Parallel Programming

Lock- and Wait-Free

Morgan Ericsson

Today

» Lock-free

» Wait-free

» Atomic variables

Locking

```go
var x int
var lock sync.Mutex

// ...

lock.Lock()
complexOp(&x)
lock.Unlock()
```

Issues with locks

- Overhead
- Risk of deadlocks and starvation
- It is difficult to “find” the right granularity
  - too much locking, reduced parallelism
  - not enough locking, races

Issues with locks

- From a software engineering perspective,
  - lock-based does not compose
  - lock-based is hard to modify and maintain

Issues with locks

- This is why we have channels!
- But channels are not always the answer
- So, helpful to know about atomic, lock- and wait-free
  - But use channels when possible!

Lock-free programming

- Can we get rid of (some) locks?
- Lock-based takes a pessimistic perspective
  - If things can go wrong, they will
- Lock-free is optimistic
  - If things go wrong, just try again

Lock-free programming

- Stronger primities for atomic access
- Optimistic algorithms

Atomic variables

- `res := a.compareAndSwap(12, 17)`
  - Sets the value of `a` to `17` if the current value is `12`
  - Atomically
  - Returns true if swap, else false

Atomic variables

- sync/atomic provides a set of atomic types with operations that are guaranteed to be atomic
- E.g., `atomic.Int64`
  - Add
  - Load
  - Store
  - Swap
  - CompareAndSwap

atomic.Int64

```go
var x atomic.Int64

x.Store(12)
x.Add(4)
fmt.Println(x.Load())
```

atomic.Int64

```go
var x atomic.Int64
var ov, nv int64

//...

for {
  ov = x.Load()
  nv = ComplexOp(ov)
  if x.CompareAndSwap(ov, nv) {
    break
  }
}
```

Note

- All of these involve some kind of compare and set (CAS) operation
- Not free, involves memory barriers to, e.g., synchronize caches

Non-blocking

- Lock-free algorithms guarantee system-wide progress
- “infinitely often, some process makes progress”

Non-blocking

- Wait-free algorithms guarantee per-process progress
- “Every process eventually makes progress”

Non-blocking

» Wait-free is stronger than lock-free

» Lock-free algorithms are free from deadlocks

» Wait-free algorithms are free from deadlocks and starvation

Example: Linked list

```
type LL struct {
    head, tail *LLNode
}
```

Example: Linked list

```go
func NewLL() LL {
	head := LLNode{key: math.MinInt}
	tail := LLNode{key: math.MaxInt}
	head.nxt = &tail

	return LL{head: &head, tail: &tail}
}
```

Example: Linked list

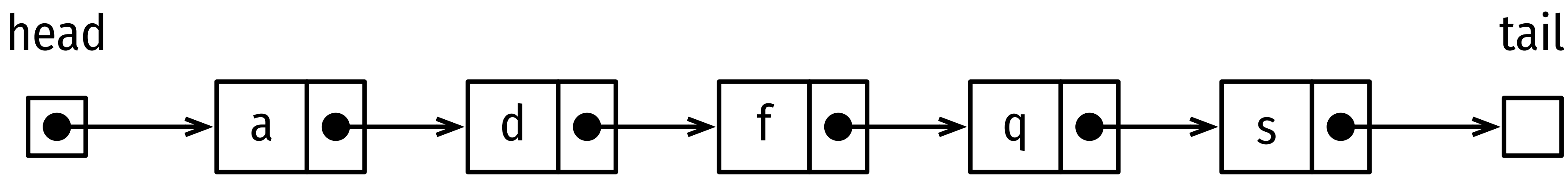
```go
func NewLL() LL {
	head := LLNode{key: math.MinInt}
	tail := LLNode{key: math.MaxInt}
	head.nxt = &tail

	return LL{head: &head, tail: &tail}
}
```

Example: Linked list

- We use a linked list to represent a set
- The list is sorted using the key’s value
- We use the min key in the head and the max key in the tail

