CS 134
Operating Systems

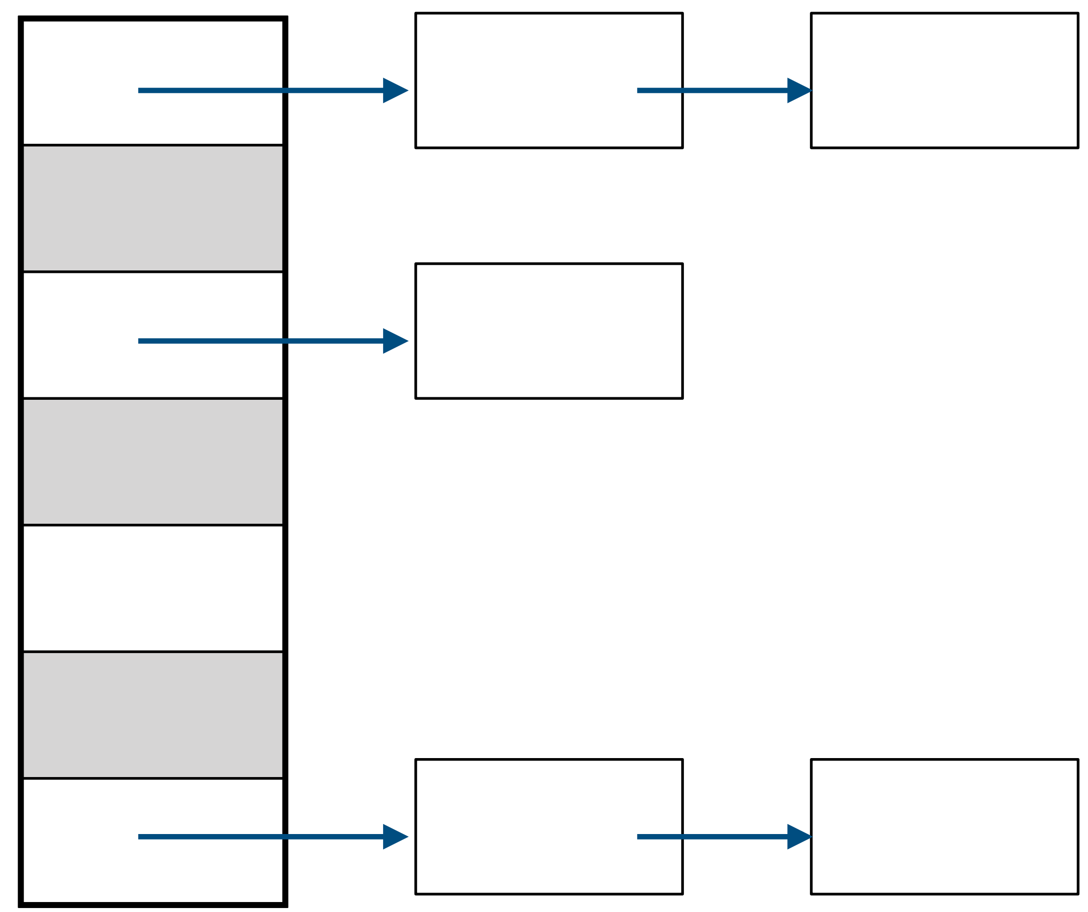
Feb 20, 2019

Multiprocessors and locking

Outline

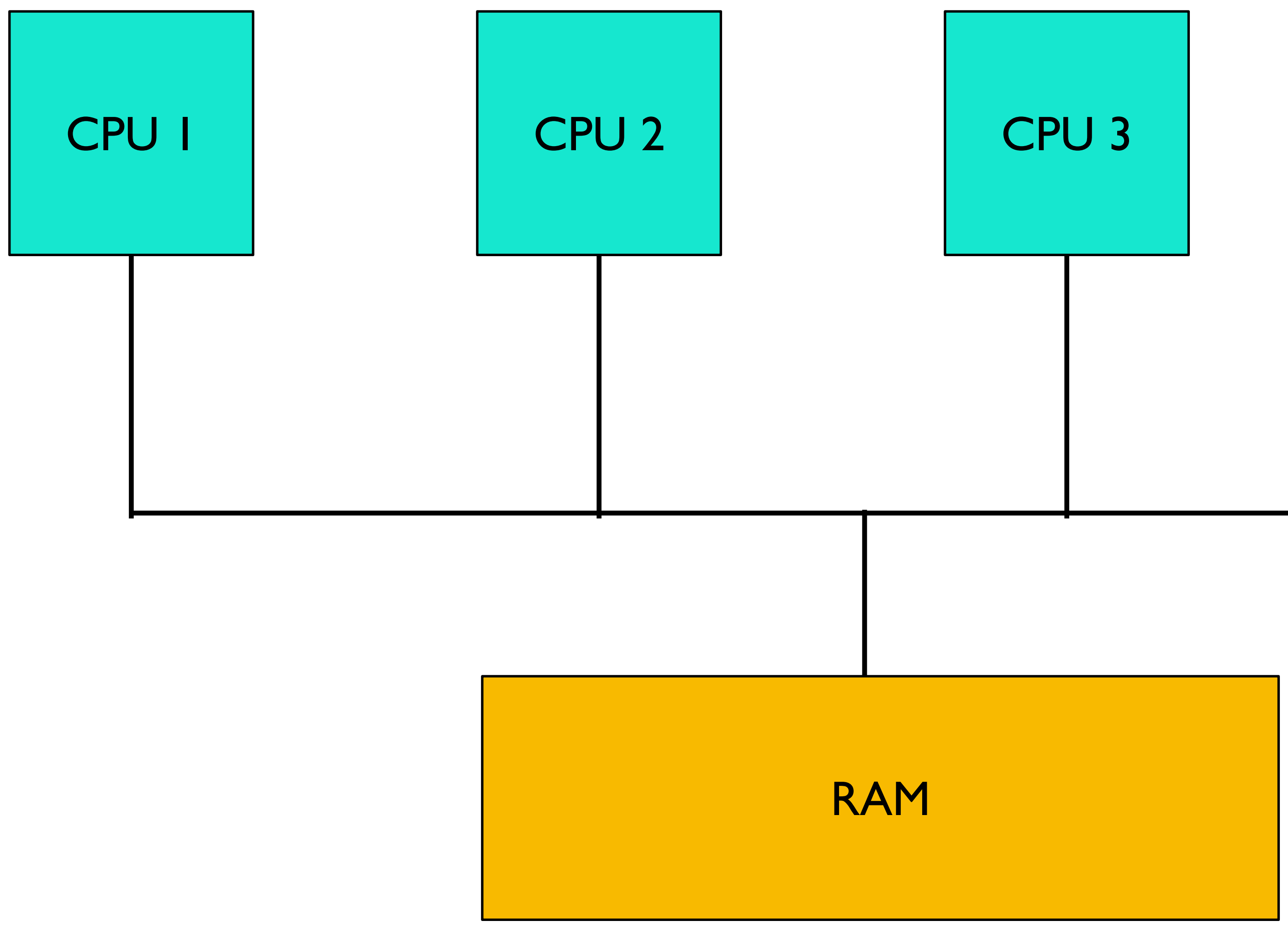
- Homework 6: locking
- Lock abstraction (and deadlocks)
- Atomic instructions and how to implement locks

Why run ph.c on multiple cores?



```
struct entry {
  int key, value;
  struct entry *next;
}
```

Why run ph.c on multiple cores?



