6.828: Locking

Adam Belay <abelay@mit.edu>

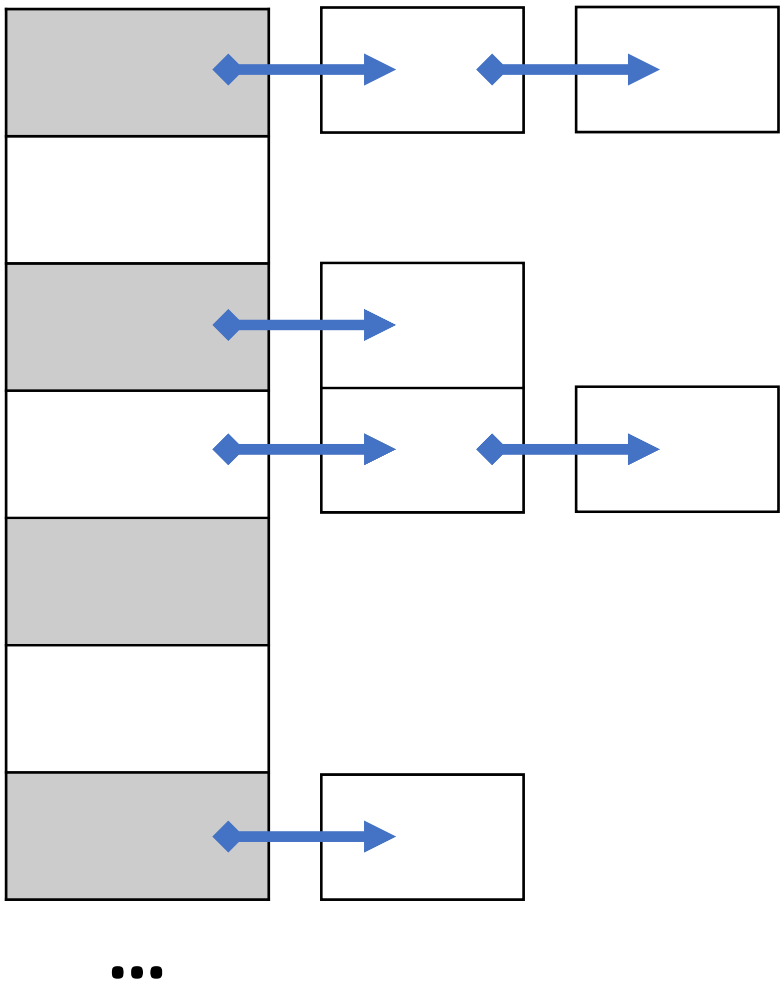
Plan for today

- Locking homework solutions
- Lock abstraction + Deadlocks
- Atomic instructions and how to implement locks

