OSTEP Concurrency: Locks

Questions answered in this lecture:

Review: Why threads and mutual exclusion for critical sections?

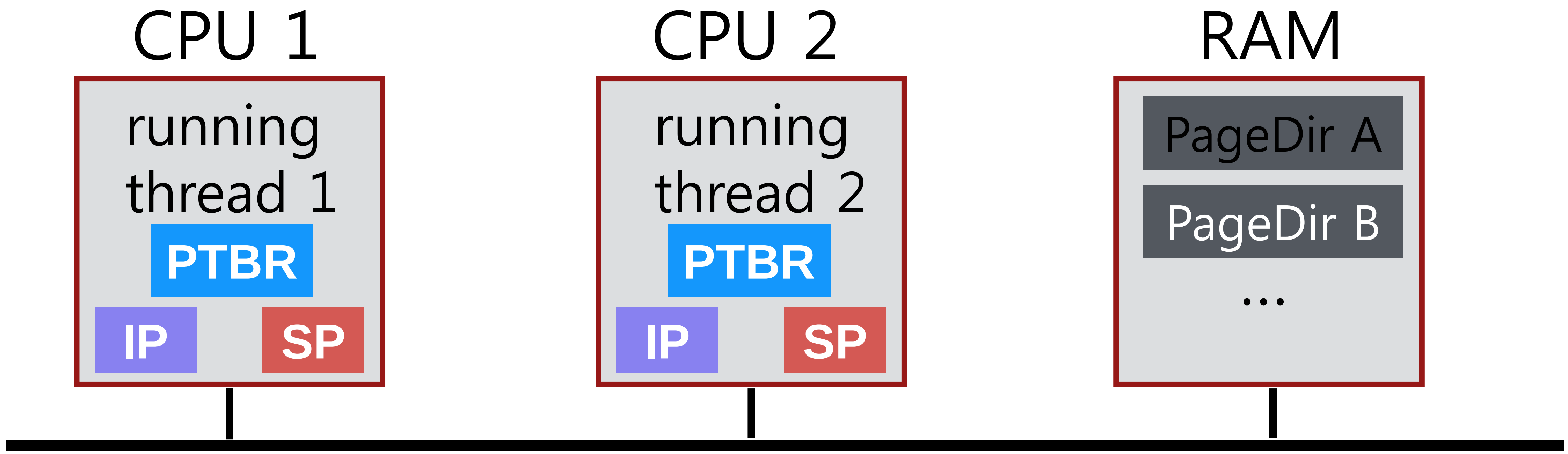
How can locks be used to protect shared data structures such as **linked lists**?

Can locks be implemented by **disabling interrupts**?

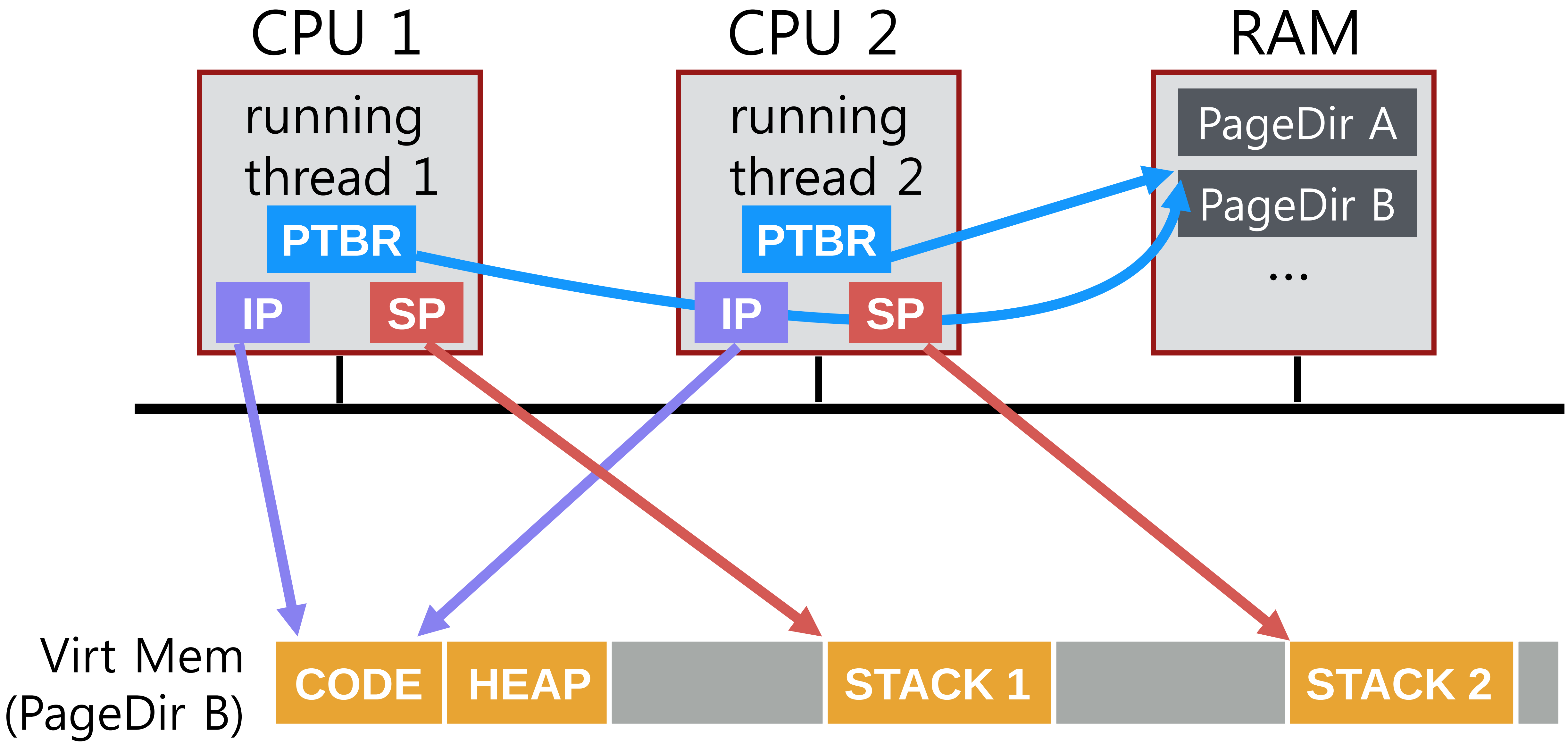
Can locks be implemented with **loads and stores**?

Can locks be implemented with **atomic hardware instructions**?

When are **spinlocks** a good idea?



Review:
Which registers store the same/different values across threads?


CPU 1
running thread 1
PTBR
IP
SP
CPU 2
running thread 2
PTBR
IP
SP
RAM
PageDir A
PageDir B
...
Virt Mem
(PageDir B)
CODE
HEAP
STACK 1
STACK 2

Locks: The Basic Idea

Ensure that any **critical section** executes as if it were a single atomic instruction.

- An example: the canonical update of a shared variable

```
balance = balance + 1;
```

- Add some code around the critical section

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks: The Basic Idea

Lock variable holds the state of the lock.

- **available** (or **unlocked** or **free**)
  - No thread holds the lock.
- **acquired** (or **locked** or **held**)
  - Exactly one thread holds the lock and presumably is in a critical section.

The semantics of the lock()

`lock()`

- **Try to** acquire the lock.
- If <u>no other thread holds</u> the lock, the thread will **acquire** the lock.
- **Enter** the *critical section*.
  - This thread is said to be <u>the owner of</u> the lock.

- Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

Pthread Locks - mutex

The name that the POSIX library uses for a lock.

- Used to provide mutual exclusion between threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

- We may be using *different locks* to protect *different variables* → Increase concurrency (a more **fine-grained** approach).

Building A Lock

Efficient locks provided mutual exclusion at low cost.

Building a lock need some help from the **hardware** and the **OS**.

Evaluating locks – Basic criteria

Mutual exclusion

- Does the lock work, preventing multiple threads from entering *a critical section*?

Fairness

- Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation)

Performance

- The time overheads added by using the lock

Controlling Interrupts

Disable Interrupts for critical sections

- One of the earliest solutions used to provide mutual exclusion
- Invented for <u>single-processor</u> systems.

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

- Problem:
  - Require too much *trust* in applications
    - Greedy (or malicious) program could monopolize the processor.
  - Do not work on multiprocessors
  - Code that masks or unmasks interrupts be executed *slowly* by modern CPUs

Why hardware support needed?

First attempt: Using a *flag* denoting whether the lock is held or not.

- The code below has problems.

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 → lock is available, 1 → held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)   // TEST the flag
          ;  // spin-wait (do nothing)
    mutex->flag = 1;  // now SET it !
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Why hardware support needed? (Cont.)

- **Problem 1**: No Mutual Exclusion (assume `flag=0` to begin)

Thread1	Thread2
call `lock()`	
`while (flag == 1)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1
`flag = 1;` // set flag to 1 (too!)	

- **Problem 2**: Spin-waiting wastes time waiting for another thread.

- So, we need an atomic instruction supported by Hardware!
  - *test-and-set* instruction, also known as *atomic exchange*

Test And Set (Atomic Exchange)

An instruction to support the creation of simple locks

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;    // fetch old value at ptr
    *ptr = new; // store 'new' into ptr
    return old; // return the old value
}
```

- **return**(testing) old value pointed to by the `ptr`.
- *Simultaneously* **update**(setting) said value to `new`.
- This sequence of operations is performed atomically.

Evaluating Spin Locks

Correctness: yes

- The spin lock only allows a single thread to entry the critical section.

Fairness: no

- Spin locks <u>don't provide any fairness</u> guarantees.
- Indeed, a thread spinning may spin *forever*.

Performance:

- In the single CPU, performance overheads can be quite *painful*.
- If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*.

Compare-And-Swap

Test whether the value at the address(`ptr`) is equal to `expected`.

- *If so*, update the memory location pointed to by `ptr` with the `new` value.
- *In either case*, return the actual value at that memory location.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
            *ptr = new;
    return actual;
}
```

Compare-and-Swap hardware atomic instruction (C-style)

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
            ; // spin
}
```

Spin lock with compare-and-swap

Compare-And-Swap (Cont.)

C-callable x86-version of compare-and-swap

```
char CompareAndSwap(int *ptr, int old, int new) {
    unsigned char ret;

    // Note that sete sets a 'byte' not the word
    __asm__ __volatile__ (
          " lock\n"
          " cmpxchgl %2,%1\n"
          " sete %0\n"
          : "=q" (ret), "=m" (*ptr)
          : "r" (new), "m" (*ptr), "a" (old)
          : "memory");
    return ret;
}
```

Load-Linked and Store-Conditional

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Load-linked And Store-conditional

The store-conditional *only succeeds* if no intermittent store to the address has taken place.

- **success**: return 1 and <u>update</u> the value at `ptr` to `value`.
- **fail**: the value at `ptr` is <u>not updates</u> and 0 is returned.

Load-Linked and Store-Conditional (Cont.)

