CS238P: Operating Systems

Lecture 12: Synchronization (and Scalability)

Anton Burtsev
March, 2018

Spinlocks

Critical sections with spinlocks

```
struct spin_lock {
    int locked;
};

insert(int data)
{
  struct list *l;
  l = malloc(sizeof *l);
  spinlock_acquire(&spin_lock);
  l->data = data;
  l->next = list;
  list = l;
  spinlock_release(&spin_lock);
}
```

- Critical section

Correct implementation

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

xchgl 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 }
```

Correct implementation

```
1573 void
1574 acquire(struct spinlock *lk)
1575 {
...
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
1584   // Tell the C compiler and the processor to not move loads or
stores
1585   // past this point, to ensure that the critical section's memory
1586   // references happen after the lock is acquired.
1587   __sync_synchronize();
...
1592 }
```

Mutexes

Critical sections with mutexes

```
struct mutex {
    int locked;
};
```

```
insert(int data)
{
  struct list *l;
  l = malloc(sizeof *l);
  mutex_acquire(&spin_lock);
  l->data = data;
  l->next = list;
  list = l;
  mutex_release(&spin_lock);
}
```

- Critical section

Mutexes

```
void
mutex_acquire(struct mutex *m)
{
    // The xchg is atomic.
    while(xchg(&m->locked, 1) != 0)
        yield();
    __sync_synchronize();
}
void
mutex_unlock(struct mutex *m)
{
    __sync_synchronize();
    m->locked = 0;
}
```

Conditional variables

Conditional variables

```
pthread_mutex_lock(&m);
while (count < 10) {
    pthread_cond_wait(&cv, &m);
}
pthread_mutex_unlock(&m);
```

```
while (1) {
  pthread_mutex_lock(&m);
  count++;
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&m);
}
```

Semaphores

Suppose a library has 10 identical study rooms, to be used by one student at a time. Students must request a room from the front desk if they wish to use a study room. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that one room has become free.

Semaphore

```
typedef struct sem_t {
  int count;
  mutex_t m;
  condition_t cv;
} sem_t;

int sem_init(sem_t *s, int value) {
  s->count = value;
  mutex_init(&s->m, NULL);
  cond_init(&s->cv, NULL);
  return 0;
}
```

Semaphores

```
sem_wait(sem_t *s) {
  mutex_lock(&s->m);
  while (s->count == 0) {
      cond_wait(&s->cv, &s->m);
  }
  s->count--;
  mutex_unlock(&s->m);
}

sem_post(sem_t *s) {
  mutex_lock(&s->m);
  s->count++;
  cond_signal(&s->cv);
  pthread_mutex_unlock(&s->m);
}
```

Semaphores as mutexes

- Mutexes can be implemented as binary semaphores
  - count = 1
  - However watch out for recursive use of the mutex

Back to spinlocks

Spinlock implementation

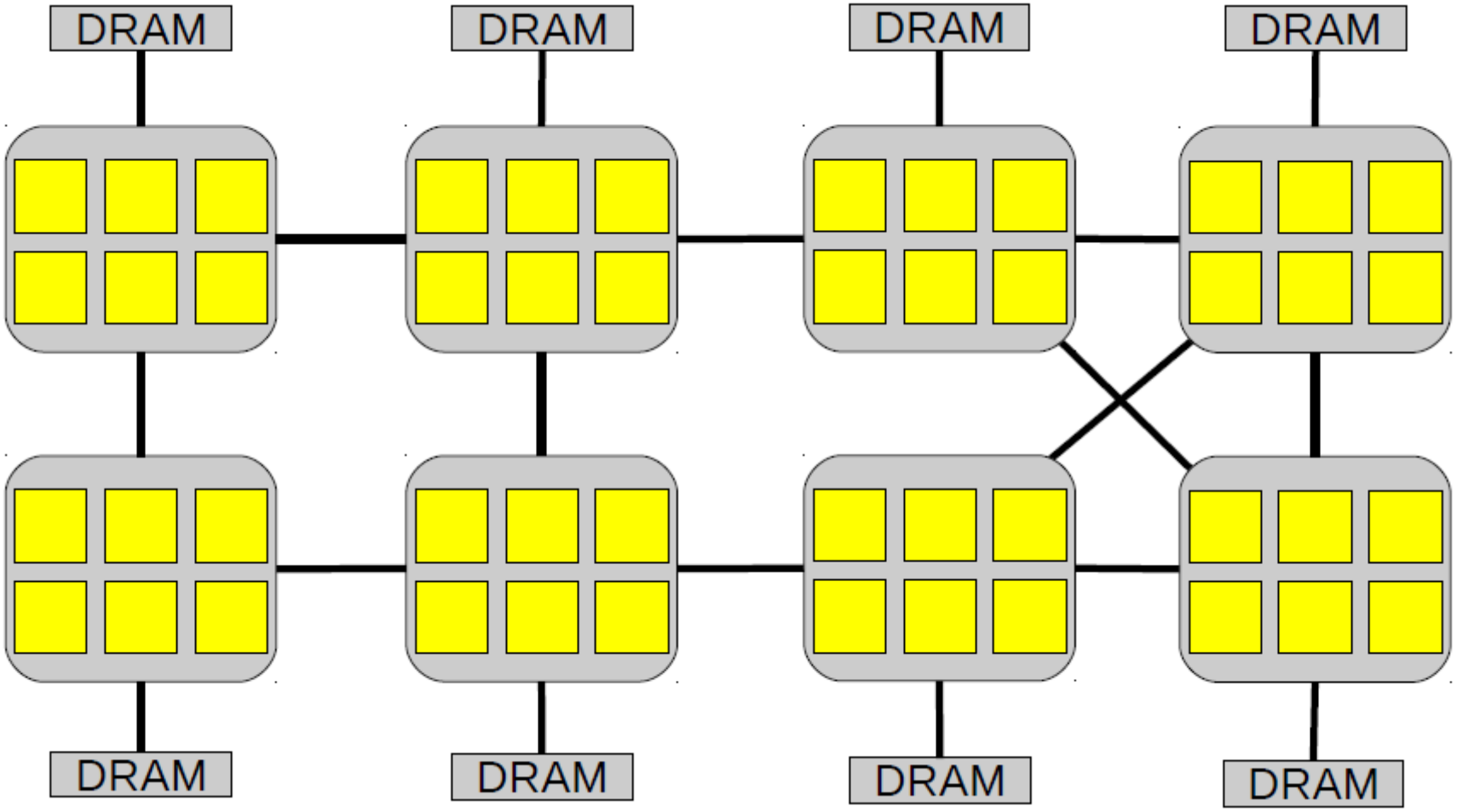
```
1573 void
1574 spinlock_acquire(struct spinlock *lk)
1575 {
...
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
...
1592 }
```

What is really wrong with locks?

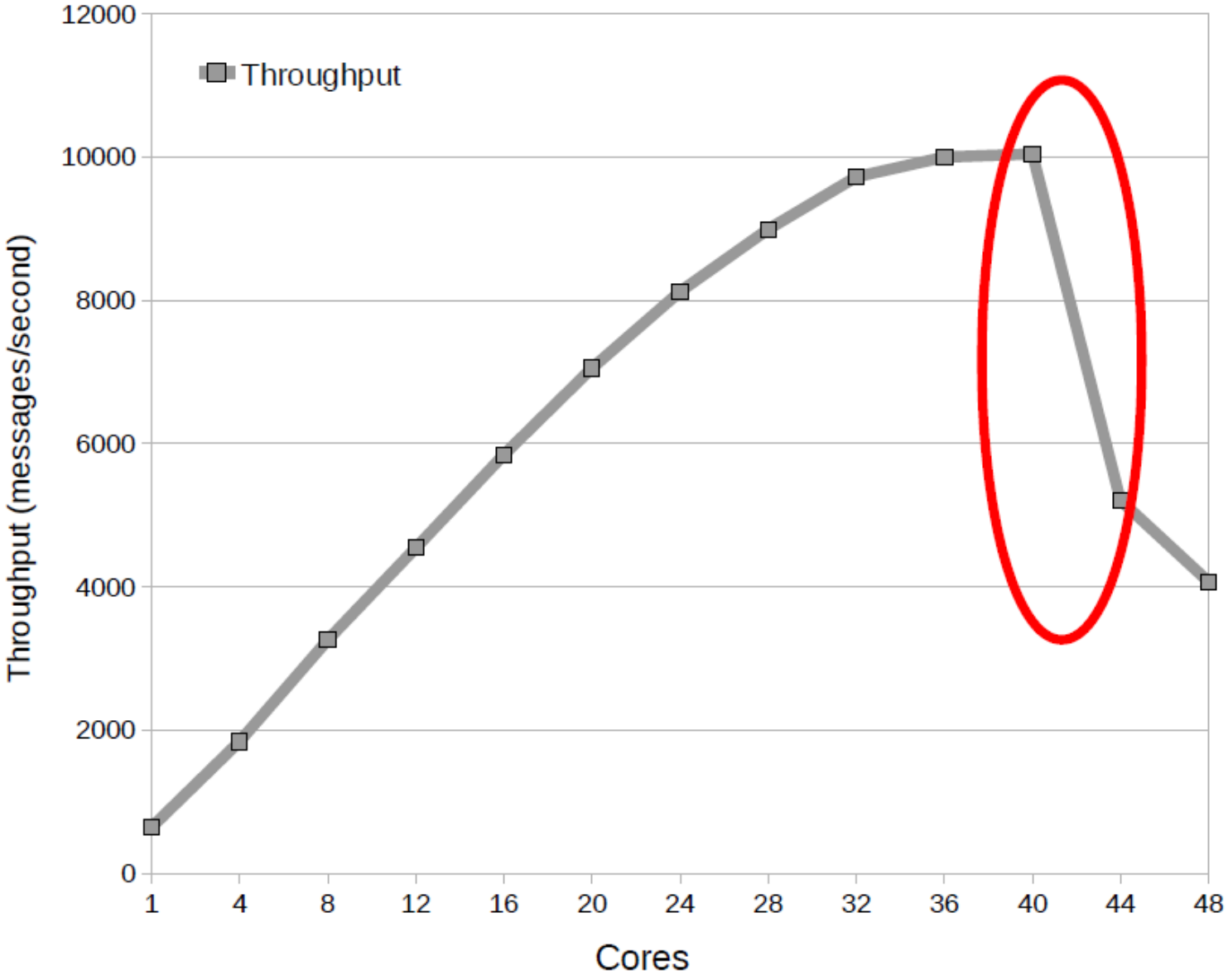
What is really wrong with locks?

- 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

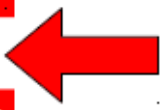
- sys_open eventually calls:

```
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

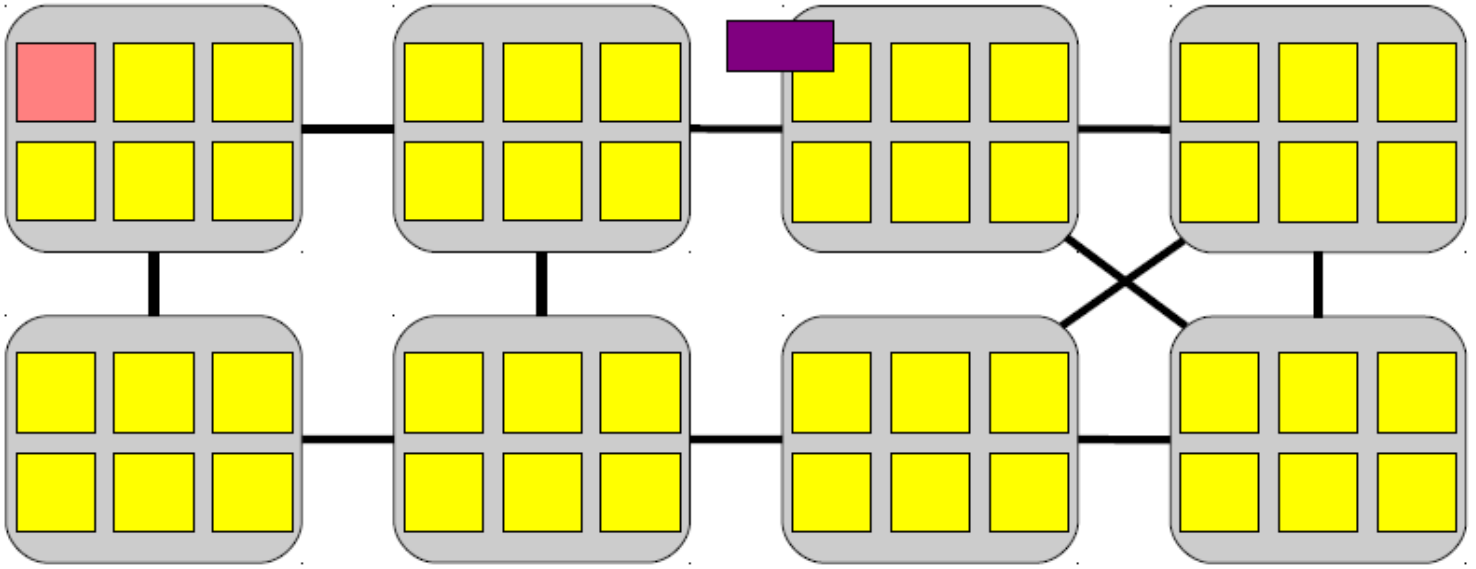
```
struct spinlock_t {
  int current_ticket ;
  int next_ticket ;
}
void spin_lock ( spinlock_t *lock)
{
  int t = atomic_fetch_and_inc (&lock -> next_ticket );
  while (t != lock -> current_ticket )
  ; /* spin */
}
void spin_unlock ( spinlock_t *lock)
{
  lock -> current_ticket ++;
}
```

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;
}
```



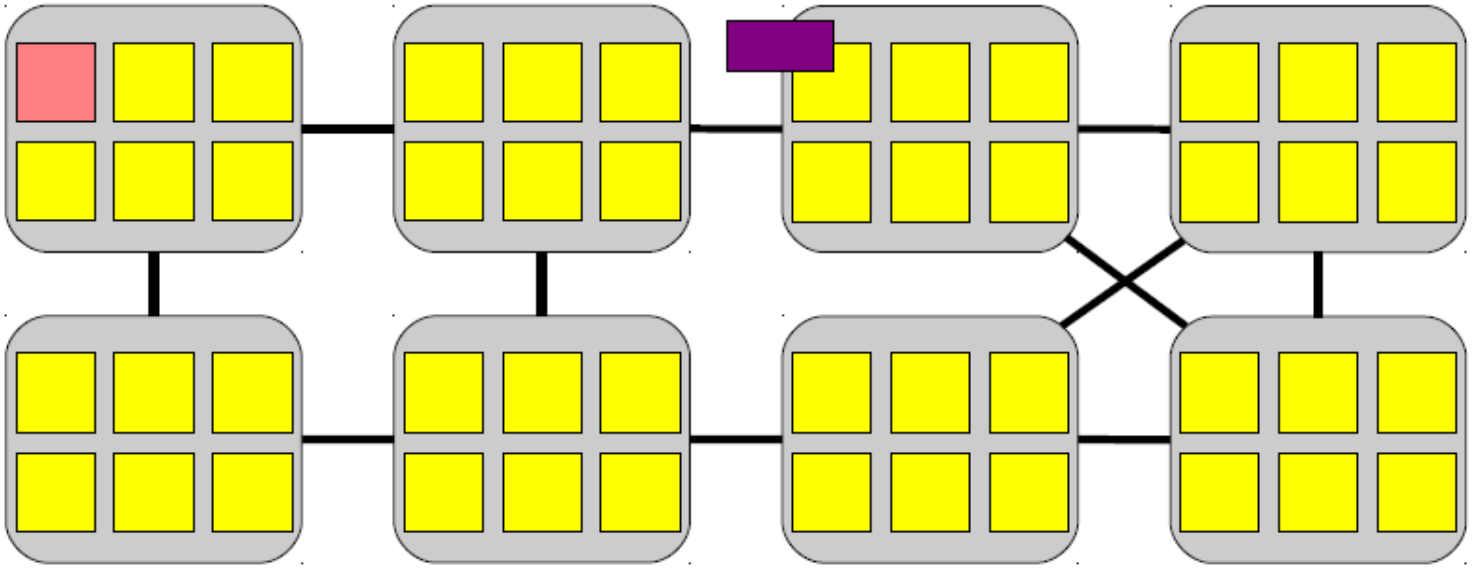
Spin lock implementation

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;
}
```