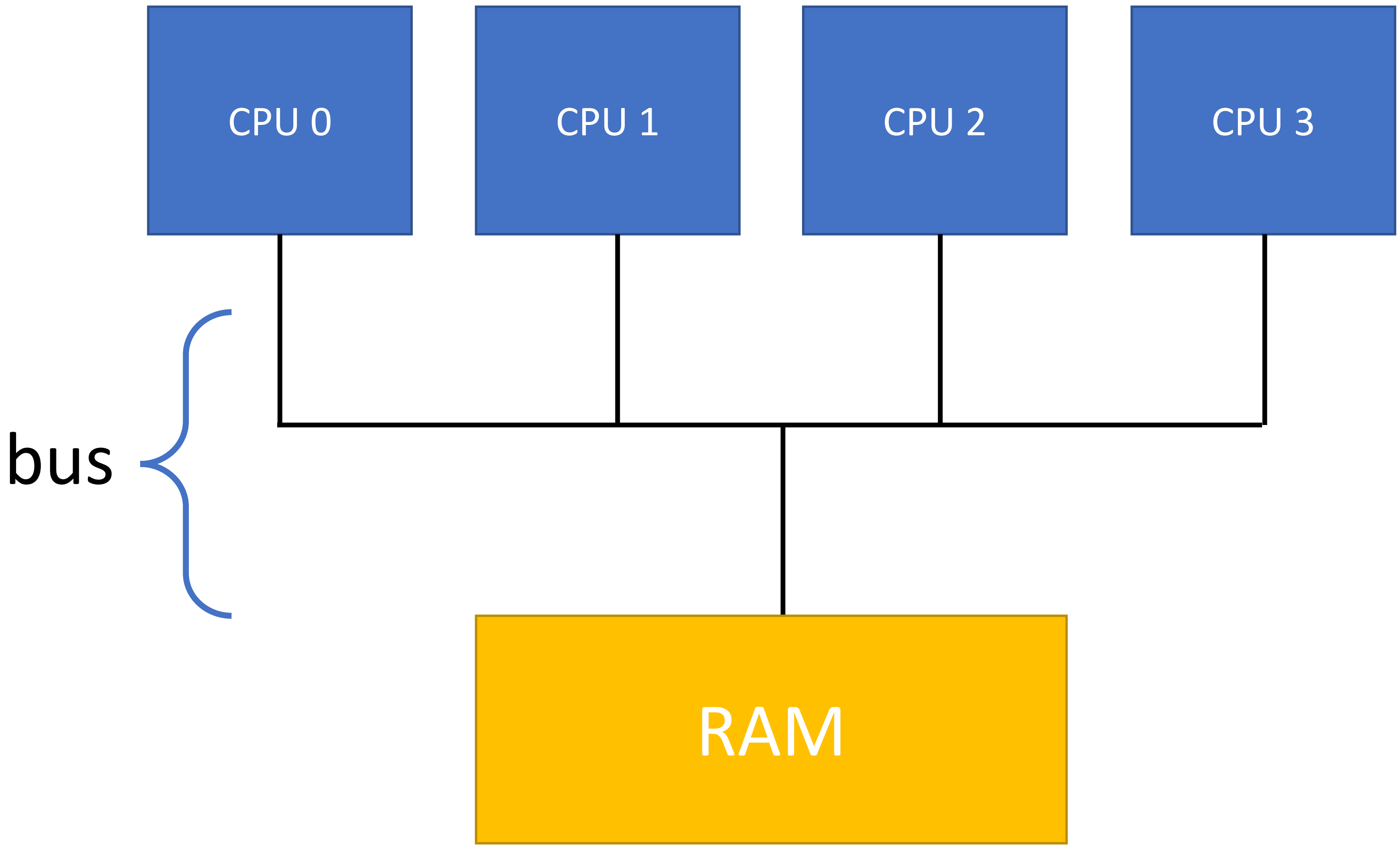
HW: Multithreaded hash table

- Concurrent operations
- Put() and Get()
- Collisions resolved with chaining

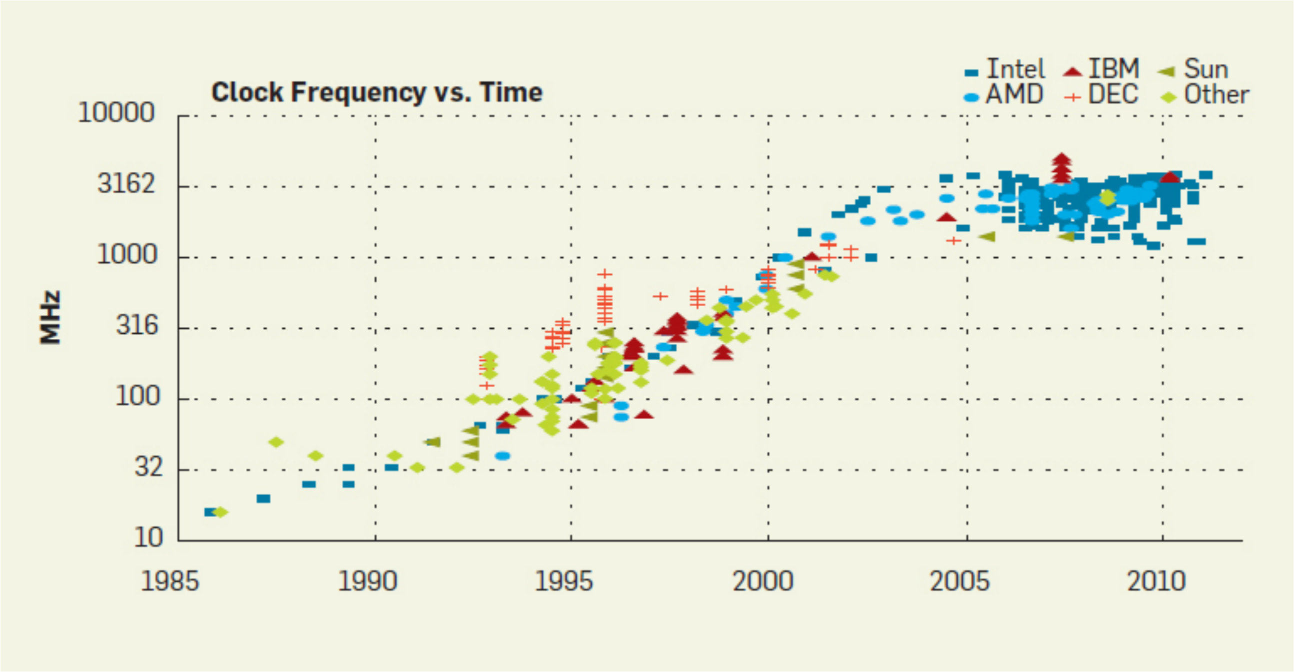
```
struct entry {
  int key, value;
  struct entry *next;
};
```