ph0.c

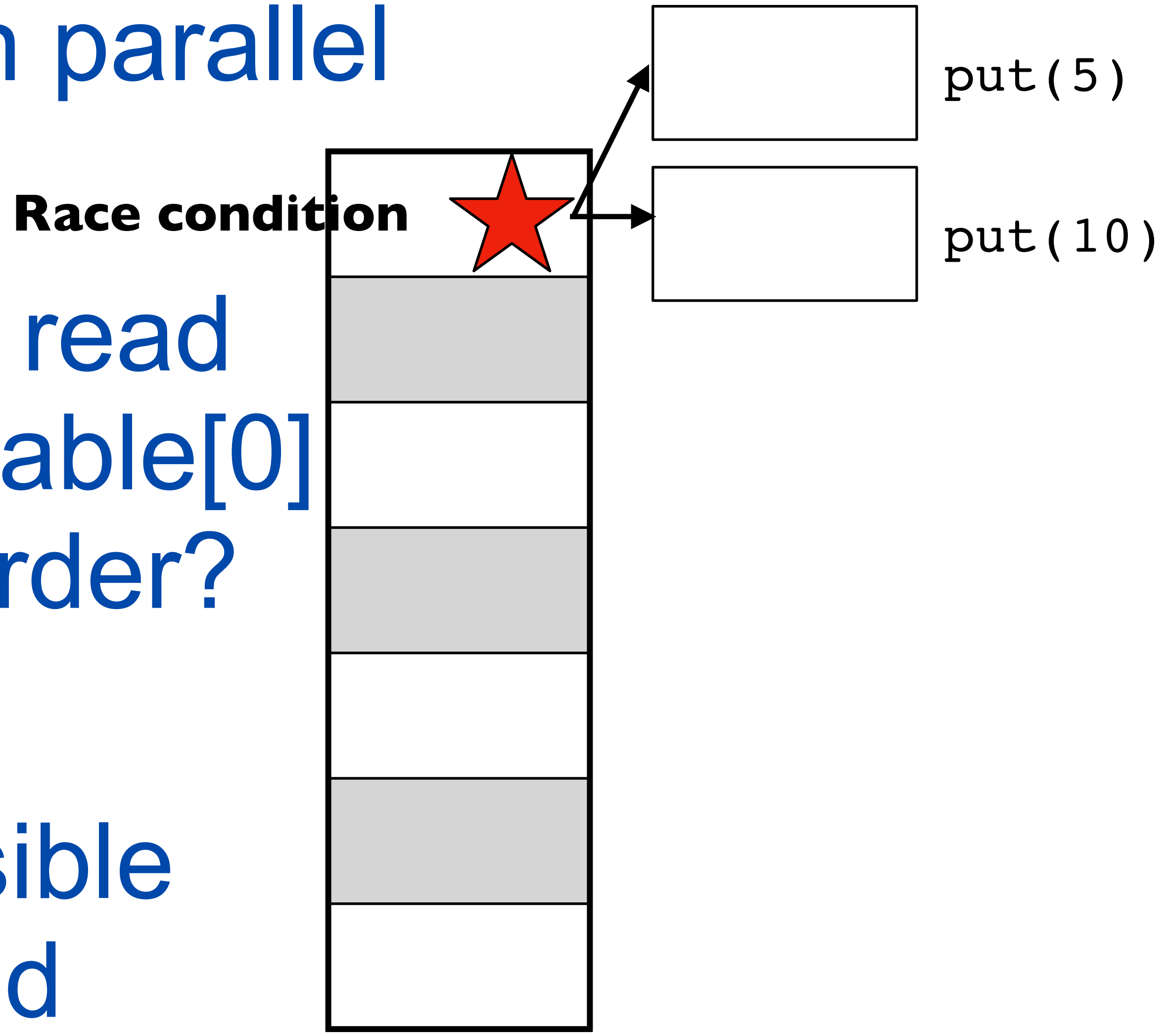
- Plan: no synchronization

Where are the missing keys?

- Plan: no synchronization

Where are the missing keys?