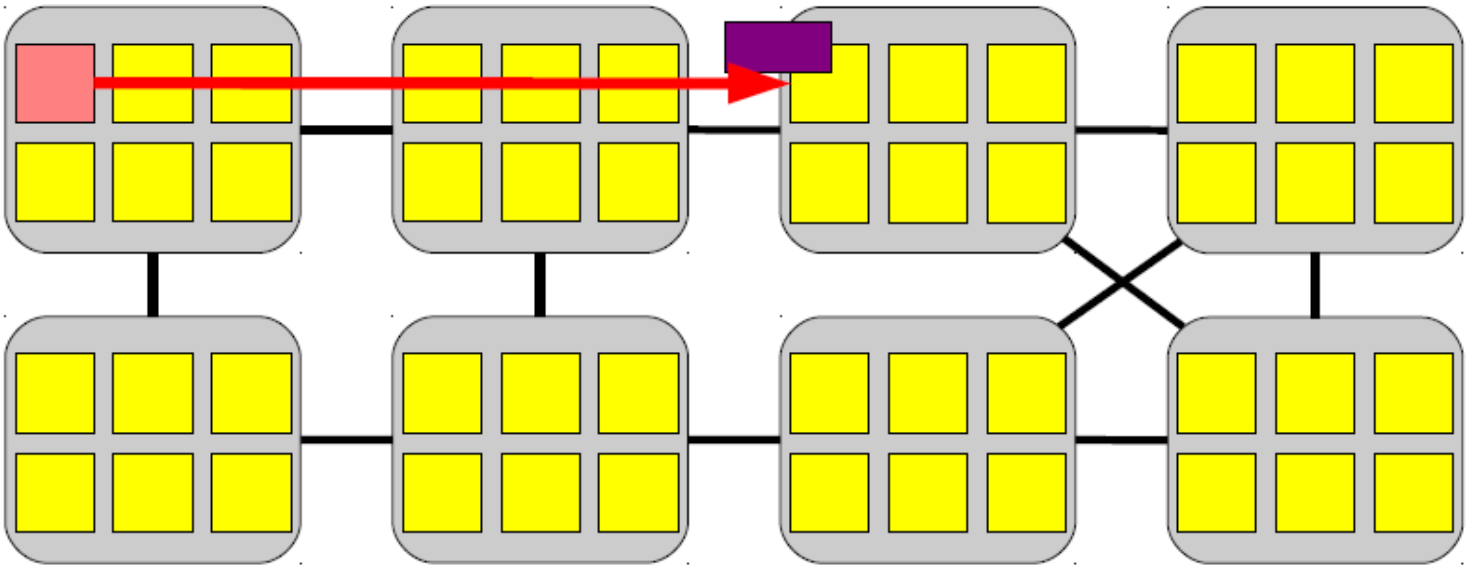
Spin lock implementation

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;
}
```



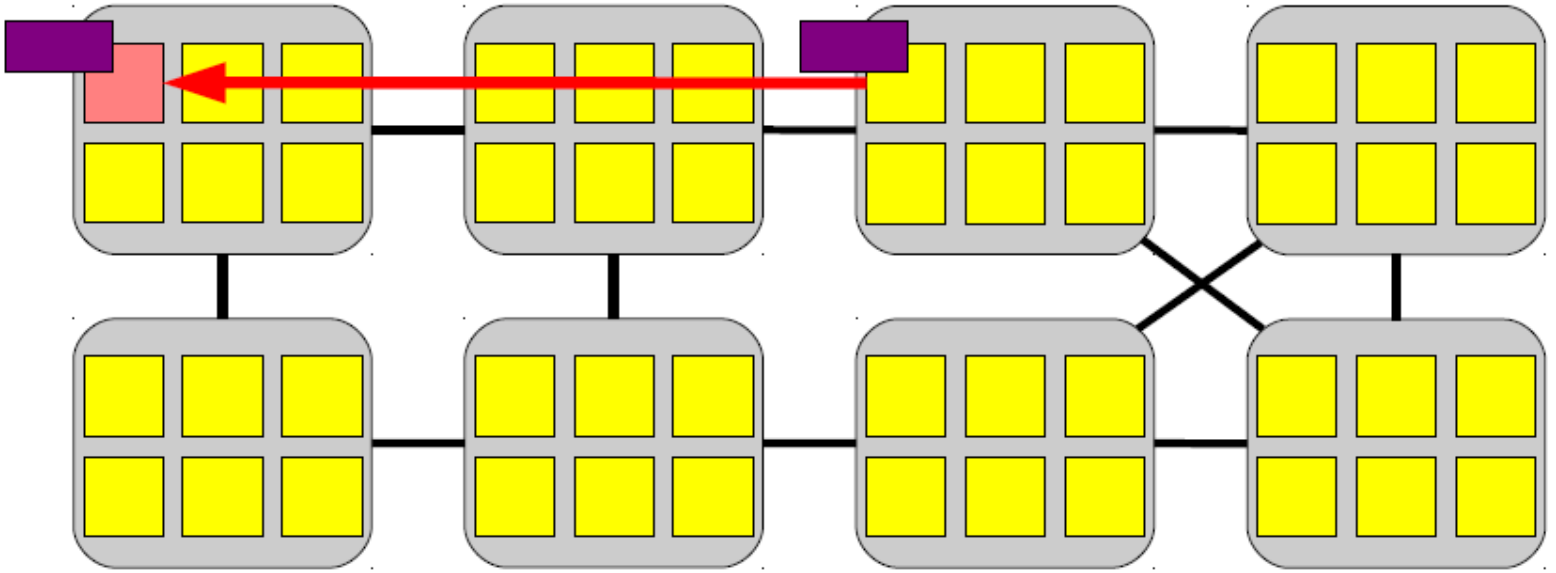
Spin l entation

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

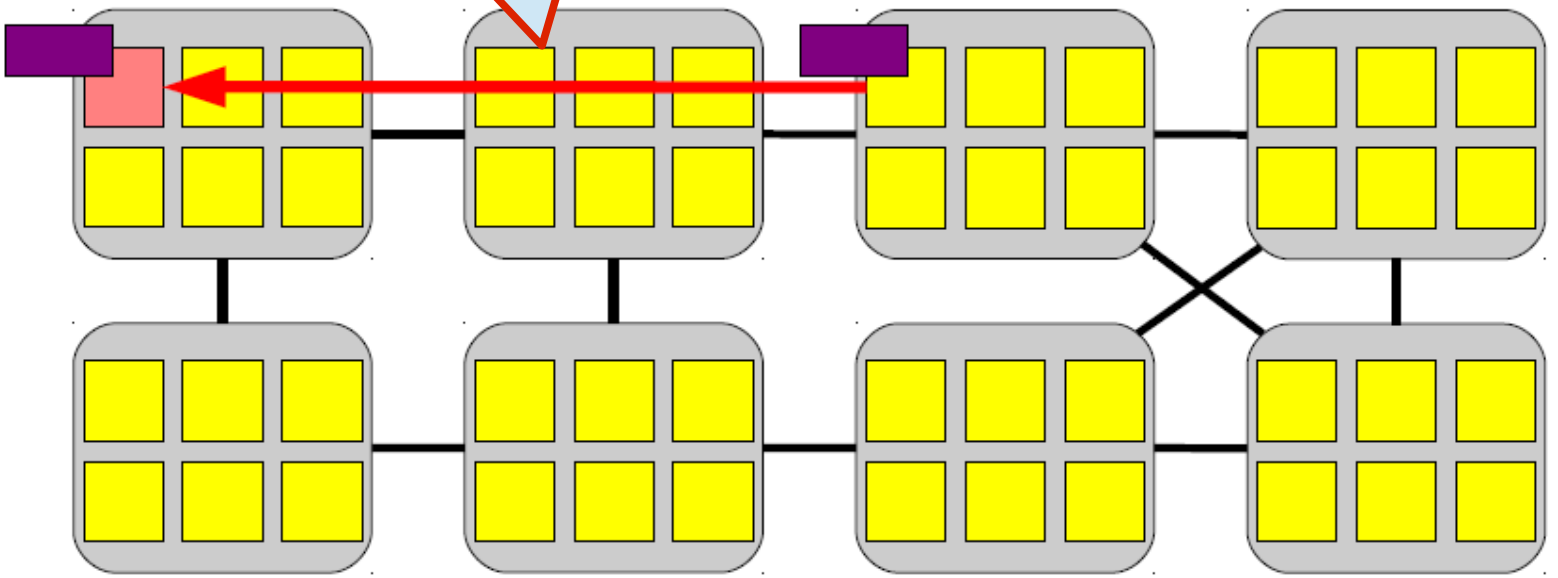
Allocate a ticket

