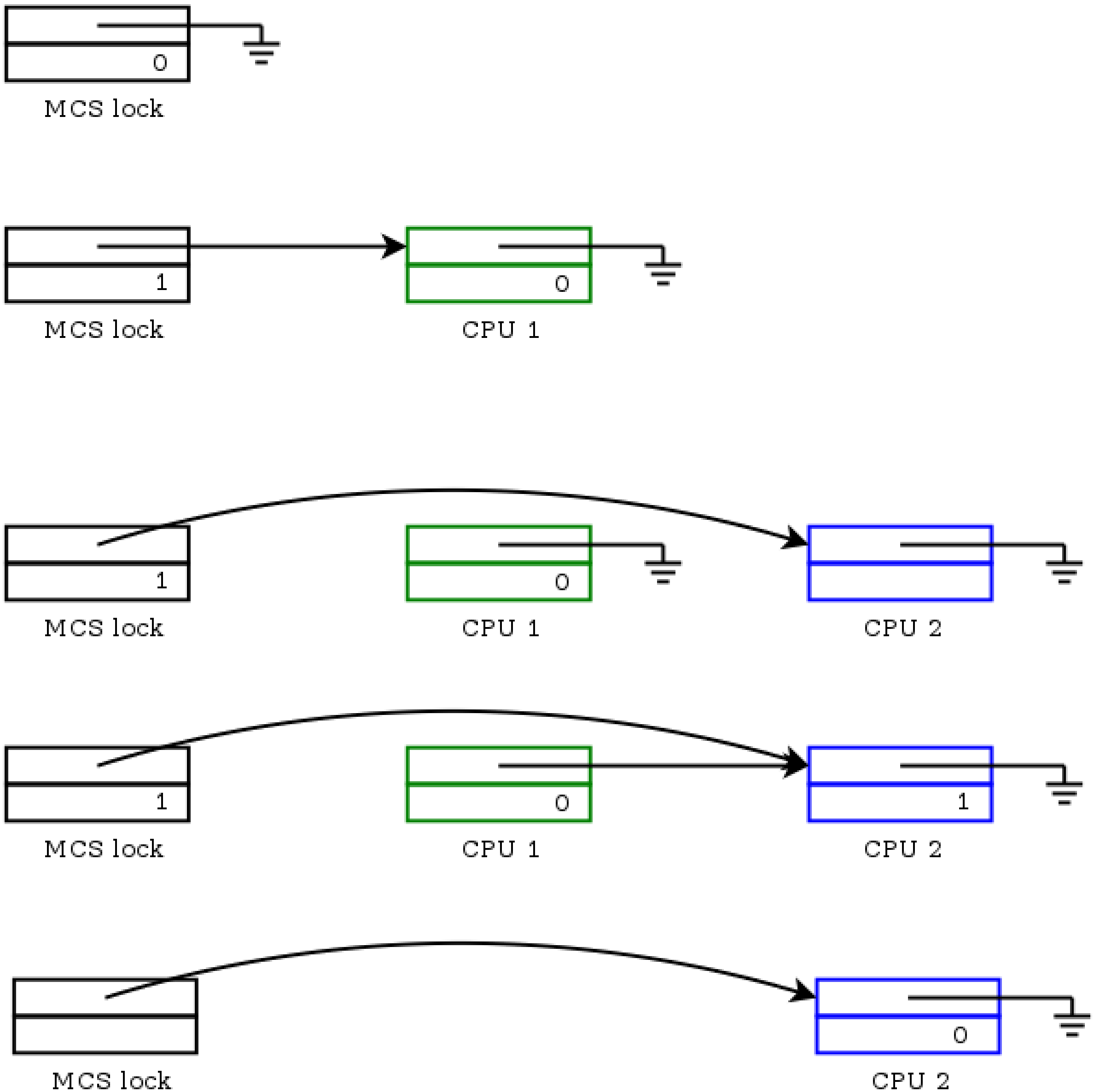
What can we do about it?

Is it possible to build scalable spinlocks?

