

CS5460/6460: Operating Systems

Lecture 12: Synchronization and scalability

Anton Burtsev
February, 2014

Recap from last time

- Main synchronization paradigm
 - Critical sections
 - Implemented as spinlocks
- Optimistic concurrency is possible
 - Lock-free synchronization
 - Algorithms are hard

What is really wrong with locks?

What is really wrong with locks?

- Scalability

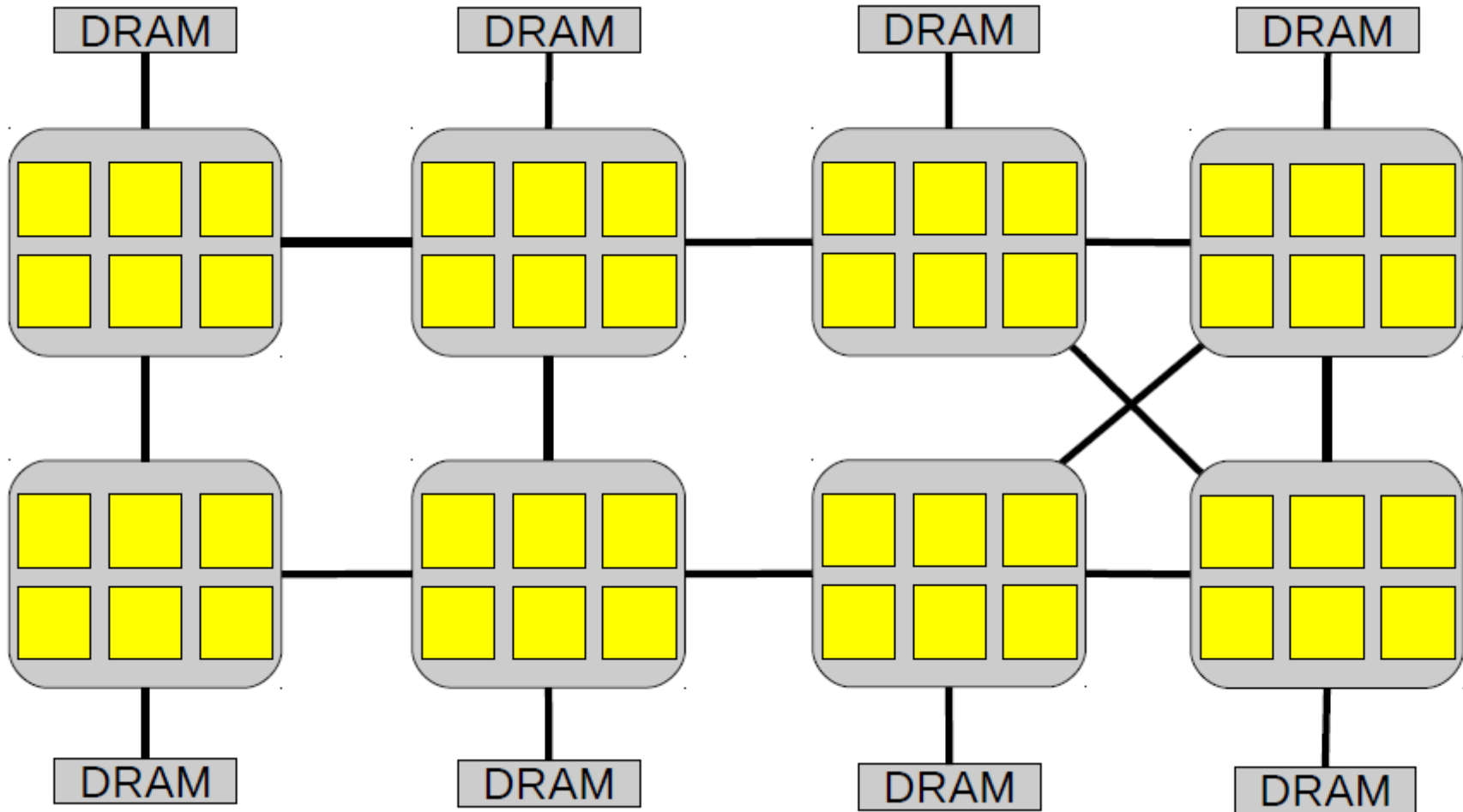
Ticket lock in Linux

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

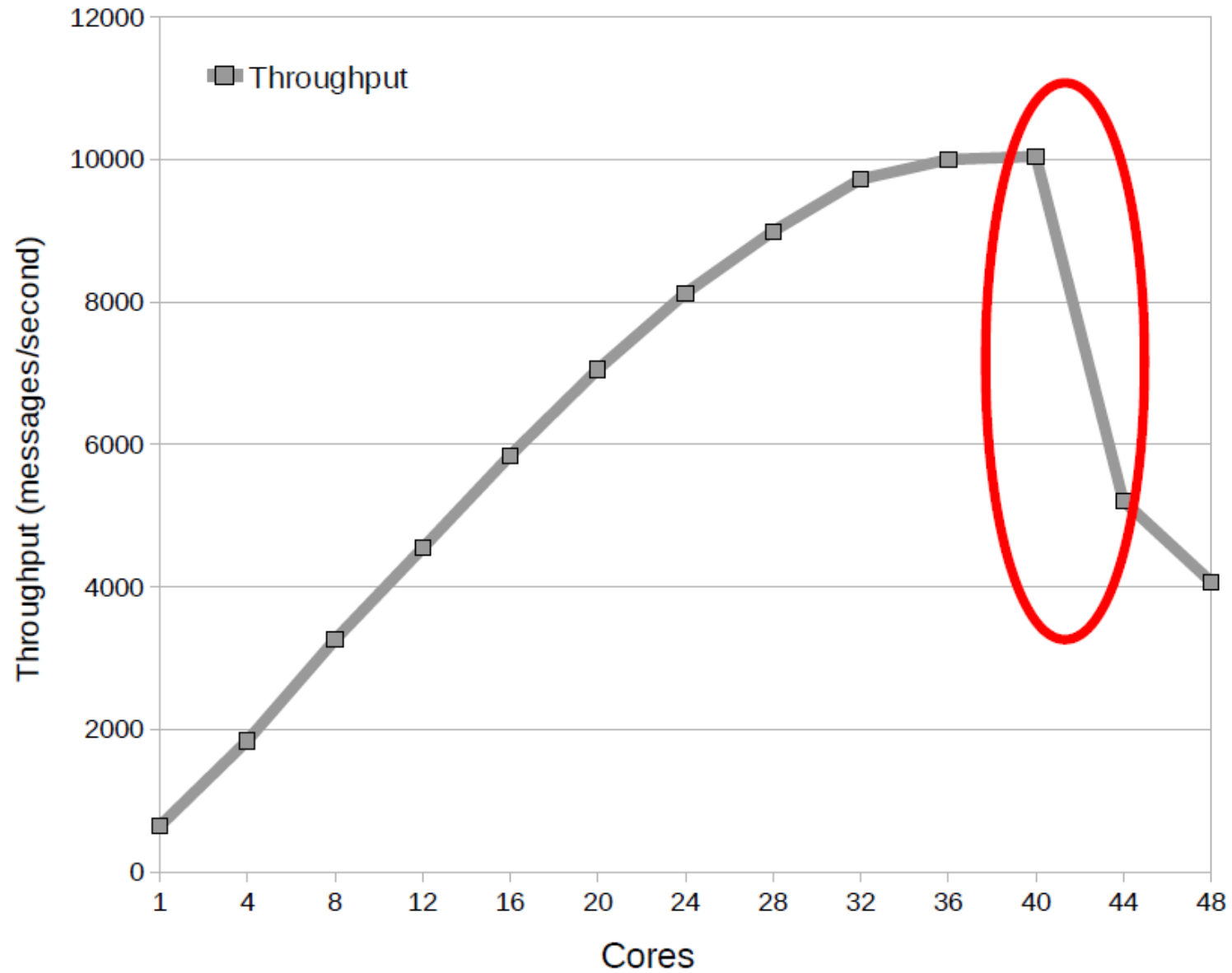
void spin_lock ( spinlock_t *lock)
{
    int t = atomic_fetch_and_inc (&lock -> next_ticket );
    while (t != lock -> current_ticket )
        ; /* spin */
}

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

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:

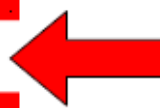
```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    return mnt;
}
```

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?



