Linux Performance

국민대학교 임베디드 연구실
경 주 현

2017 LG MC 인재육성팀 교육

Outline

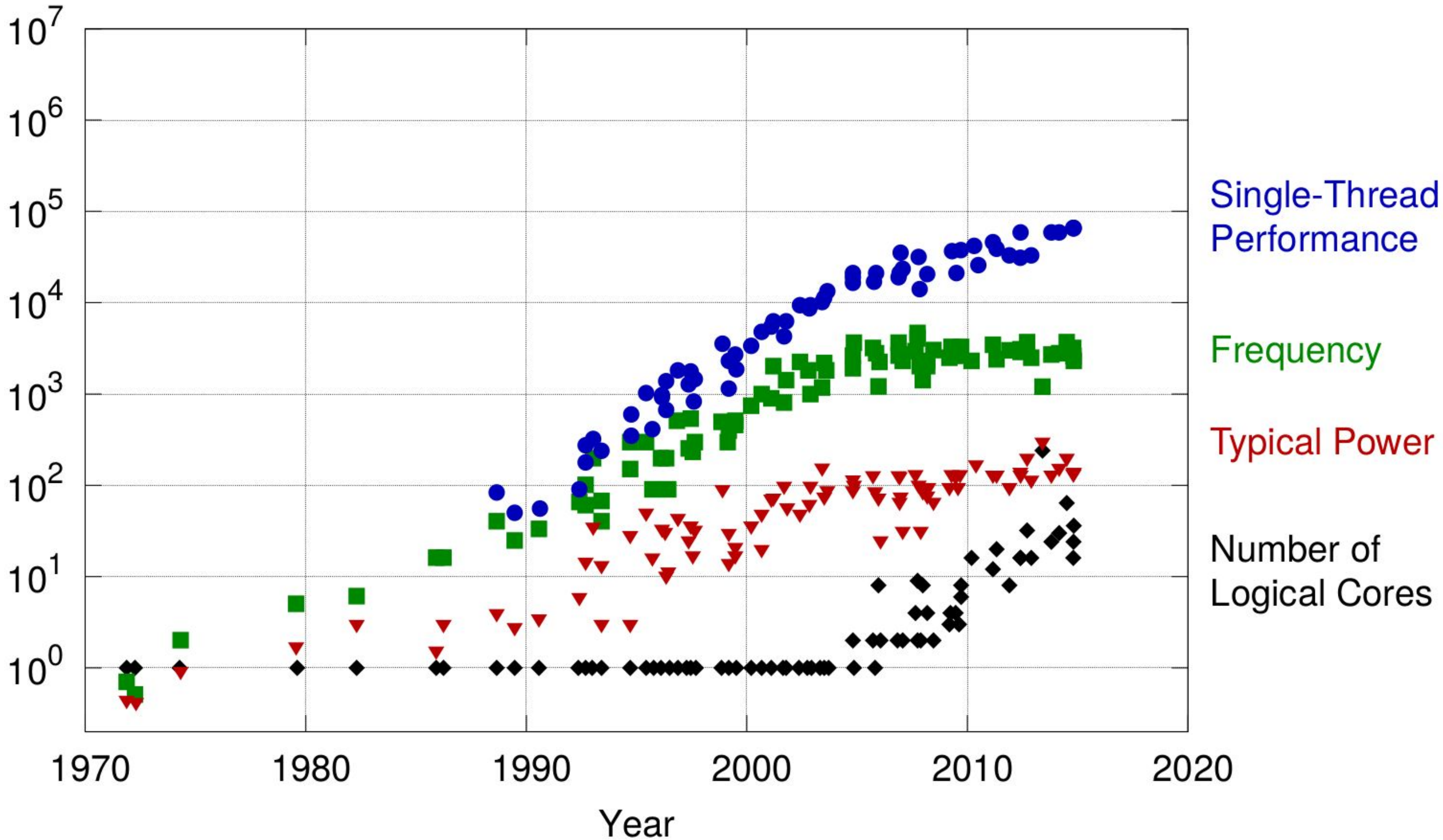
- **Performance**
- **Locking**
- **Multi-core Scalability**
- **Linux Kernel Synchronization History**
  - Ticket Lock
  - Queued Lock(MCS)
  - RCU
  - Nonblocking list(Linux Lock-less list)

Why talk about performance?

Why talk about performance?

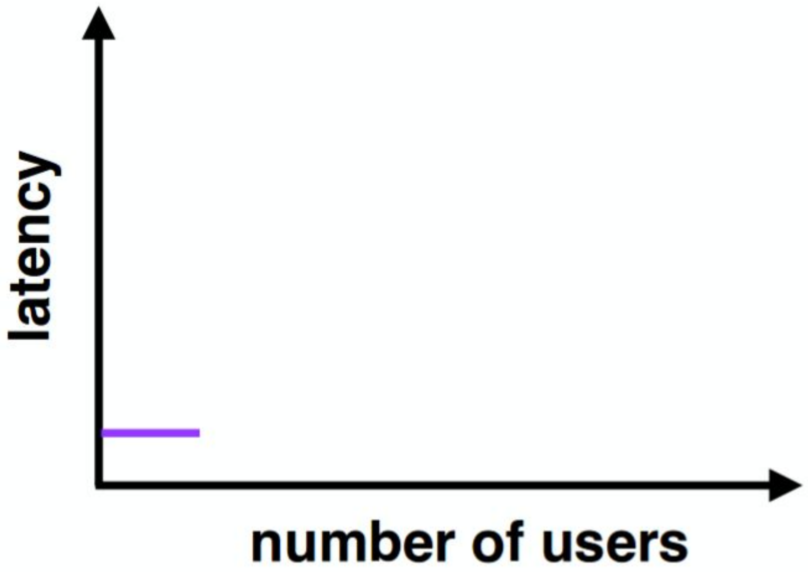
- Servers? Vs. Embedded Systems?
- Performance Scalability Vs. Energy
- Many kernel developers focused on performance scalability.

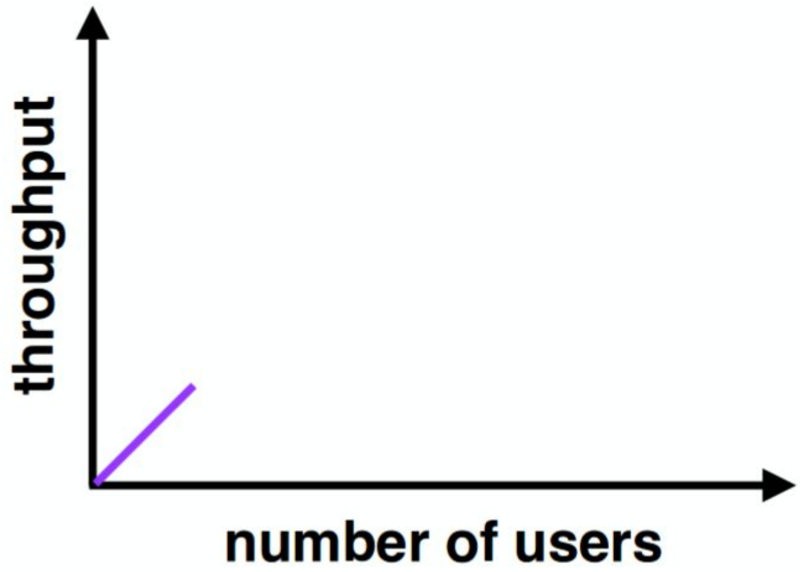
CPU trends



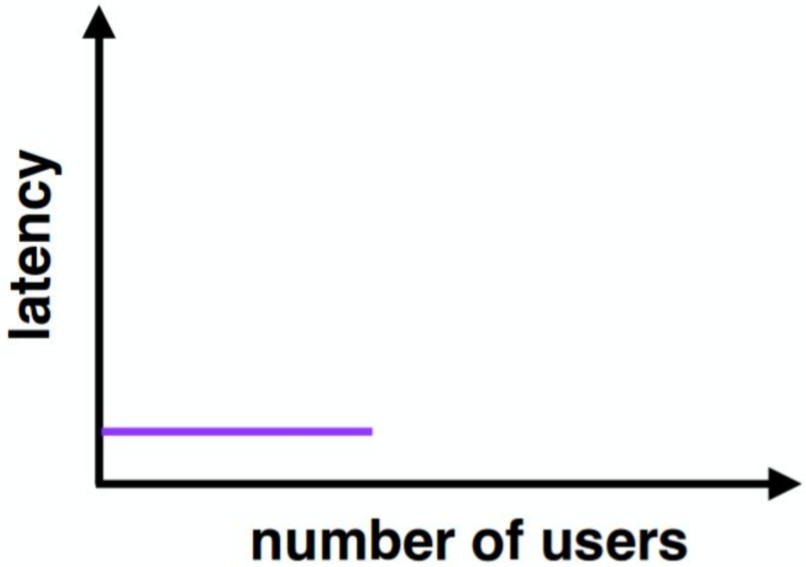
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Single-Thread Performance
Frequency
Typical Power
Number of Logical Cores

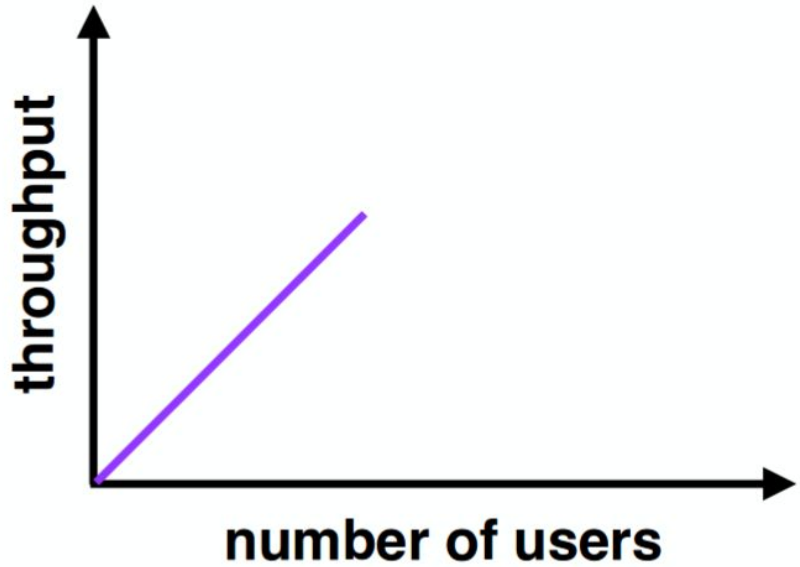
Performance - few users



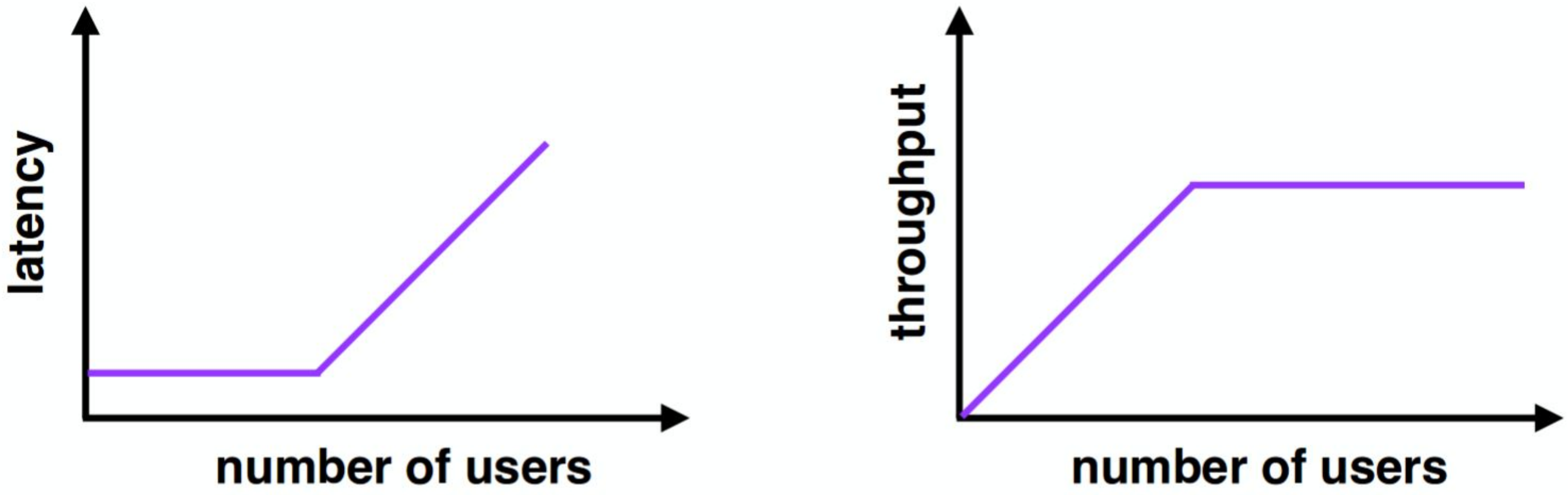


Performance - moderate users





Performance - many users



Performance

- How to Improve Performance?