Example

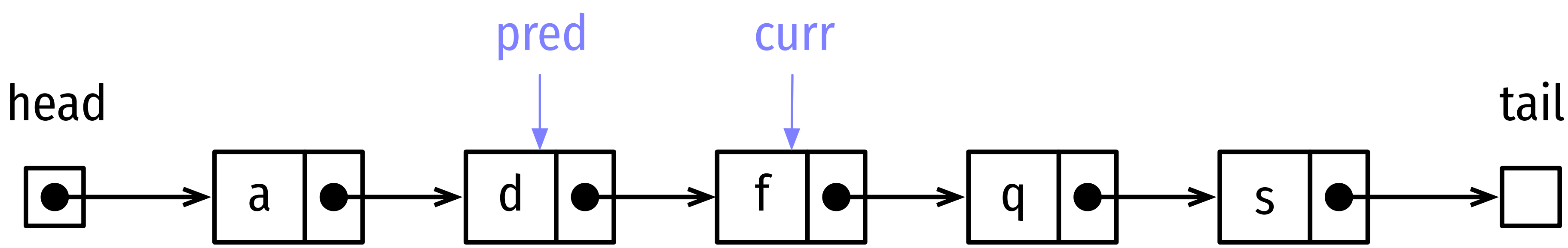


Representation of the set { a, s, d, f, q }

Find

```
func (lst LL) find(k int) (pred *LLNode, curr *LLNode) {
    curr = lst.head
    for {
        pred = curr
        curr = curr.nxt
        if curr.key >= k {
            break
        }
    }
    return pred, curr
}
```

Find



Contains

```go
func (lst LL) Contains(k int) bool {
    _, curr := lst.find(k)

    return k == curr.key
}
```

Add

```
func (lst LL) Add(k int) bool {
    pred, curr := lst.find(k)
    if curr.key == k {
        return false
    } else {
        n := LLNode{key: k, nxt: curr}
        pred.nxt = &n
        return true
    }
}
```

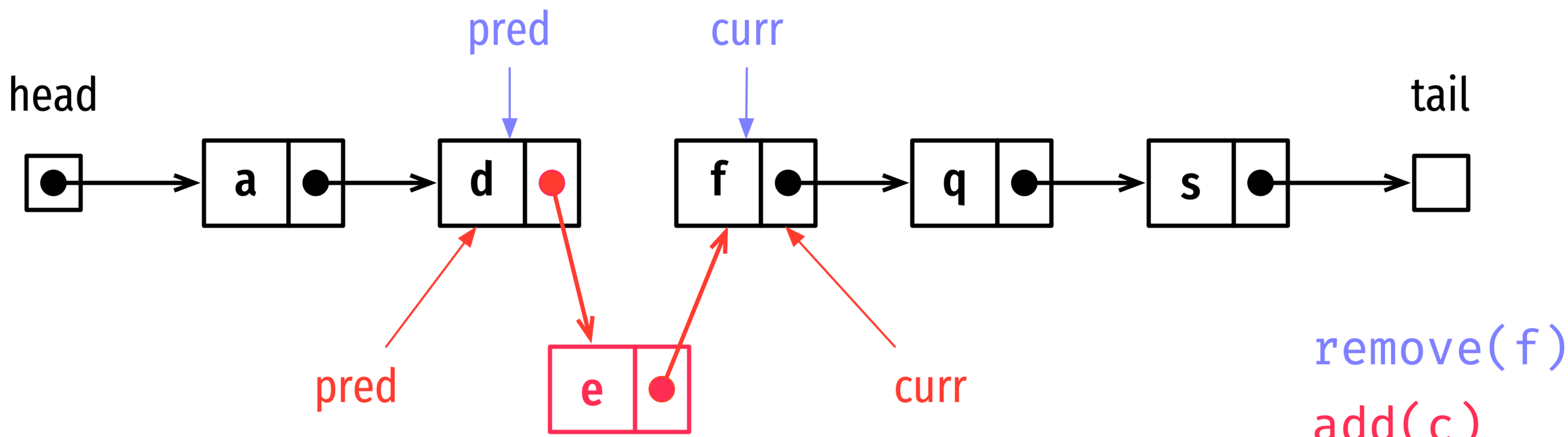
Remove

```
func (lst LL) Remove(k int) bool {
    pred, curr := lst.find(k)
    if curr.key > k {
        return false
    } else {
        pred.nxt = curr.nxt
        return true
    }
}
```

