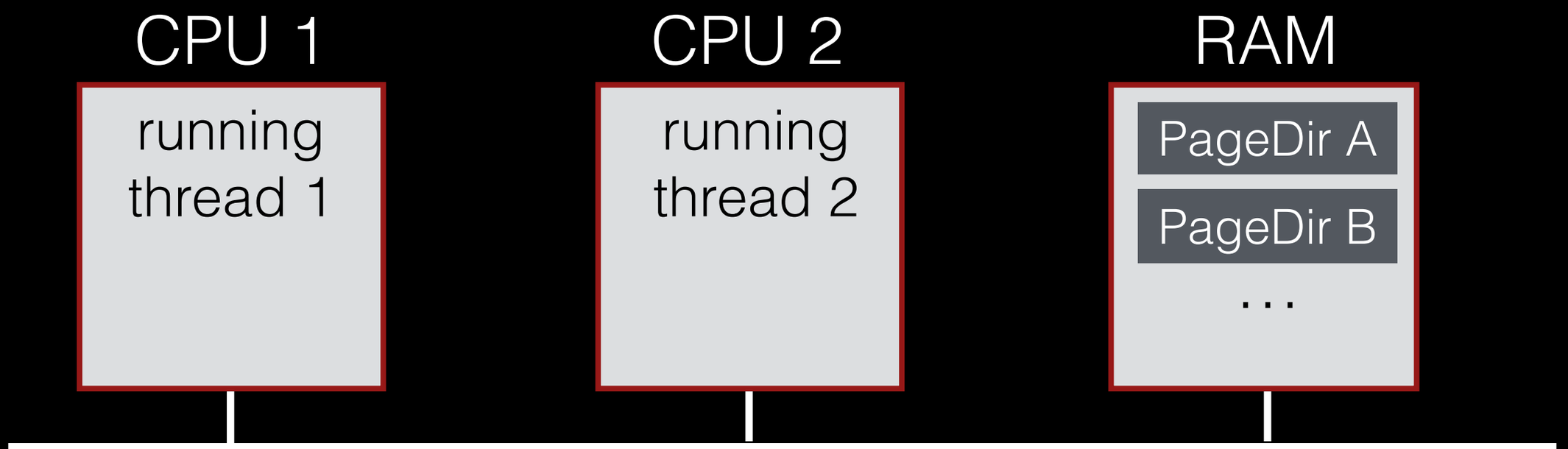
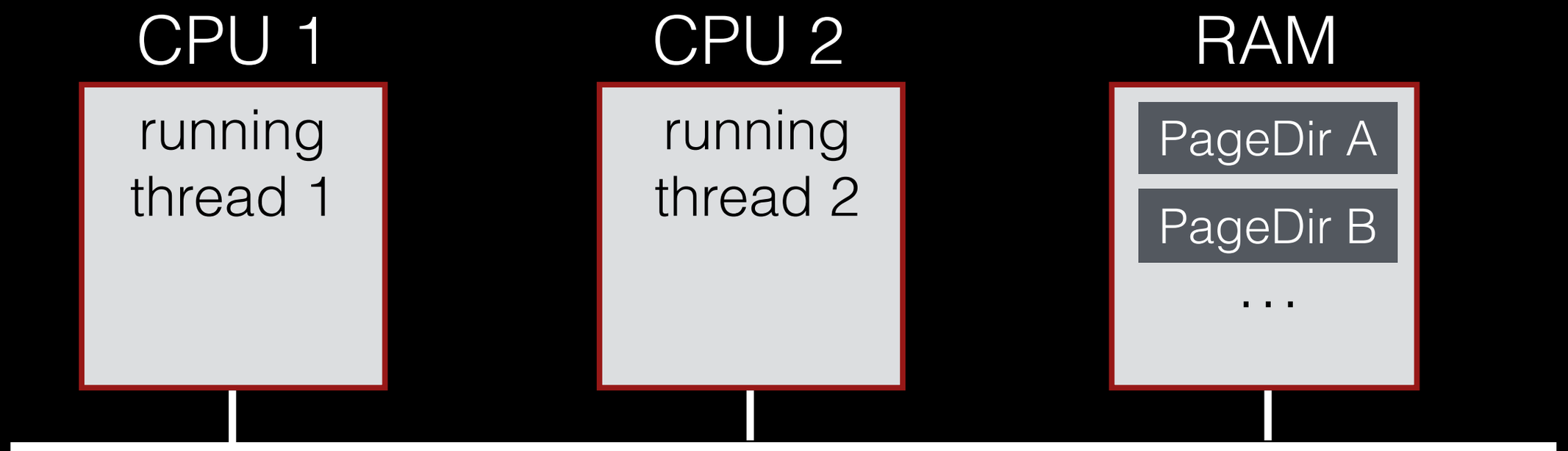
[537] Locks