Two Easy Steps

- **Measure** the system to find the bottleneck

Two Easy Steps

- **Measure** the system to find the bottleneck
- Reduce the **bottleneck**

Two Easy Steps

- Measure the system to find the bottleneck
- Reduce the bottleneck

  1. Better algorithms

Two Easy Steps

- Measure the system to find the bottleneck
- Reduce the bottleneck

  1. Better algorithms

  2. Cache data : Ex) page cache, slab

Two Easy Steps

- **Measure the system to find the bottleneck**
- **Reduce the bottleneck**

1. Better algorithms

2. Cache data : Ex) page cache

3. Concurrency

- CPU: --A-- --B-- --C--
- Disk: --A-- --B-- --C--

Apply concurrency

- CPU: --A----B----C-- ...
- Disk: --A----B--

4. Parallelism

- CPU1: --A-- --D--
- CPU2: --B-- --E--
- CPU3: --C-- --F--

Today talk about multi-core speed-up

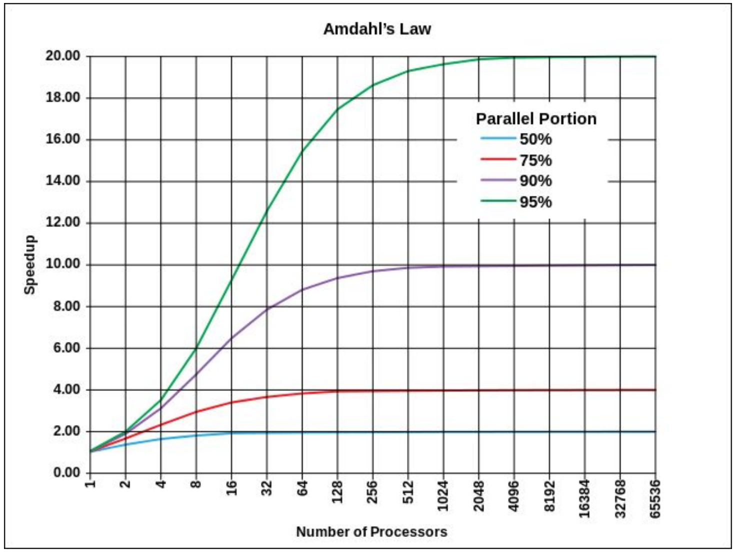
- Concurrency and Parallelism
- **Apps want to use multi-core processors for parallel speed-up**
- **kernel must deal with parallel system calls**
- **Parallel access to kernel data**
  - buffer cache, processes

What is multi-core scalability problem?

What is multi-core scalability problem?

- **Sequential Parts of a Parallel Program!!!**
  - Amdahl's Law: S = 1 / (1 – p + p/n)

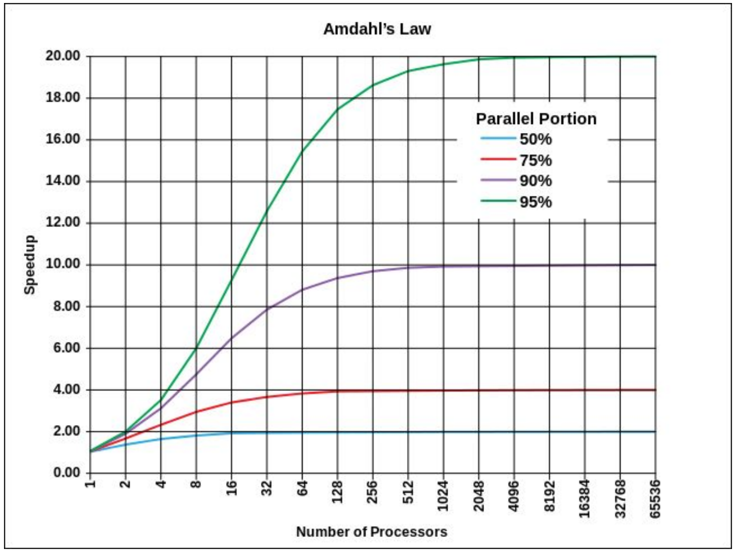
- **90% parallelizable software**
  - 10 processors: 5.3x speedup
  - 20 processors: 6.9x speedup
  - 40 processors: 8.2x speedup



- **and**

What is multi-core scalability problem?

- **Sequential Parts of a Parallel Program!!!**
  - Amdahl's Law: $S = 1 / (1 - p + p/n)$
- **90% parallelizable software**
  - 10 processors: 5.3x speedup
  - 20 processors: 6.9x speedup
  - 40 processors: 8.2x speedup



- **and Locking!!**

KESL Kookmin Univ. Embedded System Lab

Why talk about locking?

- **Locks help with correct sharing of data**

- **Locks can limit parallel speedup**

Locking

- Allow only one CPU to be inside a piece of code at a time.

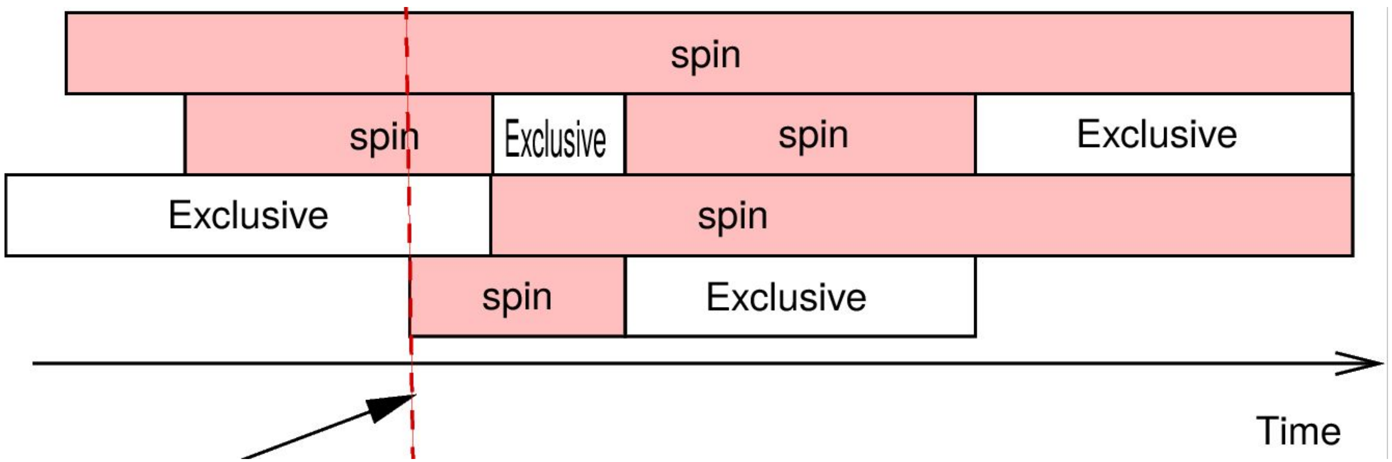
- The lock abstraction

```
lock l
acquire(l)
  x = x + 1 -- "critical section"
release(l)
```

The type of Locks

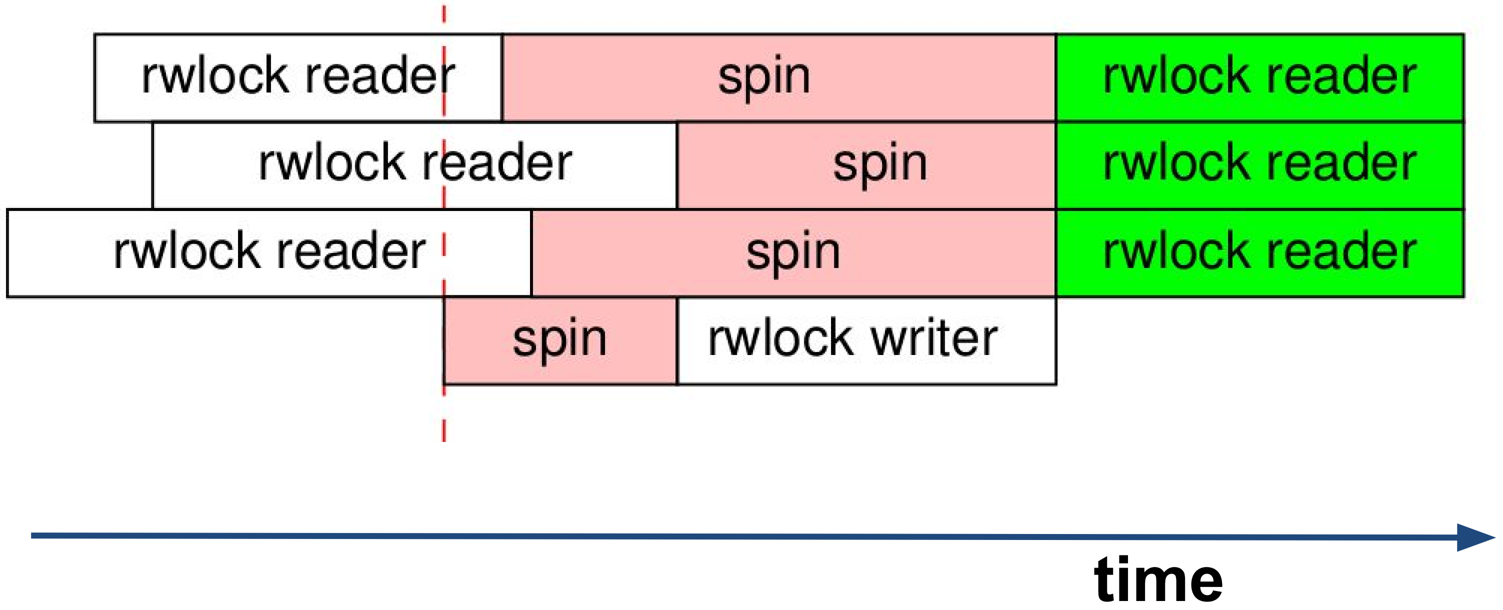
- **Exclusive Lock**
  - spinlock, mutex
- **Reader-writer lock**
  - reader-writer semaphore
- **RCU(Read Copy Update)**
  - RCU, SRCU