```
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



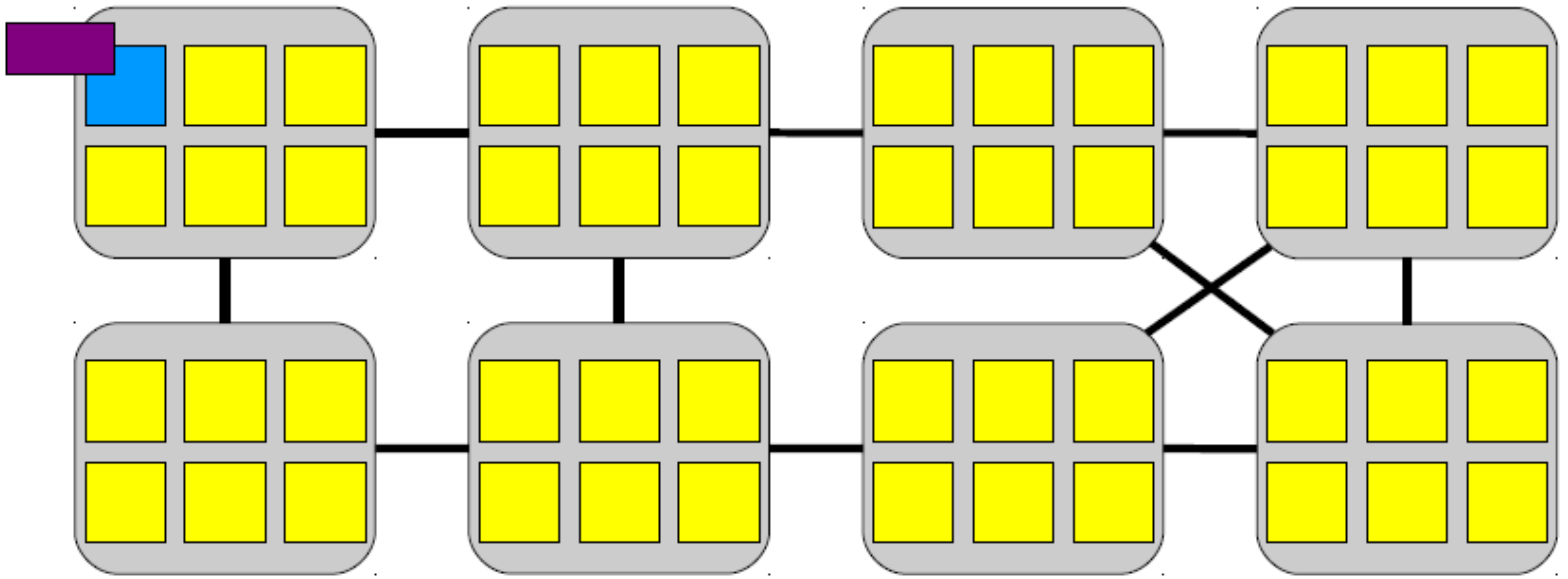
Spin lock impleme

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;
}
```



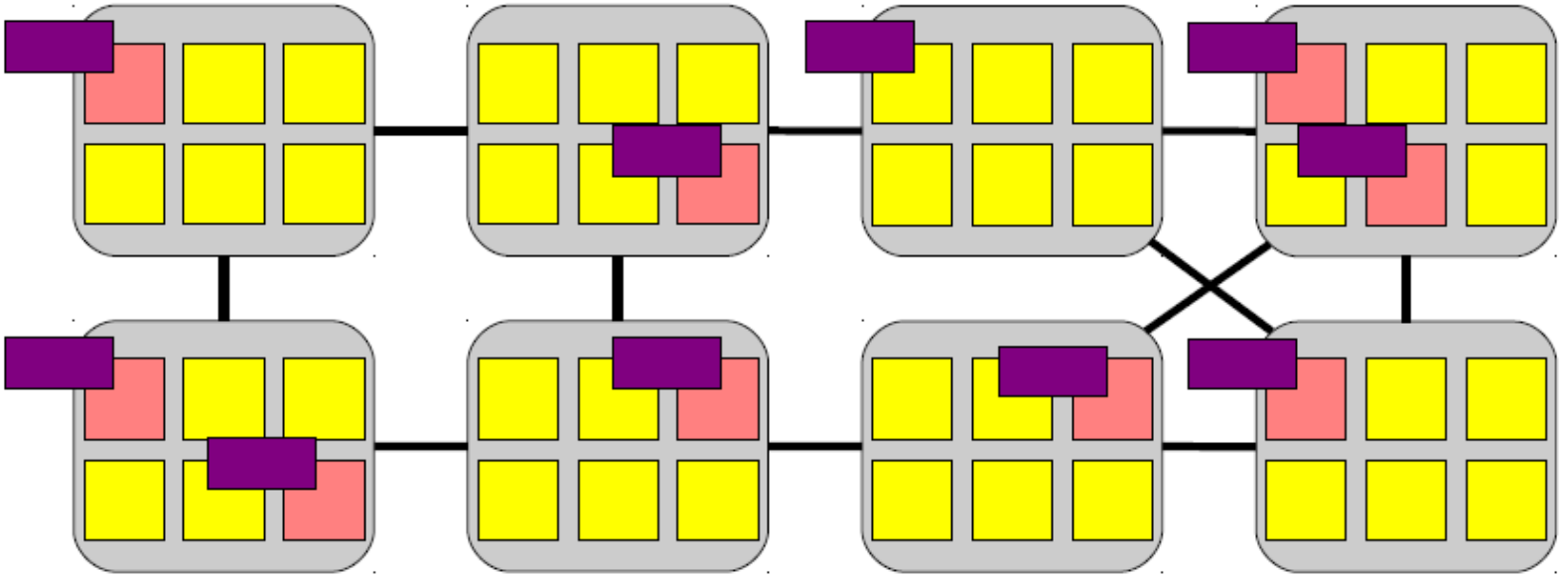
ck implementation

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;
}
```



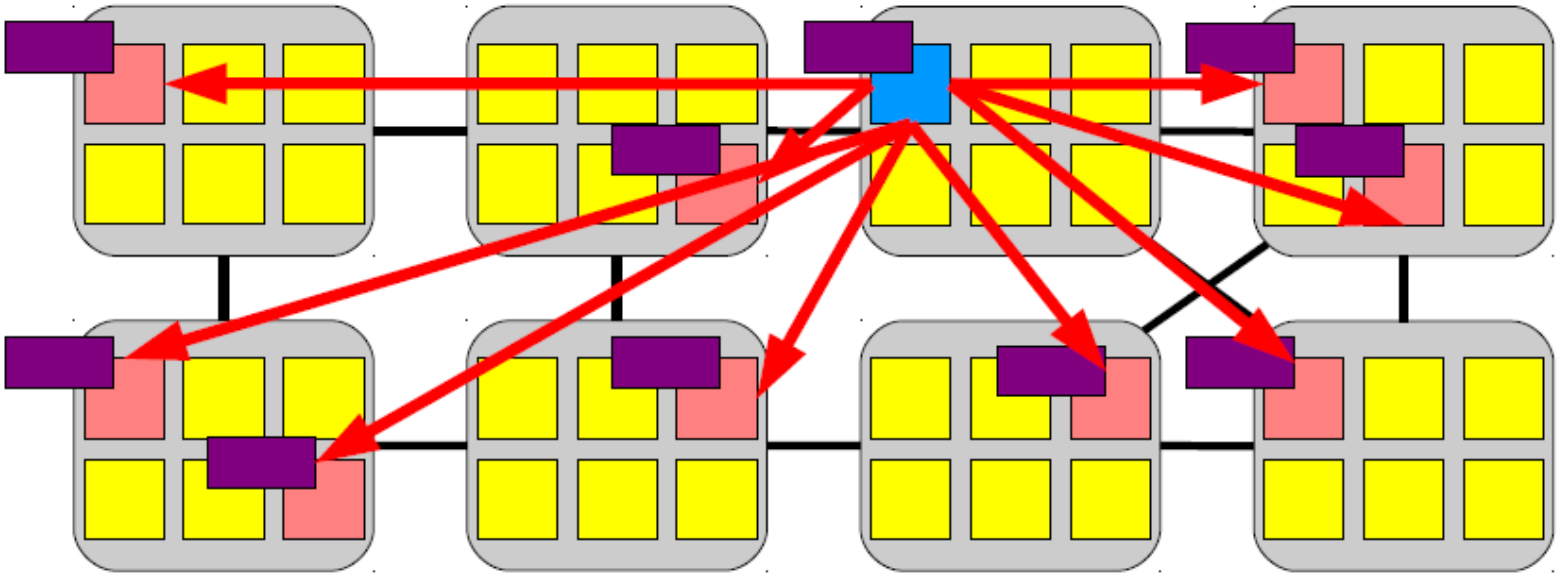
Spin lock implementation

```
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)

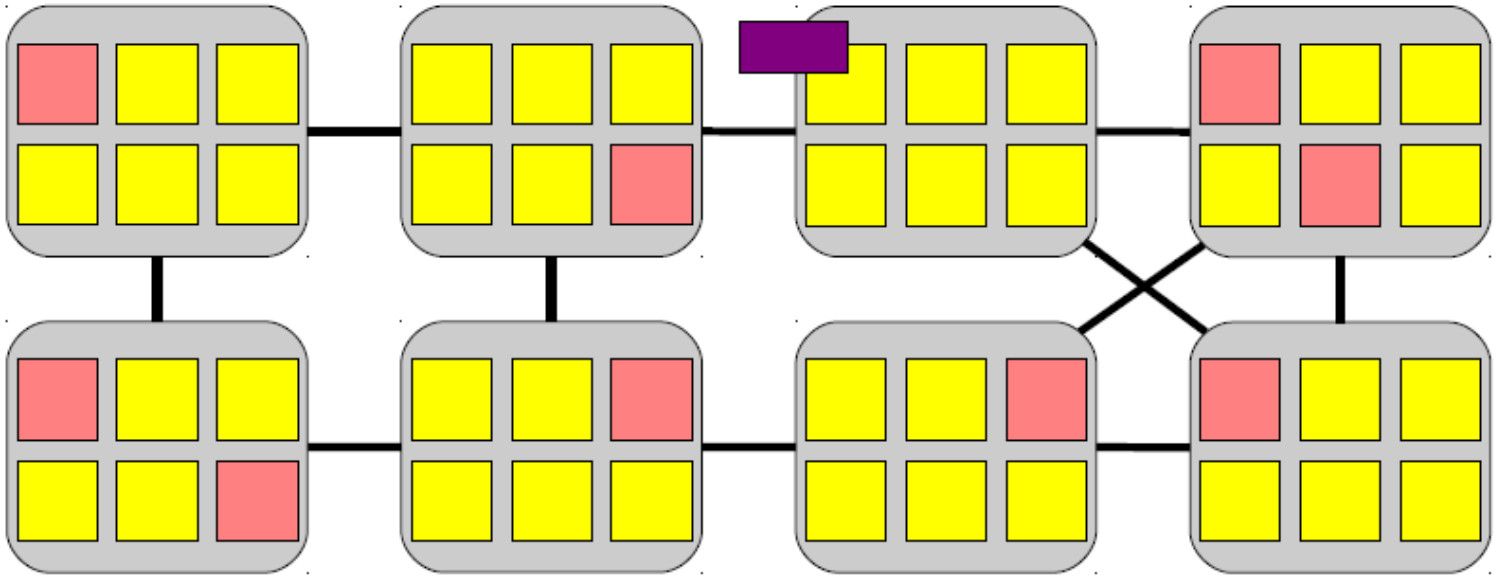


Spin lock implementation