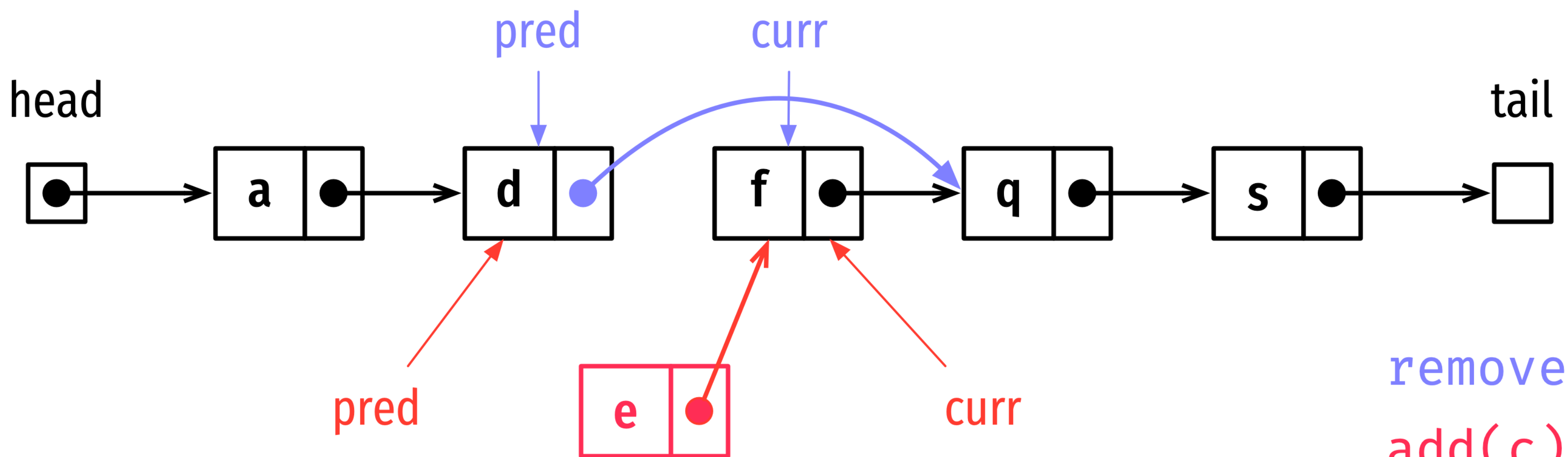
Linked list is sequential

- Linked list is designed for sequential access
- Will not work with multiple goroutines
- But there is an easy fix...

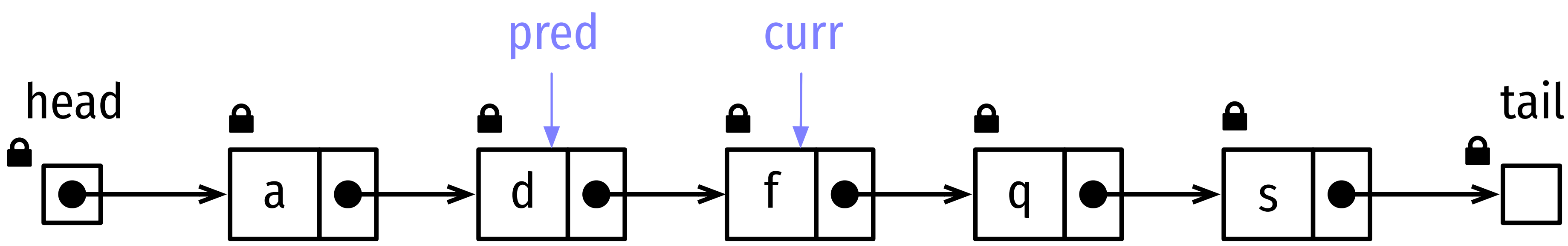
Sequential



Sequential



Coarse locking



Coarse locking

```go
type CoarseLL struct {
    head, tail *LLNode
  lock sync.Mutex
}
```

Add

```
func (lst CoarseLL) Add(k int) bool {
  lst.lock.Lock()
  defer lst.lock.Unlock()
    pred, curr := lst.find(k)
    if curr.key == k {
        return false
    } else {
        n := LLNode{key: k, nxt: curr}
        pred.nxt = &n
        return true
    }
}
```