Exclusive Lock

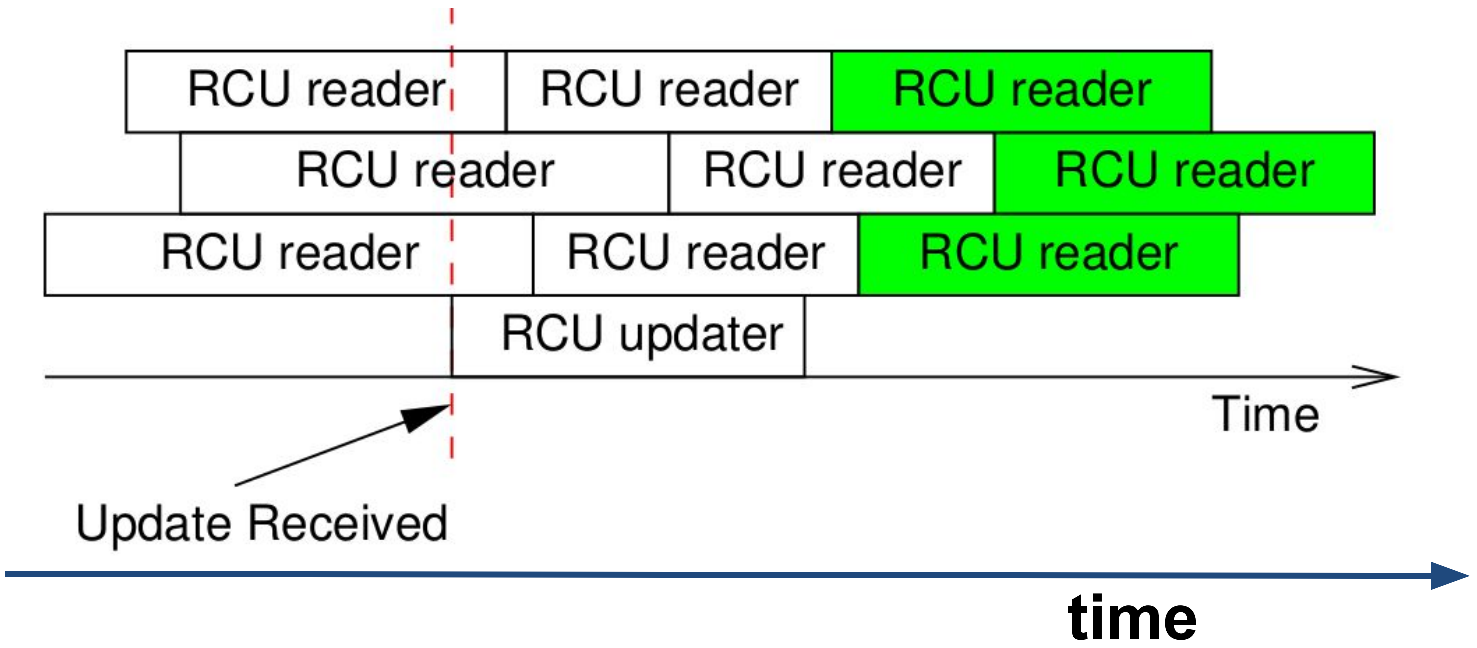


Reader-writer locks

- **ex) reader-writer semaphore**
  - spin -> block



RCU(Read Copy Update)



Exclusive Lock

How to implement locks?

How to implement locks?

```
struct lock { int locked; }
struct lock l;

 acquire(l) {
   while(1){
     if(l->locked == 0) {
       l->locked = 1;
       return;
     }
   }
 }
```

How to implement locks?

- **why not?**

```
struct lock { int locked; }
struct lock l;

acquire(l) {
  while(1){
    if(l->locked == 0) {    // A
      l->locked = 1;        // B
      return;
    }
  }
}
```

Solution

- **Atomic exchange instruction**
  - **SWAP or xchg** : swap operation

Simple spin lock

- **Atomic exchange instruction**
  - **SWAP or xchg** : swap operation

Now:

```
acquire(l){
  while(1){
    if(xchg(&l->locked, 1) == 0){
      break
    }
  }
}
```

Why spin locks?

- **Don't they waste CPU while waiting?**

Why spin locks?

- **Don't they waste CPU while waiting?**
- **Spin lock guidelines:**
  - Hold spin locks for very short times
  - Don't yield CPU while holding a spin lock

Why spin locks?

- **Don't they waste CPU while waiting?**
- **Spin lock guidelines:**
  - Hold spin locks for very short times
  - Don't yield CPU while holding a spin lock
- **Systems often provide "blocking" locks for longer critical sections**
  - waiting threads yield the CPU

    but overheads are typically higher

Lock’s Problem

- **The problem is multi-core caching**

How to ensure caches aren't stale?

- core 1 : reads + caches x=10,
- core 2 : writes x=11, core 1 reads x=?

How to ensure caches aren't stale?

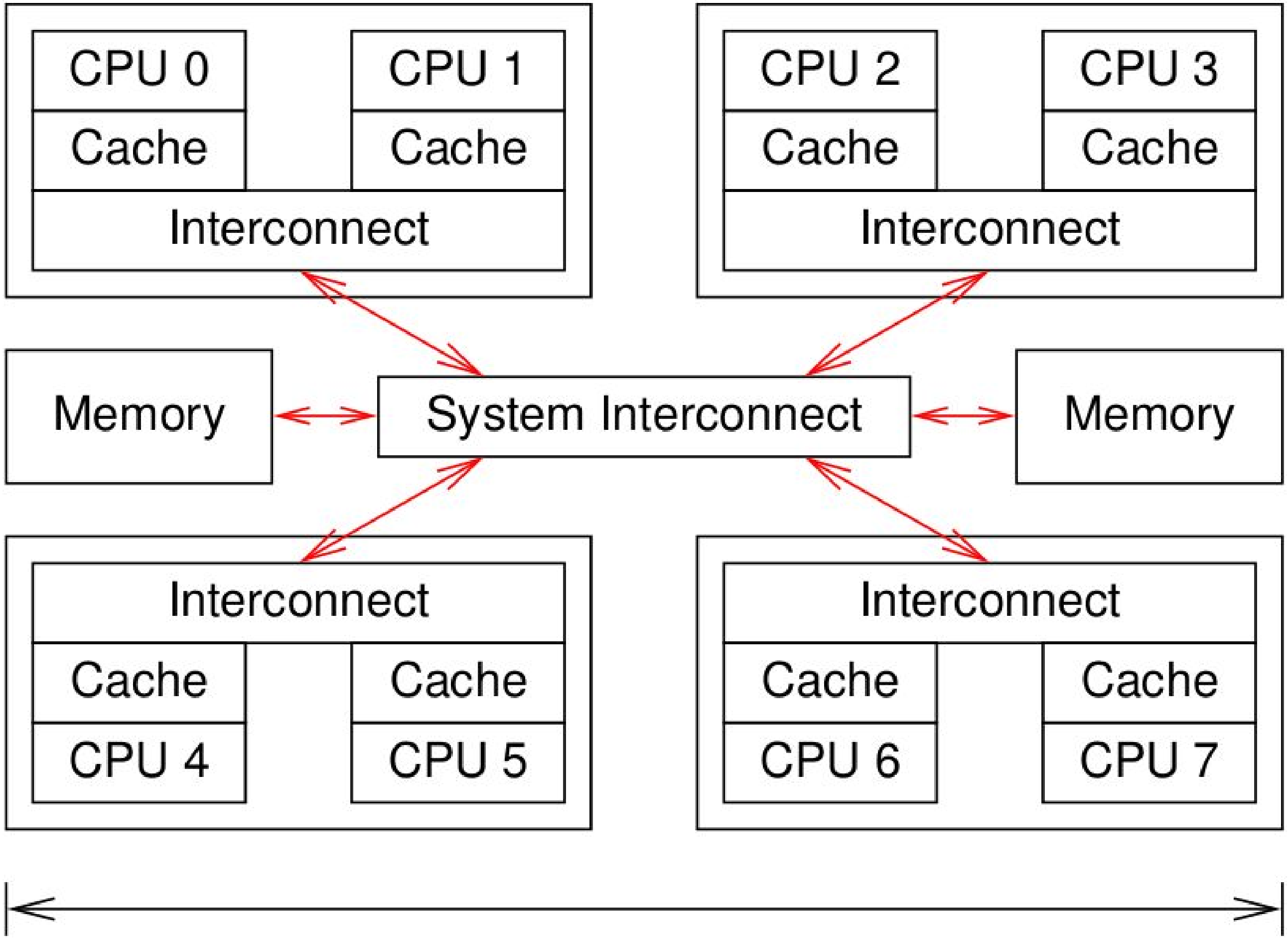
- core 1 : reads + caches x=10,
- core 2 : writes x=11, core 1 reads x=?

- answer:

"**cache coherence protocol**"

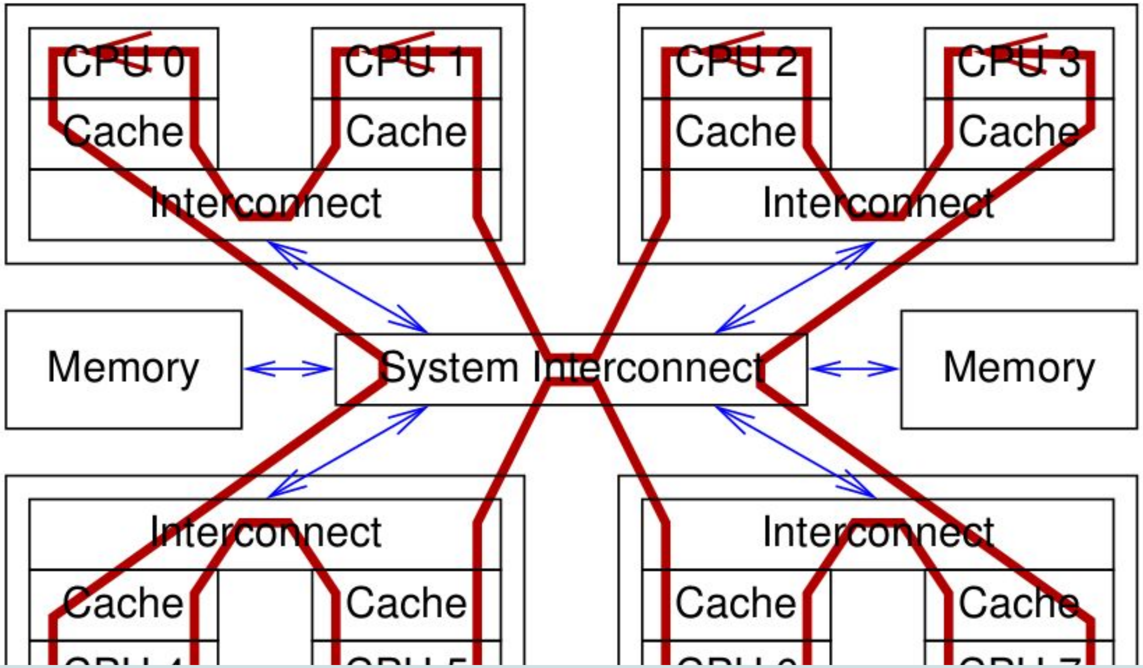
System Interconnect

- Cache coherence protocol
- All messages are broadcast to all cores



Data Flow For Global Atomic Increment

- Thread a : atomic_inc(&global_value;
- Thread b : atomic_inc(&global_value);
- Thread c : atomic_inc(&global_value);
- Thread d : atomic_inc(&global_value);