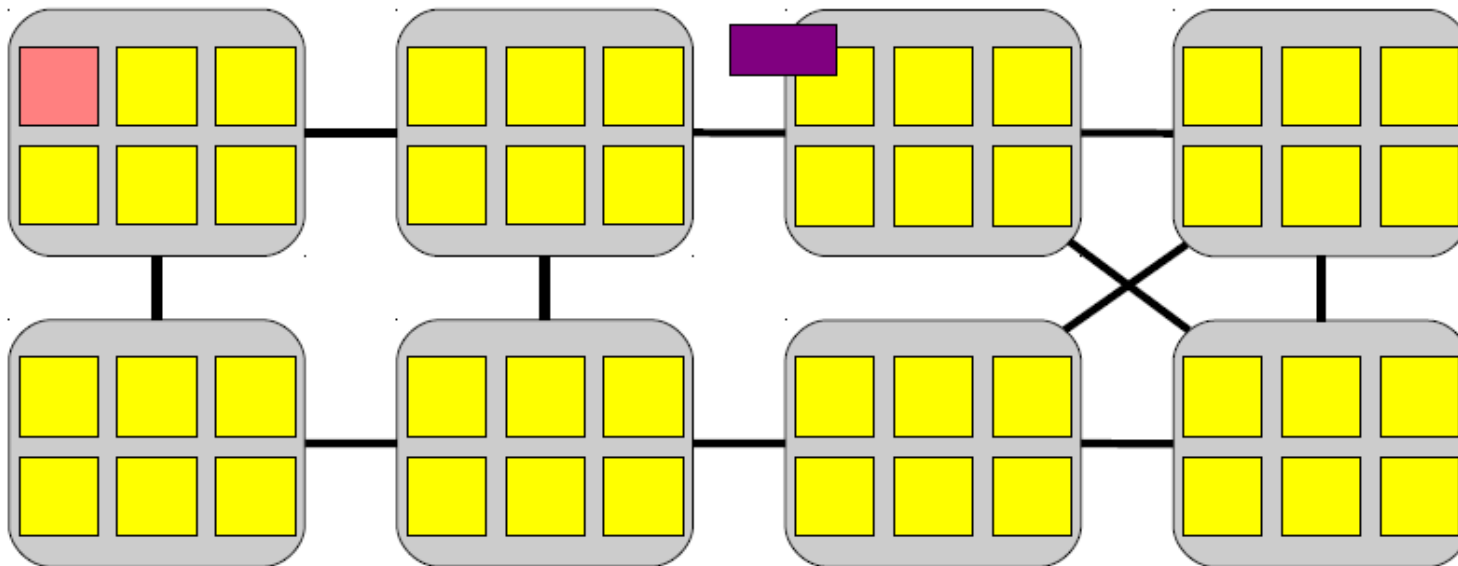
- `spin_lock` and `spin_unlock` use many more cycles than the critical section

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;
}
```



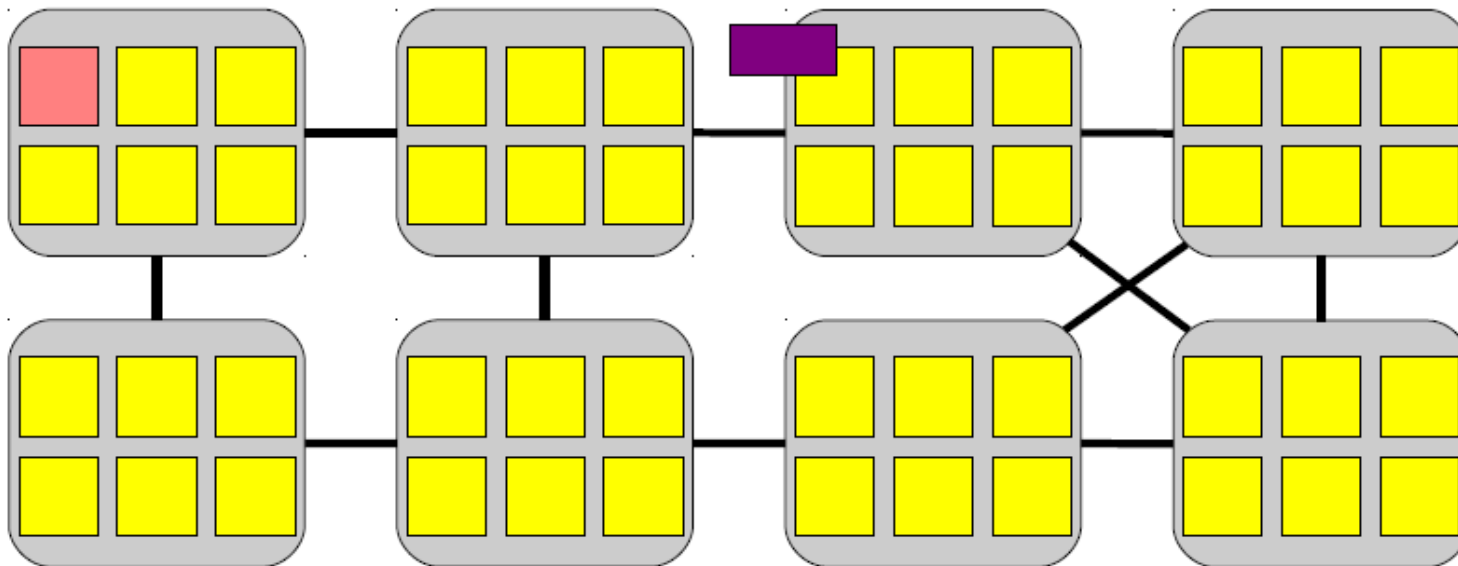
Spin lock implementation

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;
}
```



