6.S081: Locking

Adam Belay <abelay@mit.edu>

Plan for today

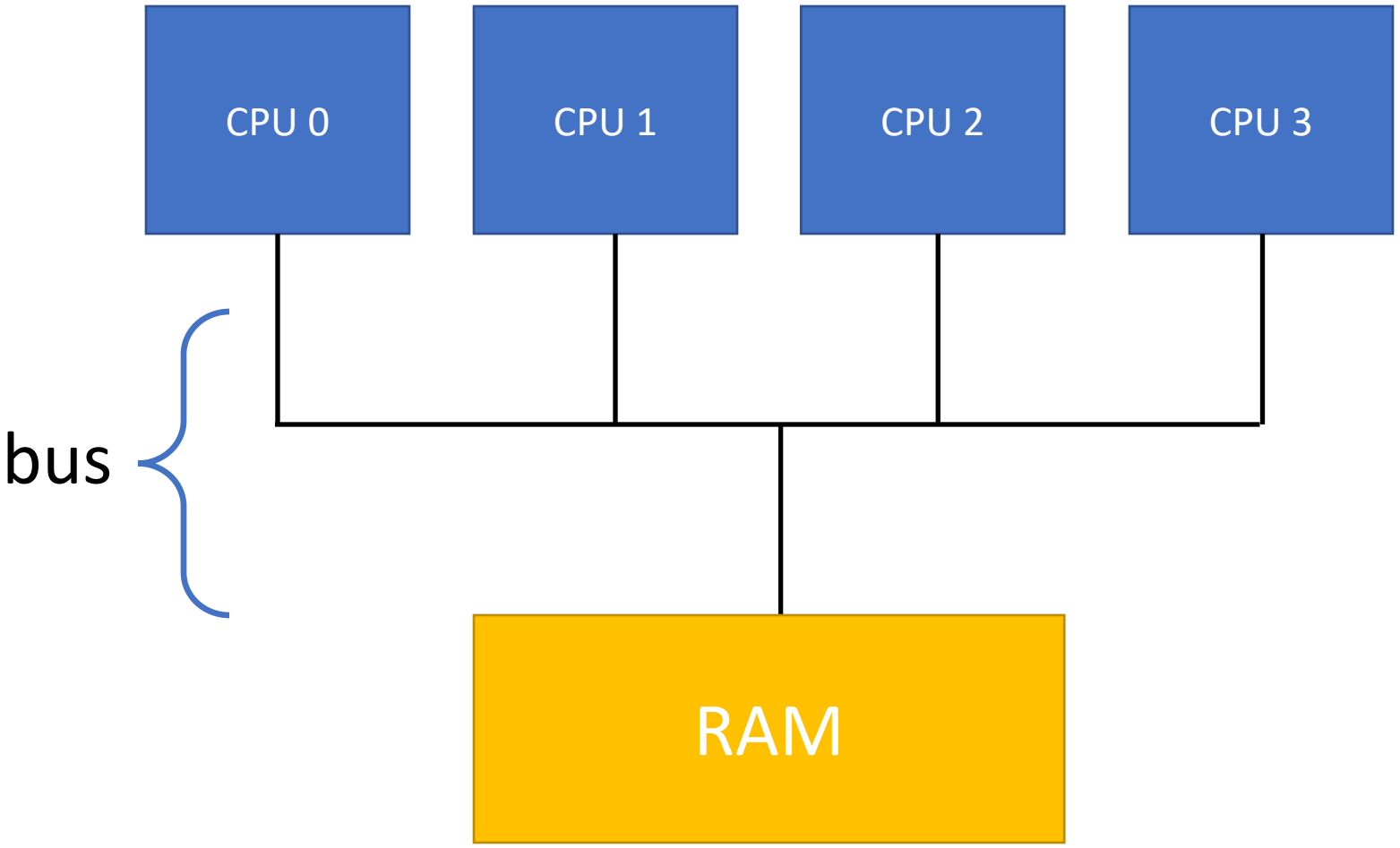
- Examples of code that needs locks
- Lock abstraction + Deadlocks
- Atomic instructions and how to implement locks