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) ;
        }
}
```

MCS lock



0
MCS lock
1
0
MCS lock
CPU 1
1
0
MCS lock
CPU 1
CPU 2
1
0
1
MCS lock
CPU 1
CPU 2
0
MCS lock
CPU 2

unlock

```
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;
}
```

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

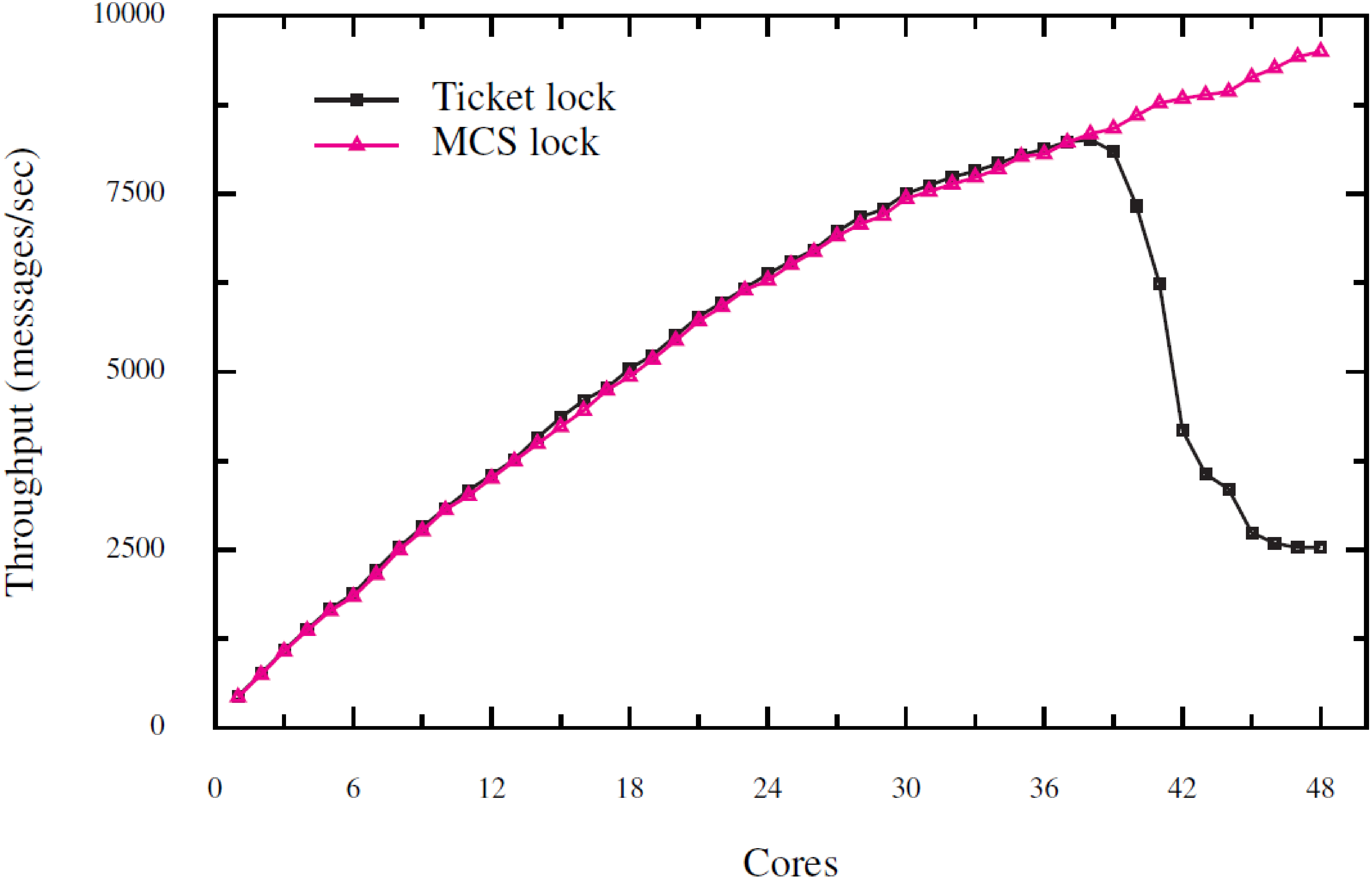
```c
arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
        struct qnode *predecessor;
        mynode->next = NULL;
        predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
        if (predecessor) {
                mynode->locked = 1;
                barrier();
       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



Thank you!