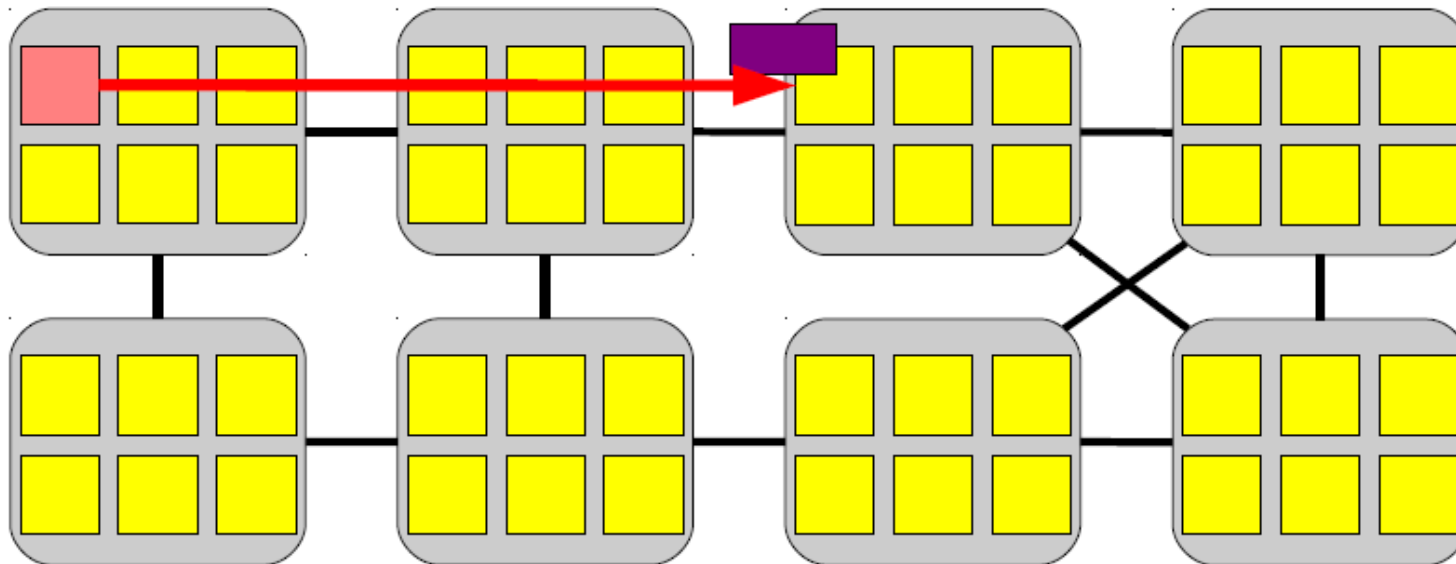
Spin lock implementation

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;
}
```



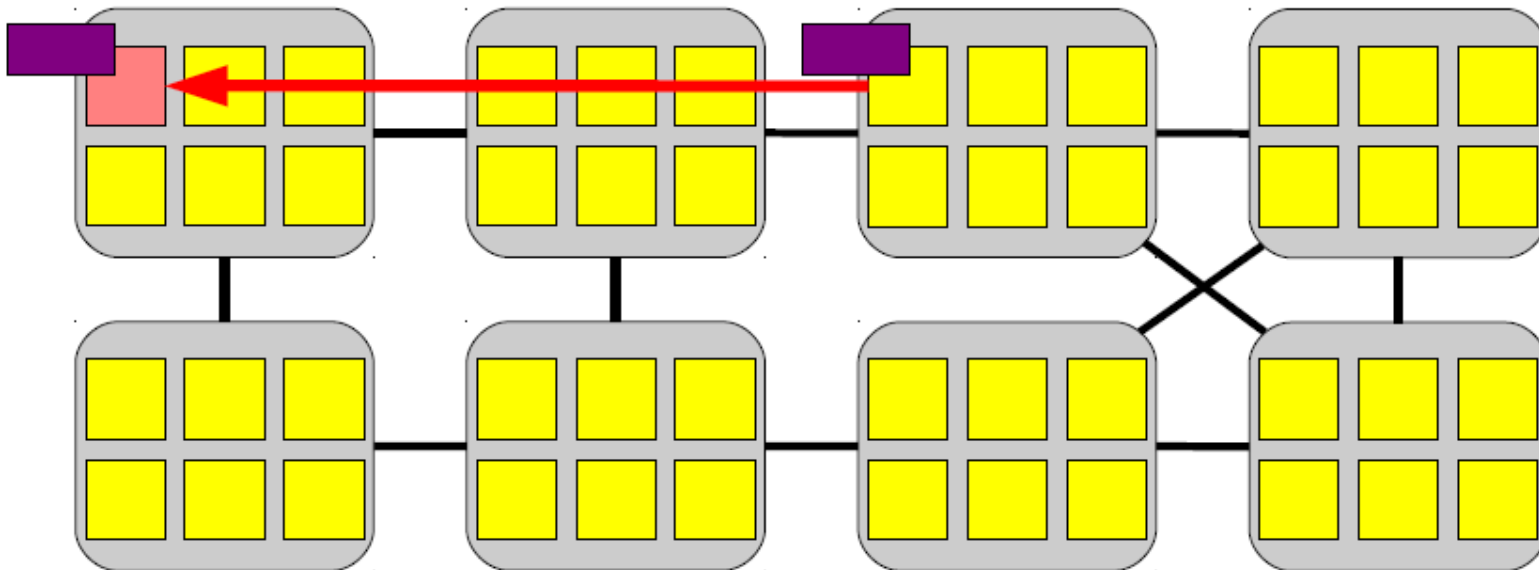
Spin lock implementation

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

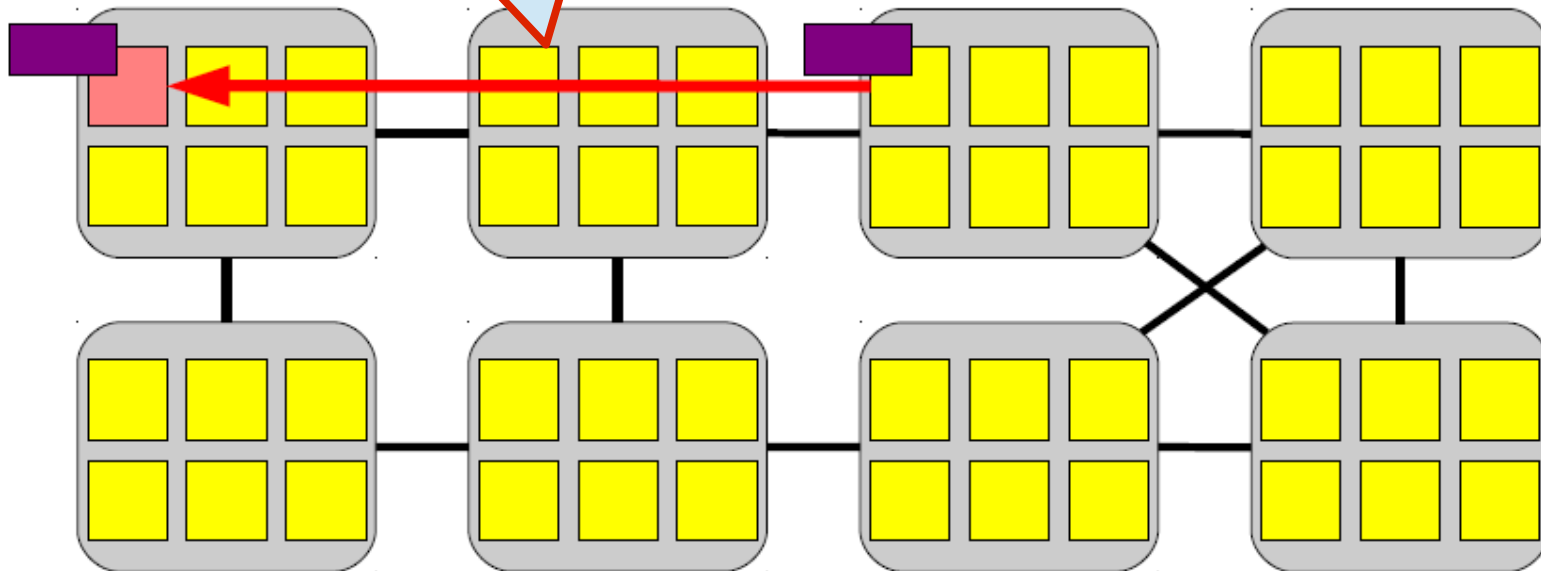
Allocate a ticket