Kfree() example in xv6

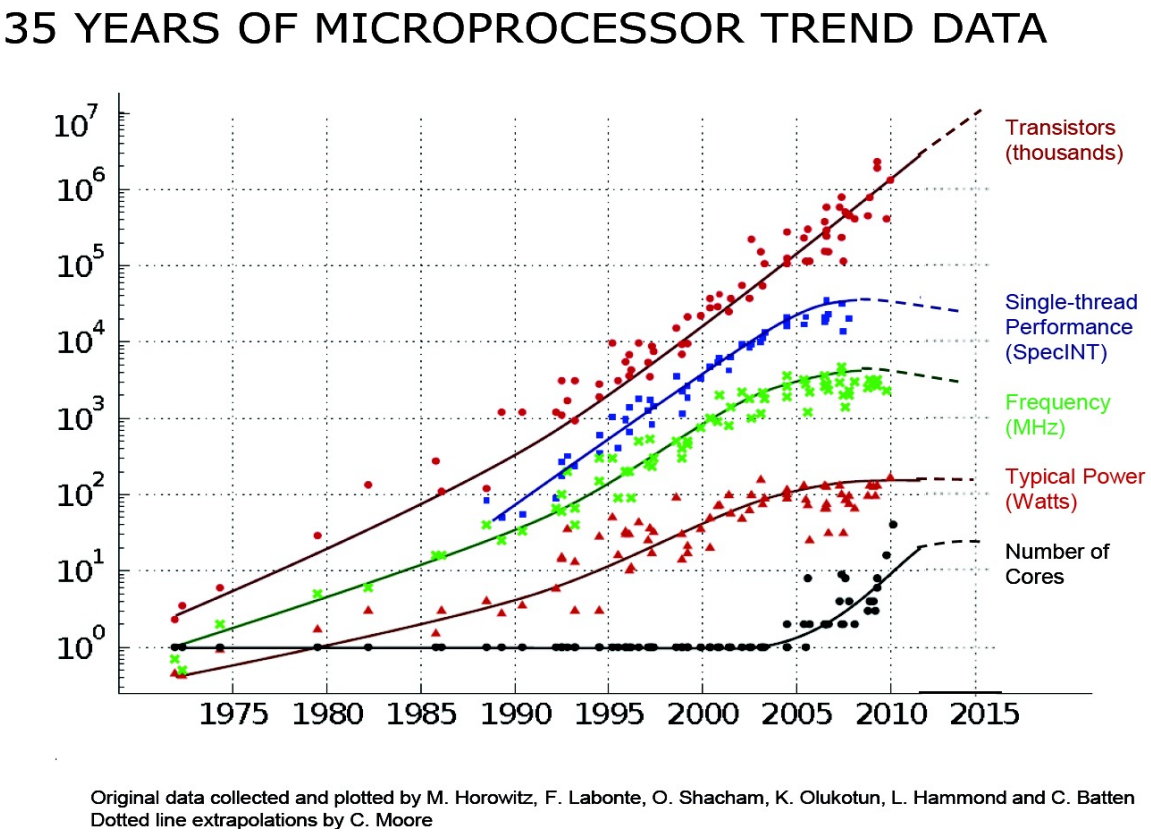
Why locks?

- Apps want to use multiple cores for parallel speedups
  - Kernel must handle parallel system calls too
- Implication: parallel access to datastructures
- Pro: Locks can enforce correct access to data
- Con: Locks limit parallel speedup

Why multiple cores?