Why run ph.c on multiple cores?



Reality: Parallelism is unavoidable



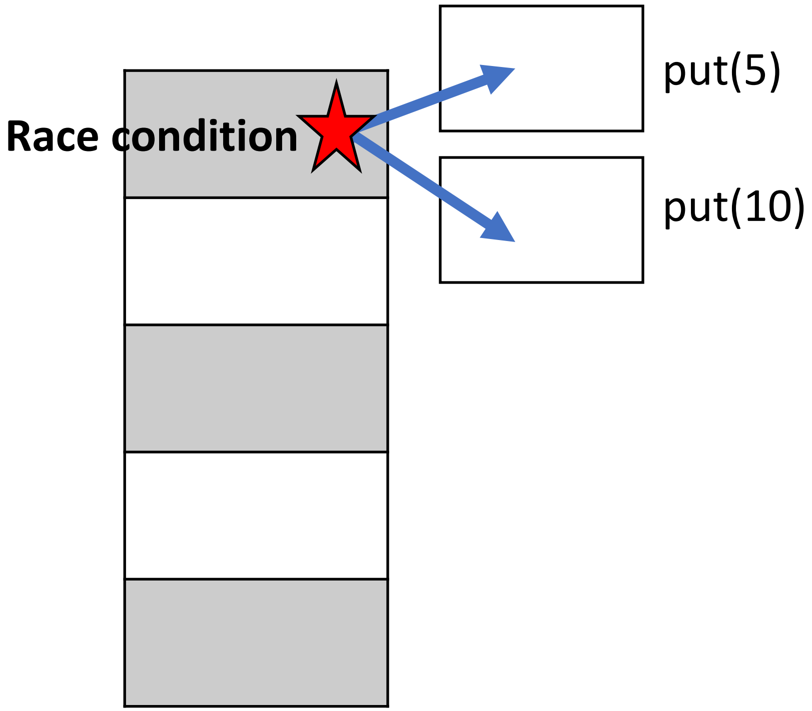
- **ILP wall**: Increasingly difficult to find enough parallelism in instruction stream to keep a powerful single core busy
- **Power wall**: power usage is V^2

ph0.c

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, it's known as a **race condition**



Race condition example

Thread 1: put(5)	Thread 2: put(10)
READ table[0] -> tmp	
	READ table[0] -> tmp
WRITE tmp -> e->next	
	WRITE tmp -> e->next
WRITE e -> table[0]	
	WRITE e -> table[0]