```
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;
}
```

120-420 cycles



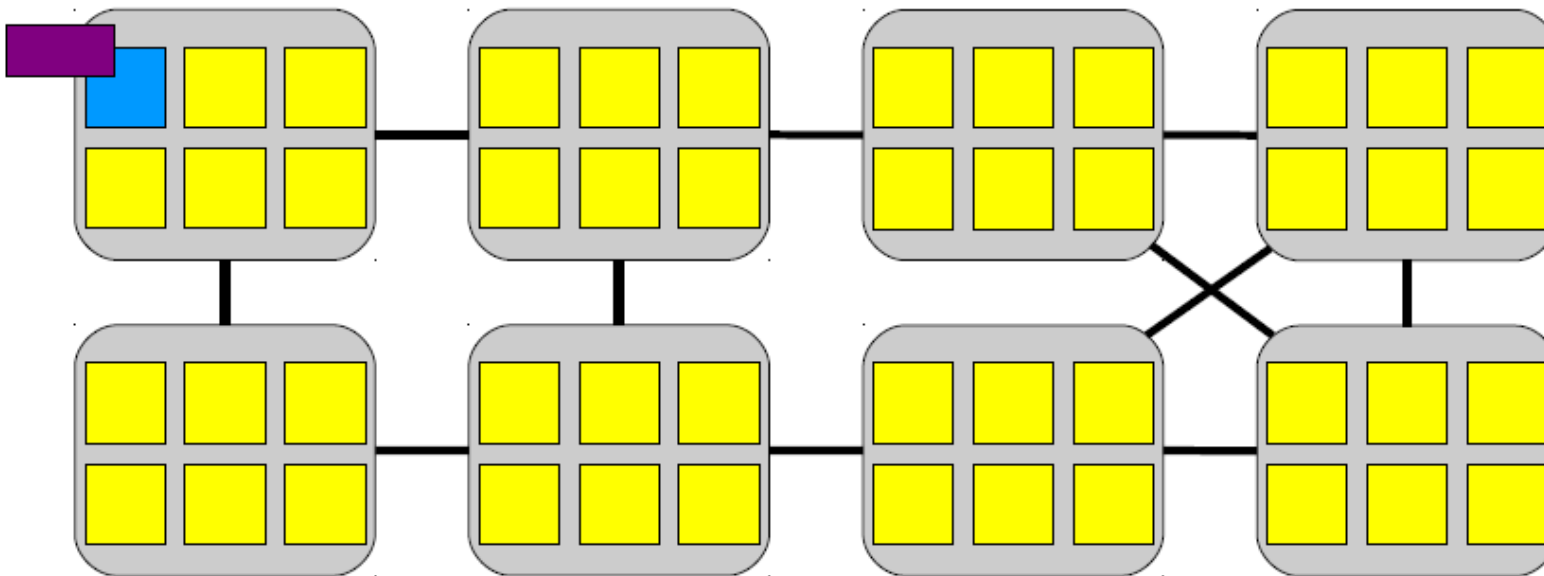
Spin lock implementation

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;
}
```