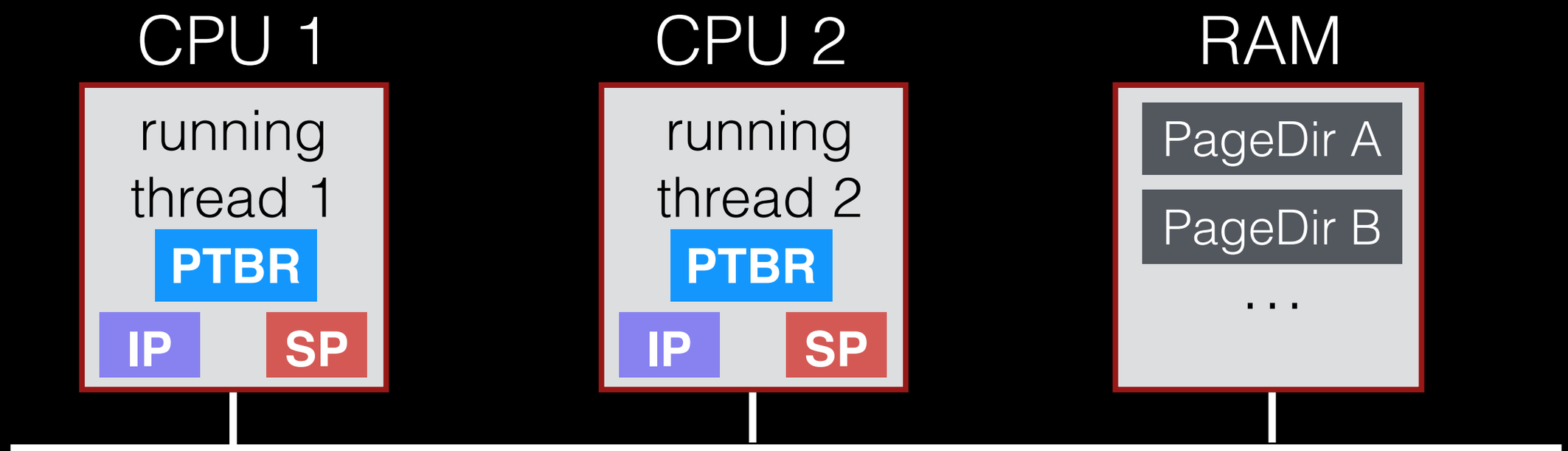
Chapters 28
Tyler Harter
10/06/14

Review: Threads+Locks

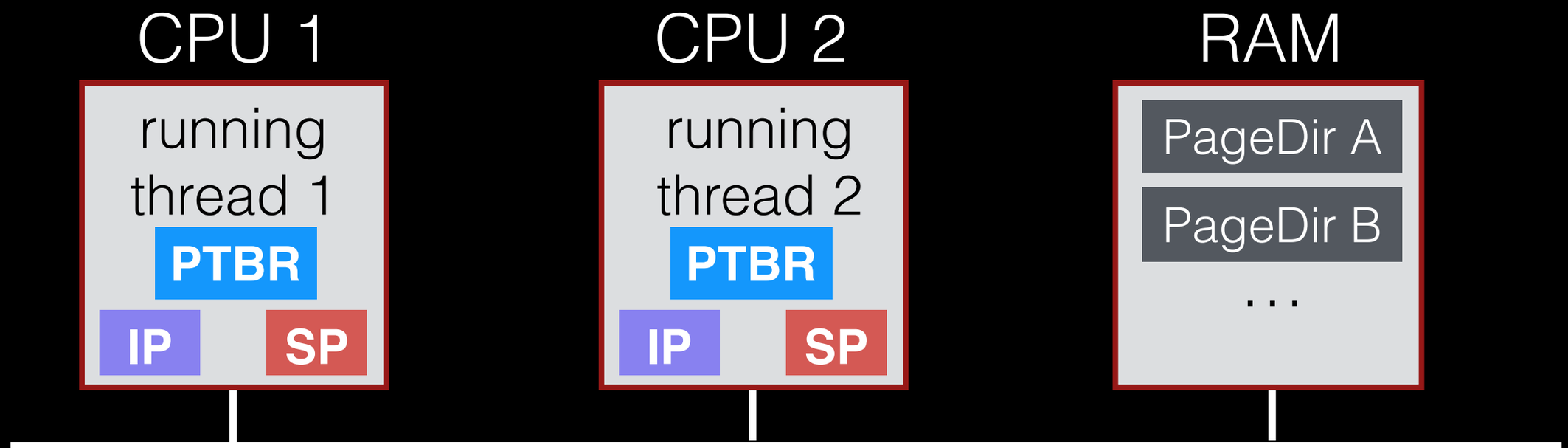

CPU 1
running
thread 1
CPU 2
running
thread 2
RAM
PageDir A
PageDir B
...


CPU 1
running
thread 1
CPU 2
running
thread 2
RAM
PageDir A
PageDir B
...

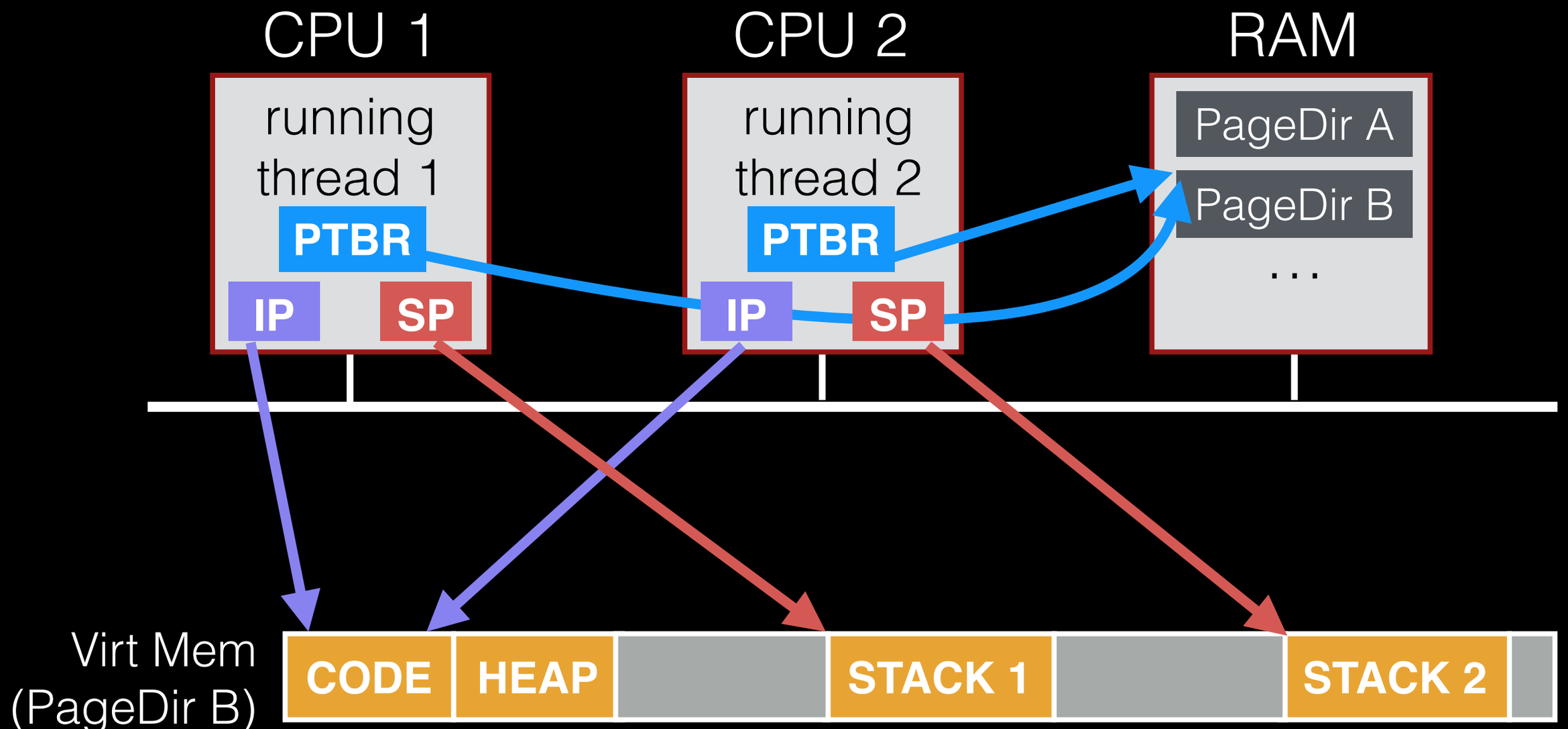

Virt Mem
(PageDir B)
CODE
HEAP


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



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

Context Switch

Why is switching between threads cheaper than switching between processes?

Why is switching between threads not free?

Why is concurrency hard?