```
void lock(lock_t *lock) {
    while (1) {
            while (LoadLinked(&lock->flag) == 1)
                    ; // spin until it's zero
            if (StoreConditional(&lock->flag, 1) == 1)
                    return; // if set-it-to-1 was a success: all done
                            otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
     lock->flag = 0;
}
```

Using LL/SC To Build A Lock

```
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
            ; // spin
}
```

A more concise form of the `lock()` using LL/SC

Fetch-And-Add

Atomically increment a value while returning the old value at a particular address.

```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Fetch-And-Add Hardware atomic instruction (C-style)

Ticket Lock (1)

Ticket lock can be built with fetch-and add.

- Ensure progress for all threads. → fairness

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn < myturn)
          ; // spin
}
void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

Ticket Lock (2)

Ticket lock can be built with fetch-and add.

- Ensure progress for all threads. → fairness

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
          ; // spin
}
void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

So Much Spinning

Hardware-based spin locks are simple and they work.

In some cases, these solutions can be quite inefficient.

- Any time a thread gets caught *spinning*, it **wastes an entire time slice** doing nothing but checking a value.

How To Avoid *Spinning*?
We'll need OS Support too!

A Simple Approach: Just Yield

When you are going to spin, give up the CPU to another thread.

- OS system call moves the caller from the *running state* to the *ready state*.
- The cost of a **context switch** can be substantial and the **starvation** problem still exists.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Lock with Test-and-set and Yield

Using Queues: Sleeping Instead of Spinning

- **Queue** to keep track of which threads are waiting to enter the lock.

- `park()`
  - Put a calling thread to sleep

- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`.

Using Queues: Sleeping Instead of Spinning

```
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
…
```

Lock With Queues, Test-and-set, Yield, And Wakeup

Using Queues: Sleeping Instead of Spinning

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

Wakeup/waiting race

- In case of releasing the lock (*thread A*) just before the call to `park()` (*thread B*) → Thread B would sleep forever (potentially).
- **Solaris** solves this problem by adding a third system call: `setpark()`.
  - By calling this routine, a thread can indicate it *is about to* `park`.
  - If it happens to be interrupted and another thread calls `unpark` before `park` is actually called, the subsequent `park` returns immediately instead of sleeping.

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
park();
```

Code modification inside of `lock()`

Futex

Linux provides a futex (is similar to Solaris's `park` and `unpark`).

- `futex_wait(address, expected)`
  - Put the calling thread to sleep
  - If the value at `address` is not equal to `expected`, the call returns immediately.
- `futex_wake(address)`
  - Wake one thread that is waiting on the queue.

Futex (Cont.)

Snippet from `lowlevellock.h` in the **nptl** library

- The high bit of the integer `v`: track whether the lock is held or not
- All the other bits : the number of waiters

```
void mutex_lock(int *mutex) {
    int v;
    /* Bit 31 was clear, we got the mutex (this is the fastpath) */
    if (atomic_bit_test_set(mutex, 31) == 0)
            return;
    atomic_increment(mutex);
    while (1) {
            if (atomic_bit_test_set(mutex, 31) == 0) {
                    atomic_decrement(mutex);
                    return;
                    }
                    /* We have to wait now. First make sure the futex value
                we are monitoring is truly negative (i.e. locked). */
            v = *mutex;
            …
```

Linux-based Futex Locks

Futex (Cont.)

```
        if (v >= 0)
            continue;
        futex_wait(mutex, v);
    }
}

void mutex_unlock(int *mutex) {
    /* Adding 0x80000000 to the counter results in 0 if and only if
       there are not other interested threads */
    if (atomic_add_zero(mutex, 0x80000000))
        return;
    /* There are other threads waiting for this mutex,
       wake one of them up */
    futex_wake(mutex);
}
```

Linux-based Futex Locks (Cont.)

Two-Phase Locks

A two-phase lock realizes that spinning can be useful if the lock *is about to* be released.

- **First phase**
  - The lock spins for a while, *hoping that* it can acquire the lock.
  - If the lock is not acquired during the first spin phase, a second phase is entered,
- **Second phase**
  - The caller is put to sleep.
  - The caller is only woken up when the lock becomes free later.

Lock-based Concurrent Data Structures

Lock-based Concurrent Data structure

Adding locks to a data structure makes the structure **thread safe**.

- How locks are added determine both the correctness and performance of the data structure.

Example: Concurrent Counters without Locks

Simple but not scalable

```
typedef struct __counter_t {
        int value;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
}

void increment(counter_t *c) {
        c->value++;
}

void decrement(counter_t *c) {
        c->value--;
}

int get(counter_t *c) {
        return c->value;
}
```

Example: Concurrent Counters with Locks

Add a **single lock**.

- The lock is acquired when calling a routine that manipulates the data structure.

```
typedef struct __counter_t {
        int value;
        pthread_lock_t lock;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
        Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value++;
        Pthread_mutex_unlock(&c->lock);
}

```

Example: Concurrent Counters with Locks (Cont.)

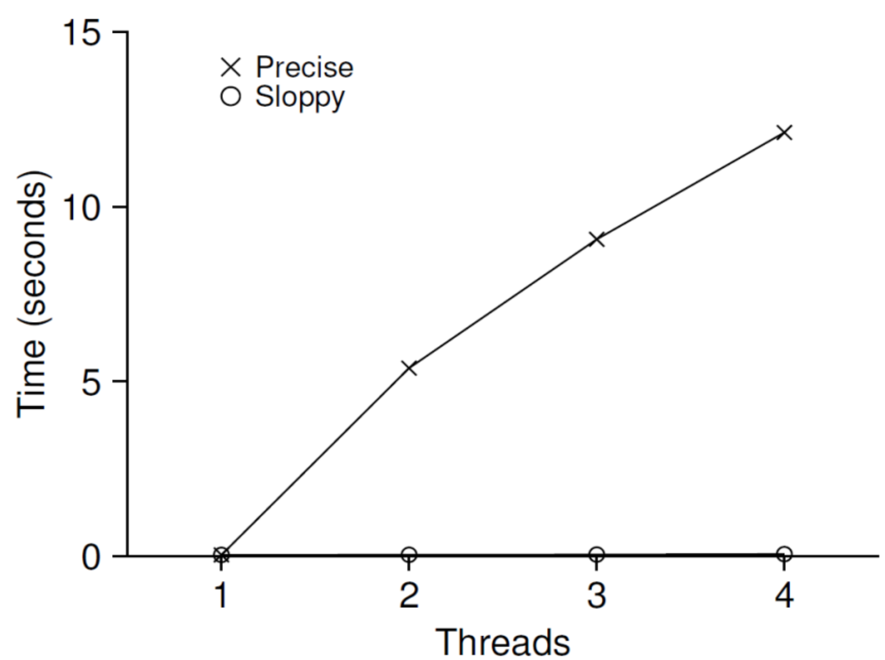
```
(Cont.)
void decrement(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value--;
        Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        int rc = c->value;
        Pthread_mutex_unlock(&c->lock);
        return rc;
}
```

The performance costs of the simple approach

Each thread updates a single shared counter.

- Each thread updates the counter one million times.
- iMac with four Intel 2.7GHz i5 CPUs.



Performance of
Traditional vs. Sloppy Counters
(Threshold of Sloppy, S, is set to 1024)

Synchronized counter scales poorly.

Perfect Scaling

- Even though more work is done, it is **done in parallel**.

- The time taken to complete the task is *not increased*.

Sloppy counter

- The sloppy counter works by representing …
  - A single **logical counter** via numerous local physical counters, on per CPU core
  - A single **global counter**
  - There are **lock**s:
    - One for each local counter and one for the global counter
- Example: on a machine with four CPUs
  - Four local counters
  - One global counter

The basic idea of sloppy counting

- When a thread running on a core wishes to increment the counter.
  - It increment its local counter.

  - Each CPU has its own local counter:
    - Threads across CPUs can update local counters *without contention*.
    - Thus counter updates are scalable.

  - The local values are periodically transferred to the global counter.
    - Acquire the global lock
    - Increment it by the local counter's value
    - The local counter is then reset to zero.

The basic idea of sloppy counting (Cont.)

- How often the local-to-global transfer occurs is determined by a threshold, S (sloppiness).
  - The smaller S:
    - The more the counter behaves like the *non-scalable counter*.
  - The bigger S:
    - The more scalable the counter.
    - The further off the global value might be from the *actual count*.

Sloppy counter example

Tracing the Sloppy Counters

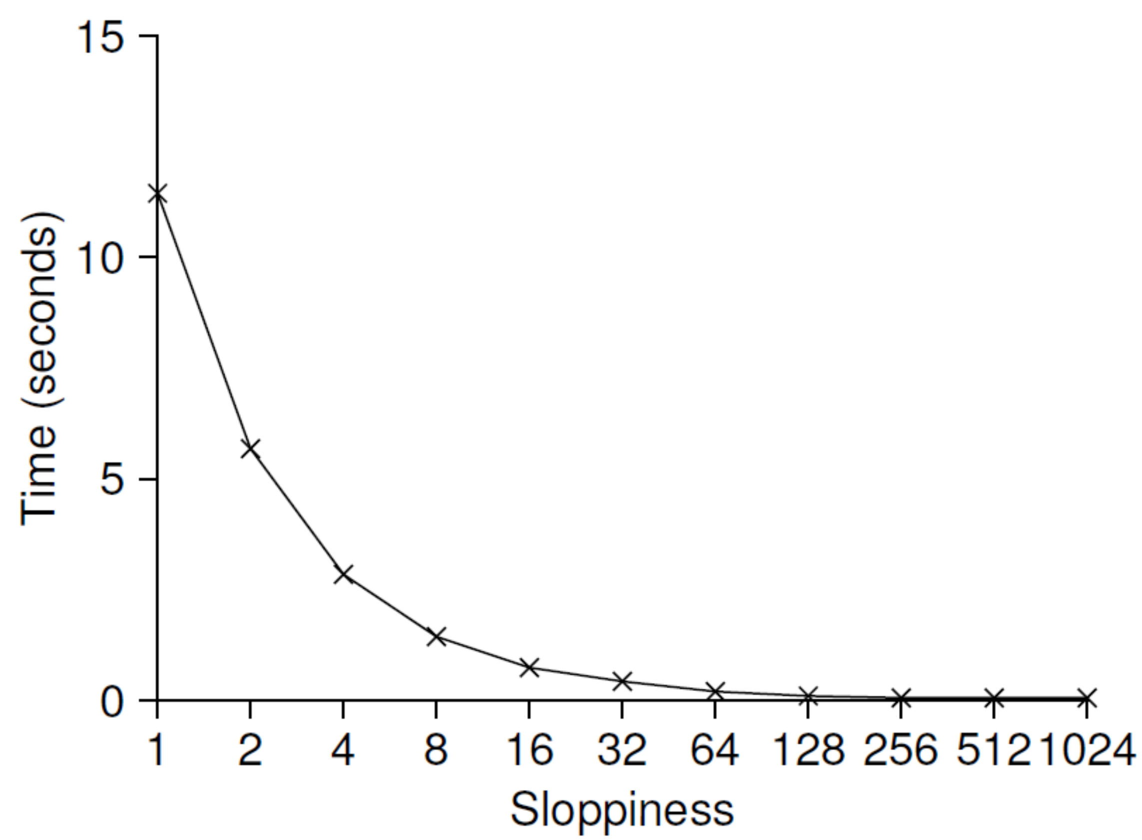
- The threshold S is set to 5.
- There are threads on each of four CPUs
- Each thread updates their local counters $L_1 \ldots L_4$.

Time	$\mathbf{L_1}$	$\mathbf{L_2}$	$\mathbf{L_3}$	$\mathbf{L_4}$	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

Importance of the threshold value S

Each four threads increments a counter 1 million times on four CPUs.

- Low S → Performance is **poor**, The global count is always quire **accurate**.
- High S → Performance is **excellent**, The global count **lags**.



Scaling Sloppy Counters

Sloppy Counter Implementation