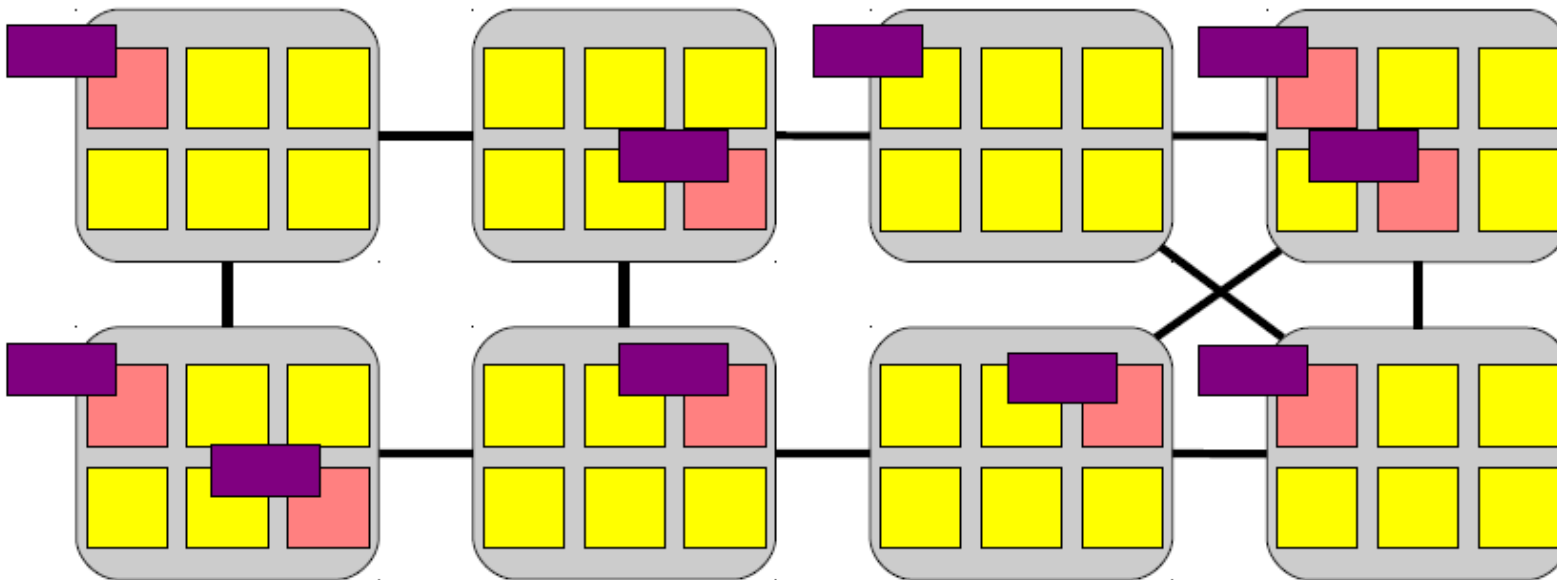
Bunch of cores are spinning

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;
}
```



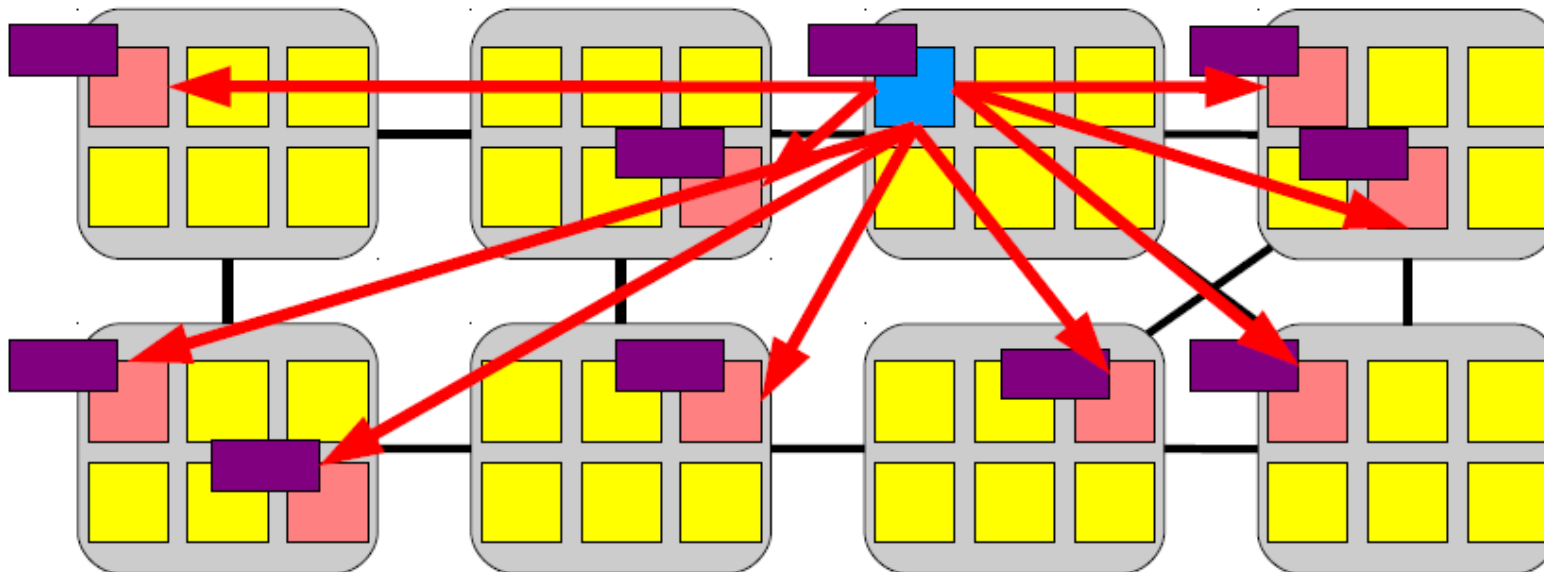
Spin lock implementation

```
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++;
}
```

Broadcast message
(invalidate the value)

```
int next_ticket;
```

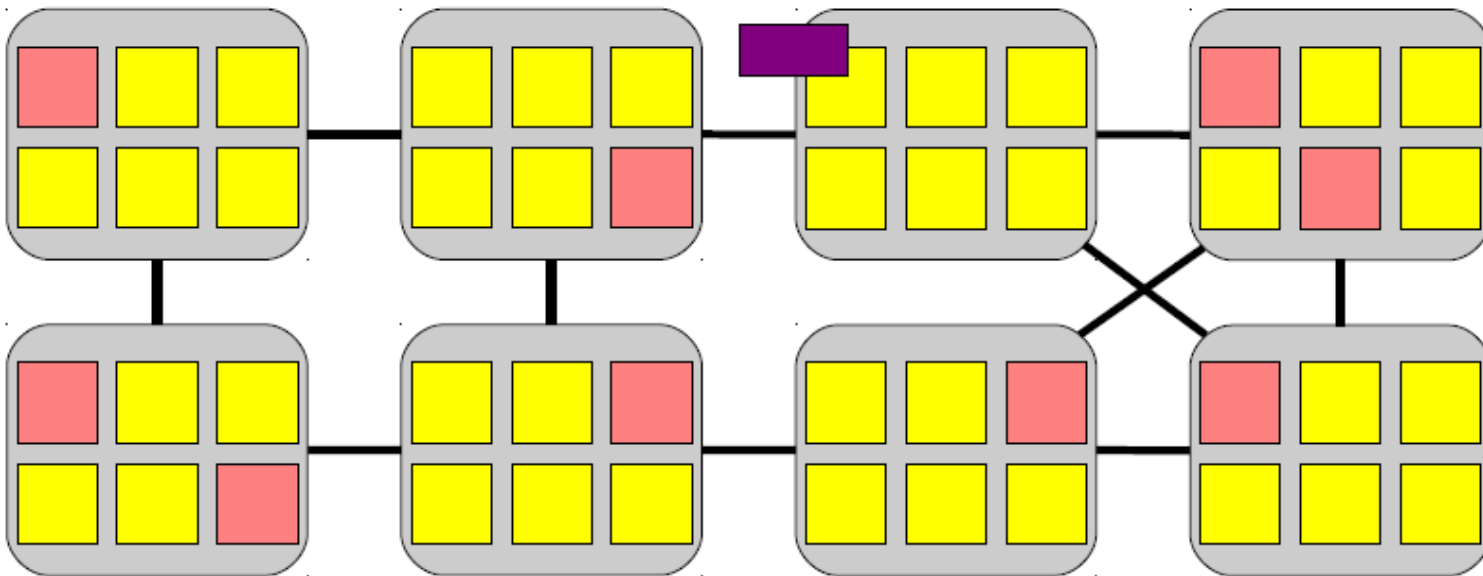


Spin lock implementation

```
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++;
    lock->current_ticket = lock->next_ticket;
}
```

Cores don't have the value of current_ticket



Spin lock implementation

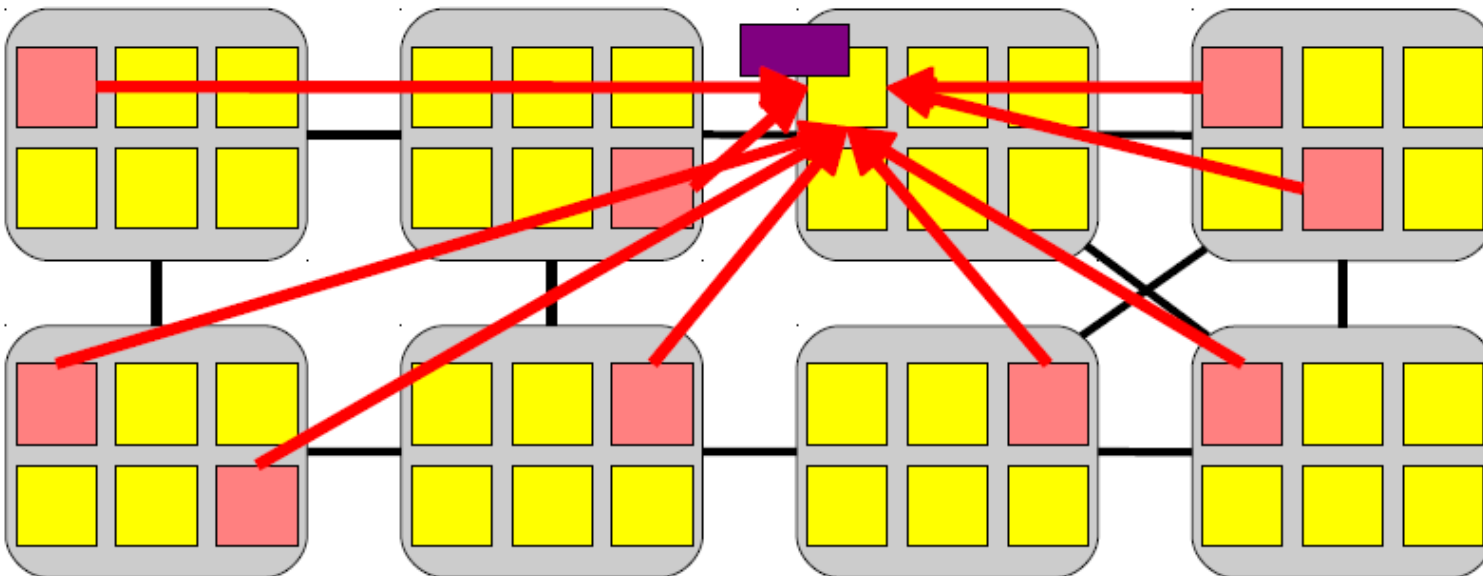
```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