Parallelism is unavoidable

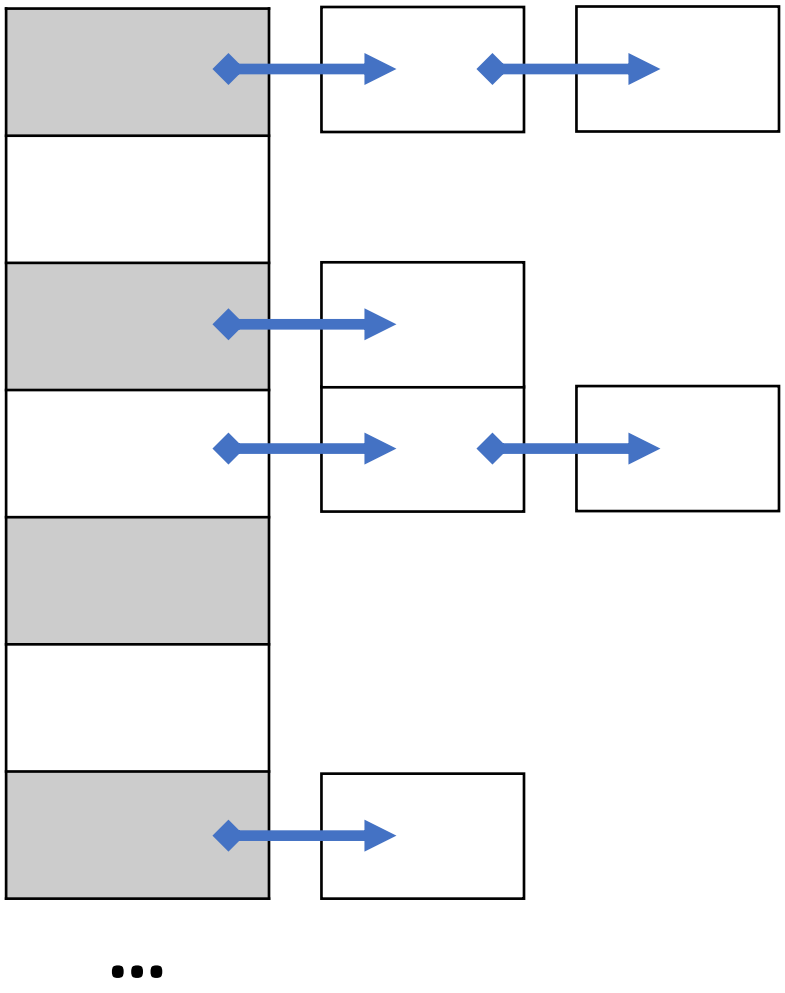


- **ILP wall**: Increasingly difficult to find enough parallelism in instruction stream to keep a powerful single thread busy
- Use multiple hardware threads (harts) instead

Locking example: Hash table

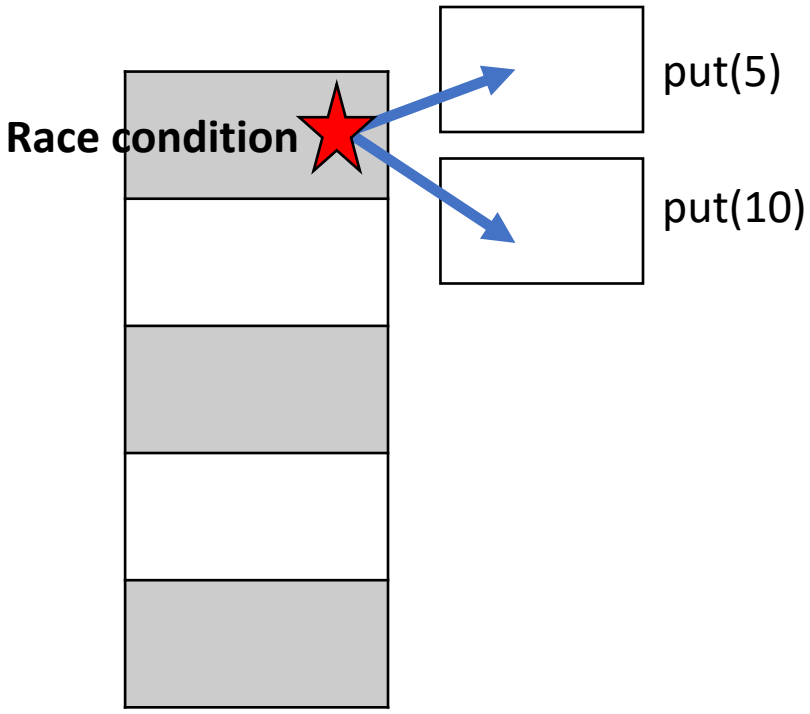
- Parallel operations
- Put() and Get()
- Collisions resolved with chaining