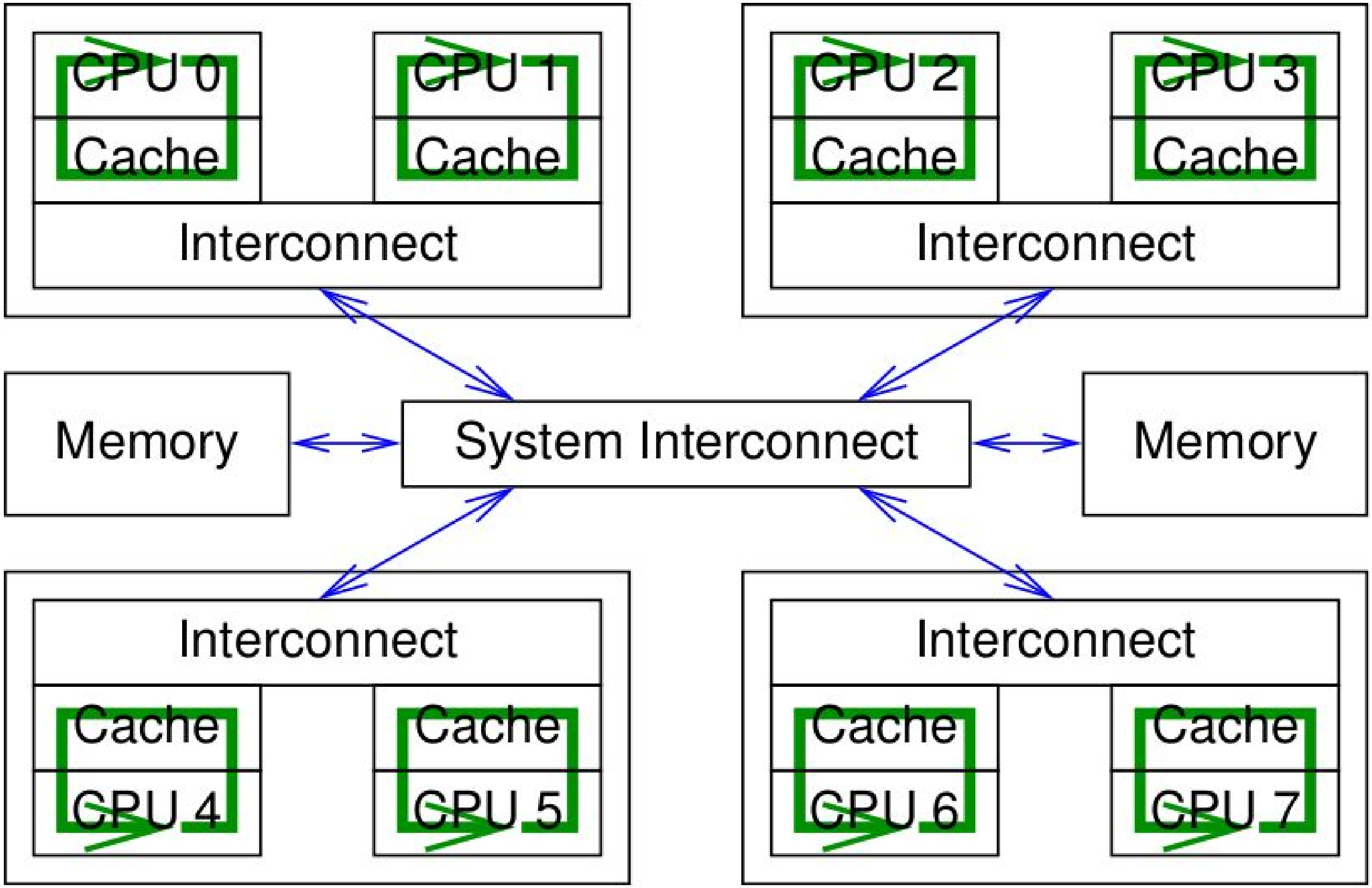
- Bottleneck is usually the interconnect

Goal : per-core partitioning

- **How to improve the scalability?**

Goal : per-core partitioning

- **How to improve the scalability?**



Locks and parallelism

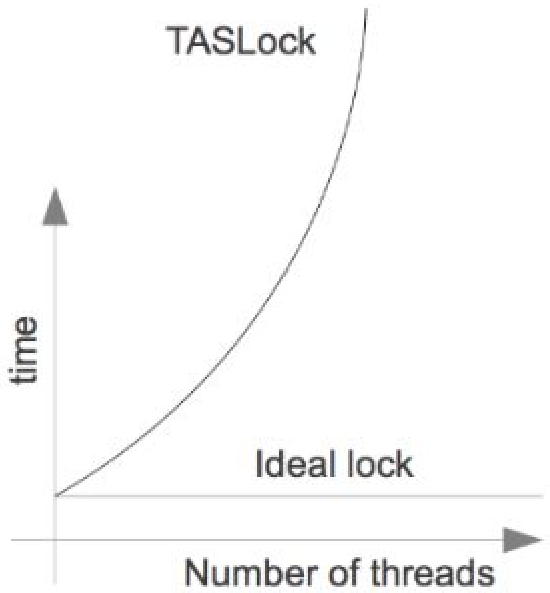
- **Locks prevent parallel execution to get parallelism**
- **How to improve parallelism.**
- **First, use fine grained locks**
  - minimizing the lock scope.
- **Second, You may need to re-design code to make it work well in parallel**

Real World?

Test and Set(Old method)

```
T&S (*addr) {
    old = *addr;
    *addr = 1;
    return (old == 0);
}
```

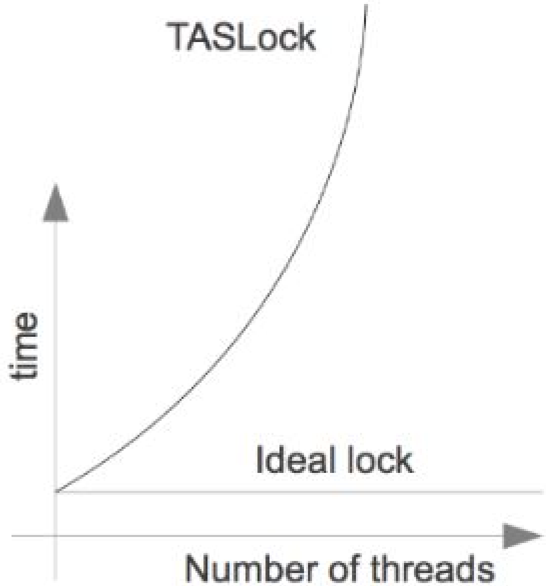
```
bool locked = false

void lock () {
    while ( !T&S(locked) );
}

void unlock() {
     locked = false;
}
```



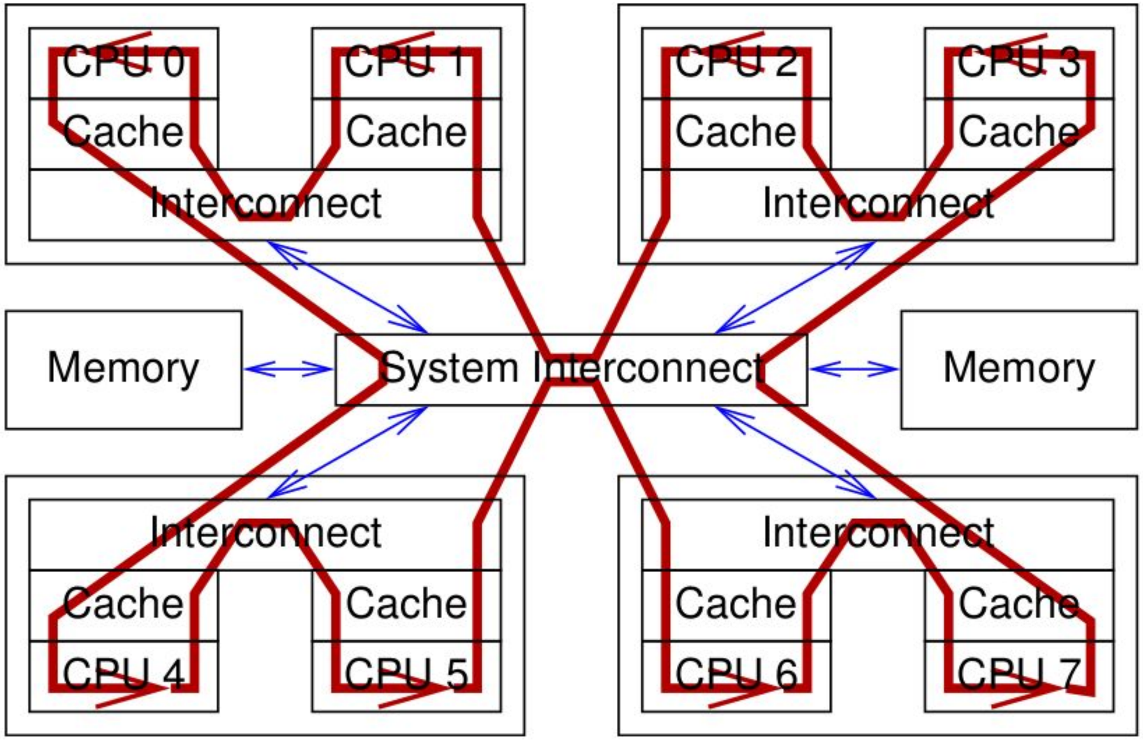
Test and Set(Old method)

```
T&S (*addr) {
    old = *addr;
    *addr = 1;
    return (old == 0);
}
```

```
bool locked = false

void lock () {
    while ( !T&S(locked) );
}

void unlock() {
     locked = false;
}
```



Data Flow For Global Atomic Increment

- 

1. **Bottleneck is usually the interconnect**
2. **t-s locks aren't fair**

Ticket locks (linux):

- Goal:
- Read-only spin loop, rather than repeated atomic instruction
- Fairness (turns out t-s locks aren't fair)

- Idea:
- Assign numbers, wake up one at a time
  avoid constant t-s atomic instructions by waiters

Ticket locks

```
int now_serving = 0;
int next_ticket = 0;

void lock () {
    my_ticket = F&I(next_ticket);
    while (my_ticket != now_serving);
}

void unlock() {
     now_serving++;
}
```

```
F&I (*addr) {
    old = *addr;
    *addr++;
    return old;
}
```

Ticket locks

```
int now_serving = 0;
int next_ticket = 0;

void lock () {
    my_ticket = F&I(next_ticket);
    while (my_ticket != now_serving);
}

void unlock() {
     now_serving++;
}
```

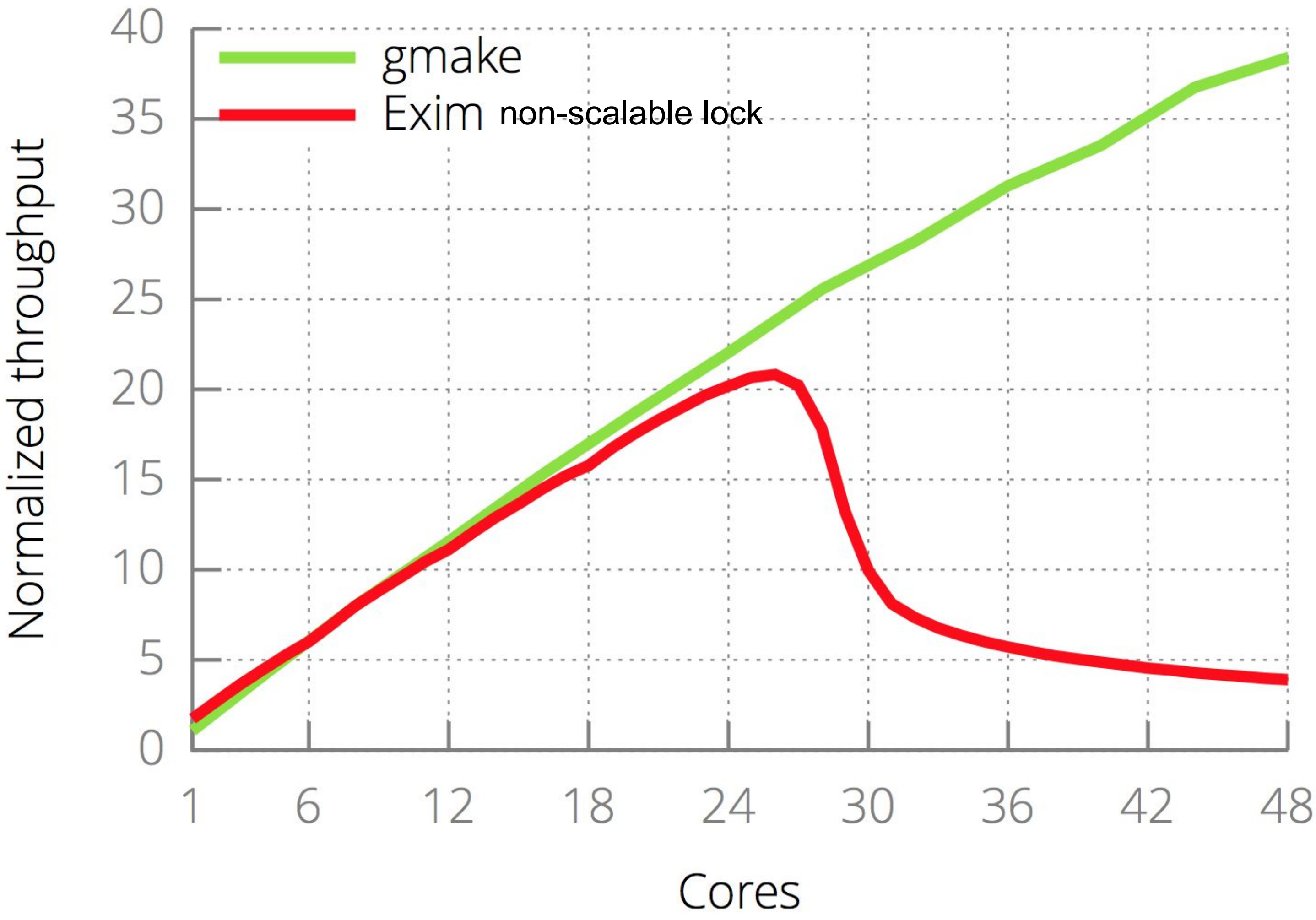
```
F&I (*addr) {
    old = *addr;
    *addr++;
    return old;
}
```

Ticket locks Problem

- Read-only spin loop, rather than repeated atomic instruction
- But still bottlenecked due to cache invalidation message.
- Release lock
  - invalidate all cpu.