- Suppose put(5) and put(10) run in parallel



- Both threads read and write to table[0] but in what order?
- When a possible ordering could cause incorrect behavior, that's called a *race condition*

Race condition example

Time

Thread 1: `put(5)`

Read: `table[0]` → `tmp`

Write: `tmp` → `e->next`

Write: `table[0]` → `tmp`

Thread 2: `put(10)`

Read: `table[0]` → `tmp`

Write: `tmp` → `e->next`

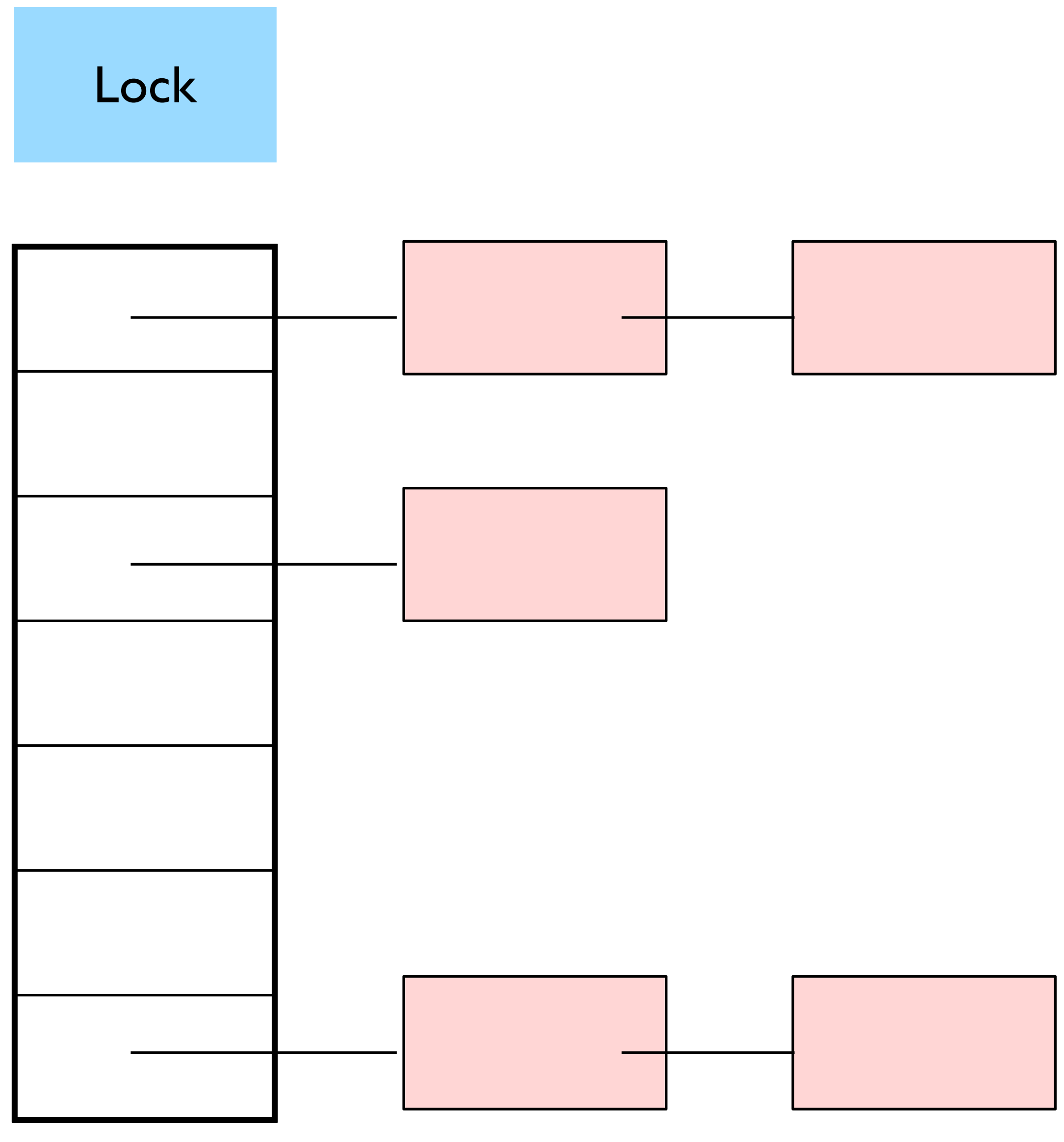
Write: `table[0]` → `tmp`

Last writer wins!