Contains

```go
func (lst CoarseLL) Contains(k int) bool {
  lst.lock.Lock()
    _, curr := lst.find(k)
  lst.lock.Unlock()

    return k == curr.key
}
```

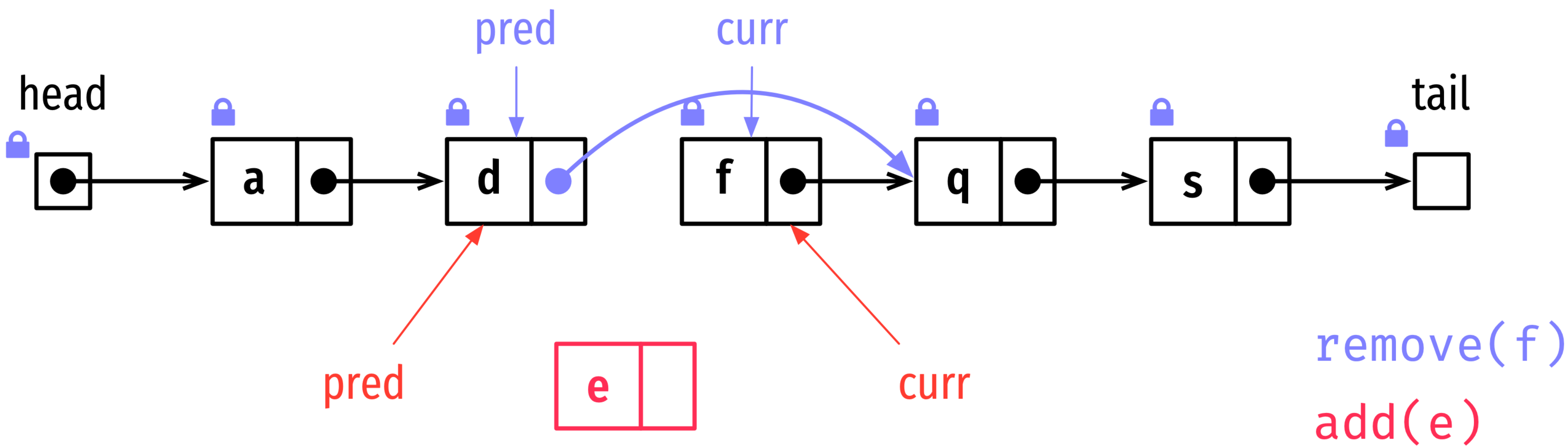
Others

- Should also fix, e.g., Remove
- But do *NOT* add locking to find!

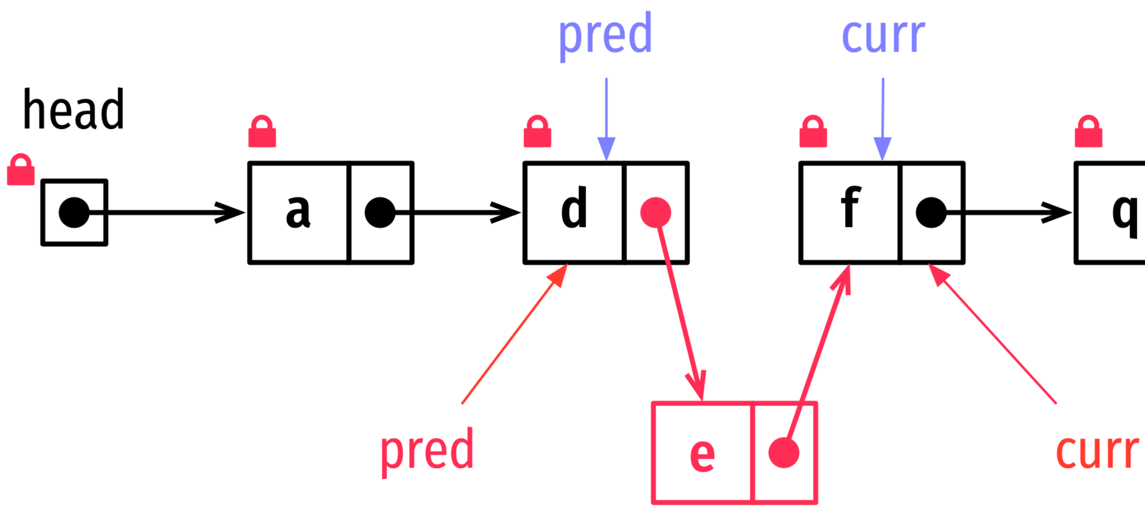
Good?