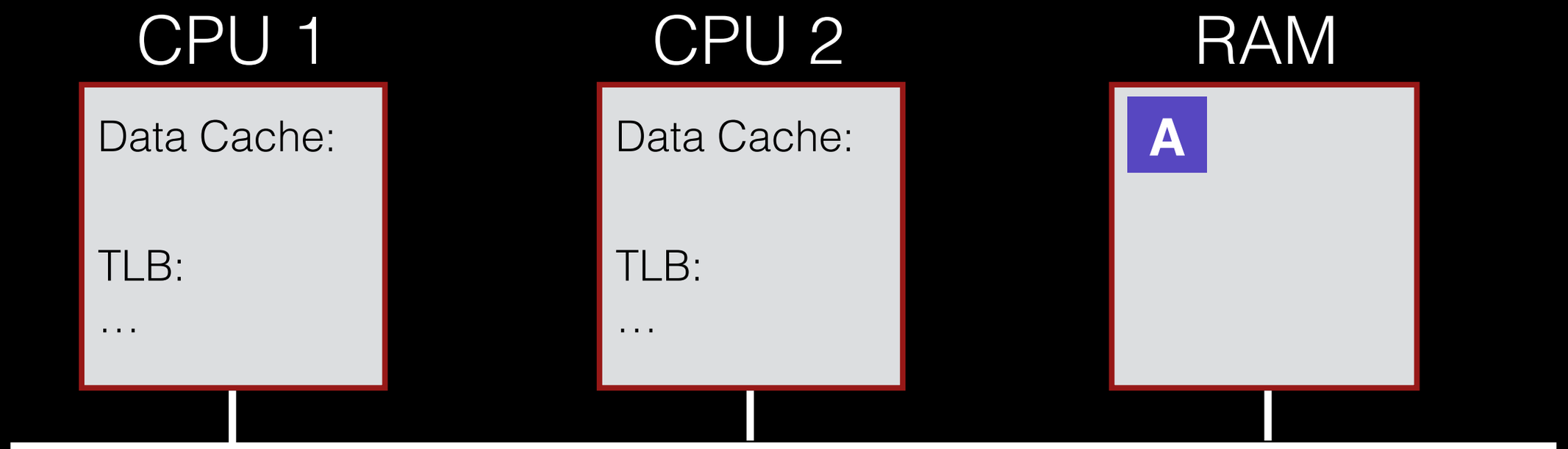
H/W caches

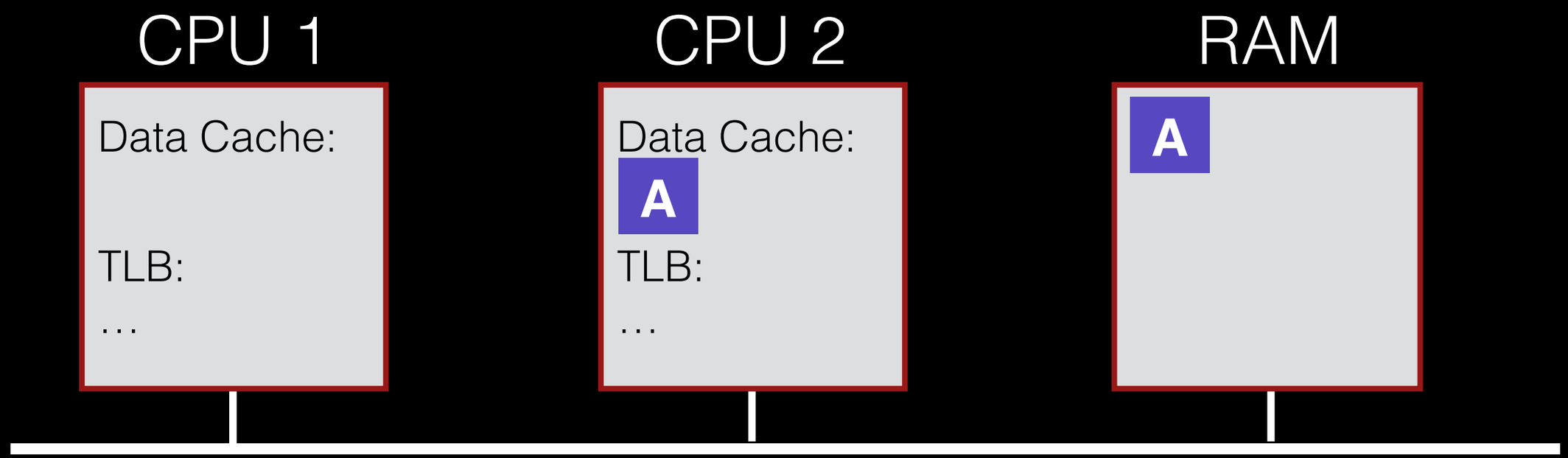
OS scheduler

Why is concurrency hard?