ph1.c

- Plan: big lock/coarse-grained synchronization

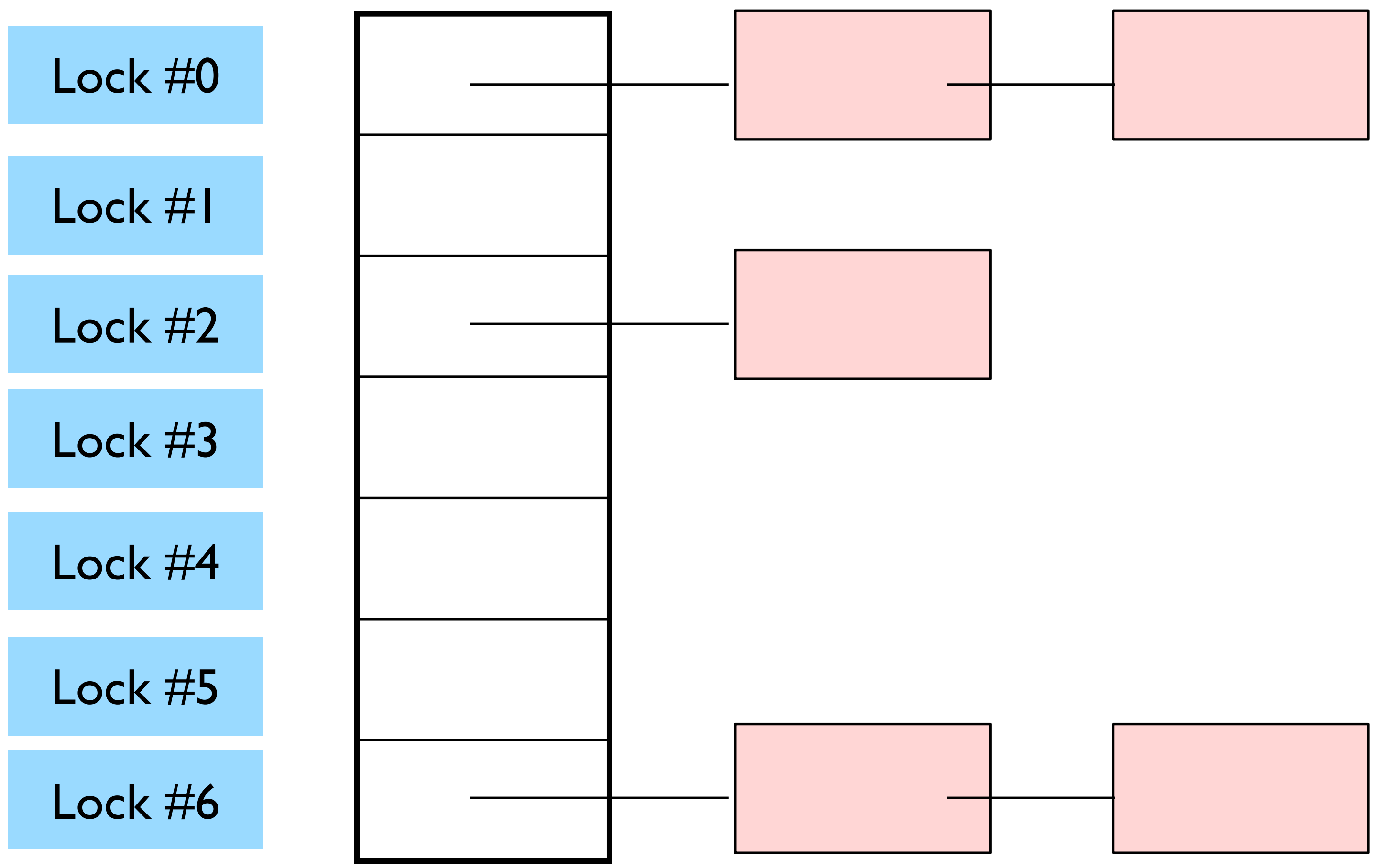
Big lock



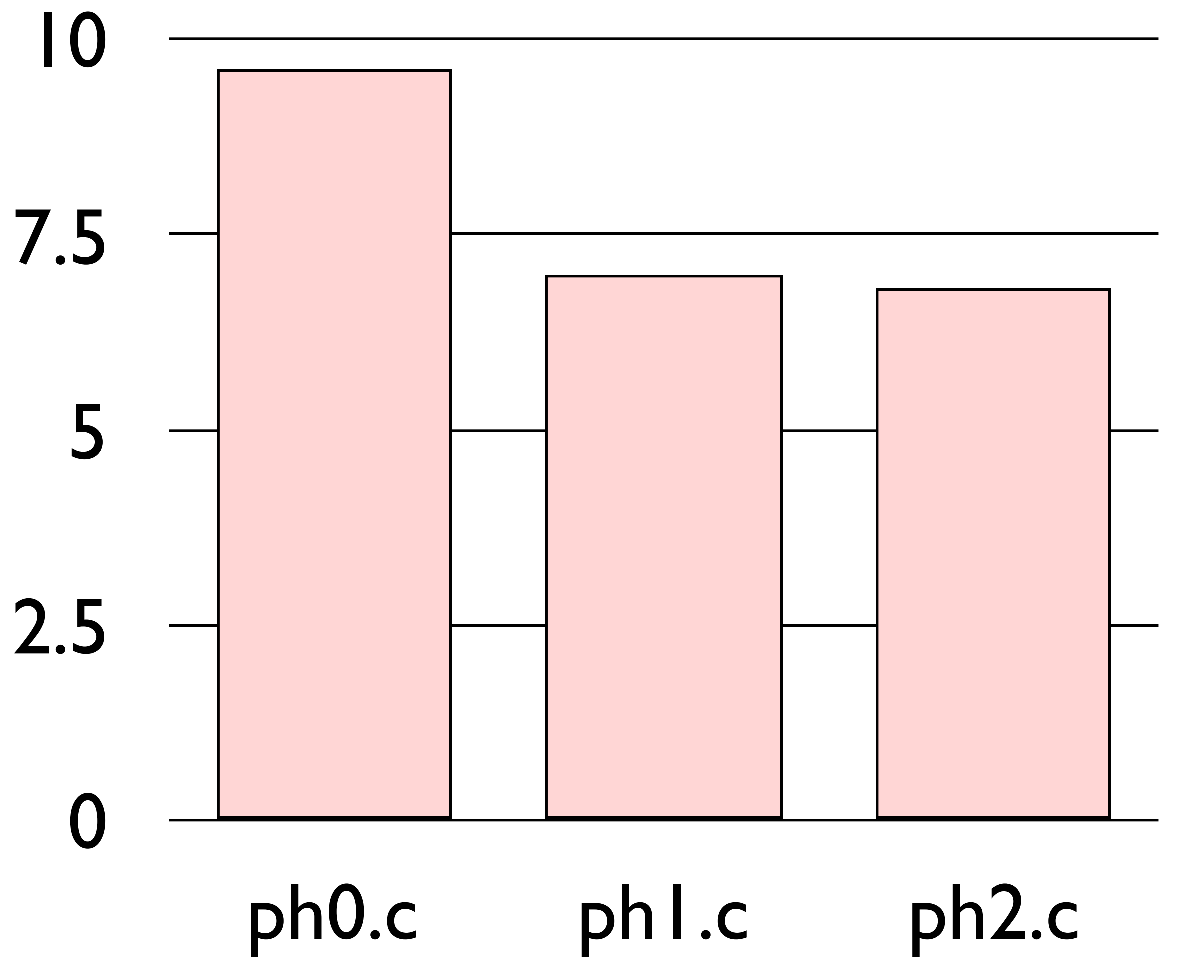
ph2.c

- Plan: bucket locks/fine-grained synchronization

Bucket Locks



ph[0-2].c runtime with 4 cores



Atomic operation

- Indivisible
  - Either completely finishes, or doesn’t do anything
  - Can’t be interrupted
- Loads and stores of single value atomic (in HW)
  - `movl $52, eax`
- Loads and stores of aggregate values not atomic
  - ```
    struct MyStruct a, b;

    a = b
    ```
    ```

Concurrent hash table questions

- Does `get()` need a lock in ph.c?
- Does `get()` need a lock with concurrent `put()`?
- Would `get()` need a lock if we supported deletes?

The lock abstraction

- Using locks:

```
lock l

acquire(&l)
  x = x + 1  // critical section
release(&l)
```

- Suppose multiple threads call acquire(&l)
  - Only one returns right away
  - Others must wait for release(&l)
- Protect different data with different locks
  - Allows independent critical sections to run in parallel
- Locks not implicitly tied to data; programmer must plan

When to lock

1. Do two or more threads touch a memory location?
2. Does at least one thread write to that memory location?

If yes to both, you need a lock!

Too conservative: sometimes deliberate races are fine!

Too liberal: Think about invariants of entire data structures (not just single memory locations)

What locks achieve

- Help avoid lost updates
- Help you create multi-step atomic operations, hiding intermediate states
- Help maintain invariants on data structures
  - Assume: invariants true at start of critical region
  - Intermediate states may violate invariants
  - Restore invariants before releasing lock

Problem: Locks can cause deadlock

Time

CPU 0:

in `rename("a/f", “b/f”)`

```
acquire(&a)
acquire(&b)
…
release(&b)
release(&a)
```

CPU 1:

in `rename("b/f", “a/f”)`