```
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

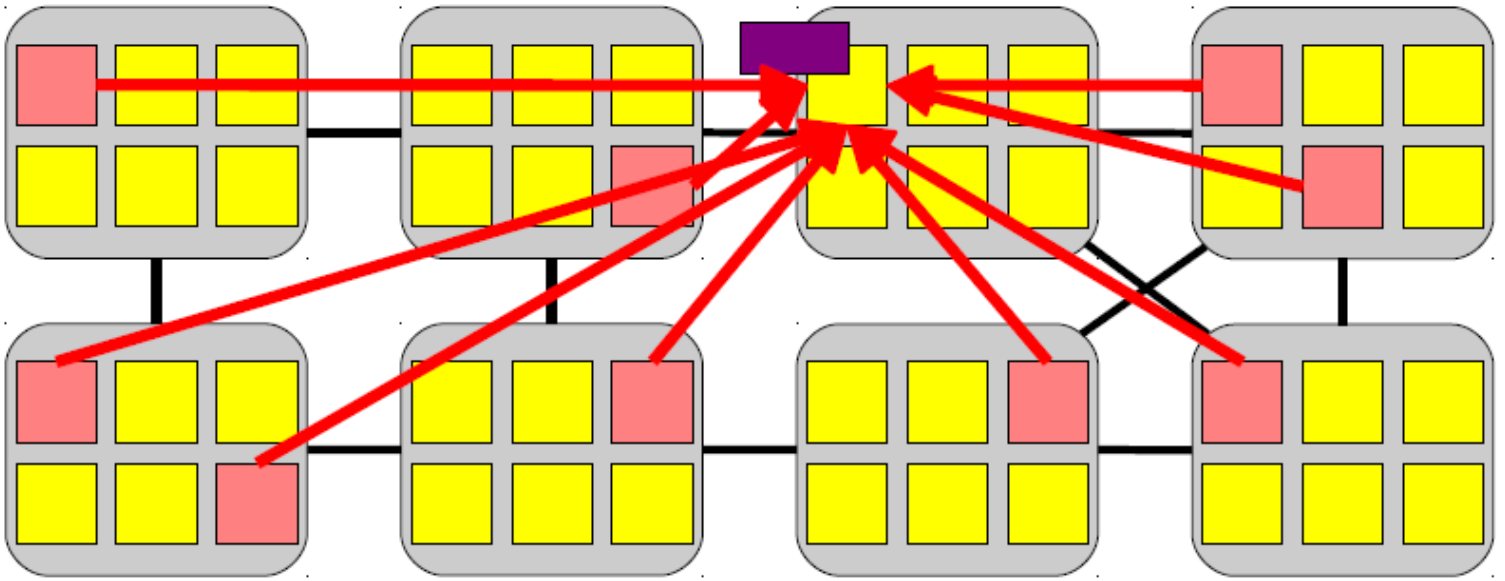


Spin lock implementation

```
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

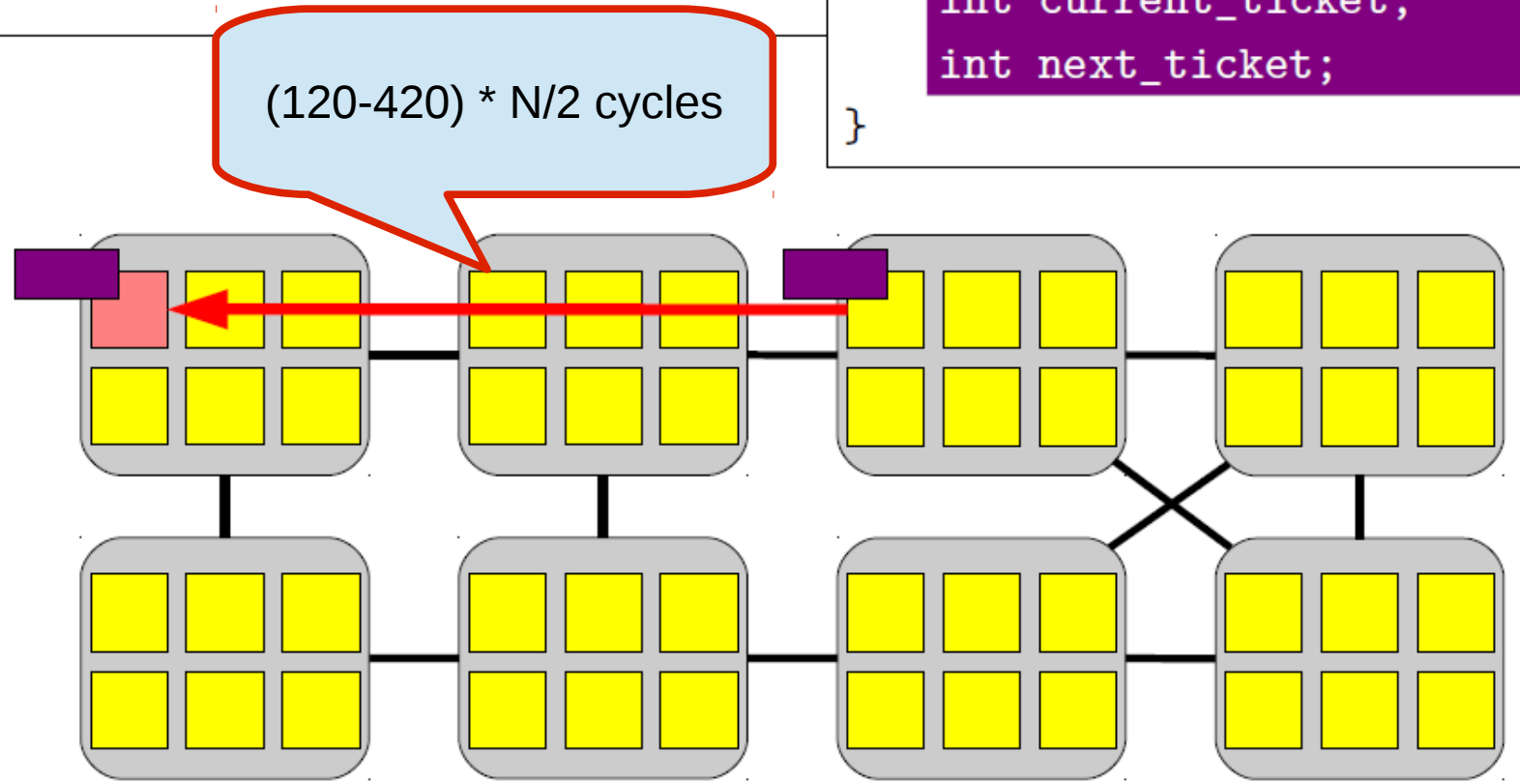


Spin lock implementation