- It avoids race conditions and deadlocks, so correct
- If the locks are fair, so is the set

Safe



Safe



`remove(f)`
`add(c)`

Good?

- Access is essentially sequential
- We could parallelize more
- If many threads are competing for access, our set implementation can be quite slow

Readers-writers

- Can we use RW locks to improve performance
- Left as an exercise

Locks and find

- All operations use find to find the position
- Can we lock after we find?
- Would reduce the size of the critical section.

No!

- The list can be modified between when a thread calls find and then accuires a lock
- But we do not need to lock the entire set

LLFine

```
type LLFine struct {
	head, tail *LLLockNode
}
```

LLLockNode

```
type LLLockNode struct {
	lock sync.Mutex
	key int
	nxt *LLLockNode
}
```

LLFine

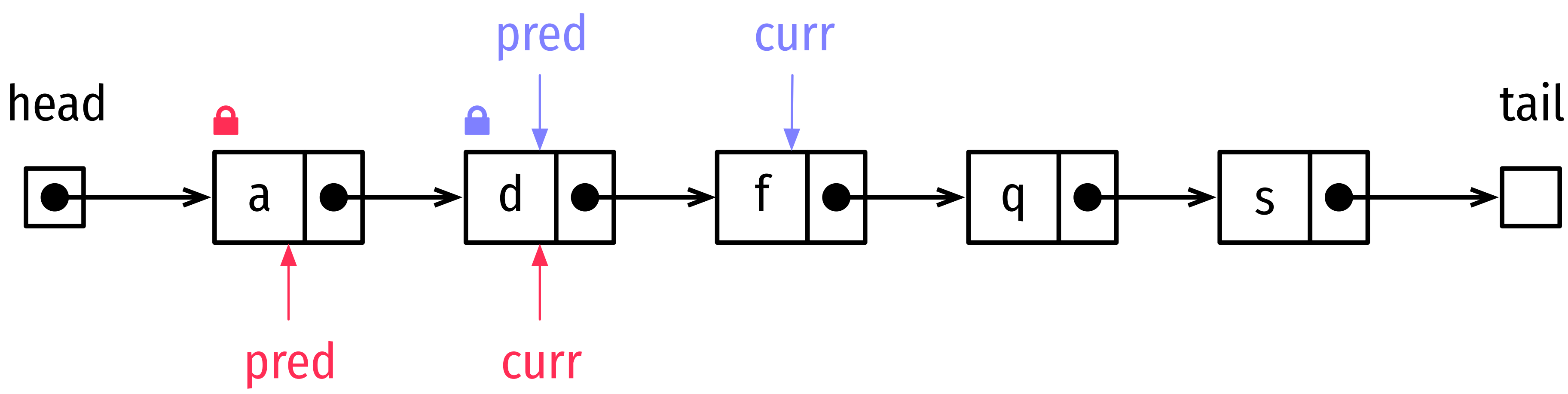
```
func NewLLFine() LLFine {
	head := LLFineNode{key: math.MinInt}
	tail := LLFineNode{key: math.MaxInt}
	head.nxt = &tail

	return LL{head: &head, tail: &tail}
}
```

How many should we lock?