```
typedef struct __counter_t {
    int global;                // global count
    pthread_mutex_t glock;         // global lock
    int local[NUMCPUS];    // local count (per cpu)
    pthread_mutex_t llock[NUMCPUS]; // ... and locks
    int threshold;  // update frequency
} counter_t;

// init: record threshold, init locks, init values
//        of all local counts and global count
void init(counter_t *c, int threshold) {
    c->thres hold = threshold;

    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);

    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}

```

Sloppy Counter Implementation (Cont.)

```
(Cont.)
    // update: usually, just grab local lock and update local amount
    //         once local count has risen by 'threshold', grab global
    //         lock and transfer local values to it
    void update(counter_t *c, int threadID, int amt) {
        pthread_mutex_lock(&c->llock[threadID]);
        c->local[threadID] += amt;   // assumes amt > 0
        if (c->local[threadID] >= c->threshold) { // transfer to global
            pthread_mutex_lock(&c->glock);
            c->global += c->local[threadID];
            pthread_mutex_unlock(&c->glock);
            c->local[threadID] = 0;
        }
        pthread_mutex_unlock(&c->llock[threadID]);
    }

    // get: just return global amount (which may not be perfect)
    int get(counter_t *c) {
        pthread_mutex_lock(&c->glock);
        int val = c->global;
        pthread_mutex_unlock(&c->glock);
        return val;     // only approximate!
    }
```

Concurrent Linked Lists

```
// basic node structure
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

(Cont.)
```

Concurrent Linked Lists

```
(Cont.)
    int List_Insert(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
            perror("malloc");
            pthread_mutex_unlock(&L->lock);
        return -1; // fail
        new->key = key;
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
        return 0; // success
    }
(Cont.)
```

Concurrent Linked Lists (Cont.)

```
(Cont.)

    int List_Lookup(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
            if (curr->key == key) {
                pthread_mutex_unlock(&L->lock);
                return 0; // success
            }
            curr = curr->next;
        }
        pthread_mutex_unlock(&L->lock);
        return -1; // failure
    }
```

Concurrent Linked Lists (Cont.)

- The code **acquires** a lock in the insert routine upon entry.
- The code **releases** the lock upon exit.
  - If `malloc()` happens to *fail*, the code must also release the lock before failing the insert.
  - This kind of exceptional control flow has been shown to be quite error prone.
  - **Solution**: The lock and release *only surround* the actual critical section in the insert code

Concurrent Linked List: Rewritten

```
void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

void List_Insert(list_t *L, int key) {
        // synchronization not needed
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                return;
        }
        new->key = key;

        // just lock critical section
        pthread_mutex_lock(&L->lock);
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
}
```

Concurrent Linked List: Rewritten (Cont.)

```
(Cont.)
int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv; // now both success and failure
}
```

Scaling Linked List

Hand-over-hand locking (lock coupling)

- Add **a lock per node** of the list instead of having a single lock for the entire list.
- When traversing the list,
  - First grabs the next node's lock.
  - And then releases the current node's lock.

- Enable a high degree of concurrency in list operations.
  - However, in practice, the overheads of acquiring and releasing locks for each node of a list traversal is *prohibitive*.

Michael and Scott Concurrent Queues

There are two locks.

- One for the **head** of the queue.
- One for the **tail**.
- The goal of these two locks is to enable concurrency of *enqueue* and *dequeue* operations.

Add a dummy node

- Allocated in the queue initialization code
- Enable the separation of head and tail operations

Concurrent Queues (Cont.)

```
typedef struct __node_t {
        int value;
        struct __node_t *next;
} node_t;

typedef struct __queue_t {
        node_t *head;
        node_t *tail;
        pthread_mutex_t headLock;
        pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
        node_t *tmp = malloc(sizeof(node_t));
        tmp->next = NULL;
        q->head = q->tail = tmp;
        pthread_mutex_init(&q->headLock, NULL);
        pthread_mutex_init(&q->tailLock, NULL);
}

(Cont.)
```

Concurrent Queues (Cont.)

```
(Cont.)
    void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        tmp->value = value;
        tmp->next = NULL;

        pthread_mutex_lock(&q->tailLock);
        q->tail->next = tmp;
        q->tail = tmp;
        pthread_mutex_unlock(&q->tailLock);
    }
(Cont.)
```

Concurrent Queues (Cont.)

```
(Cont.)
int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent Hash Table

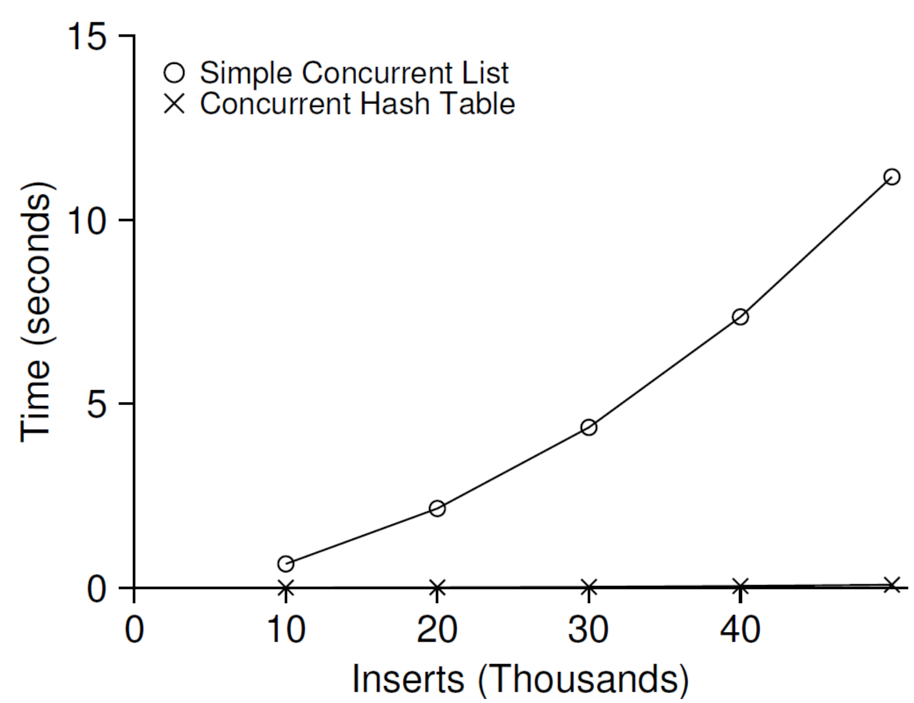
Focus on a simple hash table

- The hash table does not resize.
- Built using the concurrent lists
- It uses a lock per hash bucket each of which is represented by *a list*.

Performance of Concurrent Hash Table

From 10,000 to 50,000 concurrent updates from each of four threads.

- iMac with four Intel 2.7GHz i5 CPUs.



The simple concurrent hash table scales magnificently.

Concurrent Hash Table

```
#define BUCKETS (101)

typedef struct __hash_t {
        list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
        int i;
        for (i = 0; i < BUCKETS; i++) {
                List_Init(&H->lists[i]);
        }
}

int Hash_Insert(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Lookup(&H->lists[bucket], key);
}
```

Context Switch

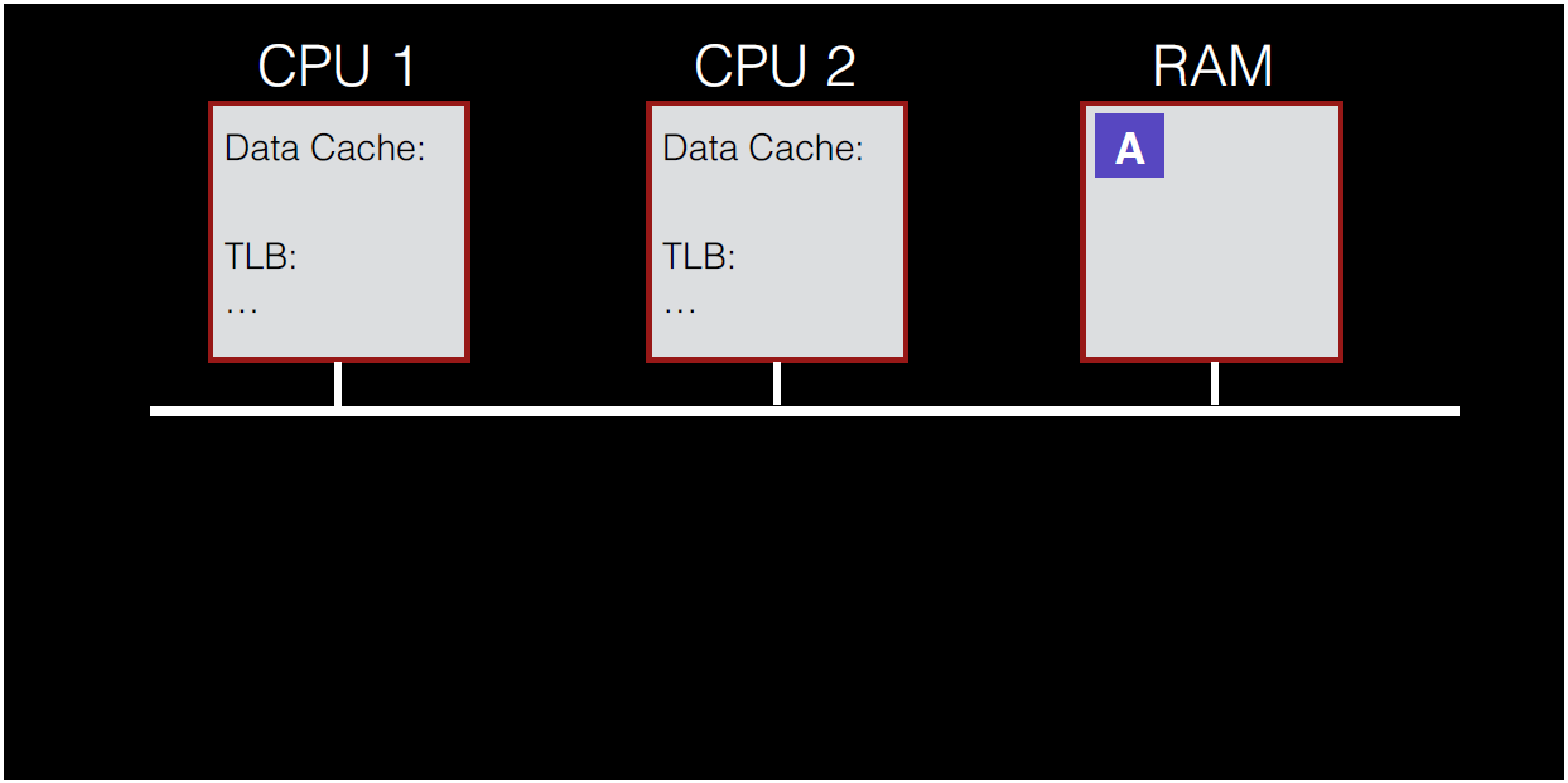
Why is switching between threads cheaper than switching between processes?