```
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++;
}
```

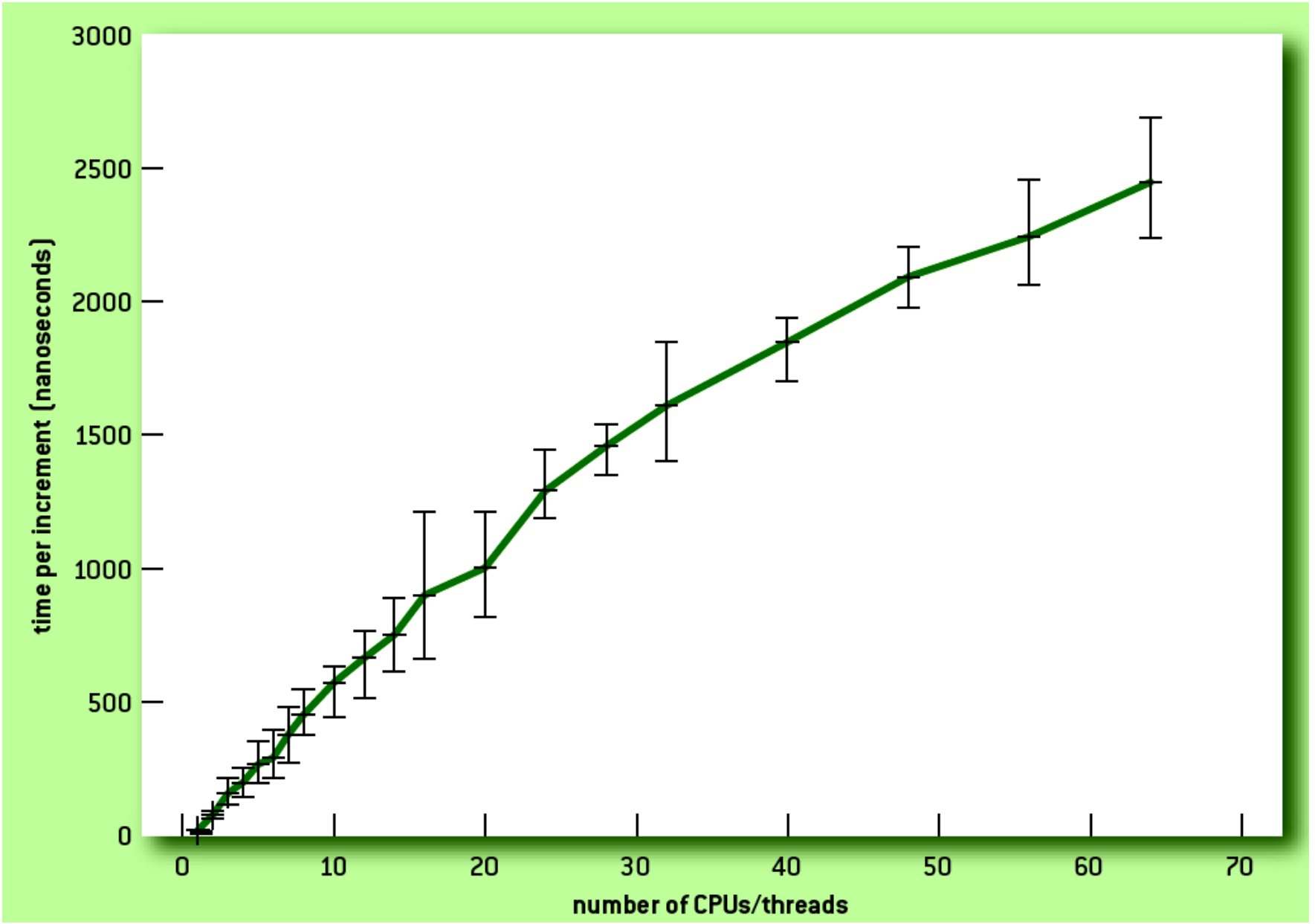
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



- 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;
}
```