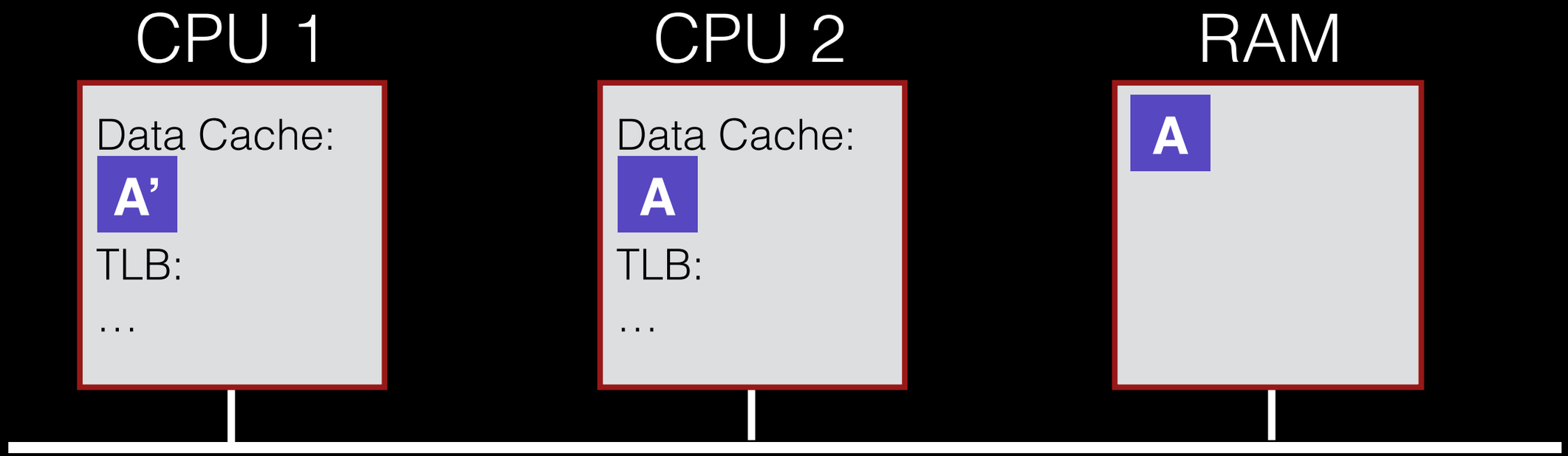
H/W caches

OS scheduler


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

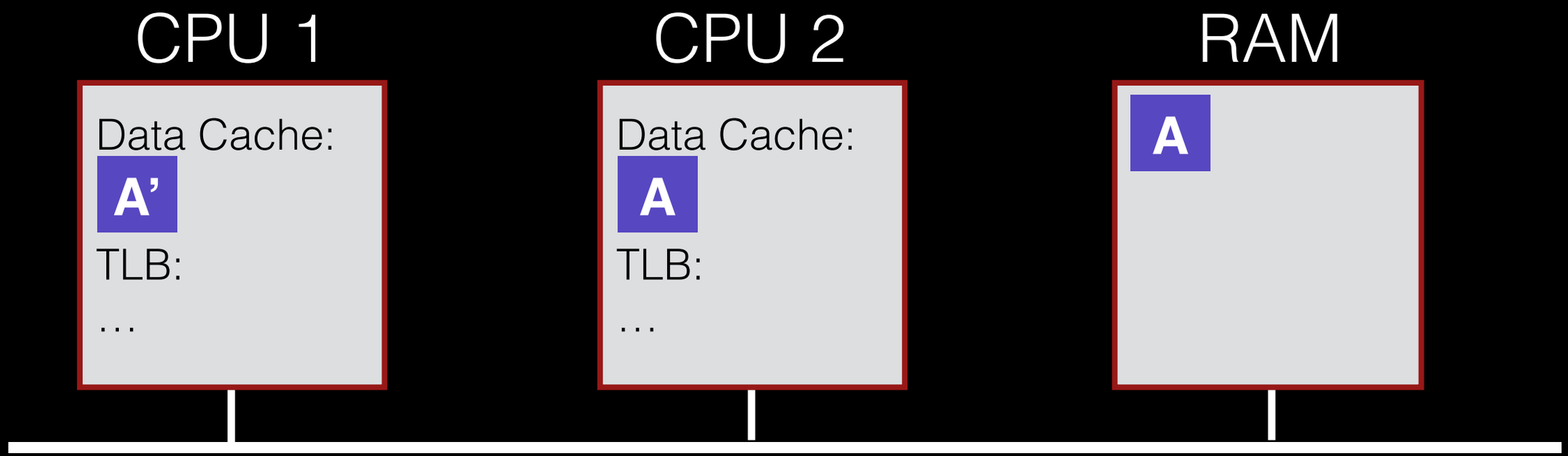


CPU 2: memory load returns **A**



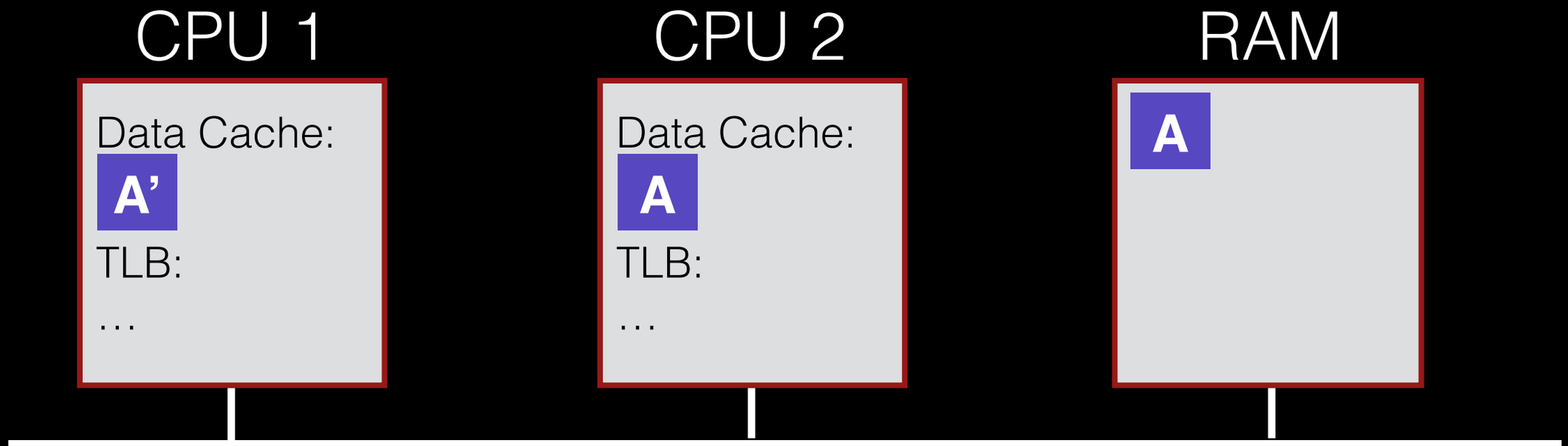
CPU 2: memory load returns **A**

CPU 1: memory store of **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;
}
```

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

```
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;
}
```

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

Why is concurrency hard?

H/W caches [552 and other courses]

OS scheduler [537's primary focus]

What if multiple threads run this?

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

Balance Adder

Thread 1

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

Thread 2

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

How much is added?

Balance Adder

Thread 1

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

Thread 2

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

How much is added?

Balance Adder

Thread 1

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

Thread 2

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

How much is added?

Balance Adder

Thread 1

mov 0x123, %eax (eax = 101)
add %0x1, %eax (eax = 102)
mov %eax, 0x123 (0x123 = 102)

Thread 2

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

How much is added?