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

Re-read the value

```
{
    ticket;
```

```
int next_ticket;
```



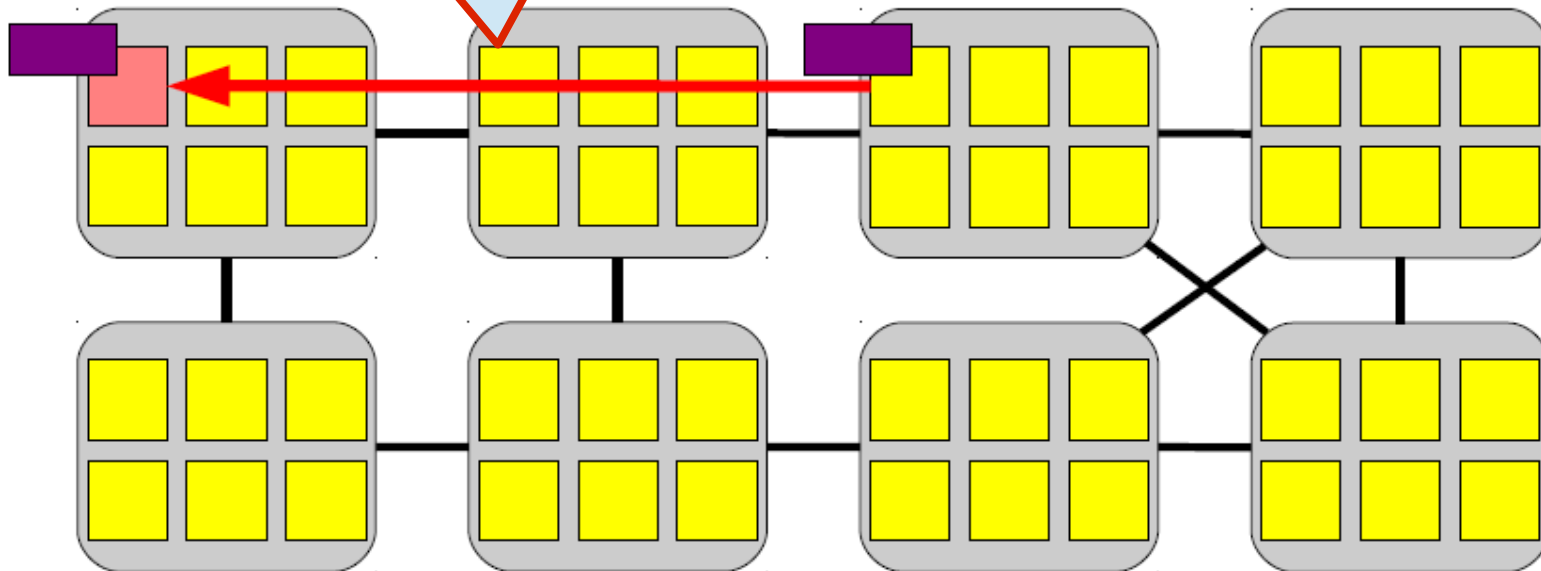
Spin lock implementation

```
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++;
}
```

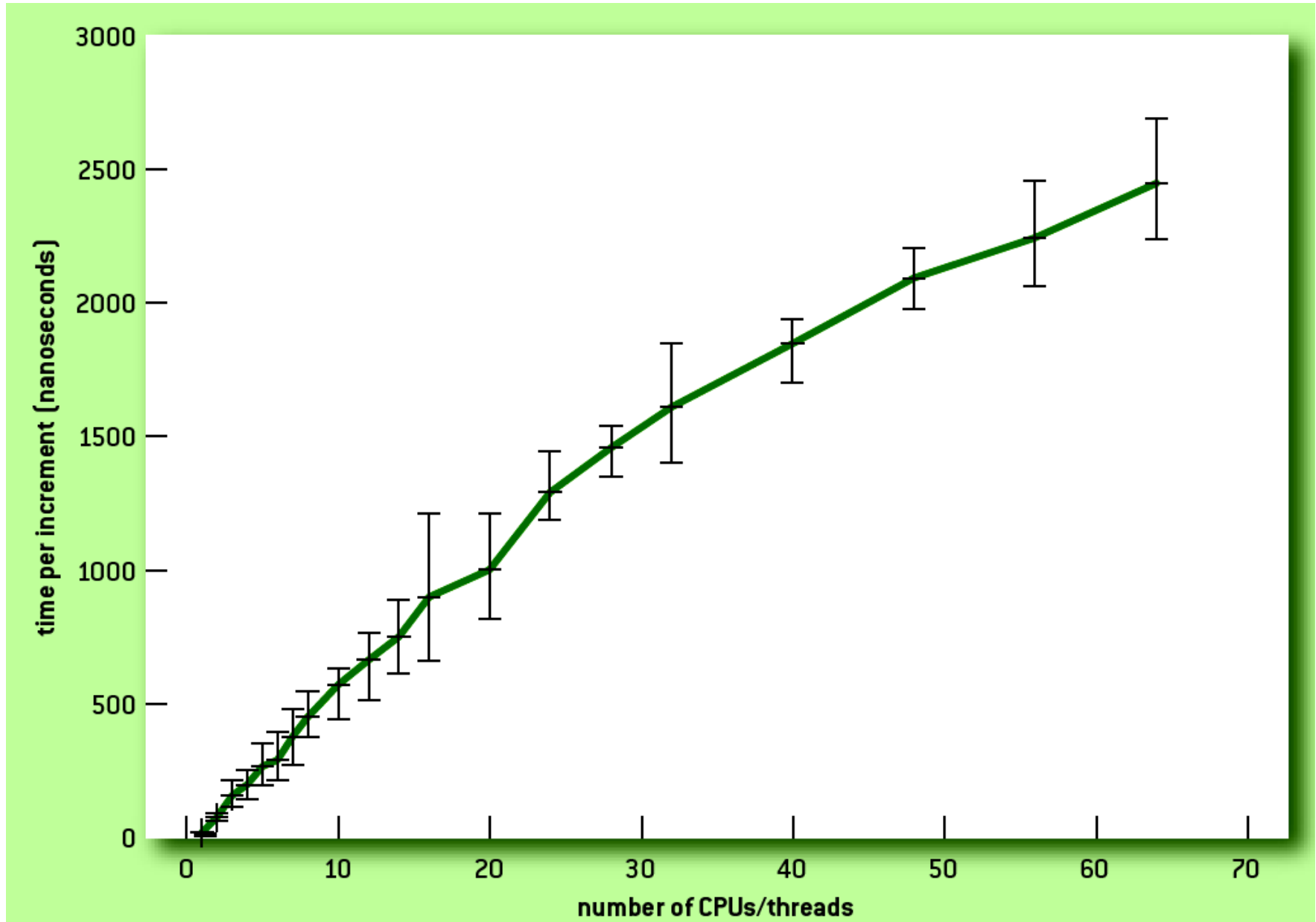
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