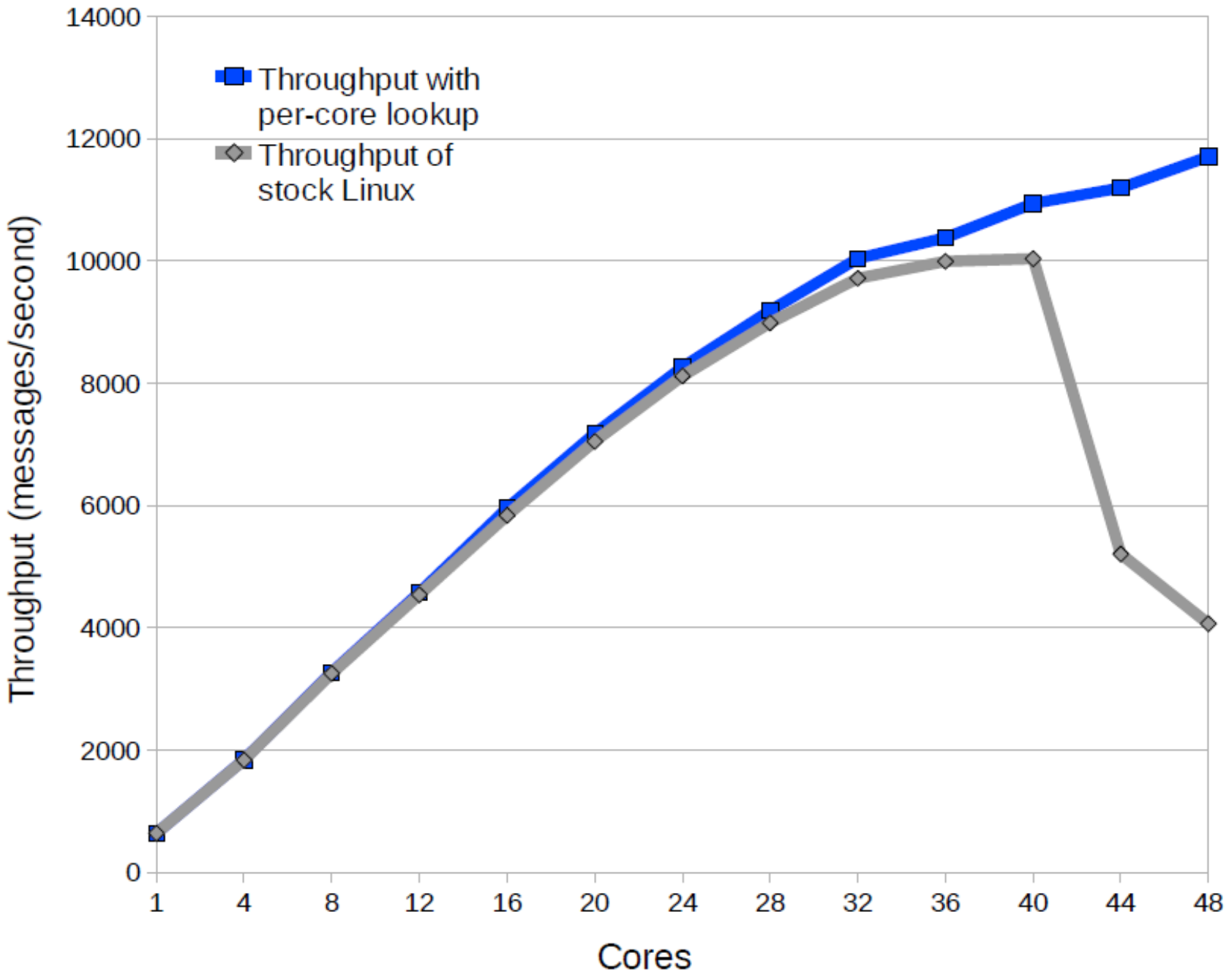
- Fast path: local hash lookup

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

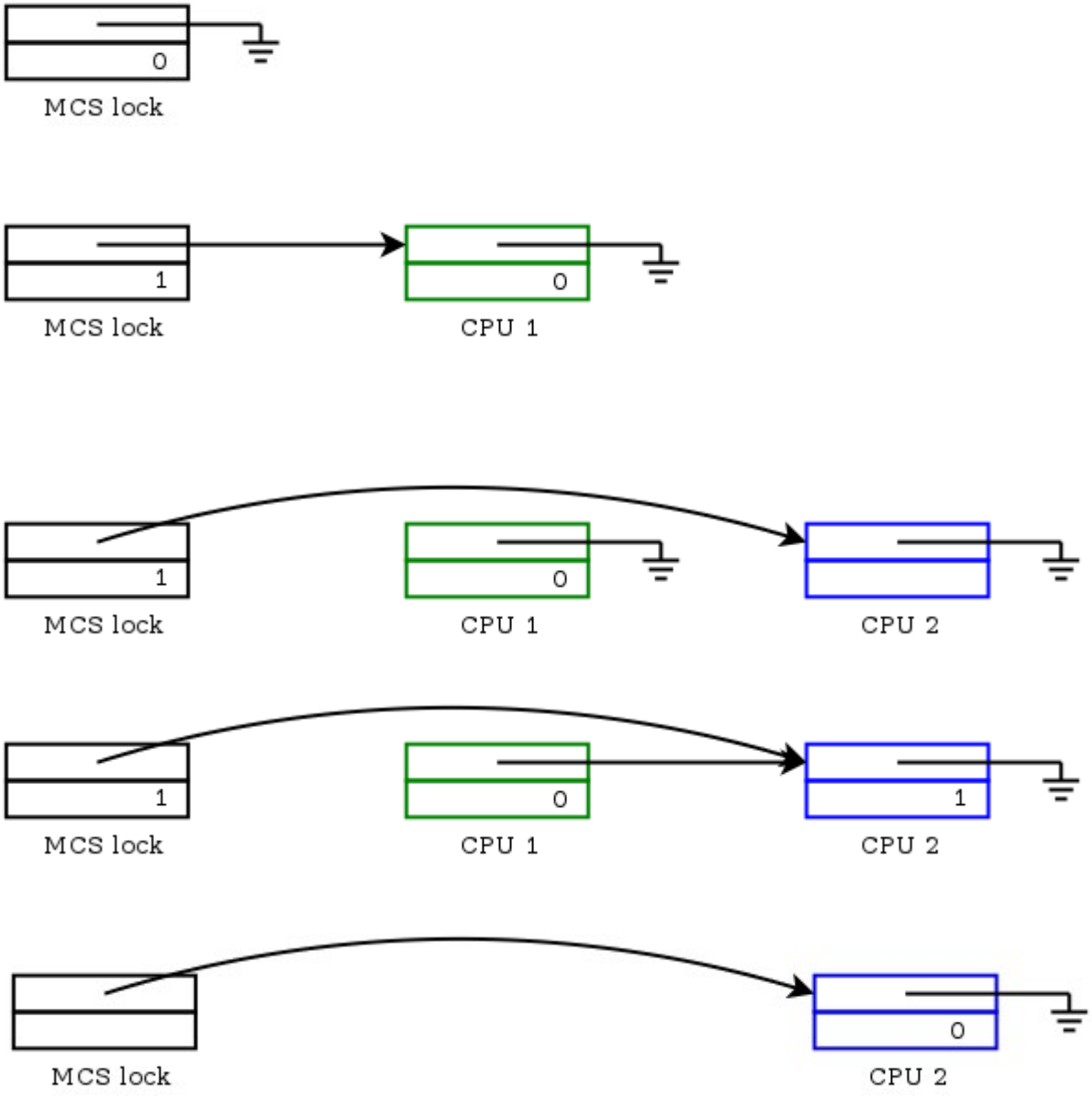

Throughput with per-core lookup
Throughput of stock Linux
Throughput (messages/second)
14000
12000
10000
8000
6000
4000
2000
0
1
4
8
12
16
20
24
28
32
36
40
44
48
Cores

Solution #2:
Is it possible to build scalable spinlocks?

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

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

```
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;
}
```

unlock

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

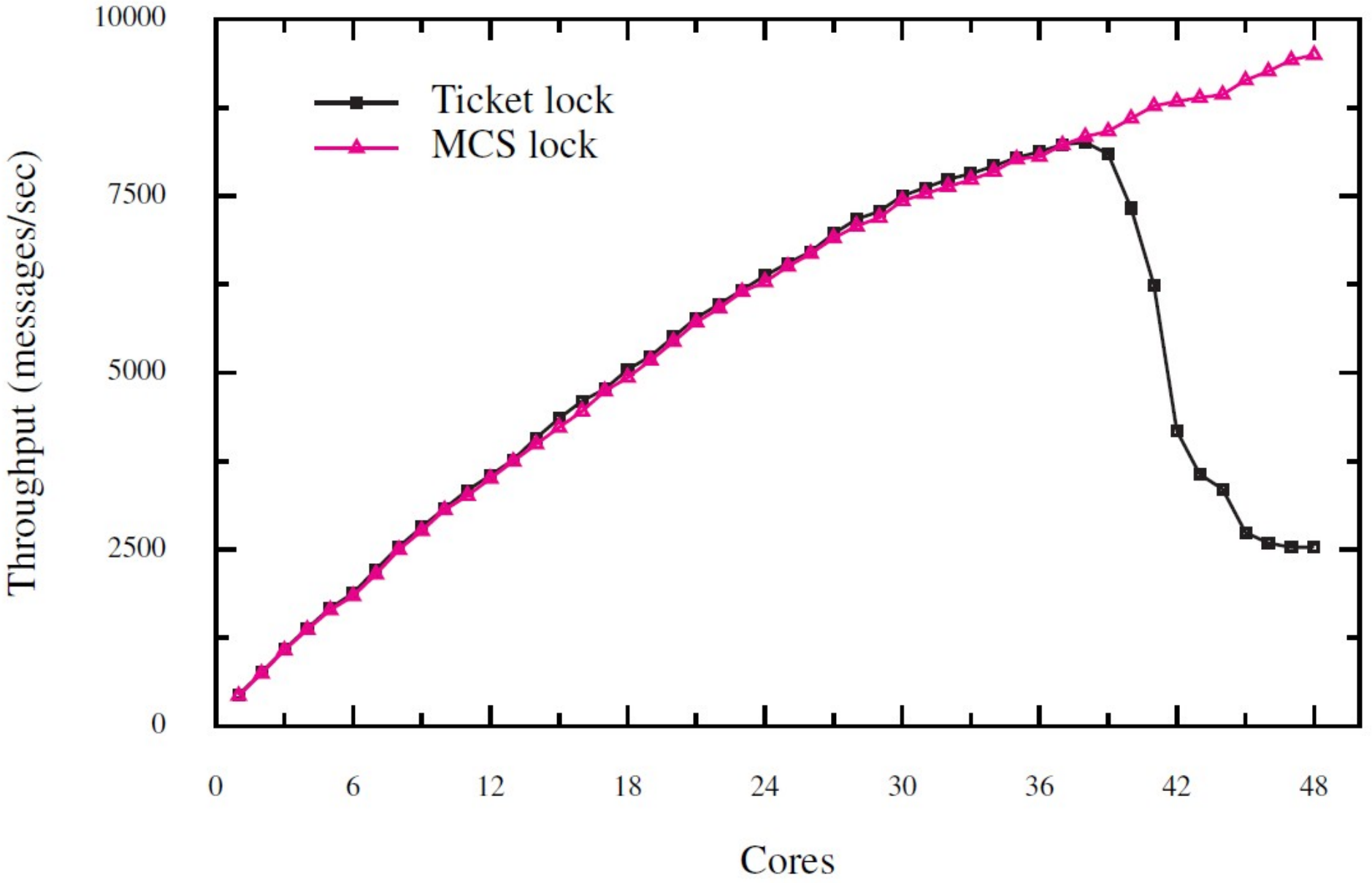
```
arch_mcs_lock(mcslock_t *l, volatile struct qnode *mynode) {
        struct qnode *predecessor;
        mynode->next = NULL;
        predecessor = (struct qnode *)xchg((long *)&l->v, (long)mynode);
        if (predecessor) {
                mynode->locked = 1;
                barrier();
        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

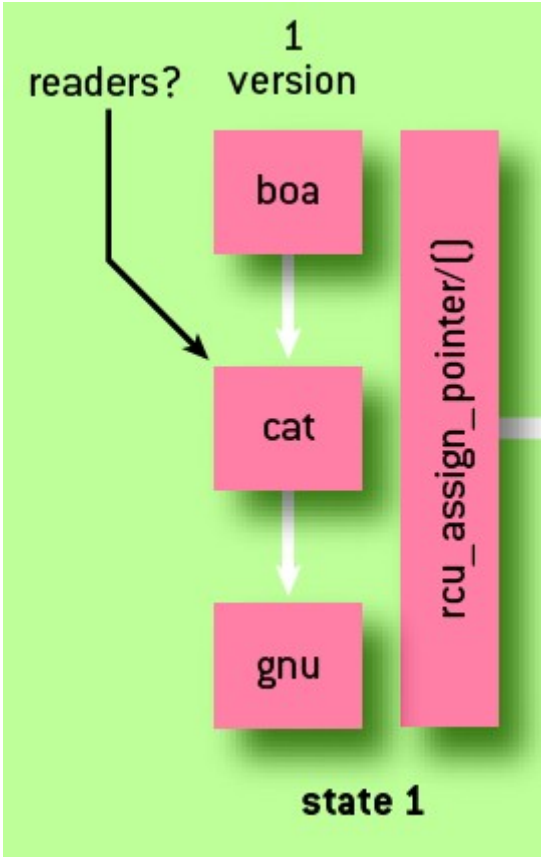
```
struct qnode {
        volatile void *next;
        volatile char locked;
        char __pad[0] __attribute__((aligned(64)));
};