Balance Adder

Thread 1

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

Thread 2

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

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

Worksheet

Problem 1.

Worksheet

Problem 1.

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

Using Locks

Worksheet

What about problems more complex than “balance”?

Problem 2 code.

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	

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 point to old head.

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 point to old head.
Only one (which one?) can be the new head.

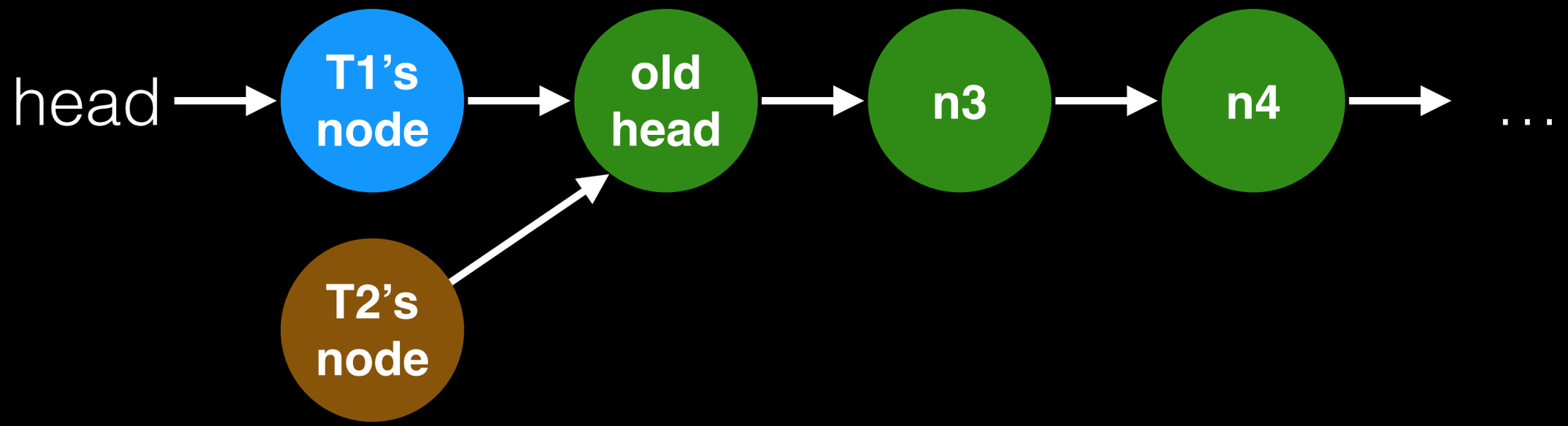
Thread 1

new->key = key
new->next = L->head

L->head = new

Thread 2

new->key = key
new->next = L->head
L->head = new



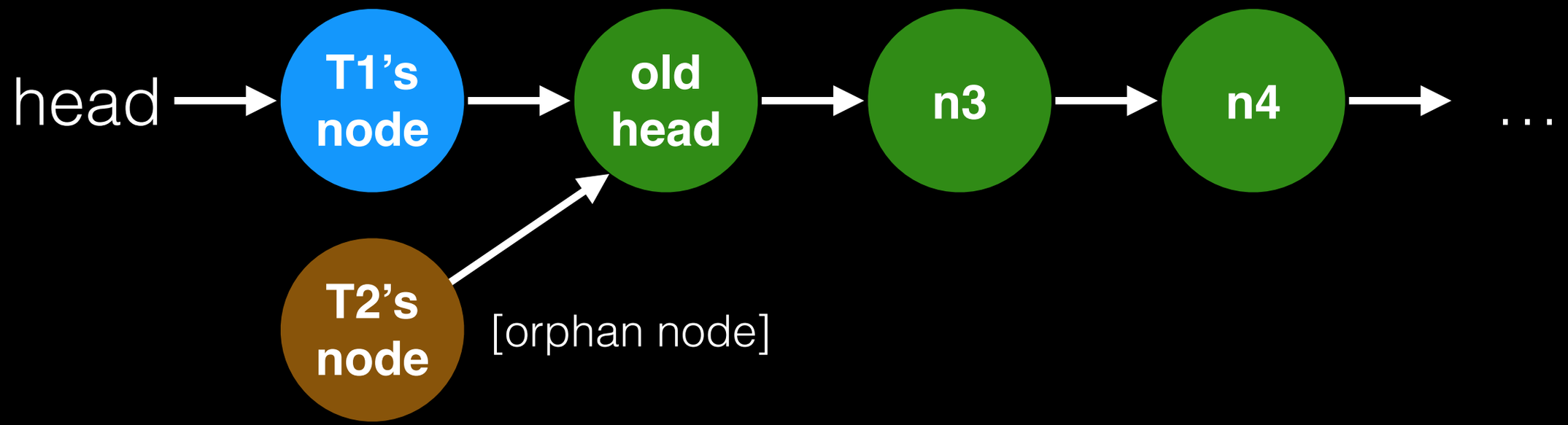
Thread 1

new->key = key
new->next = L->head

L->head = new

Thread 2

new->key = key
new->next = L->head
L->head = new



Worksheet

What about problems more complex than “balance”?

Problem 2 code.

Add locks to linked list!

Worksheet

What about problems more complex than “balance”?

Problem 2 code.

Add locks to linked list!

- talk about style (e.g., List_Lookup and __List_Lookup)

Building Locks

Lock Goals

Correctness

Fairness

Performance

Lock Goals

Correctness [need mutual exclusion between critical sections]

Fairness

Performance

Peterson's Algorithm