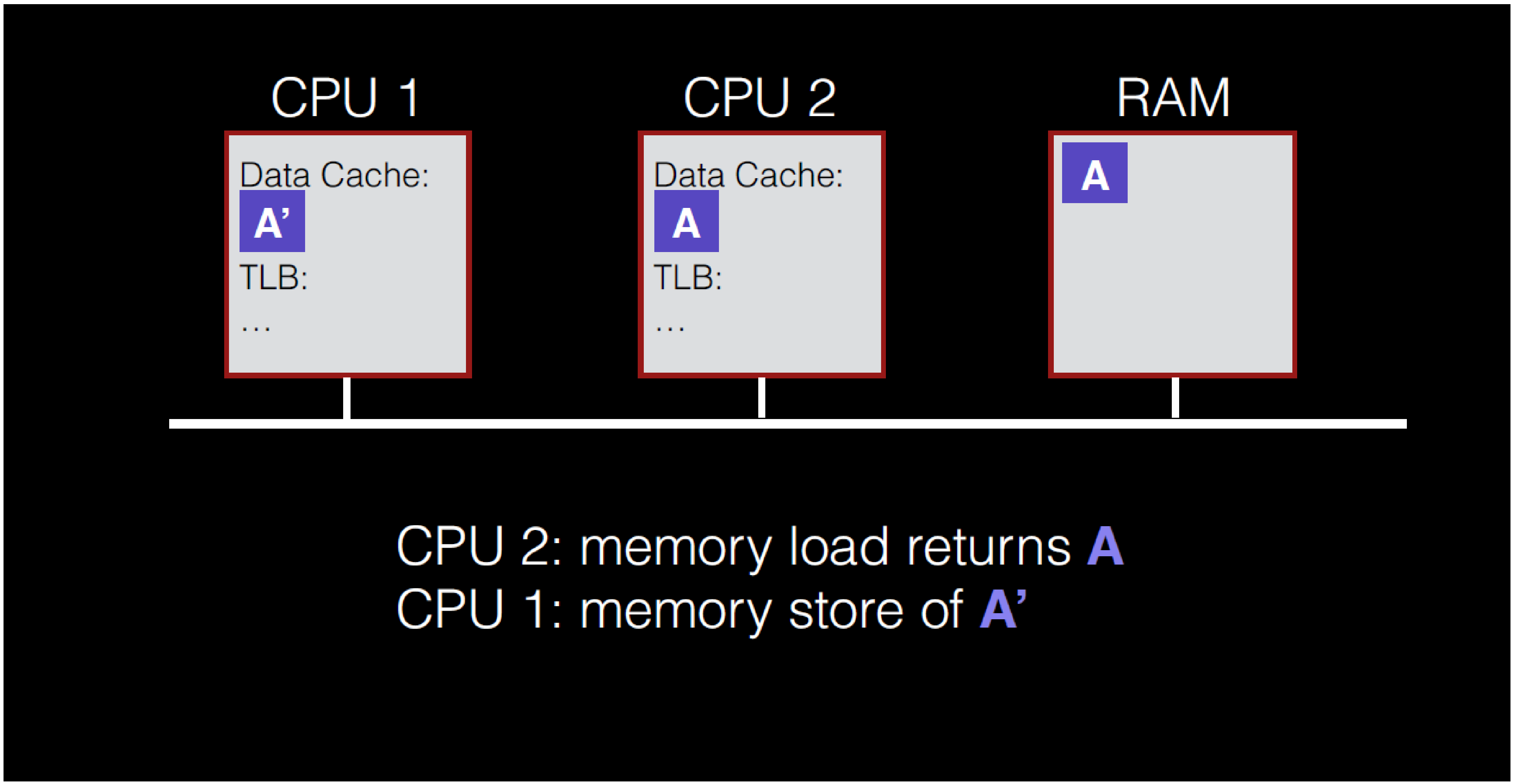
Why is switching between threads not free?

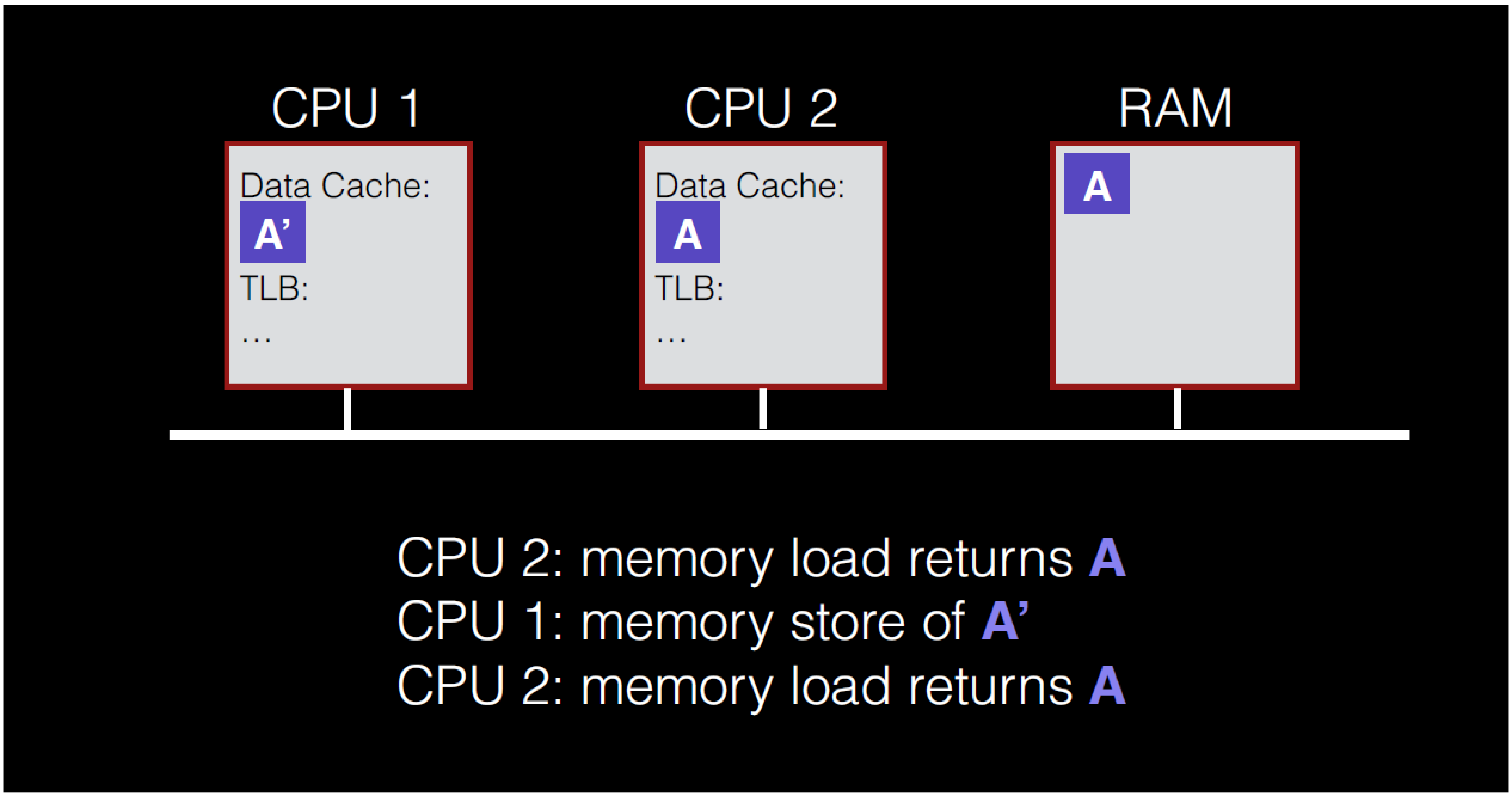
Why is concurrency hard?