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

Exim: MCS vs ticket lock



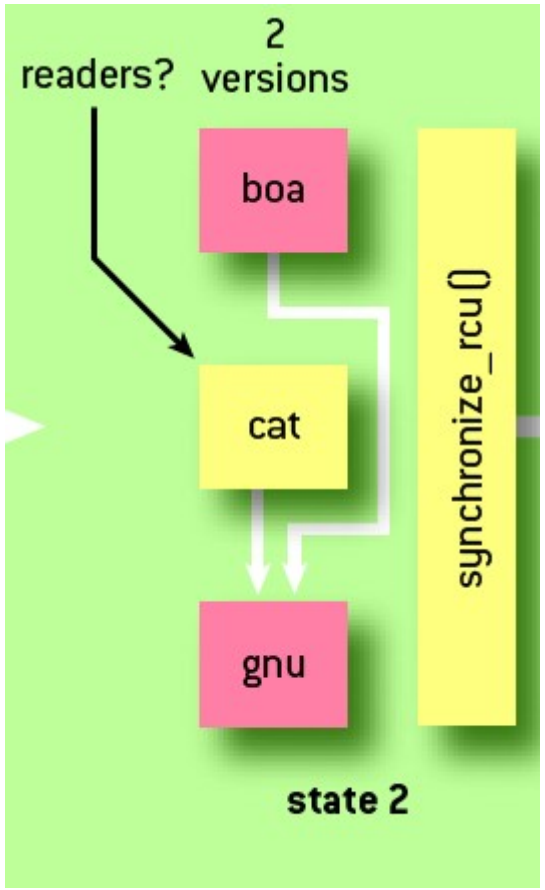
RCU: Read Copy Update

Read copy update



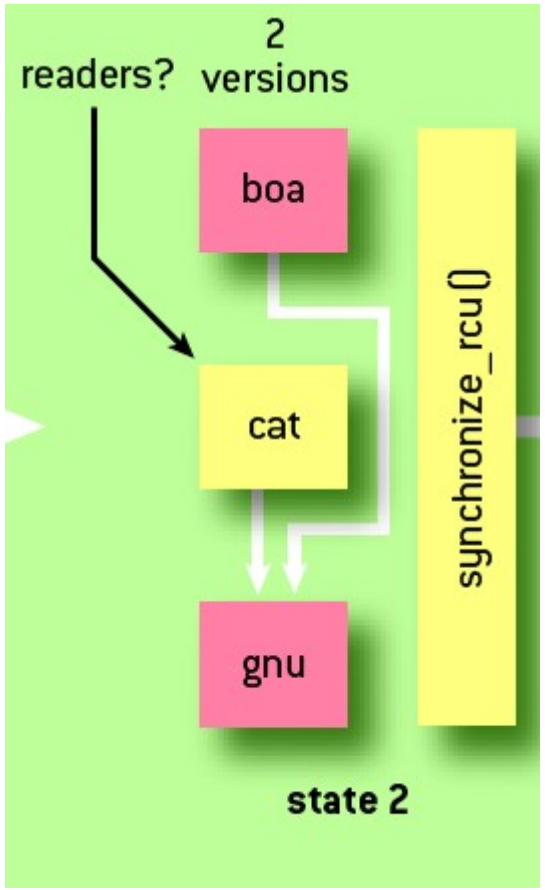
- Goal: remove “cat” from the list
  - There might be some readers of “cat”
- Idea: control the pointer dereference
  - Make it atomic

Read copy update (2)



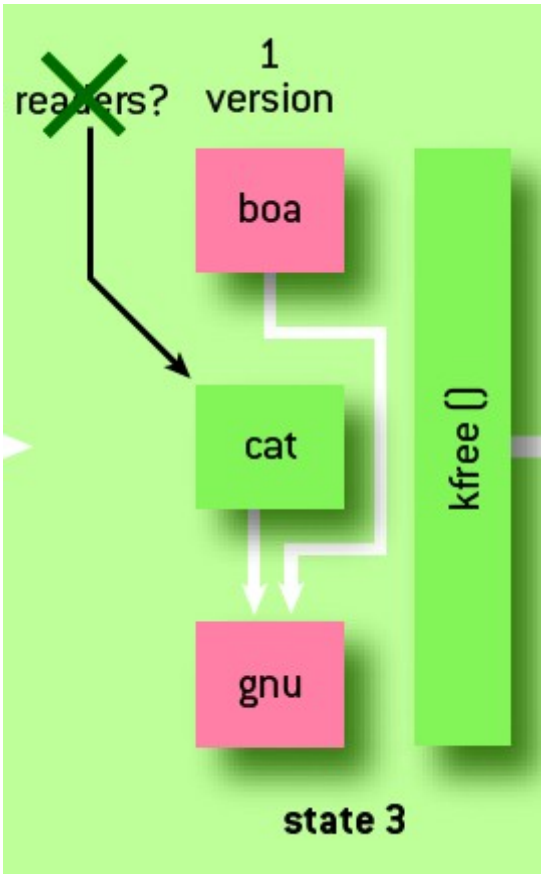
- Remove “cat”
  - Update the “boa” pointer
  - All subsequent readers will get “gnu” as boa->next

Read copy update (2)



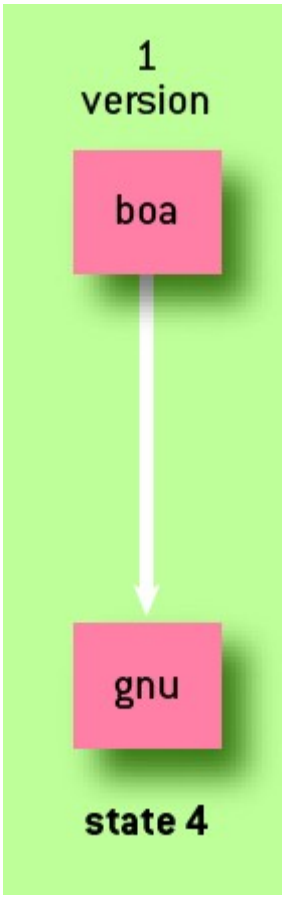
- Wait for all readers to finish
  - synchronize_rcu()

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)
```

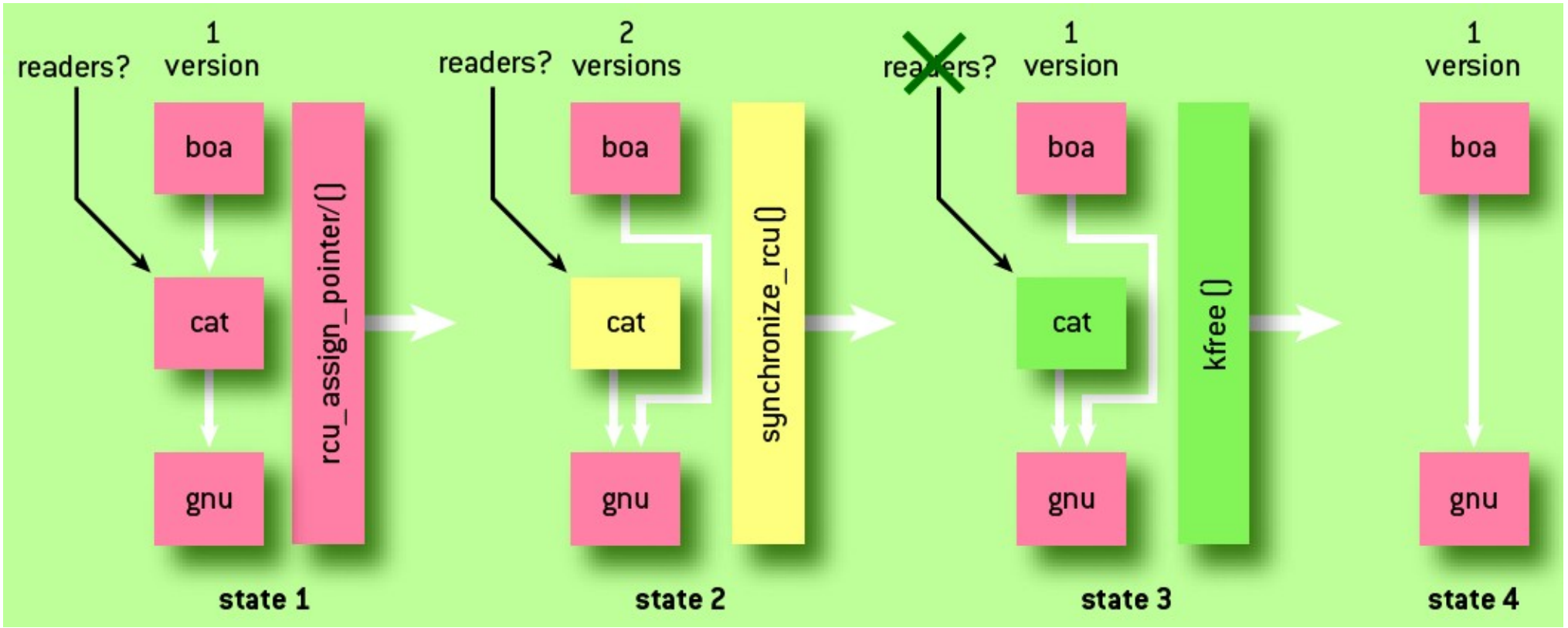
```
syscall_t *table;
spinlock_t table_lock;

int invoke_syscall(int number, void *args...)
{
  syscall_t *local_table;
  int r = -1;

  rcu_read_lock();
  local_table = rcu_deference(table);
  if (local_table != NULL)
    r = local_table[number](args);
  rcu_read_unlock();

  return r;
}
```

Example: dynamic system call table

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



Conclusion

- What RCU is good for?

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?

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?
  - Fine-grained, scalable spinlocks
  - Lock-free synchronization
  - Transactional memory

Thank you!