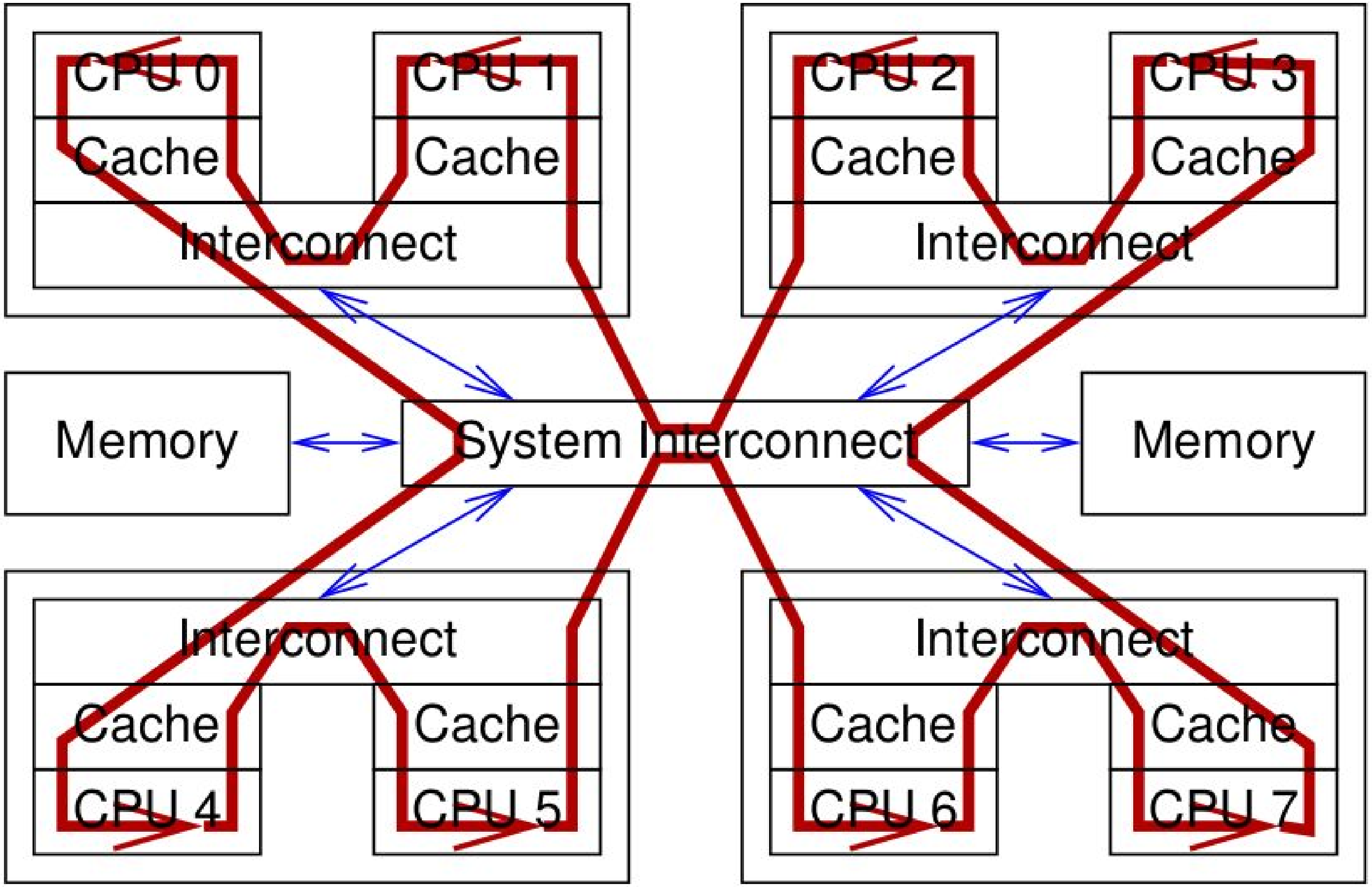
But...

- **Ticket lock's problem**



Ticket locks (linux):

•



How to make locks scale well?

- A partitioning method
- What if each core spins on a *different* cache line?
- Avoid useless invalidations
- By keeping a queue of threads

Anderson lock

- **Invalidate next holder's slots[]**
- **Each thread Notifies next in line**
  - Without bothering the others
- **Problem**

Anderson lock

- **Invalidate next holder's slots[]**

- **Each thread Notifies next in line**
  - Without bothering the others

- **Problem: high space cost**
  - O(lock * cores)
  - N slots per lock

MCS

- **Goal:**
  - as scalable as anderson, but less space used

- **Idea: one list element per thread, since a thread can wait on only one lock**
  - so total space is O(locks + threads), not O(locks*threads)

- **Idea: linked list of waiters per lock**

  1. Pushes caller's element at end of list
  2. Spins on a variable in its own element
  3. pops its own element

MCS

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

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

```c
static inline void
mcs_lock(mcslock_t *l, volatile struct qnode *mynode)
{
    struct qnode *predecessor;

    mynode->next = NULL;
    predecessor = xchg(&l->tail, mynode);

    if (predecessor) {
        mynode->locked = 1;
        predecessor->next = mynode;
        while (mynode->locked)  // local spin
            ;
    }
}
```

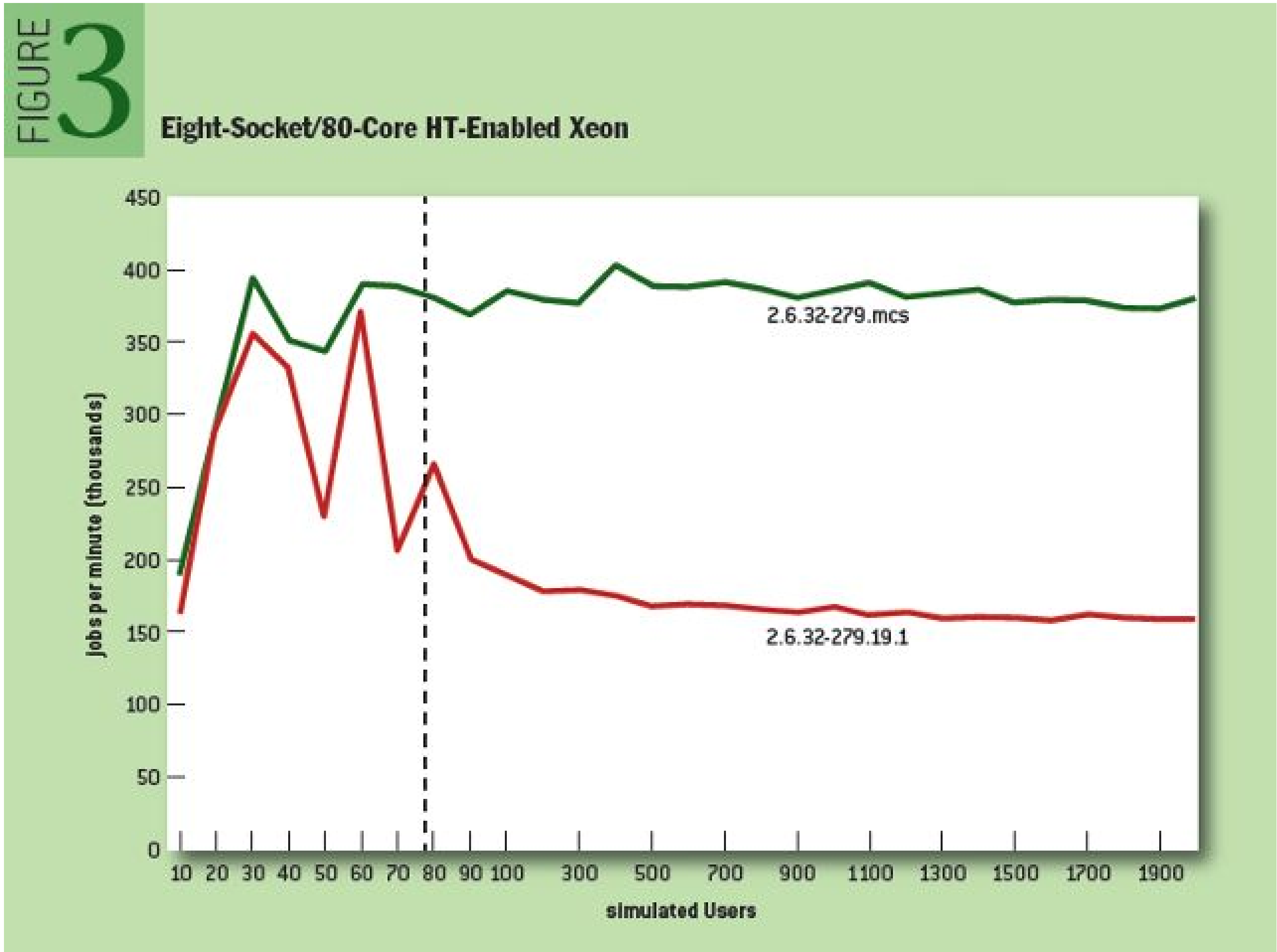
1. Pushes caller's element at end of list

2. Spins on a variable in its own element

```c
static inline void
mcs_unlock(mcslock_t *l, volatile struct qnode *mynode)
{
    if (!mynode->next) {
        if (cmpxchg(&l->tail, mynode, 0) == mynode)
            return;
        while (!mynode->next) ;
    }
    mynode->next->locked = 0;  //unlock
}
```