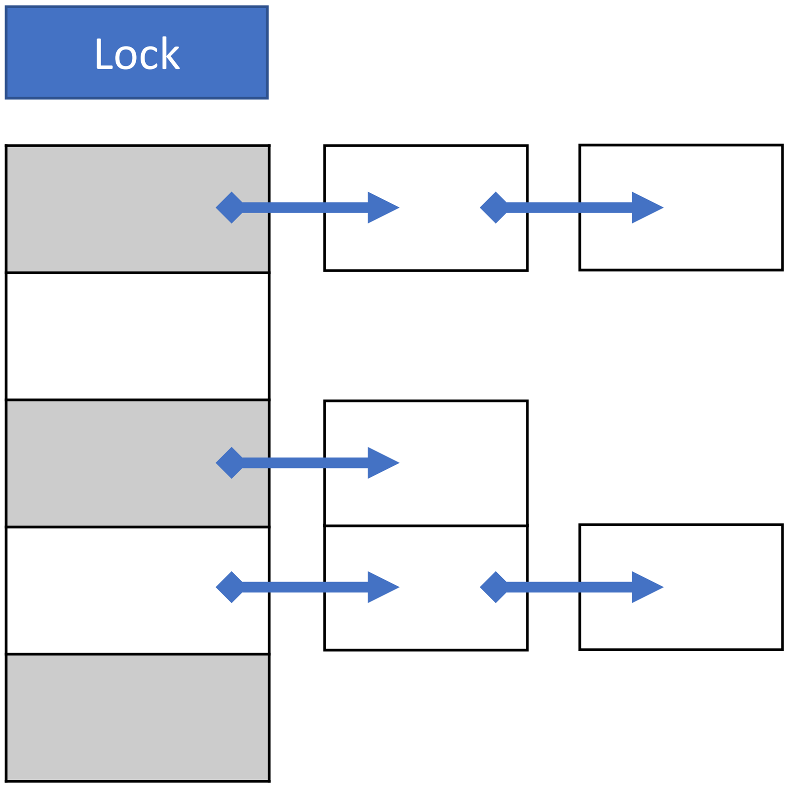
Time

Last writer wins!