```
acquire(&b)
acquire(&a)
…
release(&a)
release(&b)
```

Could end up with both hung forever

Solution to lock deadlocks

- Programmer works out an order in which locks are to be acquired
  - One idea: use the VA of the lock, least to greatest
- Always acquire locks in the same order
- Complex!

Tradeoff between locking and modularity

- Locks make it hard to hide details inside modules
  - E.g., to avoid deadlock, you have to know which locks are acquired by each function
- Locks aren't necessarily the private business of each individual module
- Too much abstraction can make it hard to write correct, well-performing locking

What about performance?

- We want parallel speedup

Locks prevent parallelism

- To maintain parallelism, split up data and locks
- Choosing the best design is a challenge
  - Whole ph.c table, each table[] row, each entry?
  - Whole file system, each file/directory, each block?
- May need to make design changes to promote parallelism
  - Example: break a single free list into a per-core free list

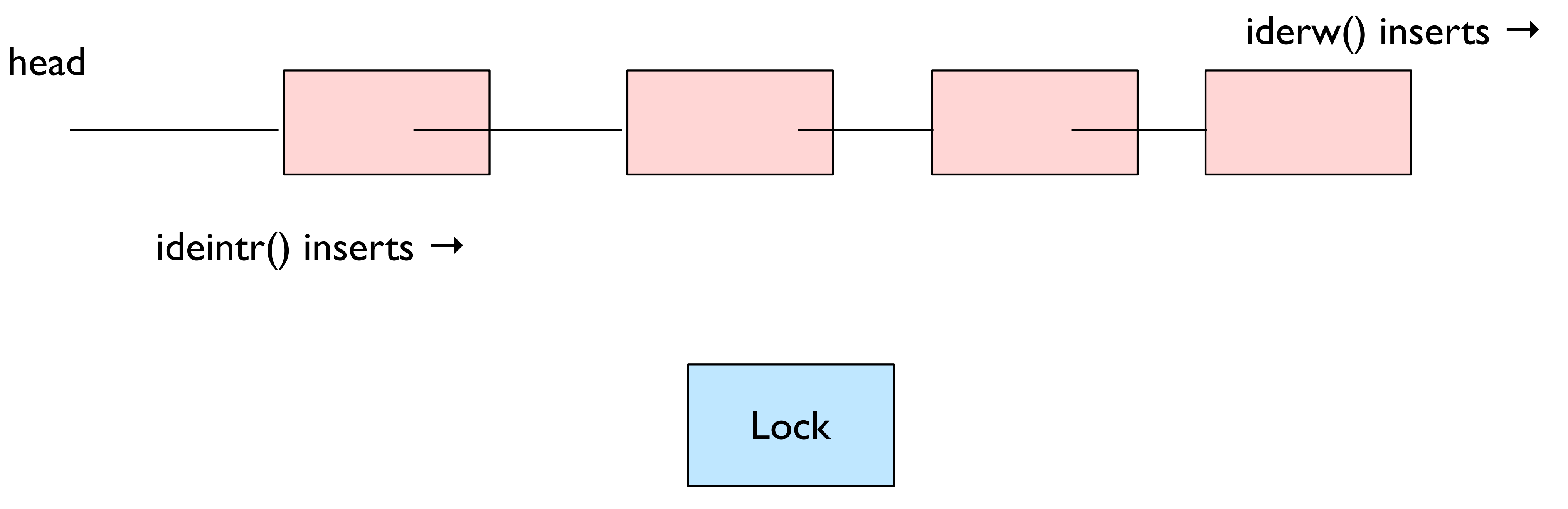
Lock granularity

- Start with big locks—one per module, perhaps
  - Less opportunity for deadlock
  - Less reasoning about invariants
- Then measure to see if there's a problem
  - Big locks could be enough, maybe not much time is spent in the module
  - Redesign only if you have to

"You can't optimize too early"

Example: xv6 IDE driver

- `iderw()` issues a block request
- `ideintr()` completes a block request



How to implement locks

```
struct lock {int locked};

acquire(struct lock *lk) {
  for (;;) {