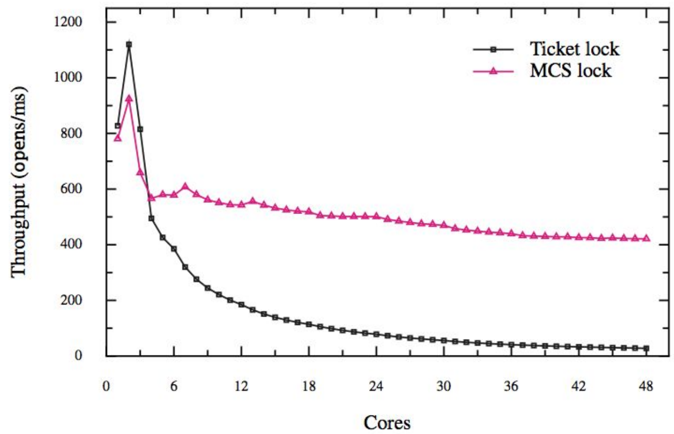
3. unlock own element

MCS and Linux scalability

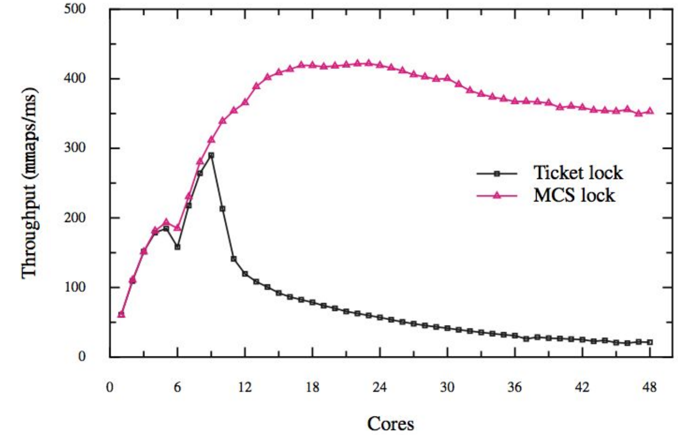


"AIM7 file-system benchmark" http://queue.acm.org/detail.cfm?id=2698990

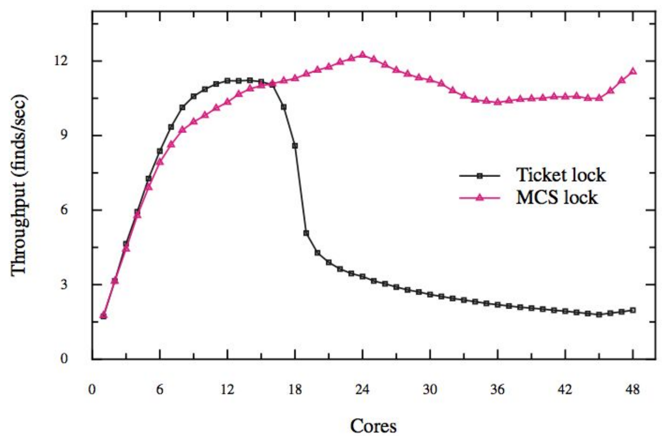
MCS and Linux scalability



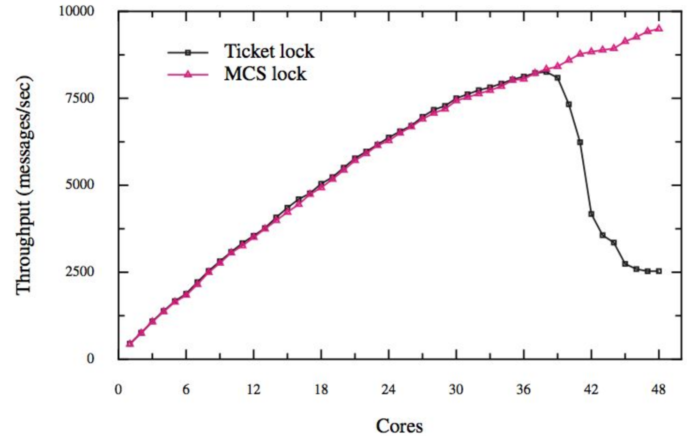
(a) Performance for FOPS.



(b) Performance for MEMPOP.



(c) Performance for PFIND.



(d) Performance for EXIM.

Notes on Linux and MCS locks

- **The Linux developers came up with “qspinlocks”**
  - Avoid bloating the size of the MCS

- **The old and unused ticket-spinlock implementation was deleted from the mainline in 2016.**

What about blocking lock?

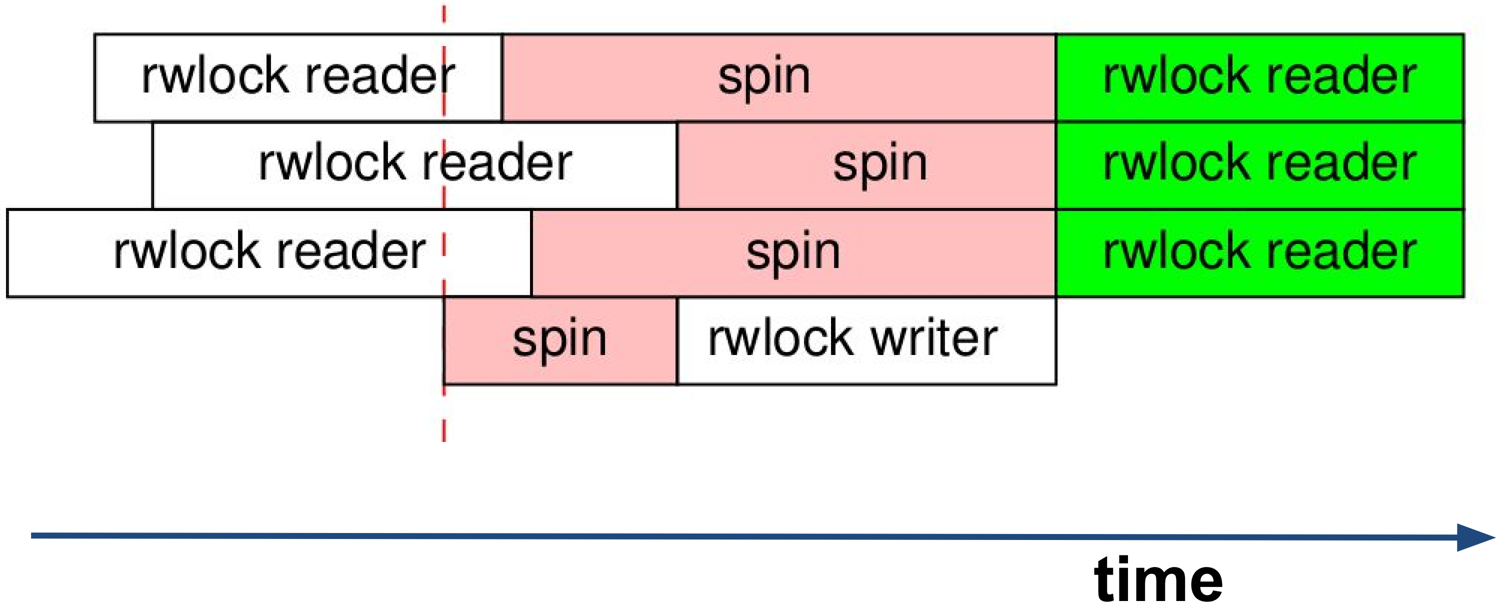
- **Linux kernel mutex or rw_semaphore (reader-writer semaphore)**

Reader-writer lock

What about blocking lock?

- **Linux kernel mutex or rw_semaphore (reader-writer semaphore)**

Reader-writer locks



Blocking lock Problem

- **Problem :**
  - Scheduling overhead because threads yield the CPU
  - Block -> Runqueue -> Run

Davidlohr Bueso. 2014. Scalability techniques for practical synchronization primitives. *Commun. ACM* 58

Blocking lock

- **Problem :**
  - Scheduling overhead because threads yield the CPU
  - Block -> Runqueue -> Run
- **Solution :**

  Hybrid Locking Models and Optimistic Spinning

Davidlohr Bueso. 2014. Scalability techniques for practical synchronization primitives. *Commun. ACM* 58

Hybrid Locking Models and Optimistic Spinning

- **Fastpath.**
  - Acquire the lock atomically by modifying an internal counter such as fetch_and_add or atomic decrementing.
- **Midpath (aka optimistic spinning)**
  - Tries to spin for acquisition while the lock owner is running
  - Spinner threads are queued up using an MCS lock
- **Slowpath.**
  - Task is added to the wait queue and sleeps

Davidlohr Bueso. 2014. Scalability techniques for practical synchronization primitives. *Commun. ACM* 58

RCU

Blocking lock Problem

- **Problem :**
  - Scheduling overhead because threads yield the CPU
  - Block -> Runqueue -> Run

Blocking lock

- **Problem :**
  - Scheduling overhead because threads yield the CPU
  - Block -> Runqueue -> Run
- **Solution :**

  RCU

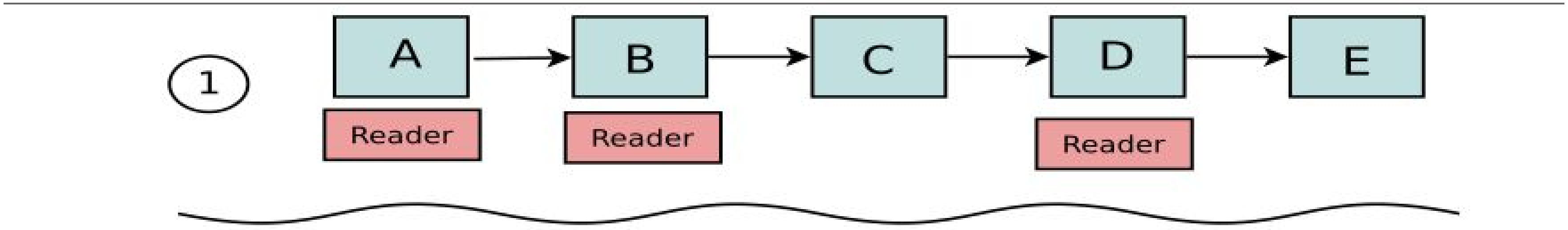
Davidlohr Bueso. 2014. Scalability techniques for practical synchronization primitives. *Commun. ACM* 58

RCU

- **A per-core partitioning algorithm**
- **Read-mostly data structure**
- **Reader Lock / Unlock**
  - enter() / exit() in per-core memory
- **Writer Lock / Unlock**
  - look-up all thread's local value and delayed free

RCU Usage (Route Table Lookup)

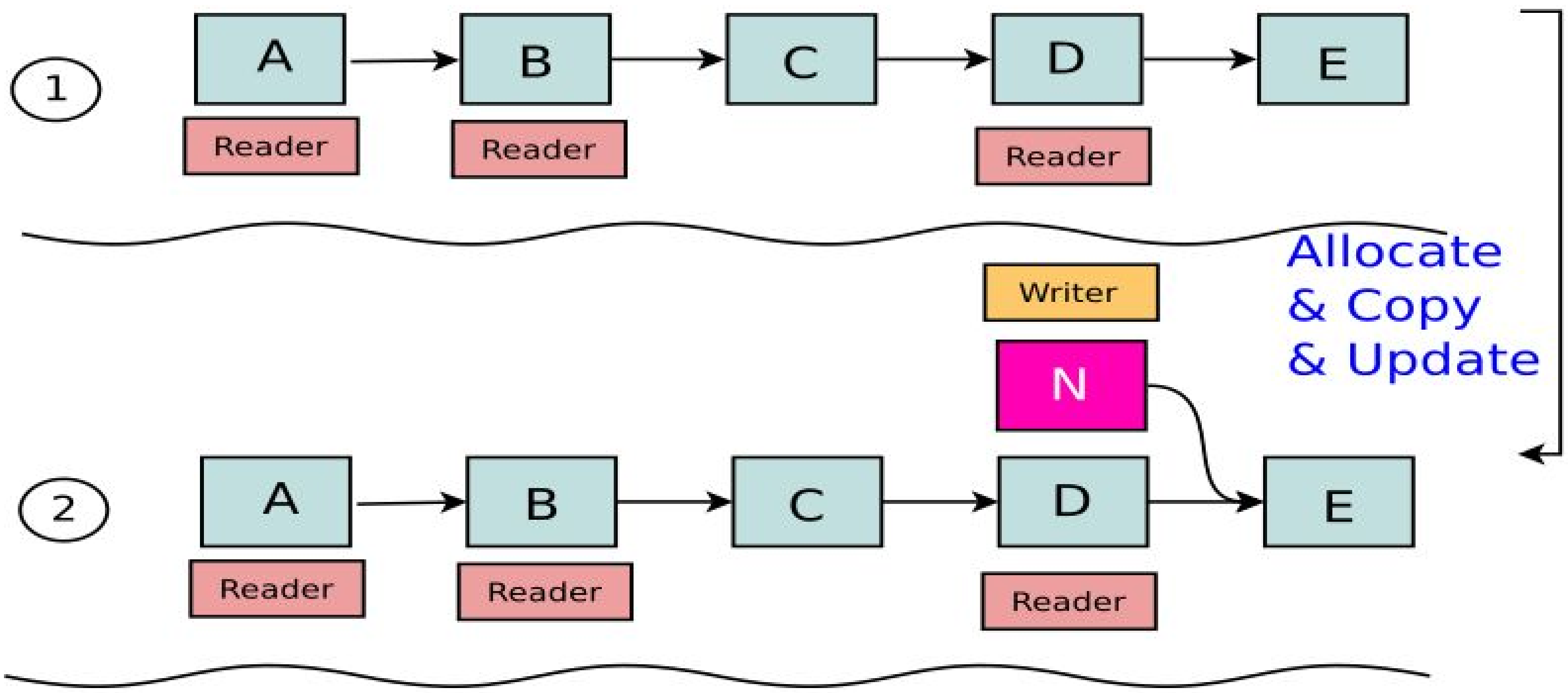
- **Packet routing list**
- **No cache coherence protocol problem**
  - Using per-core marking