ph1.c

Plan: Big lock / coarse-grained synchronization

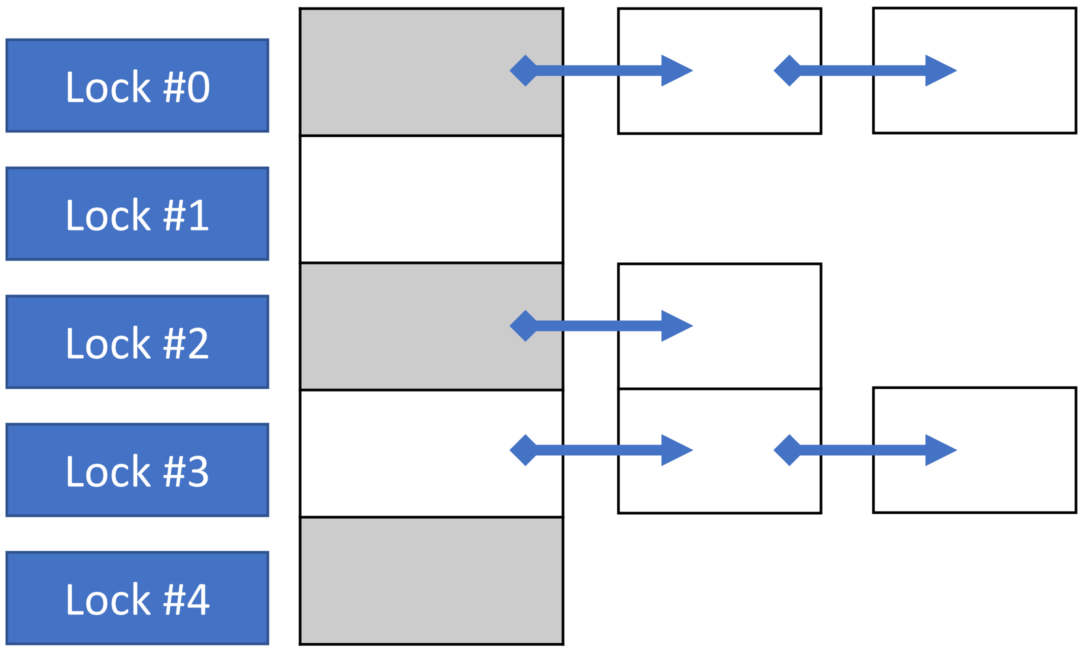
Big lock



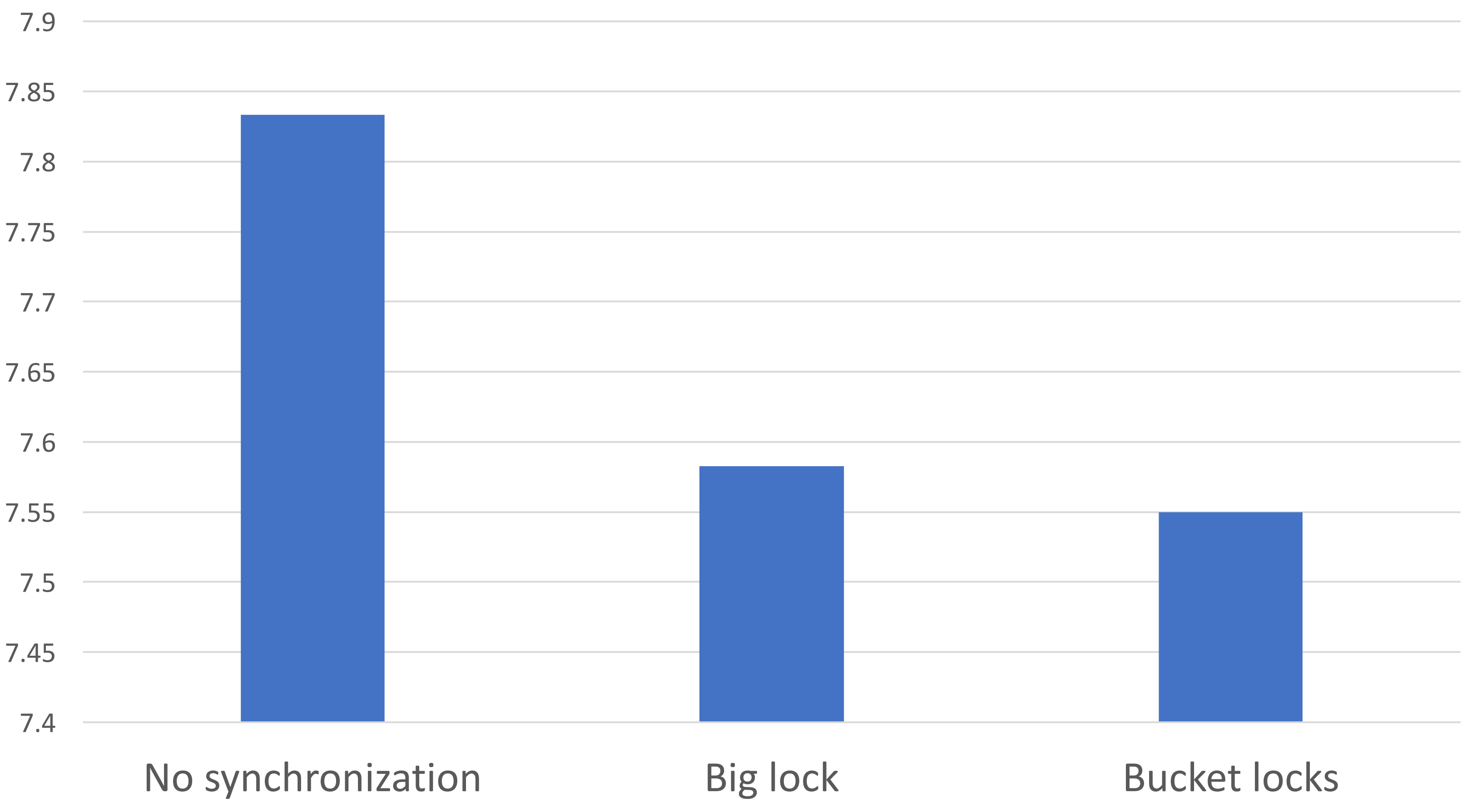
ph2.c

Plan: Bucket locks / fine-grained synchronization

Bucket locks



ph[0-2].c run-time with 4 cores



Concurrent hash table questions

1. Does get() need a lock in ph.c?
2. Does get() need a lock with concurrent puts()?
3. 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);
```

- A lock itself is an object
- Suppose multiple threads call acquire(&l):
  - Only one returns right away
  - The 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 the memory location?

If so, you need a lock!

Too conservative: Sometimes deliberate races are fine!

Too liberal: Think about invariants of entire data structure, not just single memory locations (e.g. console)

Could locking be automatic?

- Idea: The language could associate a lock with every object
  - Compiler adds acquire() and release() around every use
  - No room for programmer to forget!
- Can be awkward in practice
  - E.g. rename(“d1/foo”, ”d2/foo”);
  - Acquire d1; erase foo; release d1
  - Acquire d2; add foo; release d2
  - At one point, foo doesn’t exist at all!
- Programmer needs explicit control to hide intermediate states

Perspectives on what locks achieve

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

Problem: Locks can cause deadlock

What if:

CPU 0:
rename(”a/f”, “b/f”);
acquire(&a);
…
acquire(&b);
…

CPU 1:
Rename(“b/f”, “a/f”);
acquire(&b);
…
acquire(&a);
…

Hangs forever!

Solution to lock deadlocks

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

Reality: There's a 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

Where to place locks?

One strategy:

1. Write the module to be correct under serial execution