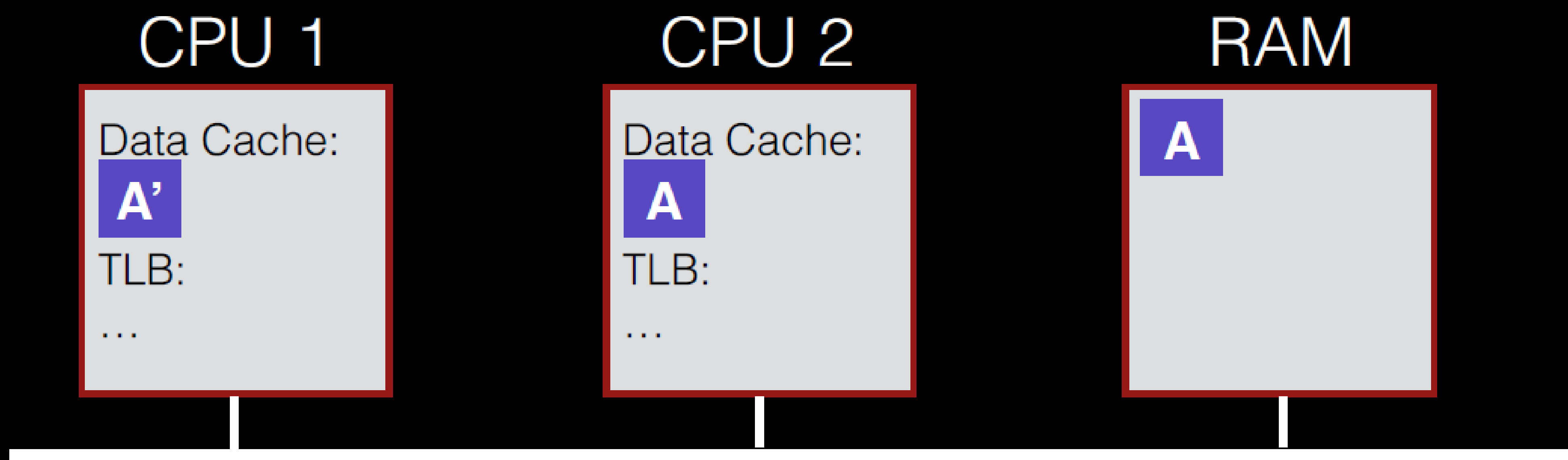
H/W caches

OS scheduler


CPU 1
Data Cache:
TLB:
...
CPU 2
Data Cache:
TLB:
...
RAM
A


CPU 1
Data Cache:
A’
TLB:
…
CPU 2
Data Cache:
A
TLB:
…
RAM
A
CPU 2: memory load returns A
CPU 1: memory store of A’


CPU 1
Data Cache:
A’
TLB:
…
CPU 2
Data Cache:
A
TLB:
…
RAM
A
CPU 2: memory load returns A
CPU 1: memory store of A’
CPU 2: memory load returns A



Updates from one critical section must be visible to others.
CPU needs to know when to flush caches (or similar).

xchg: atomic exchange, or test-and-set

```
//
// xchg(int *addr, int newval)
// return what is pointed to by addr
// at the same time, store newval into addr
//
static inline uint
xchg(volatile unsigned int *addr, unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" :
                 "+m" (*addr), "=a" (result) :
                 "1" (newval) : "cc");
    return result;
}
```

memory barrier

Test-and-set Spinlock

```c
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin
}

void SpinUnlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}
```

Test-and-set Spinlock (optimized)

```
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin


void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Works on newer x86 processors.
Not on all CPUs (sometimes due to CPU bugs!)

Why is concurrency hard?

H/W caches

OS scheduler

What if multiple threads run this?

```
for (i = 0; i < max; i++) {
    balance = balance + 1; // shared: only one
}
```

Balance Adder

Thread 1	**Thread 2**
	mov 0x123, %eax
	add %0x1, %eax
mov 0x123, %eax	
add %0x1, %eax	
mov %eax, 0x123	
	mov %eax, 0x123

How much is added?

Thread 1	Thread 2
	mov 0x123, %eax (eax = 100)
	add %0x1, %eax (eax = 101)
mov 0x123, %eax (eax = 100)	
add %0x1, %eax (eax = 101)	
mov %eax, 0x123 (0x123 = 101)	
	mov %eax, 0x123 (0x123 = 101)

How much is added?

Thread 1	**Thread 2**
	mov 0x123, %eax
	add %0x1, %eax
	mov %eax, 0x123
mov 0x123, %eax	
add %0x1, %eax	
mov %eax, 0x123	

How much is added?

Thread 1	**Thread 2**
	mov 0x123, %eax (eax = 100)
	add %0x1, %eax (eax = 101)
	mov %eax, 0x123 (0x123 = 101)
mov 0x123, %eax (eax = 101)	
add %0x1, %eax (eax = 102)	
mov %eax, 0x123 (0x123 = 102)	

How much is added?

Thread 1 | **Thread 2**

Thread 2:

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

Thread 1:

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

Need atomic sections that don’t run simultaneously, even on different CPUs!

Review: What is needed for Correctness?

Balance = balance + 1;

Instructions accessing shared memory must execute as uninterruptable group

- Need instructions to be atomic

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

critical section

More general:
Need **mutual exclusion** for critical sections

- if process A is in critical section C, process B can't
  (okay if other processes do unrelated work)

Other Examples

Consider multi-threaded applications that do more than increm
ent shared balance

Multi-threaded application with shared linked-list

- All concurrent:
  - Thread A inserting element a
  - Thread B inserting element b
  - Thread C looking up element c

Shared Linked List

```
Void List_Insert(list_t *L,
            int key) {
        node_t *new =
                malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}

int List_Lookup(list_t *L,
                        int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                        return 1;
                tmp = tmp->next;
        }
return 0;
}
```

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

What can go wrong?
Find schedule that leads to problem?

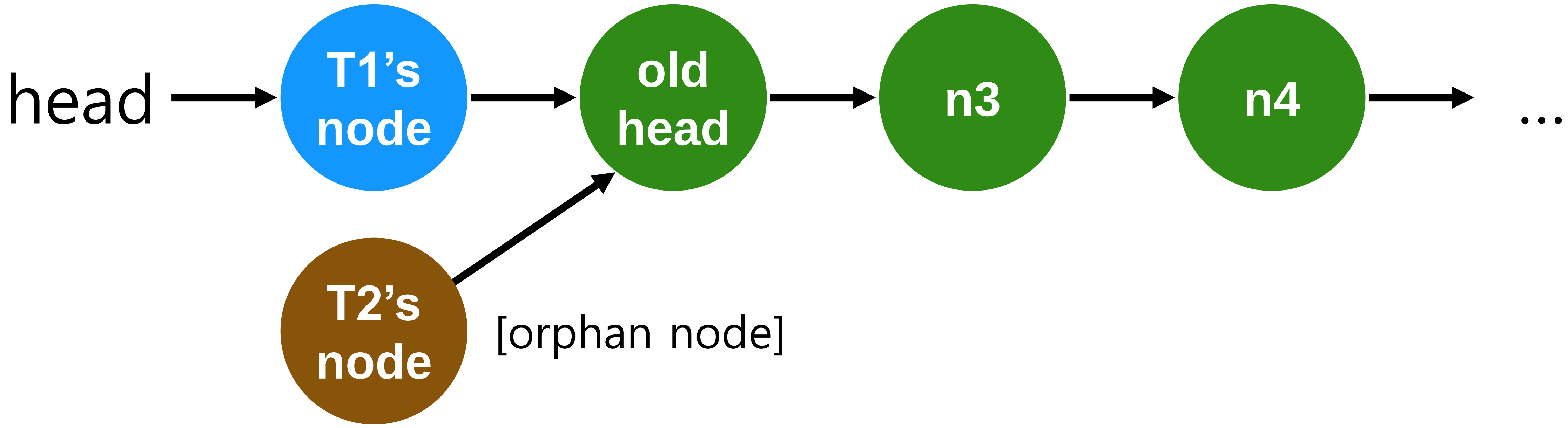
Linked-List Race

Thread 1	Thread 2
new->key = key	
new->next = L->head	
	new->key = key
	new->next = L->head
	L->head = new
L->head = new	

Both entries point to old head

Only one entry (which one?) can be the new head.

Resulting Linked List



Locking Linked Lists

```
Void List_Insert(list_t *L,
        int key) {
    node_t *new =
        malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}
int List_Lookup(list_t *L,
                int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
        return 1;
        tmp = tmp->next;
    }
return 0;
}
```

```
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

Typedef struct __list_t {
    node_t *head;
} list_t;

Void List_Init(list_t *L) {
    L->head = NULL;
}
```

How to add locks?

Locking Linked Lists

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

How to add locks?

pthread_mutex_t lock;

One lock per list

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock,
                NULL);
}
```

Locking Linked Lists : Approach #1

Consider everything critical section
Can critical section be smaller?

```
Pthread_mutex_lock(&L->lock);     →  Void List_Insert(list_t *L, int key) {
                                          node_t *new =
                                                  malloc(sizeof(node_t));
                                          assert(new);
                                          new->key = key;
                                          new->next = L->head;
Pthread_mutex_unlock(&L->lock);   →      L->head = new;
                                      }
Pthread_mutex_lock(&L->lock);     →  int List_Lookup(list_t *L, int key) {
                                          node_t *tmp = L->head;
                                          while (tmp) {
                                                  if (tmp->key == key)
                                                          return 1;
                                                  tmp = tmp->next;
Pthread_mutex_unlock(&L->lock);   →      }
                                          return 0;
                                      }
```

Locking Linked Lists : Approach #2

Critical section small as possible

Pthread_mutex_lock(&L->lock); → (before `new->key = key;`)

Pthread_mutex_unlock(&L->lock); → (after `L->head = new;`)

Pthread_mutex_lock(&L->lock); → (before `node_t *tmp = L->head;`)

Pthread_mutex_unlock(&L->lock); → (before `return 0;`)

```
Void List_Insert(list_t *L, int key) {
        node_t *new =
                malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}
int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                        return 1;
                tmp = tmp->next;
        }
        return 0;
}
```

Locking Linked Lists : Approach #3

What about Lookup()?

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

If no List_Delete(), locks not needed

Pthread_mutex_unlock(&L->lock);

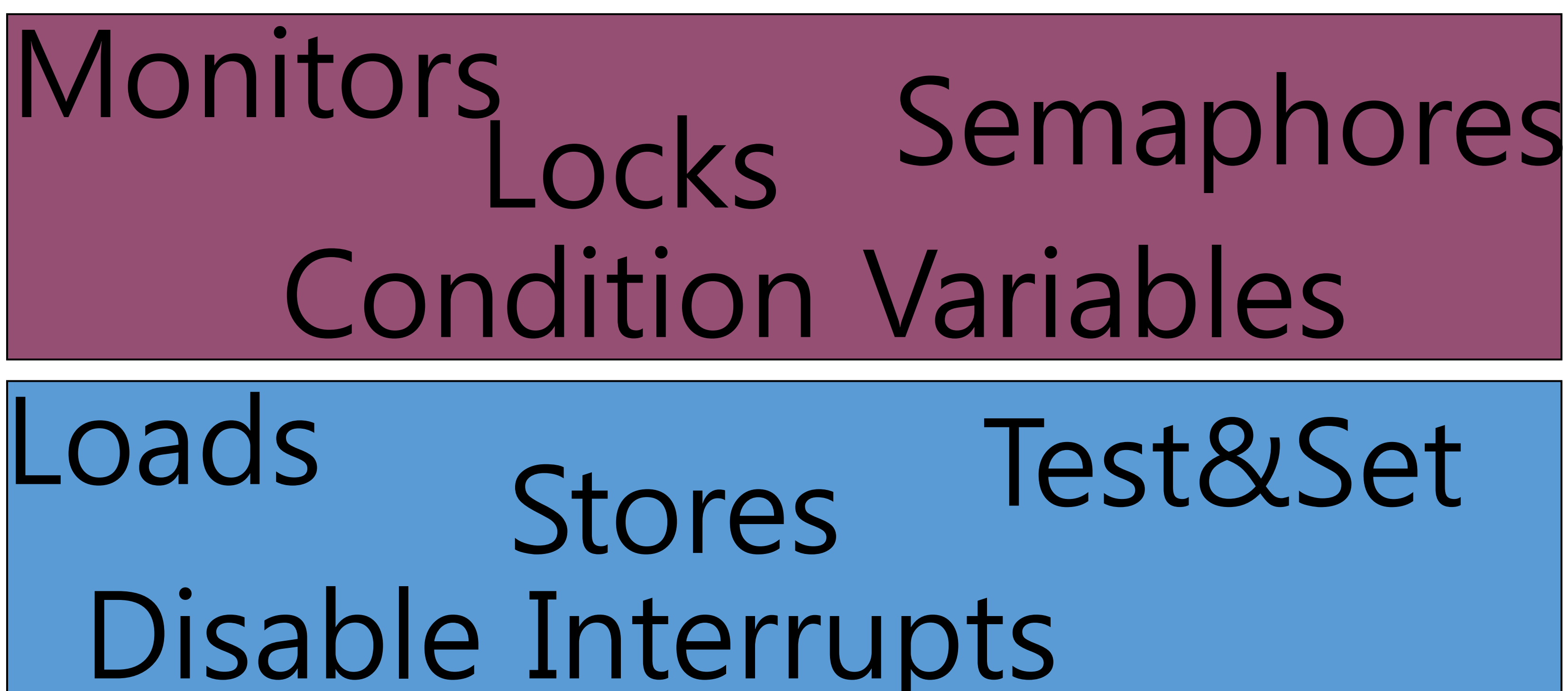
```
Void List_Insert(list_t *L, int key) {
        node_t *new =
                malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}
int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                        return 1;
                tmp = tmp->next;
        }
        return 0;
}
```