RCU Usage (Route Table Lookup)

```
struct route_entry {
  struct rcu_head rh;
  struct cds_list_head re_next;
  unsigned long addr;
  unsigned long iface;
  int re_freed;
};
CDS_LIST_HEAD(route_list);
DEFINE_SPINLOCK(routelock);

unsigned long route_lookup(unsigned long addr)
{
  struct route_entry *rep;
  unsigned long ret;

  rcu_read_lock();
  cds_list_for_each_entry_rcu(rep, &route_list,
                              re_next) {
    if (rep->addr == addr) {
      ret = rep->iface;
      if (ACCESS_ONCE(rep->re_freed))
        abort();
      rcu_read_unlock();
      return ret;
    }
  }
  rcu_read_unlock();
  return ULONG_MAX;
}
```

RCU (Linked List)

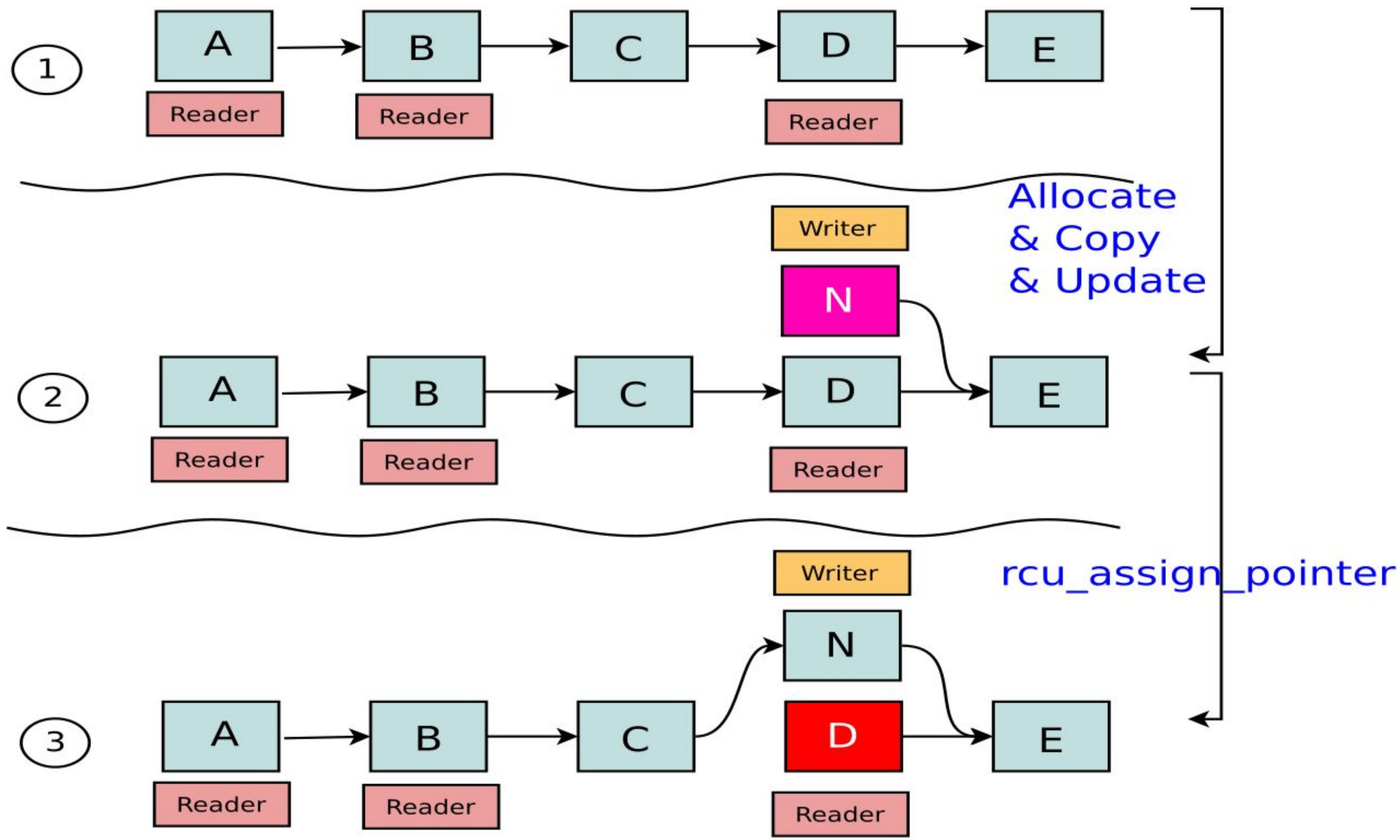


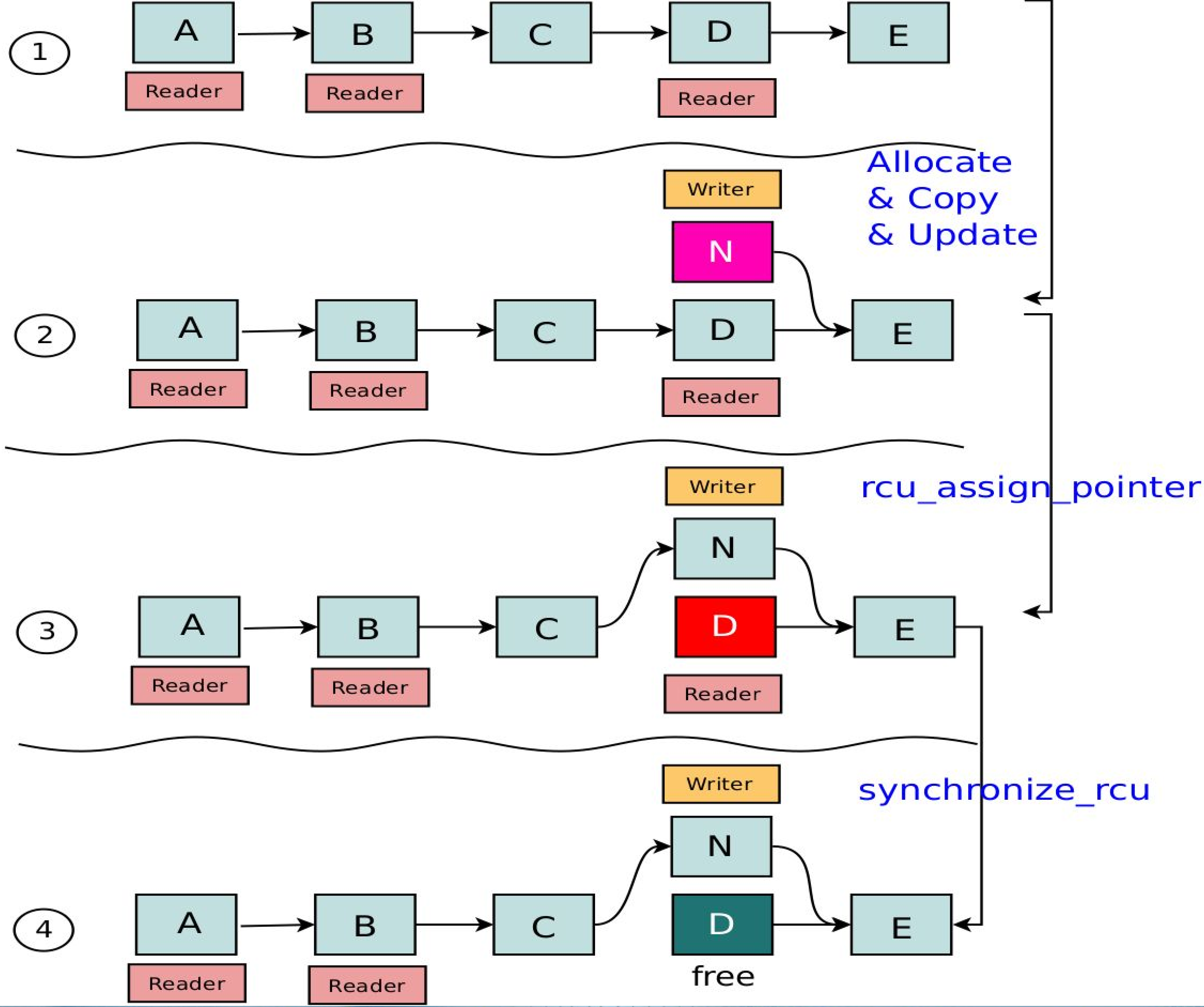
RCU Usage - add

```
int route_add(unsigned long addr,
              unsigned long interface)
{
  struct route_entry *rep;

  rep = malloc(sizeof(*rep));
  if (!rep)
    return -ENOMEM;
  rep->addr = addr;
  rep->iface = interface;
  rep->re_freed = 0;
  spin_lock(&routelock);
  cds_list_add_rcu(&rep->re_next, &route_list);
  spin_unlock(&routelock);
  return 0;
}
```

KESL Kookmin Univ. Embedded System Lab

RCU (Linked List)

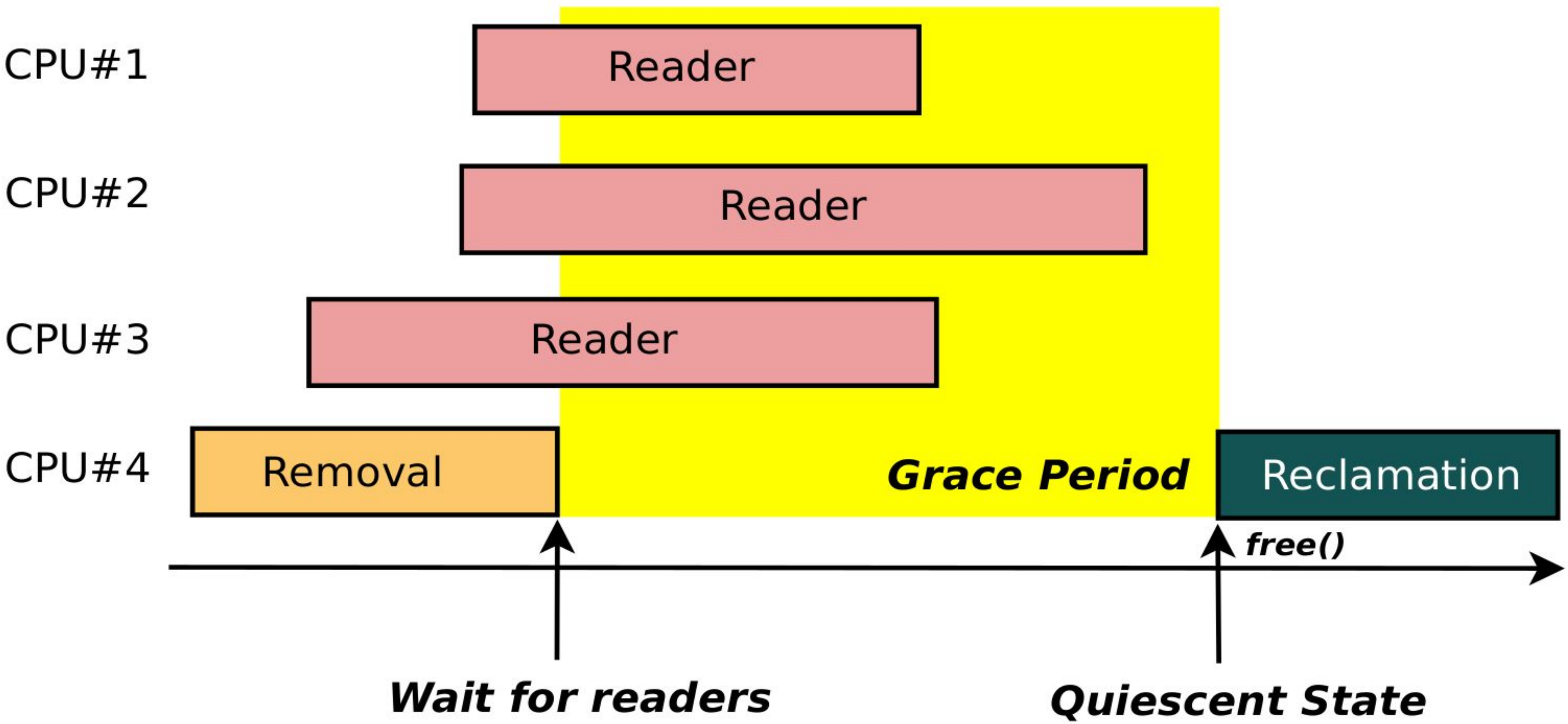



1
A
B
C
D
E
Reader
Reader
Reader
Allocate
& Copy
& Update
Writer
N
2
A
B
C
D
E
Reader
Reader
Reader
rcu_assign_pointer
Writer
N
3
A
B
C
D
E
Reader
Reader
Reader
synchronize_rcu
Writer
N
4
A
B
C
D
E
free
Reader
Reader