```
int flag[2];
int turn;

void init() {
      flag[0] = flag[1] = 0;   // 1 implies thread want to grab lock
      turn = 0;                // whose turn?  (thread 0 or 1)
}

void lock() {
      flag[self] = 1;     // self: thread ID of caller
      turn = 1 - self;    // make it other thread's turn
      while(flag[1-self] && (turn == 1 - self))
            ; // spin
}

void unlock() {
      flag[self] = 0;
}
```

Peterson's Algorithm

```
int flag[2];
int turn;

void init() {
      flag[0] = flag[1] = 0;   // 1 implies thread want to grab lock
      turn = 0;                // whose turn?  (thread 0 or 1)
}

void lock() {
      flag[self] = 1;     // self: thread ID of caller
      turn = 1 - self;    // make it other thread's turn
      while(flag[1-self] && (turn == 1 - self))
            ; // spin
}

void unlock() {
      flag[self] = 0;
}
```

doesn't work on modern hardware
(cache-consistency issues)

Worksheet

Build locks using other primitives (problem 3)

(a) test-and-set (already done)

(b) compare-and-swap

(c) load-linked / store-conditional

Lock Goals

Correctness [need mutual exclusion between critical sections]

Fairness

Performance

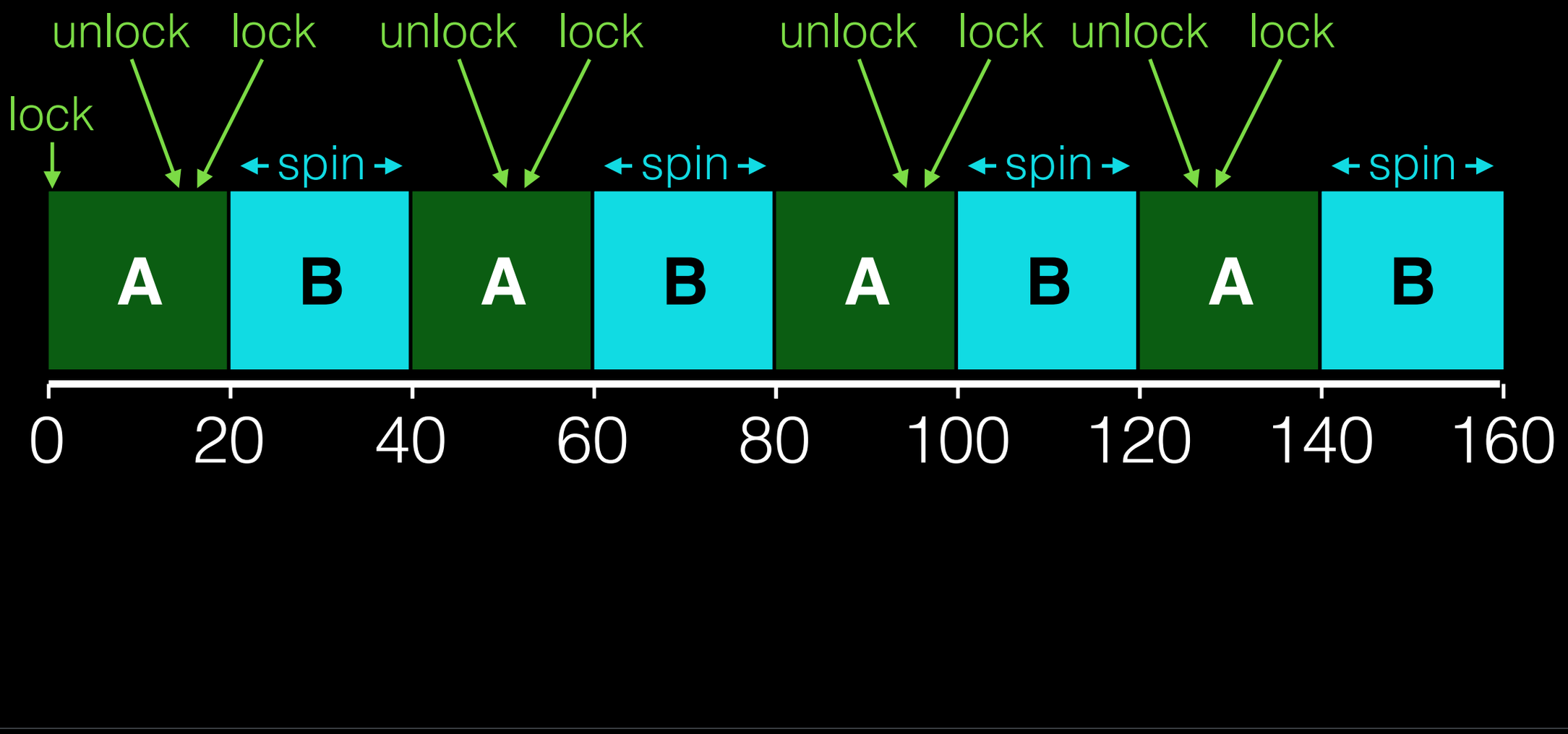
Lock Goals

Correctness [need mutual exclusion between critical sections]

Fairness [does each thread get its turn to hold the lock?]

Performance

Basic Spinlocks are Unfair



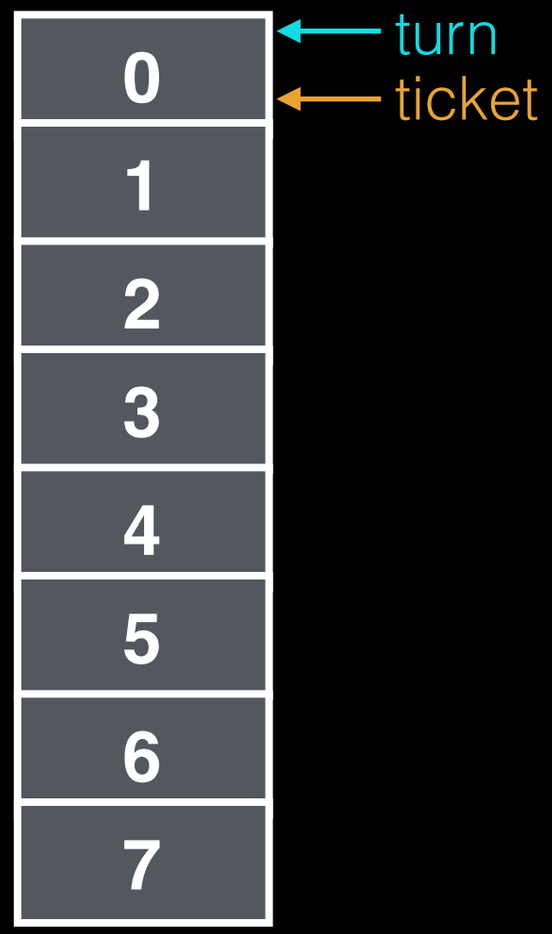
Being Fair: Ticket Locks

Idea: reserve your turn to use a lock.

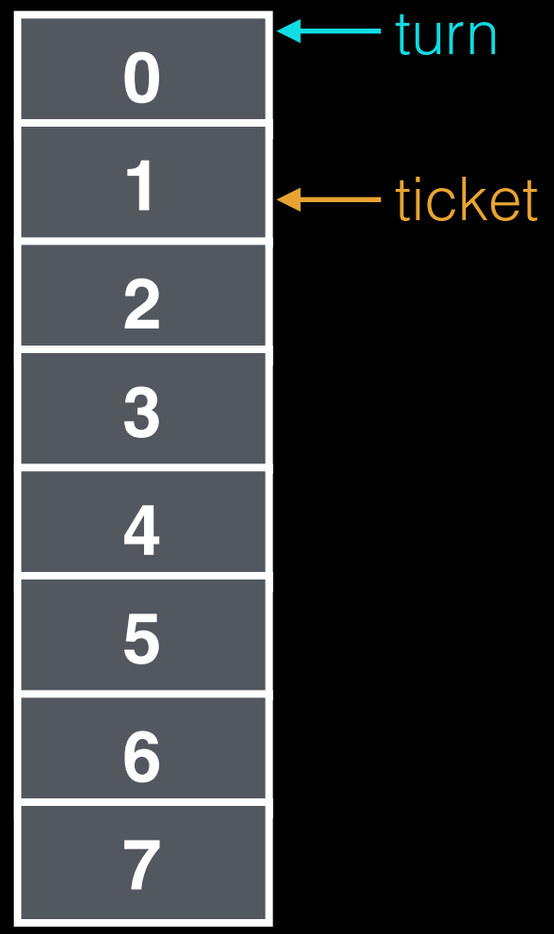
Spin until it's you turn.

Use new primitive, fetch-and-add:

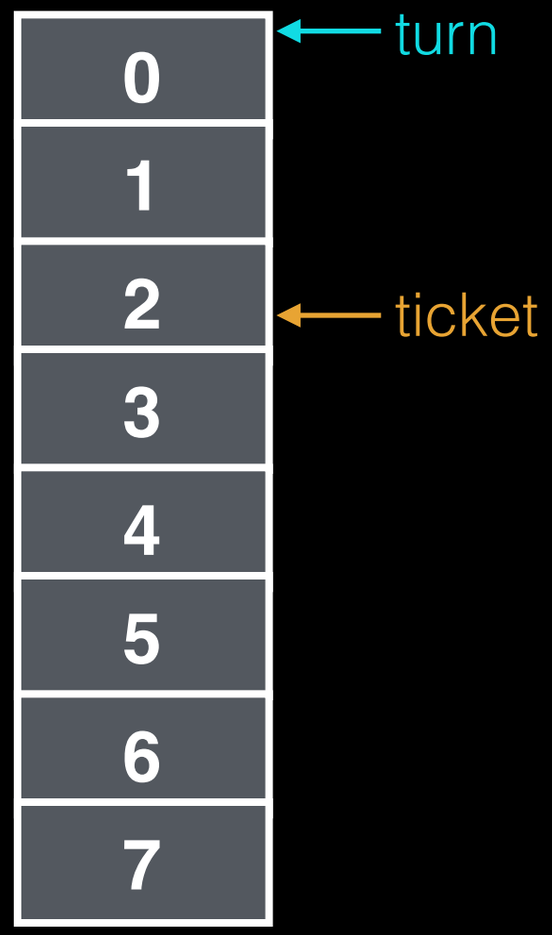
```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```