    if (lk->locked) == 0) // A
      lk->locked = 1;     // B
      break;
    }
  }
}
```

x86 has an atomic exchange instruction

```
mov $1, %eax
xchg %eax, addr
```

What `xchg` does in hardware

```
lock addr globally
temp = *addr
*addr = %eax
%eax = temp
unlock addr
```

so other cores can't use it

Correct way to implement lock

```
struct lock {int locked};

acquire(struct lock *lk) {
  for (;;) {
    if (!xchg(&l->locked, 1)) // A & B
      lk->locked = 1;
      break;
    }
  }
}
```

Spinlock.c

- xv6 support for locks

- Why does xv6 disable interrupts in `acquire` and re-enable in `release`?

Memory ordering

- The compiler and CPU can re-order reads and writes
  - They do not have to obey the source program's order of memory references
  - Legal behaviors are referred to as a *memory model*
- Calls to xchg() prevent reordering
- If you use locks, you don't have to understand memory ordering (very much)
- For exotic lock-free coding, you'll need to understand every detail

Why spin locks?

- CPU cycles wasted while lock is waiting
- Idea: Give up the CPU and switch to another process
- Guidelines:
  - Spin locks only for very short critical sections
  - What about longer critical sections?
- Blocking locks available in most systems
  - Higher overhead, typically
  - But ability to yield the CPU

Conclusion

- Don’t share if you don’t have to
- Start with coarse-grained locking
- Don’t assume; measure! Which locks prevent parallelism?
- Insert fine-grained locking only when you need more parallelism
- Use automated tools like race detectors to find locking bugs