2. Then add locks to **force** serial execution

Each locked section can only be executed by one CPU at a time, so you can reason about it as serial code!

What about performance?

Otherwise, run on a single core

Locks prevent parallelism!

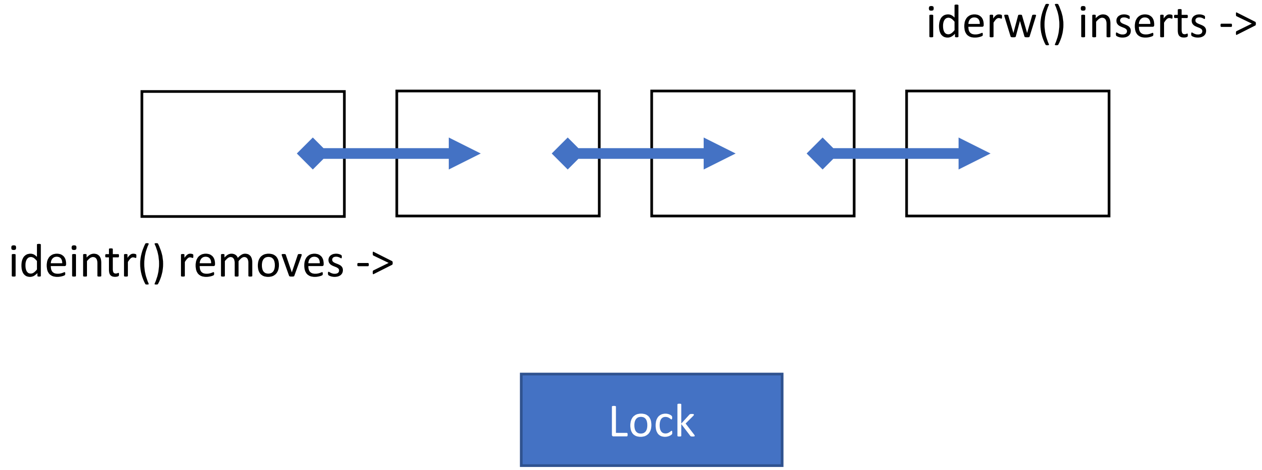
- To maintain parallelism split up data and locks
- Choosing the best split is a design challenge
  - Whole ph.c table, each table[] row, or each entry?
  - Whole FS, each file/directory, or each disk block?
- May need to make design changes to promote parallelism
  - Example: Break single free list into 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 little time is spent in the module
  - Redesign only if you have to

Example: xv6 ide driver

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



How to implement locks?

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

x86 has an atomic exchange instruction

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

Does this in HW:

```
lock addr globally (other cores can't use it)
temp = *addr
*addr = %eax
%eax = temp
unlock addr
```

How to really implement a lock

```
struct lock { int locked; };
acquire(l){
  while(1){
    if(!xchg(&l->locked, 1)) // A and B
      break;
  }
}
```

spinlock.c

xv6 support for locks

Memory ordering

- The compiler and CPU can reorder 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
- For exotic lock-free code, you'll need to know 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 for very short critical sections
  - What about longer critical sections?
- Blocking locks available in most systems
  - Higher overheads 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 automatized tools like race detectors to find locking bugs