Implementing Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right



Lock Implementation Goals

Correctness

- Mutual exclusion
  - Only one thread in critical section at a time
- Progress (deadlock-free)
  - If several simultaneous requests, must allow one to proceed
- Bounded (starvation-free)
  - Must eventually allow each waiting thread to enter

Fairness

Each thread waits for same amount of time

Performance

CPU is not used unnecessarily (e.g., spinning)

Implementing Synchronization

To implement, need atomic operations

Atomic operation: No other instructions can be interle aved

Examples of atomic operations

- Code between interrupts on uniprocessors
  - Disable timer interrupts, don't do any I/O
- Loads and stores of words
  - Load r1, B
  - Store r1, A
- **Special hw instructions**
  - **Test&Set**
  - **Compare&Swap**

Implementing Locks: W/ Interrupts

Turn off interrupts for critical sections

- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

```
Void acquire(lockT *l) {
        disableInterrupts();
}
Void release(lockT *l) {
        enableInterrupts();
}
```

Disadvantages??

- Only works on uniprocessors
- Process can keep control of CPU for arbitrary length
- Cannot perform other necessary work

Implementing LOCKS: w/ Load+Store

Code uses a single **shared** lock variable

```
Boolean lock = false; // shared variable
Void acquire(Boolean *lock) {
        while (*lock) /* wait */ ;
        *lock = true;
}
Void release(Boolean *lock) {
        *lock = false;
}
```

Why doesn’t this work?
Example schedule that fails with 2 threads?

Race Condition with LOAD and STORE

*lock == 0 initially

Thread 1	Thread 2
while(*lock == 1)	
	while(*lock == 1)
	*lock = 1
*lock = 1	

Both threads grab lock!
Problem: Testing lock and setting lock are not atomic

Demo

Critical section not protected with faulty lock implementation

Peterson's Algorithm

Assume only two threads (tid = 0, 1) and use just loads an
d stores

```
int turn = 0; // shared
Boolean lock[2] = {false, false};
Void acquire() {
        lock[tid] = true;
        turn = 1-tid;
        while (lock[1-tid] && turn == 1-tid) /* wait */ ;
}
Void release() {
        lock[tid] = false;
}
```

Different Cases: All Work

Only thread 0 wants lock

```
Lock[0] = true;
turn = 1;
while (lock[1] && turn ==1);
```

Thread 0 and thread 1 both want lock;

```
Lock[0] = true;

turn = 1;
                                        Lock[1] = true;

                                        turn = 0;

while (lock[1] && turn ==1);

                                        while (lock[0] && turn == 0);
```

Different Cases: All Work

Thread 0 and thread 1 both want lock

Lock[0] = true;

Lock[1] = true;

turn = 0;

turn = 1;

while (lock[1] && turn ==1);

while (lock[0] && turn == 0);

Different Cases: All Work

Thread 0 and thread 1 both want lock;

Lock[0] = true;

turn = 1;

Lock[1] = true;

while (lock[1] && turn ==1);

turn = 0;

while (lock[0] && turn == 0);

while (lock[1] && turn ==1);

Peterson’s Algorithm: Intuition

Mutual exclusion: Enter critical section if and only if
- Other thread does not want to enter
- Other thread wants to enter, but your turn

Progress: Both threads cannot wait forever at while() loop
- Completes if other process does not want to enter
- Other process (matching turn) will eventually finish

Bounded waiting (not shown in examples)
- Each process waits at most one critical section

Problem: doesn’t work on modern hardware
(cache-consistency issues)

xchg: atomic exchange, or test-and-set

```
// xchg(int *addr, int newval)
// return what was pointed to by addr
// at the same time, store newval into addr

int xchg(int *addr, int newval) {
  int old = *addr;
  *addr = newval;
  return old;
}

static inline uint
xchg(volatile unsigned int *addr, unsigned int newval) {
   uint result;
   asm volatile("lock; xchgl %0, %1" :
                "+m" (*addr), "=a" (result) :
                "1" (newval) : "cc");
   return result;
}
```

LOCK Implementation with XCHG

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t *lock) {
        lock->flag = ??;
}

void acquire(lock_t *lock) {
        ????;
        // spin-wait (do nothing)
}

void release(lock_t *lock) {
        lock->flag = ??;
}
```

```
int xchg(int *addr, int newval)
```

XCHG Implementation

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t *lock) {
        lock->flag = 0;
}

void acquire(lock_t *lock) {
        while(xchg(&lock->flag, 1) == 1) ;
        // spin-wait (do nothing)
}

void release(lock_t *lock) {
        lock->flag = 0;
}
```

DEMO XCHG

Critical section protected with our lock implementation!!

Other Atomic HW Instructions

```
int CompareAndSwap(int *addr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}

void acquire(lock_t *lock) {
      while(CompareAndSwap(&lock->flag, ?, ?)
                           == ?) ;
      // spin-wait (do nothing)
}
```

Other Atomic HW Instructions

```
int CompareAndSwap(int *addr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}

void acquire(lock_t *lock) {
      while(CompareAndSwap(&lock->flag, 0, 1)
                        == 1) ;
      // spin-wait (do nothing)
}
```

Lock Implementation Goals

Correctness

- Mutual exclusion
  - Only one thread in critical section at a time
- Progress (deadlock-free)
  - If several simultaneous requests, must allow one to proceed
- Bounded (starvation-free)
  - Must eventually allow each waiting thread to enter

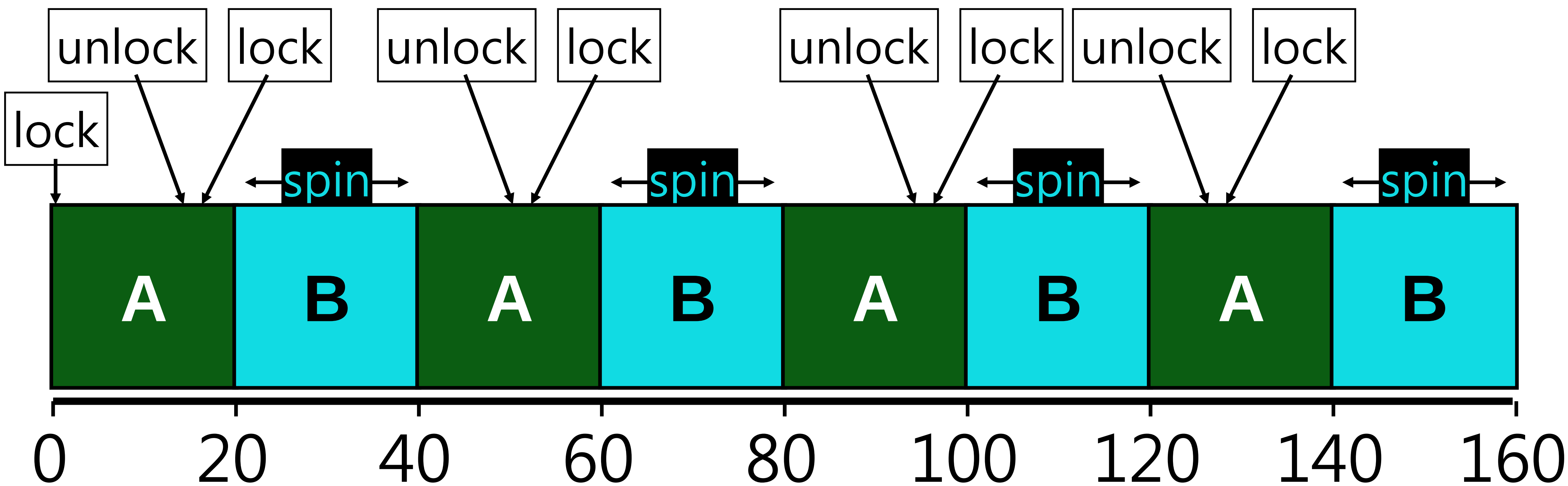
Fairness

Each thread waits for same amount of time

Performance

CPU is not used unnecessarily

Basic Spinlocks are Unfair



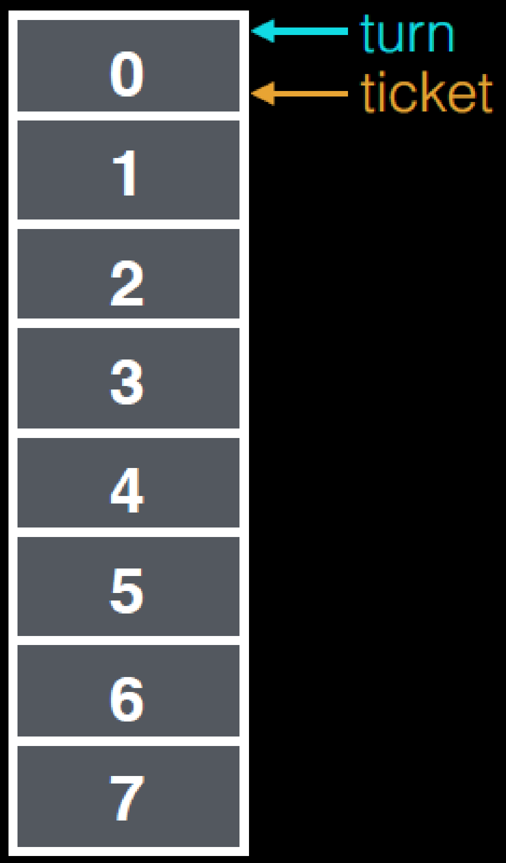
Scheduler is independent of locks/unlocks