500-4000 cycles



Atomic synchronization primitives
do not scale well

Atomic increment on 64 cores



Solution: 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: 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: 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;
}
```

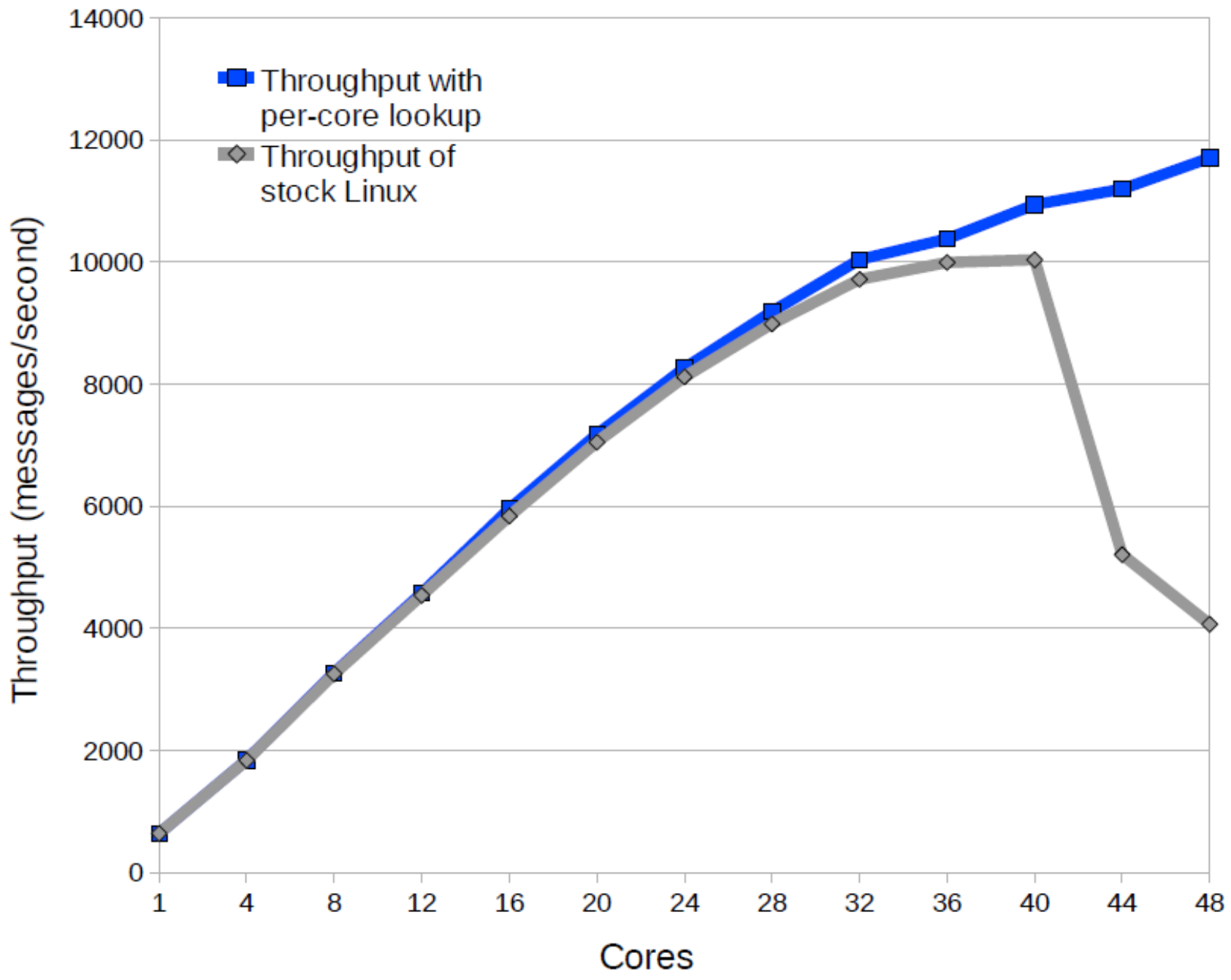
- Fast path: local hash lookup

Solution: per-core mount tables

- Observation: mount table is rarely modified

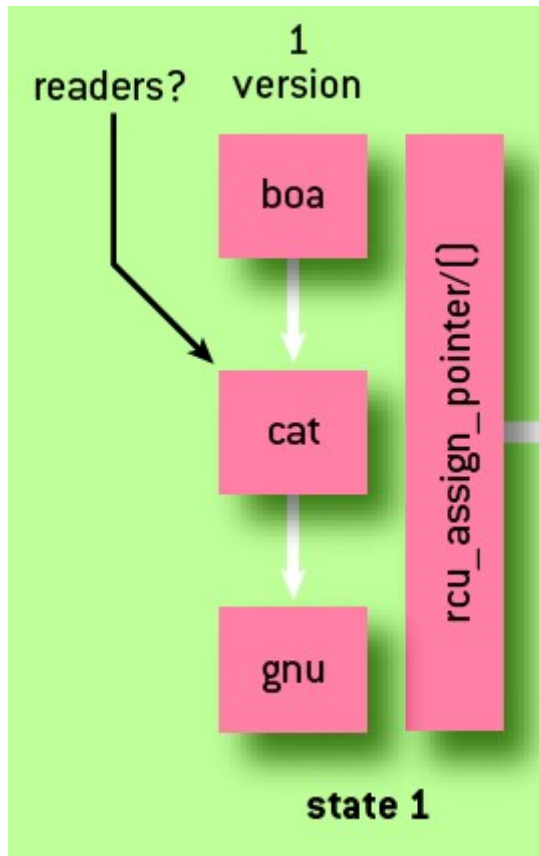
```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return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