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

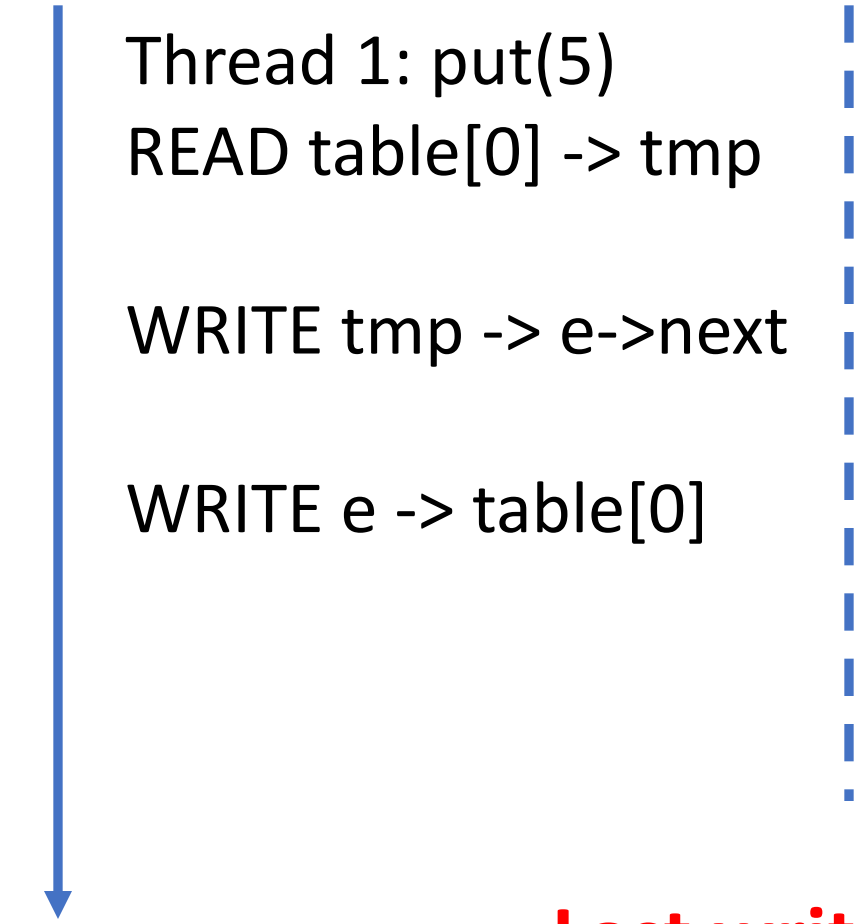


Could keys go missing?