Fairness: Ticket Locks

Idea: reserve each thread's turn to use a lock.
Each thread spins until their turn.
Use new atomic primitive, fetch-and-add:

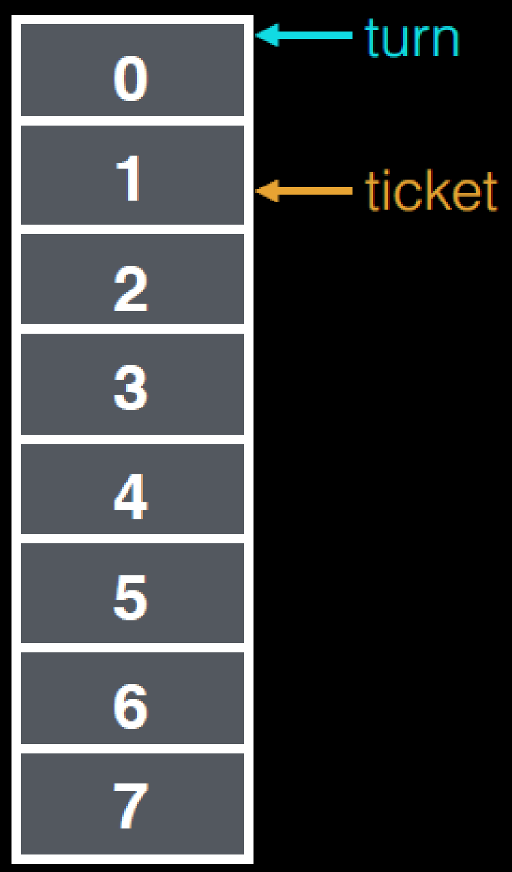
```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Acquire: Grab ticket;
Spin while not thread's ticket != turn
Release: Advance to next turn

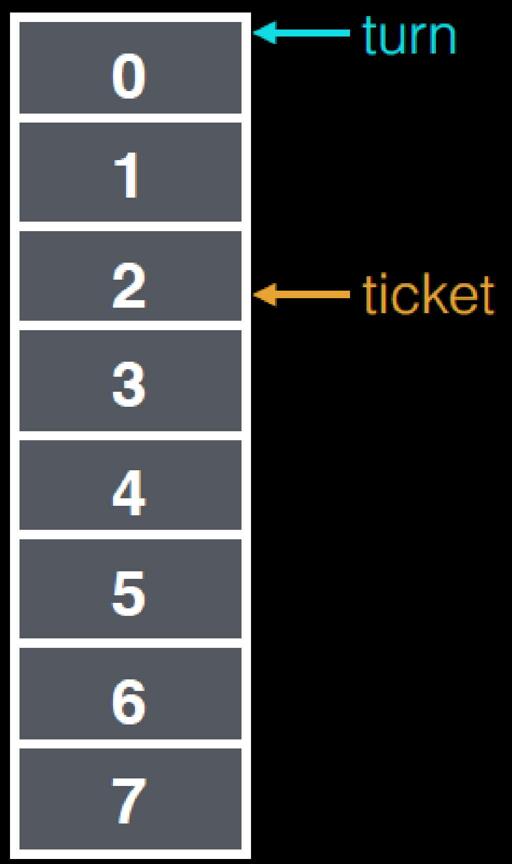


0
1
2
3
4
5
6
7
turn
ticket

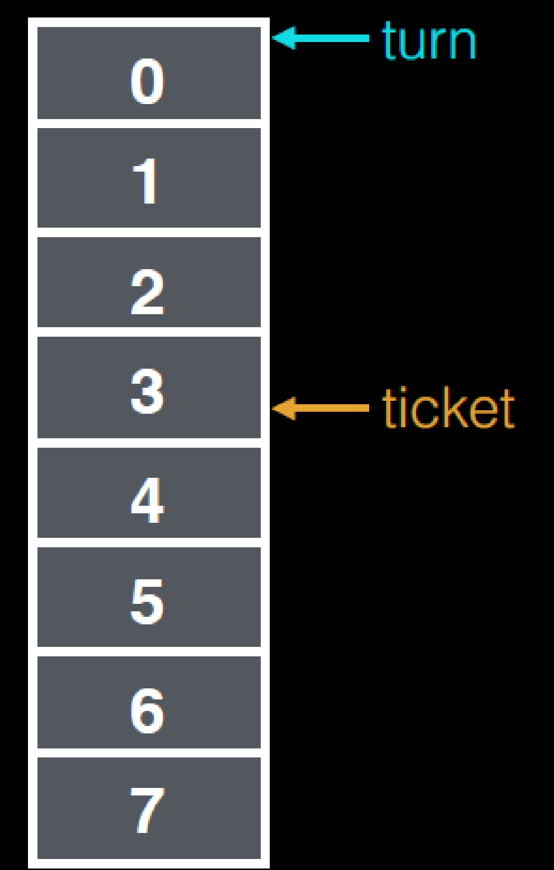
A lock(): gets ticket 0, runs



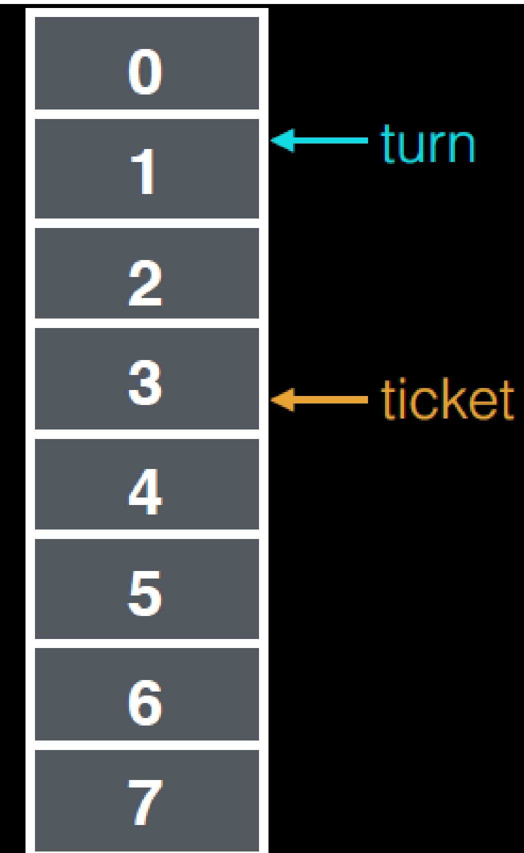
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1



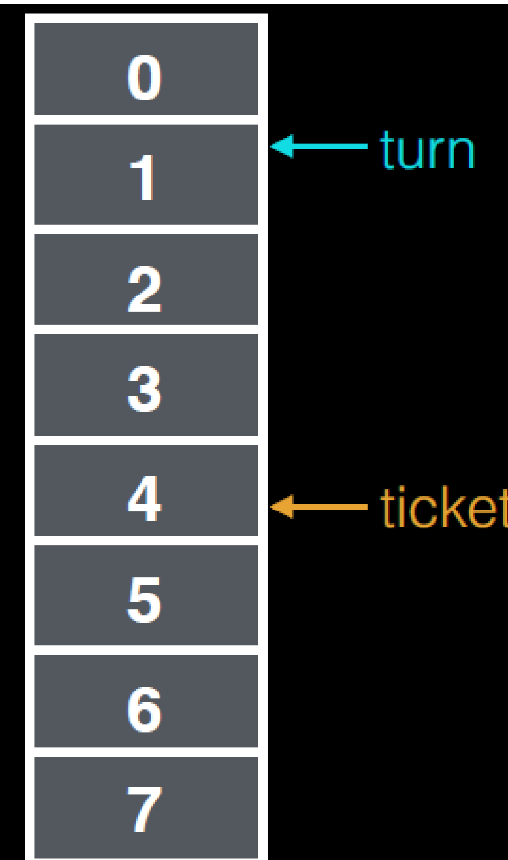
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2



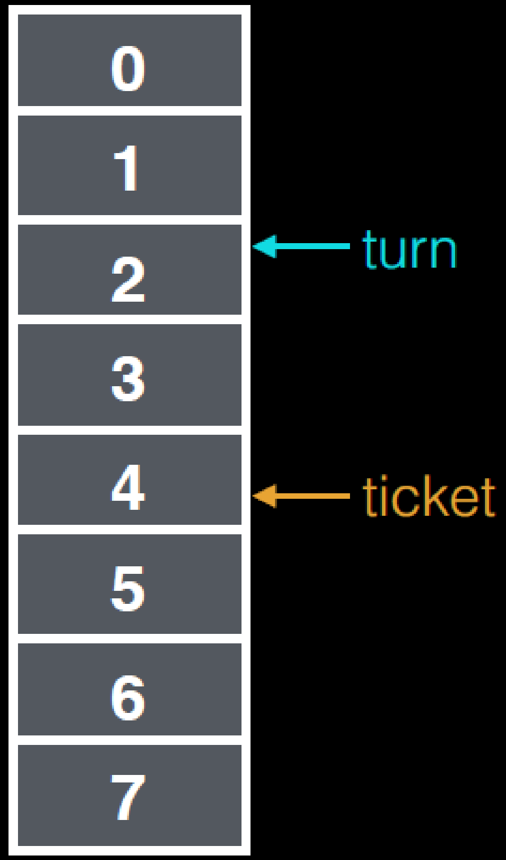
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs



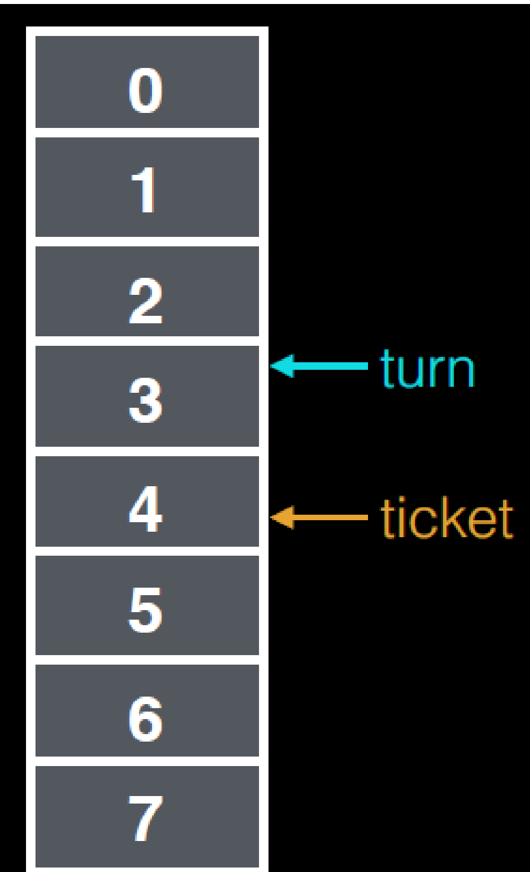
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs
A lock(): gets ticket 3, spins until turn=3



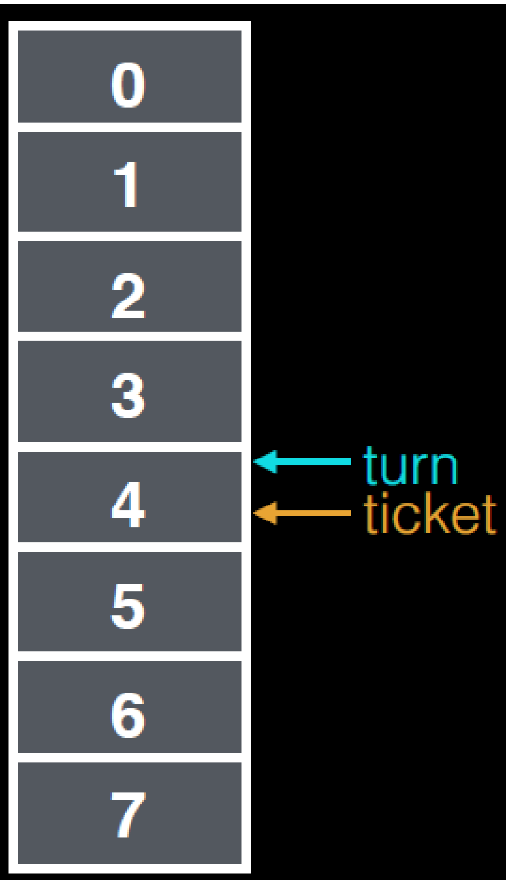
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++
C runs



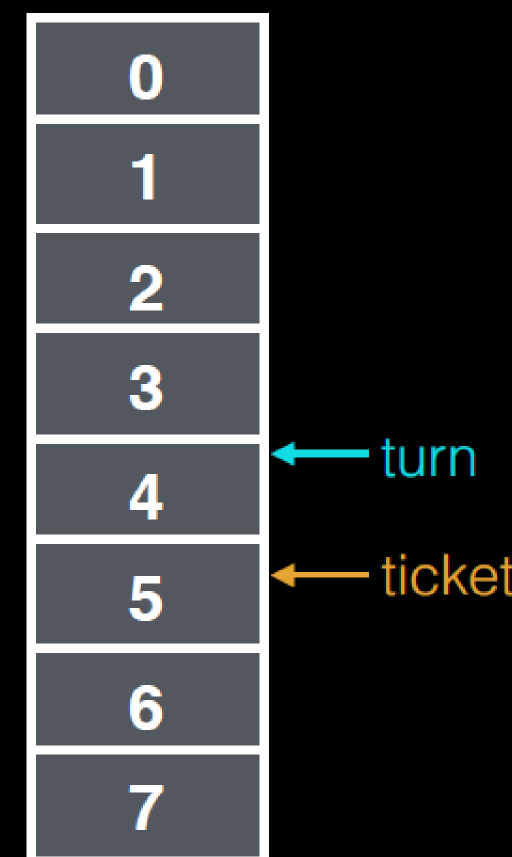
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++
C runs
C unlock(): turn++
A runs



A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++
C runs
C unlock(): turn++
A runs
A unlock(): turn++



A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++
C runs
C unlock(): turn++
A runs
A unlock(): turn++
C lock(): gets ticket 4, runs



Ticket Lock Implementation