0
1
2
3
4
5
6
7
turn
ticket

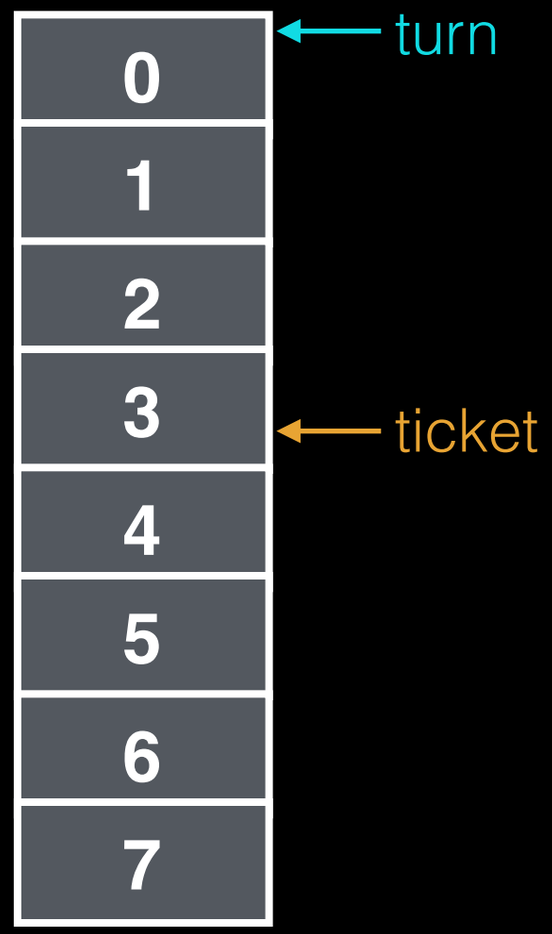
A lock(): gets ticket 0, runs



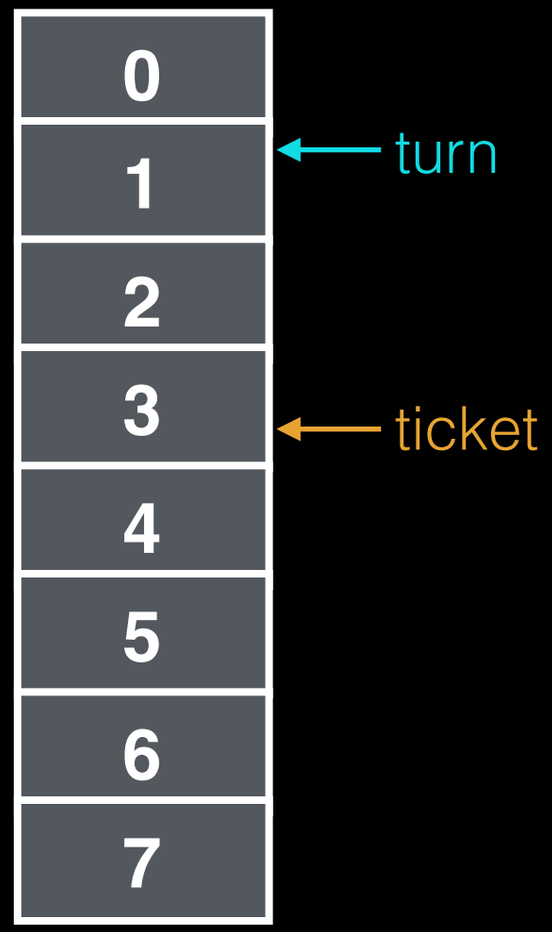
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1



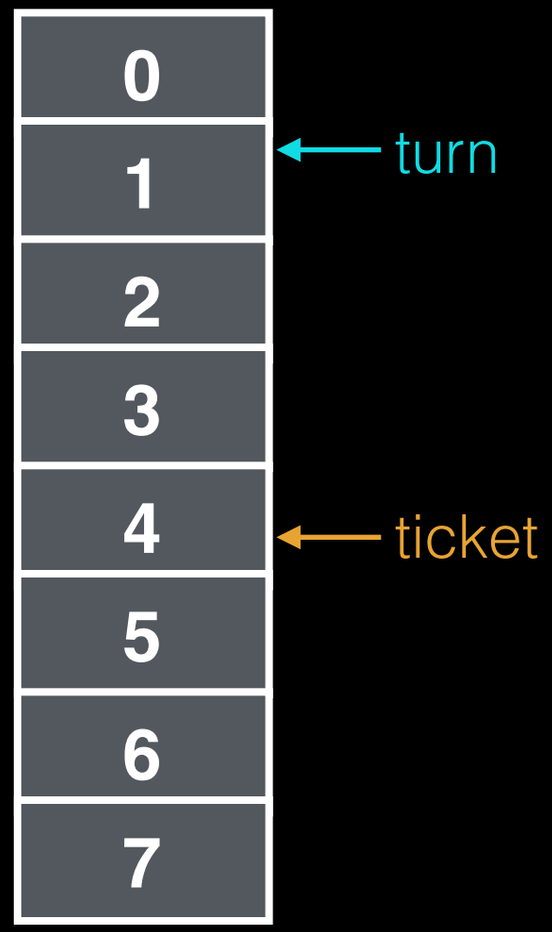
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2



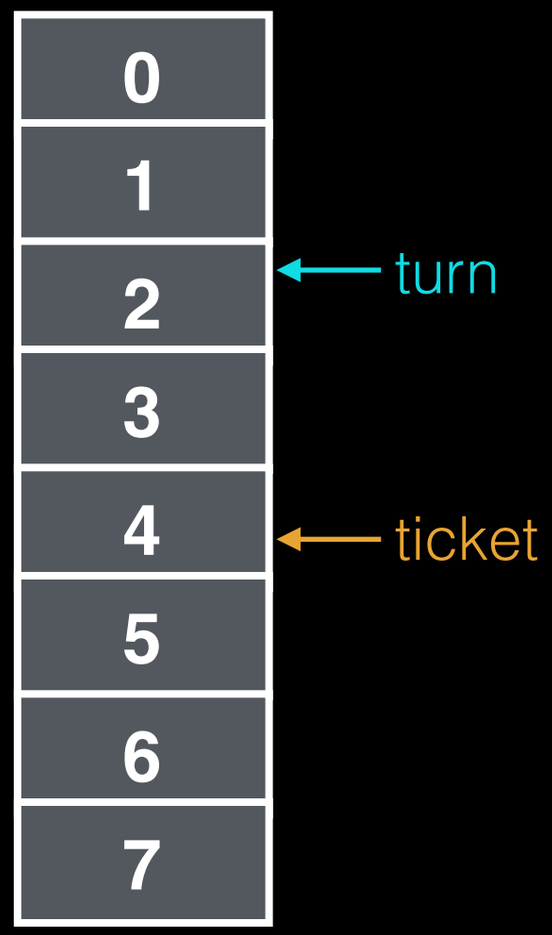
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs



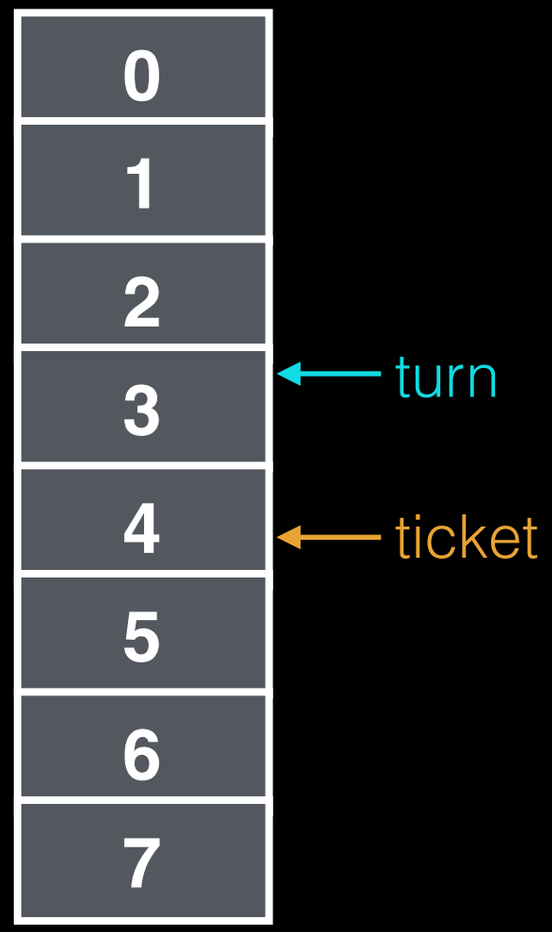
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs
A lock(): gets ticket 3, spins until turn=3