- 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**

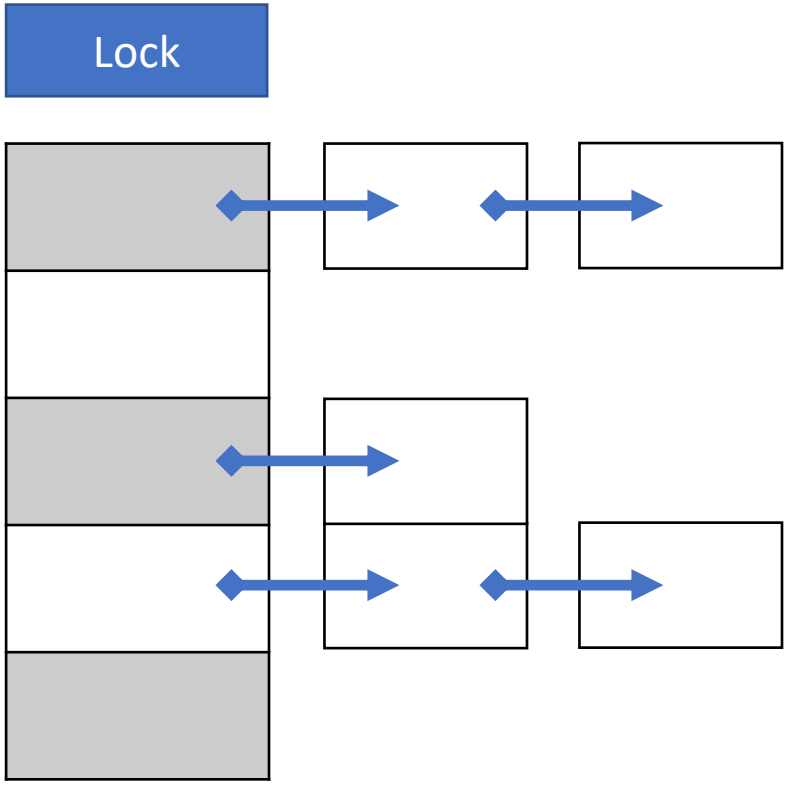


Race condition example

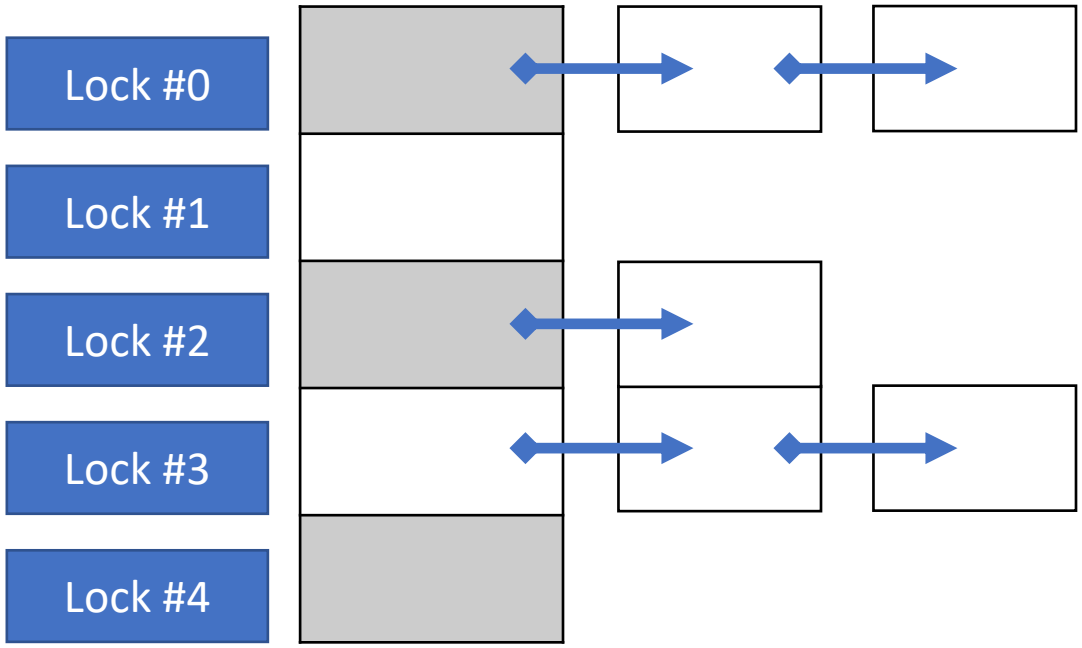


Last writer wins!

Big lock



Bucket locks



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/f1”, “b/f1”);
acquire(&a);
…
acquire(&b);
…

CPU 1:
Rename(“b/f2”, “a/f2”);
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: Top down

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!

Where to place locks?

Another strategy: Bottom up

1. Look at the data (e.g., structs, globals, etc.)
2. Ask yourself, is this data shared?
3. If so, then ask yourself which lock protects access to it?

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

How to implement locks?

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

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"
- If you use locks, you don't have to understand memory ordering
- For exotic lock-free code, you'll need to know every detail

RISC-V Atomic Instructions

- AMO* instructions
  1. v1 = *addr
  2. *addr = OP(v1, v2)
- Supported operations:
  - SWAP, ADD, AND, OR, XOR, MAX, MIN
- Read and write to memory location happens atomically

RISC-V Fences

- **fence** instruction constrains ordering between reads and writes
- **fence**(predecessor, successor): cannot observe any operation in the successor set following a FENCE before any operation in the predecessor set
- Example: FENCE(r, rw)

Special instruction for locks

- Combines ideas from fences and atomics
  - Why did the designers choose this approach?

- **amoswap.w.aq**: no later memory operations can be observed to take place before the swap
- **amoswap.w.rl**: the swap will not be observed before any memory operations that happen before it

See C/C++ acquire and release semantics for a more detailed discussion….

How to really implement a lock

```
 li           t0, 1        # Initialize swap value.
    again:
        amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
        bnez         t0, again    # Retry if held.
        # ...
        # Critical section.
        # ...
        amoswap.w.rl x0, x0, (a0) # Release lock by storing
0.
```

Excerpted from The RISC-V Instruction Set Manual.

spinlock.c

xv6 support for locks

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

Are spin locks the only option?

- No!
  - Don’t share (e.g., per-core data)
  - Deliberate races
  - Read-copy-update (later in this class)
  - Read-write locks
  - Lock-free algorithms (use atomics)
  - Etc.
- Consider taking 6.823
  - Cache coherence
  - Cache consistency
  - Link-load, store-conditional

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