```
typedef struct __lock_t {
        int ticket;
        int turn;
}

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
        int myturn = FAA(&lock->ticket);
        while (lock->turn != myturn); // spin
}

void release (lock_t *lock) {
        FAA(&lock->turn);
}
```

Lock Implementation Goals

Correctness

- Mutual exclusion
  - Only one thread in critical section at a time
- Progress (deadlock-free)
  - If several simultaneous requests, must allow one to proceed
- Bounded (starvation-free)
  - Must eventually allow each waiting thread to enter

Fairness

Each thread waits for same amount of time

Spinlock Performance

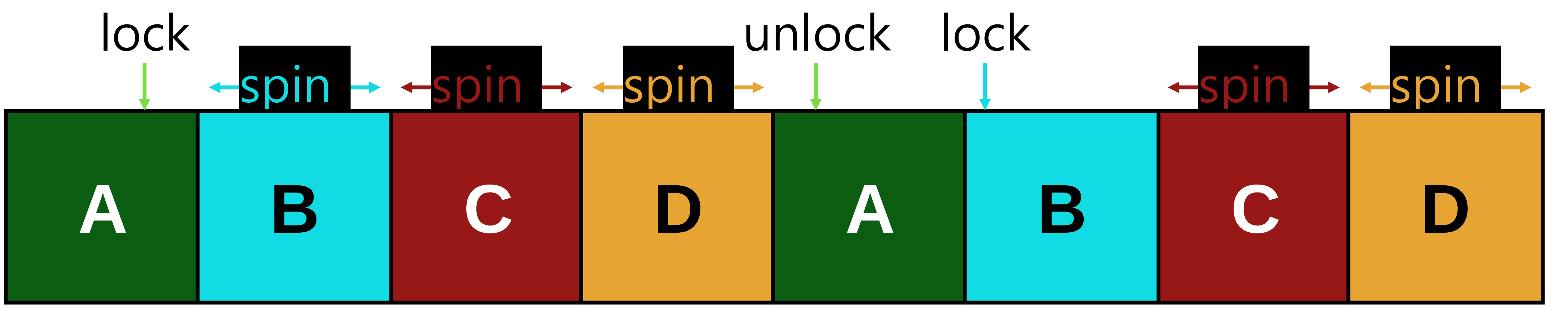
Fast when…

- many CPUs
- locks held a short time
- advantage: avoid context switch

Slow when…

- one CPU
- locks held a long time
- disadvantage: spinning is wasteful

CPU Scheduler is Ignorant



CPU scheduler may run **B** instead of **A**
even though **B** is waiting for **A**

Test-and-set Spinlock

```
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        ; // spin


void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

```
void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1)
        yield(); // spin


void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Pro: we won't waste cycles on spin now
Con: we may have to context switch many times to get the right thread

Queue Locks

Idea: put threads on queue.

Tell kernel don't schedule queued threads.

Upon unlock, tell kernel it can run thread(s) again.

Hybrid approach: spin a while, then queue self
- called "two-phase locks"

In-Kernel locking

Sometimes interrupt handlers have no context!

Queue locks cannot work. Why?

Approach: cooperative scheduling.
- use spin locks, disable interrupts

Ticket Lock with Yield()

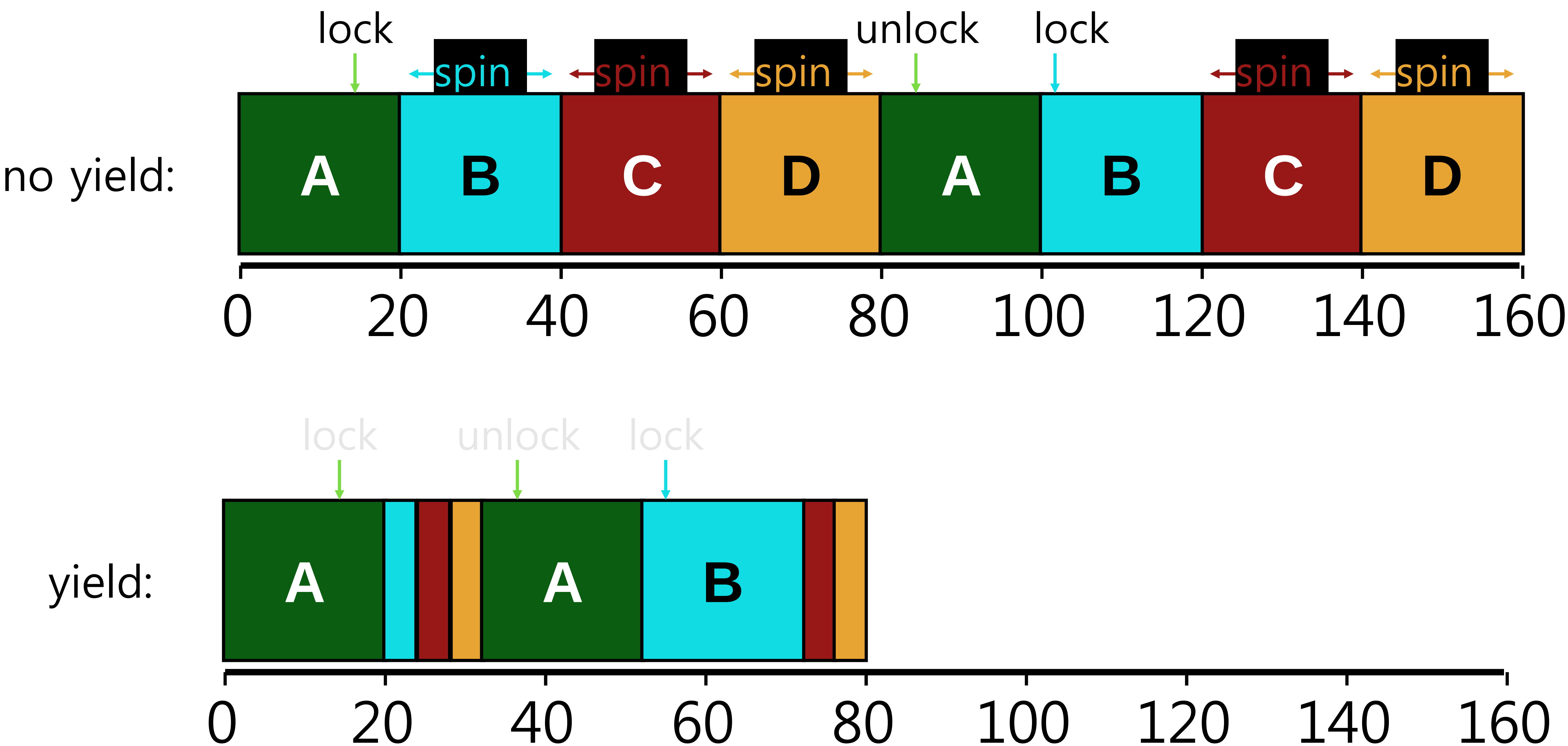
```
typedef struct __lock_t {
        int ticket;
        int turn;
}

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
        int myturn = FAA(&lock->ticket);
        while(lock->turn != myturn)
                yield();
}

void release (lock_t *lock) {
        FAA(&lock->turn);
}
```

Yield Instead of Spin



Spinlock Performance

Waste…

- Without yield: O(threads * **time_slice**)
- With yield: O(threads * **context_switch**)

So even with yield, spinning is slow with high thread contention

Next improvement: Block and put thread on waiting queue instead of spinning