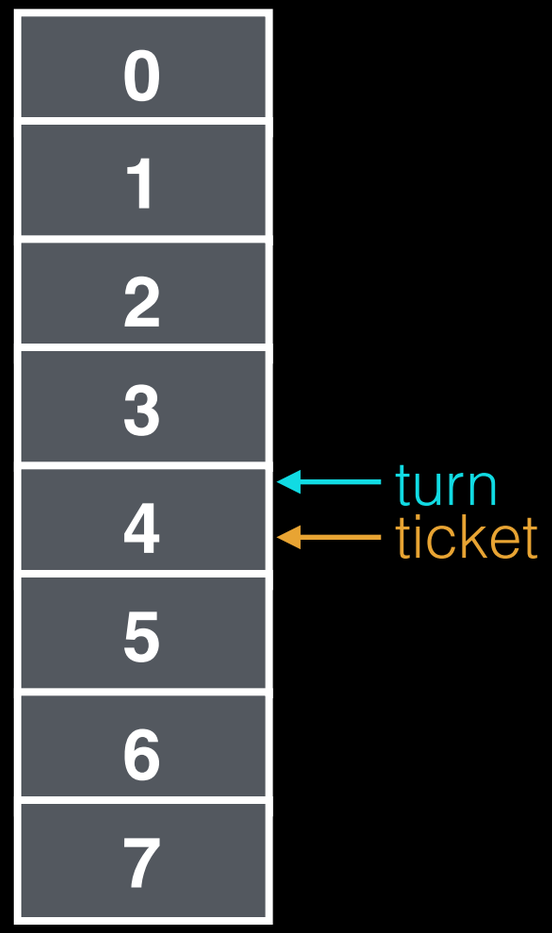
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++
C runs



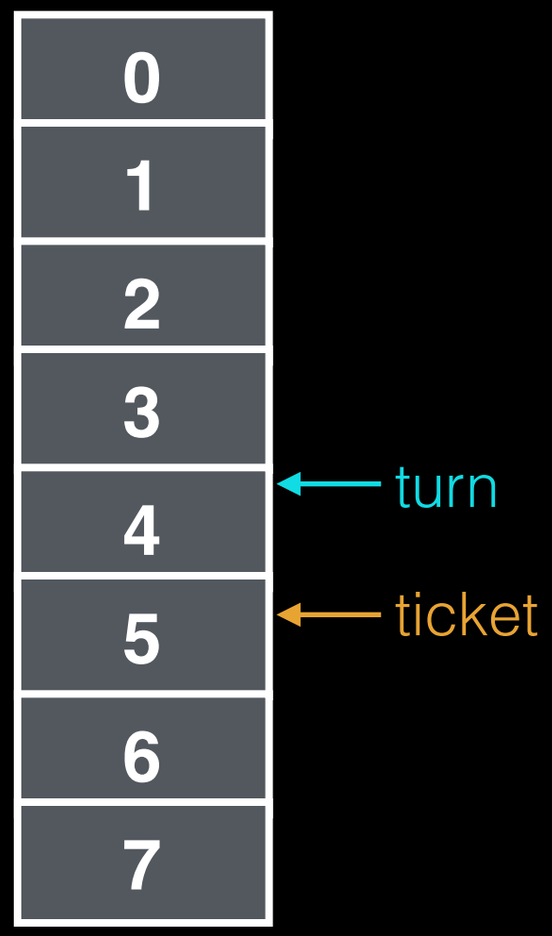
A lock(): gets ticket 0, runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++
C runs
C unlock(): turn++
A runs



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



Lock Goals

Correctness [need mutual exclusion between critical sections]

Fairness [does each thread get its turn to hold the lock?]

Performance

Lock Goals

Correctness [need mutual exclusion between critical sections]

Fairness [does each thread get its turn to hold the lock?]

Performance [how to minimize *switch overheads* and *spin waste*]

Spinlock Performance

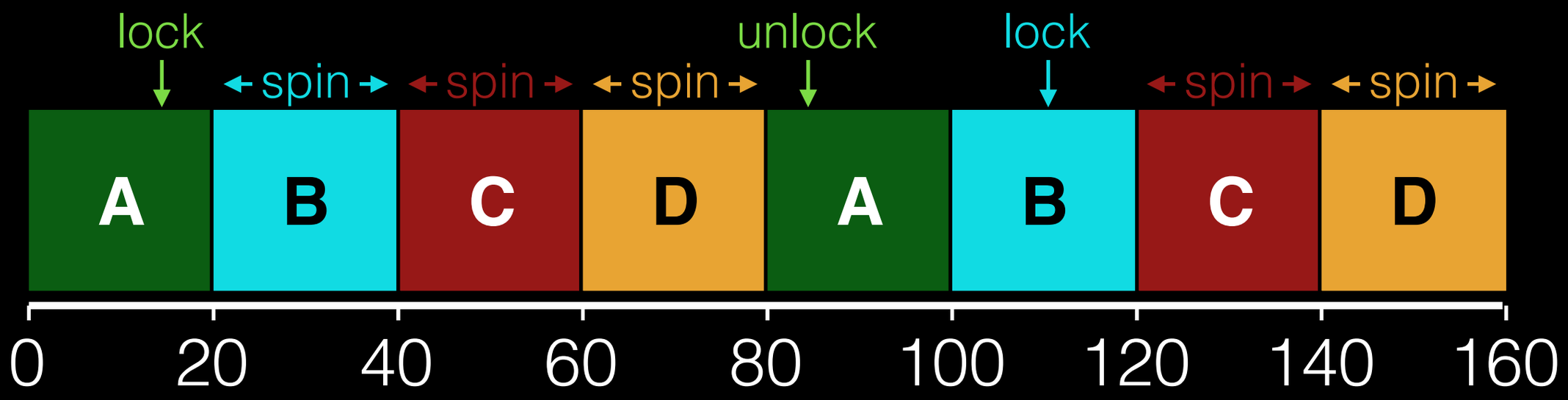
Fast when…

- 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;
}
```

Test-and-set Spinlock

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


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

Test-and-set Spinlock

```
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.

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

Locks Summary

Workload: how many threads, cores? Lock length?

Lock library: who to give lock? How to wait?

Metric: fairness, performance

Lock “algebra”, given 2 variables, find the 3rd:

f(**W**, **L**) = **M**

Announcements

p2b due this Friday.

Exam next Friday.
- Oct 17, 7-9pm, in CHEM 1351.
- Covers all material until that day.
- Read OSTEP!

Office hours today @ ~9:15am in **Galapagos lab**.

Wed lecture: **cloud computing**