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



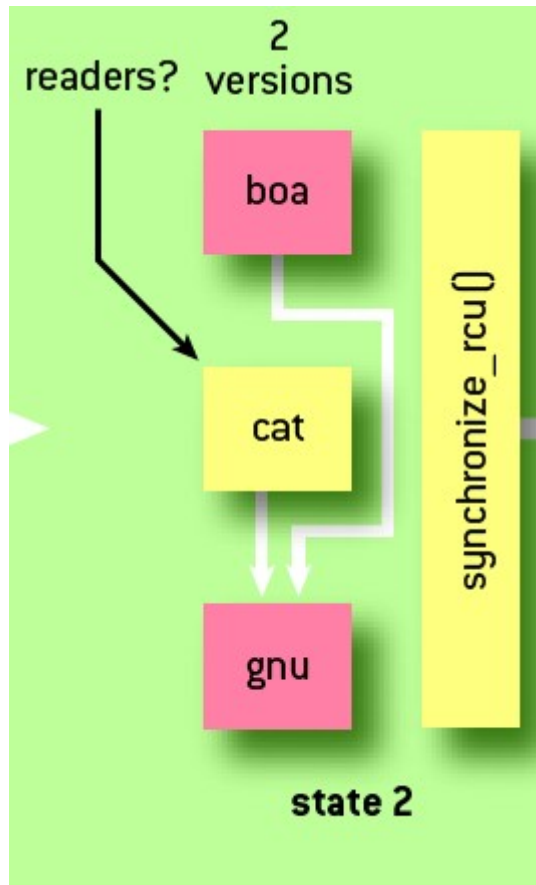
RCU: Read Copy Update

Read copy update



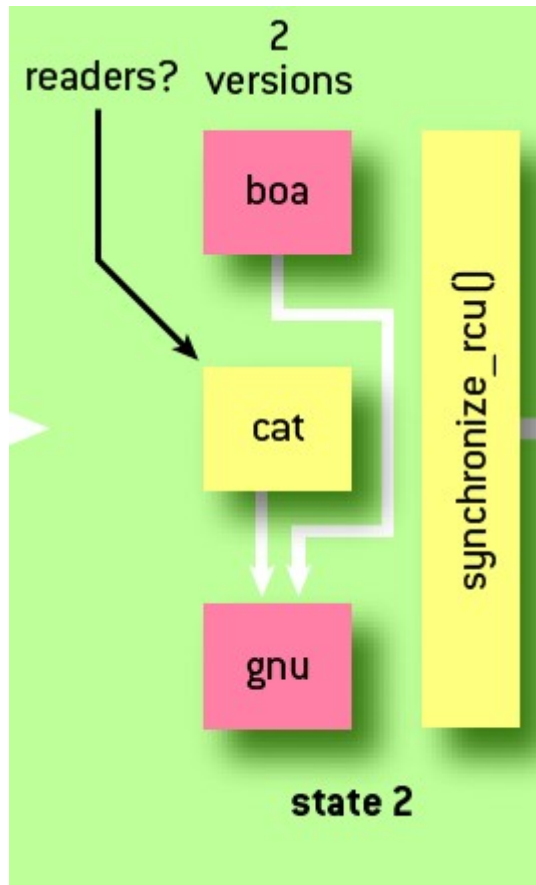
- Goal: remove “cat” from the list
 - There might be some readers of “cat”
- Idea: control the pointer dereference
 - Make it atomic

Read copy update (2)



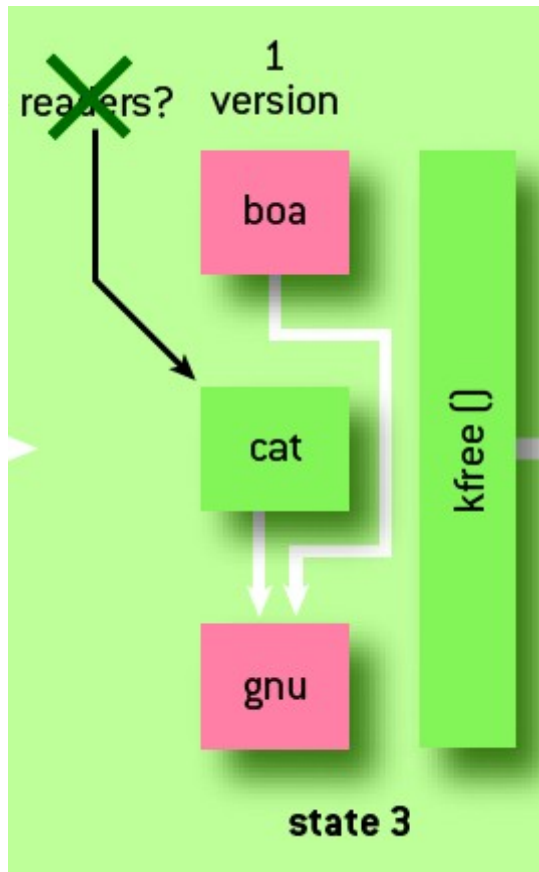
- Remove “cat”
 - Update the “boa” pointer
 - All subsequent reader will get “gnu” as `boa->next`

Read copy update (2)



- Wait for all readers to finish
 - `synchronize_rcu()`

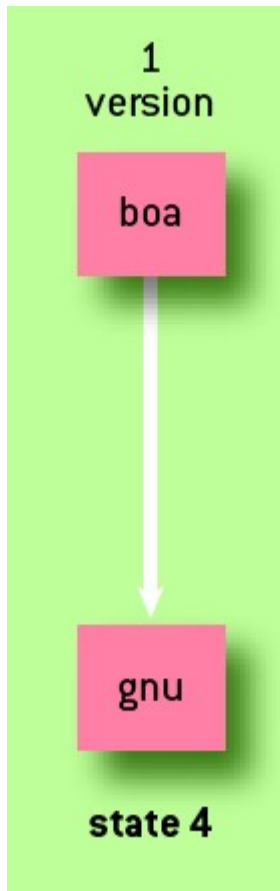
Read copy update (3)



- 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
 - A thread on that CPU was preempted
 - Thus 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;

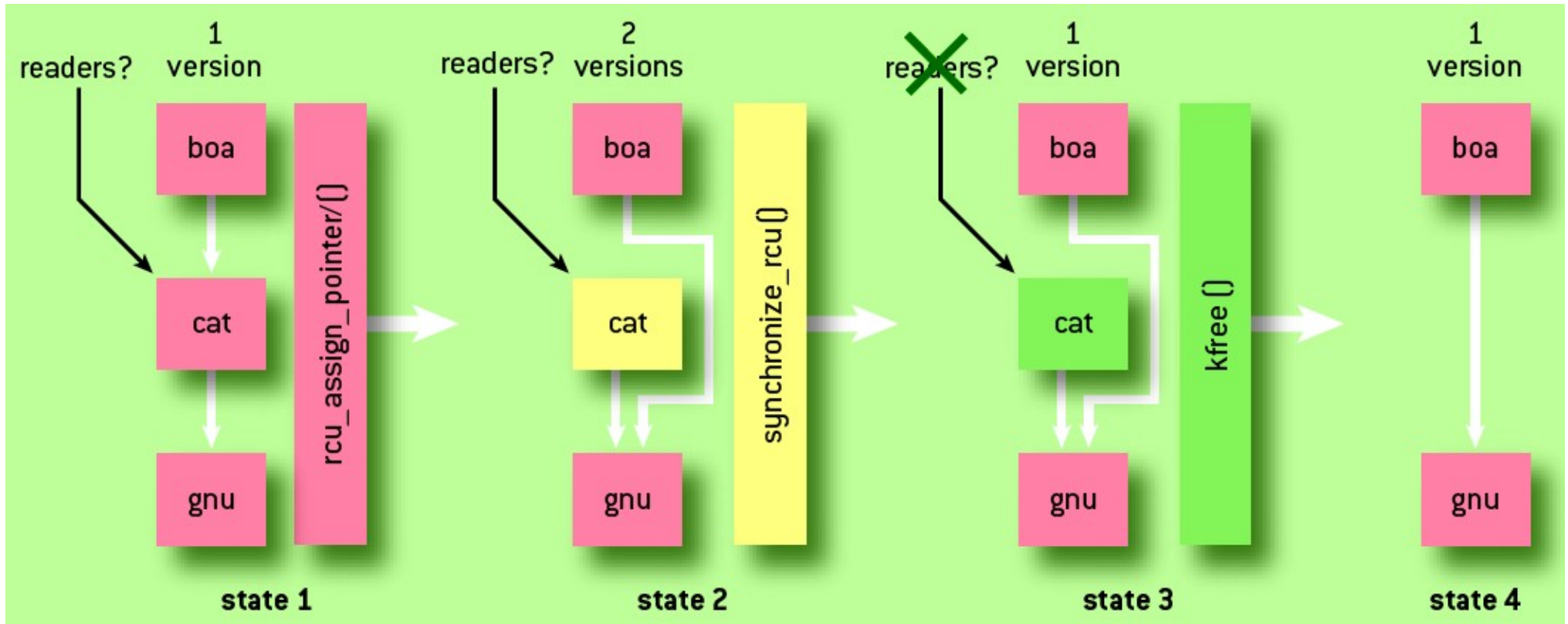
    rcu_read_lock();
    local_table = rcu_deference(table);
    if (local_table != NULL)
        r = local_table[number](args);
    rcu_read_unlock();

    return r;
}
```

**Example: dynamic
system call table**

```
void retract_table()    Table update (well,  
{                       removal)  
    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 [next time]
 - Lock-free synchronization
 - Transactional memory [next time]

Thank you!