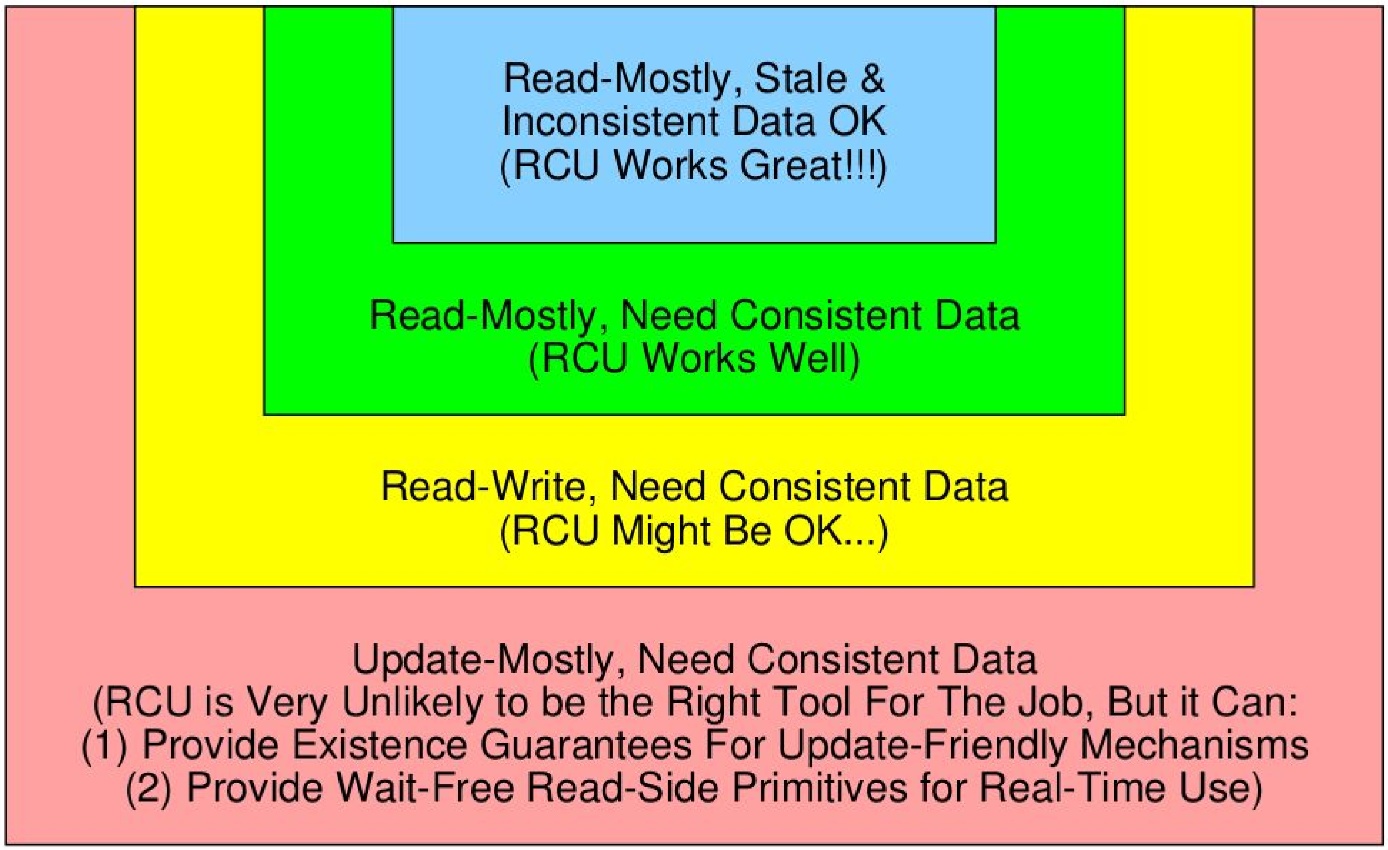
RCU Usage - remove

```
static void route_cb(struct rcu_head *rhp)
{
  struct route_entry *rep;

  rep = container_of(rhp, struct route_entry, rh);
  ACCESS_ONCE(rep->re_freed) = 1;
  free(rep);
}

int route_del(unsigned long addr)
{
  struct route_entry *rep;

  spin_lock(&routelock);
  cds_list_for_each_entry(rep, &route_list,
                          re_next) {
    if (rep->addr == addr) {
      cds_list_del_rcu(&rep->re_next);//rcu_assign_pointer
      spin_unlock(&routelock);
      call_rcu(&rep->rh, route_cb);//synchroinized_rcu
      return 0;
    }
  }
  spin_unlock(&routelock);
  return -ENOENT;
}
```

Grace Period



RCU Areas of Applicability



Non-blocking algorithms

Deadlock

core A
foo(d1/a, d2/b)
lock d1
lock d2 ...

core B
foo(d2/c, d1/d)
lock d2
lock d1 ...

Deadlock

core A
foo(d1/a, d2/b)
lock d1
lock d2 ...

core B
foo(d2/c, d1/d)
lock d2
lock d1 ...

Solution :

Works out an order for all locks
all code must acquire locks in that order

Or use a lock-less algorithm.

Deadlock

core A
foo(d1/a, d2/b)
lock d1
lock d2 ...

core B
foo(d2/c, d1/d)
lock d2
lock d1 ...

Solution :

Works out an order for all locks
all code must acquire locks in that order
predict locks, sort, acquire

Deadlock

core A
foo(d1/a, d2/b)
lock d1
lock d2 ...

core B
foo(d2/c, d1/d)
lock d2
lock d1 ...

Solution :
use a lock-less algorithm.

Lock-less algorithms(Non-blocking algorithms)

- **One example : A Lock-less algorithm**

```
struct element {
    int key;
    int value;
    struct element *next;
};

struct element *global;

void push(struct element *e)
{
retry:
    e->next = global;     //A
    if (cmpxchg(&global, e->next, e) != e->next)  //B
        goto retry;
}
```

```
struct element *pop(void)
{
retry:
    struct element *e = global; //A

    if (cmpxchg(&global, e, e->next) != e) //B
        goto retry;
    return e;
}
```

Linux's Lock-less List

- **Lock-less NULL terminated single linked list**
- **llist_add_batch**
- **llist_del_all**

Linux lock-less data structure

lib/llist.c

```c
static inline struct llist_node *llist_del_all(struct llist_head *head)
{
	return xchg(&head->first, NULL);
}
```

```c
/**
 * llist_add_batch - add several linked entries in batch
 * @new_first:	first entry in batch to be added
 * @new_last:	last entry in batch to be added
 * @head:	the head for your lock-less list
 *
 * Return whether list is empty before adding.
 */
bool llist_add_batch(struct llist_node *new_first, struct llist_node *new_last,
		     struct llist_head *head)
{
	struct llist_node *first;

	do {
		new_last->next = first = ACCESS_ONCE(head->first);
	} while (cmpxchg(&head->first, first, new_first) != first);

	return !first;
}
EXPORT_SYMBOL_GPL(llist_add_batch);
```

KESL Kookmin Univ. Embedded System Lab

Ex) tty buffer

```c
void tty_buffer_init(struct tty_port *port)
{
	struct tty_bufhead *buf = &port->buf;

	mutex_init(&buf->lock);
	tty_buffer_reset(&buf->sentinel, 0);
	buf->head = &buf->sentinel;
	buf->tail = &buf->sentinel;
	init_llist_head(&buf->free);
	atomic_set(&buf->mem_used, 0);
	atomic_set(&buf->priority, 0);
	INIT_WORK(&buf->work, flush_to_ldisc);
	buf->mem_limit = TTYB_DEFAULT_MEM_LIMIT;
}
```

Ex) tty buffer

```
void tty_buffer_free_all(struct tty_port *port)
{
	struct tty_bufhead *buf = &port->buf;
	struct tty_buffer *p, *next;
	struct llist_node *llist;

	while ((p = buf->head) != NULL) {
		buf->head = p->next;
		if (p->size > 0)
			kfree(p);
	}
	llist = llist_del_all(&buf->free);
	llist_for_each_entry_safe(p, next, llist, free)
		kfree(p);

	tty_buffer_reset(&buf->sentinel, 0);
	buf->head = &buf->sentinel;
	buf->tail = &buf->sentinel;

	atomic_set(&buf->mem_used, 0);
}
```

Today

process/thread -> performance -> multi-core performance scalability-> lock-> bottleneck-> cache cohere system problem-> per-core partition approach

Next Step.

1. **Linux power**
2. **CFS scheduler(User level, Tools, Kernel level)**
3. **Load Balancer(Group Scheduling, Bandwidth Control, PELT)**
4. **EAS features**

Reference

- https://pdos.csail.mit.edu/6.828/2016/schedule.html
- http://web.mit.edu/6.033
- http://www.rdrop.com/~paulmck/
- "**Is Parallel Programming Hard, And If So, What Can You Do About It?"**
- Davidlohr Bueso. 2014. Scalability techniques for practical synchronization primitives. *Commun. ACM* 58

http://queue.acm.org/detail.cfm?id=2698990

- **"CPUFreq and The Scheduler Revolution in CPU Power Management", Rafael J. Wysocki**
- **https://sites.google.com/site/embedwiki/oses/linux/pm/pm-qos**
- **https://intl.aliyun.com/forum/read-916**
- **User-level threads : co-routines**

  http://www.gamedevforever.com/291

  https://www.youtube.com/watch?v=YYtzQ355_Co
- **Scheduler Activations**
  - https://cgi.cse.unsw.edu.au/~cs3231/12s1/lectures/SchedulerActivations.pdf
- **https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)**
- **http://jake.dothome.co.kr/**
- **http://www.linuxjournal.com/magazine/completely-fair-scheduler?page=0,0**
- **https://www2.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/6_CPU_Scheduling.html**
- **"Energy Aware Scheduling", Byungchul Park, LG Electronic**
- **"Update on big.LITTLE scheduling experiments", ARM**
- **"EAS Update" 2015 september ARM**
- **"EAS Overview and Integration Guide", ARM TR**
- **"Drowsy Power Management", Matthew Lentz, SOSP 2015**
- **https://www.slideshare.net/nanik/learning-aosp-android-hardware-abstraction-layer-hal**
- https://www.youtube.com/watch?v=oTGQXqD3CNI
- https://www.youtube.com/watch?v=P80NcKUKpuo
- https://lwn.net/Articles/398470/
- "SCHED_DEADLINE: It's Alive!", ARM, 2017