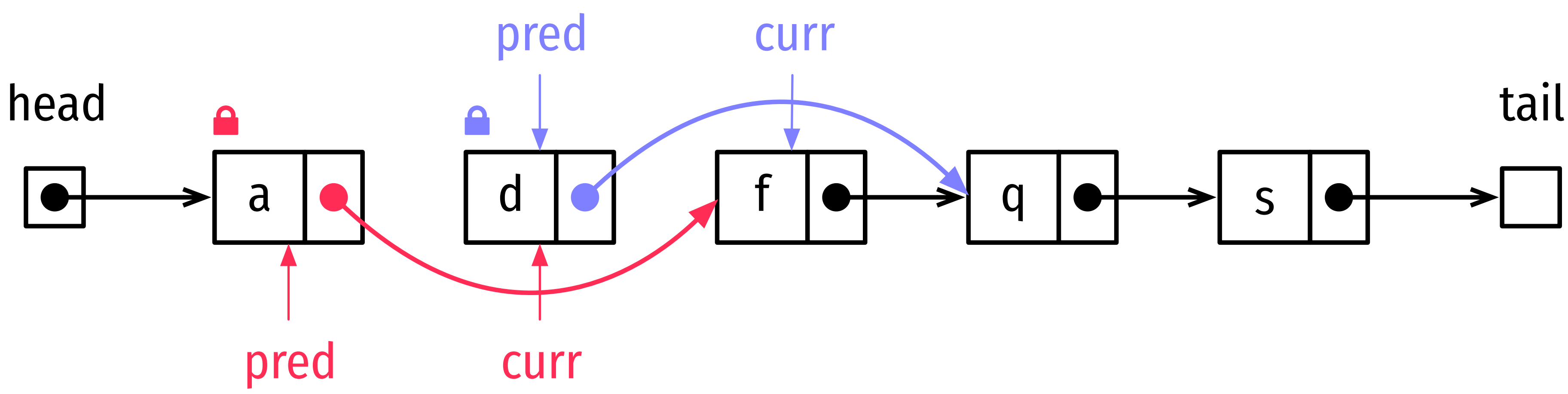
- We must lock as soon as we call find
- How many nodes do we need to lock?
- One? No, not enough.
- We need two!

Locking one



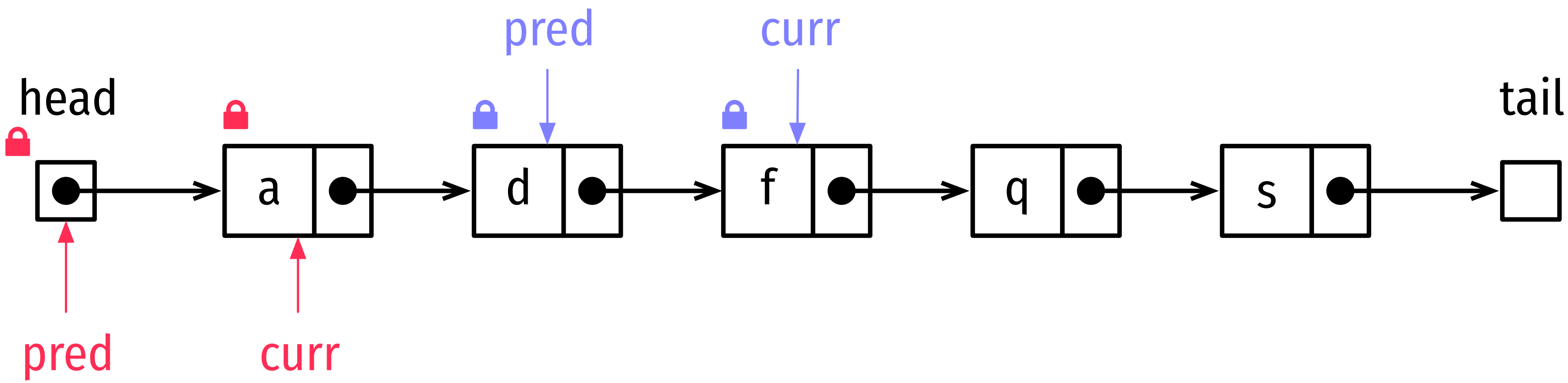
Locking pred

Locking one



Locking pred

Locking two



Locking pred and curr

Find

```
func (lst LLFine) find(k int) (pred *LLFineNode, curr *LLFineNode) {
    pred = lst.head
    curr = pred.nxt
    pred.lock.Lock()
    curr.lock.Lock()
    for curr.key < k {
        pred.lock.Unlock()
        pred = curr
        curr = curr.nxt
        curr.lock.Lock()

    }
    return pred, curr
}
```

Hand over hand locking

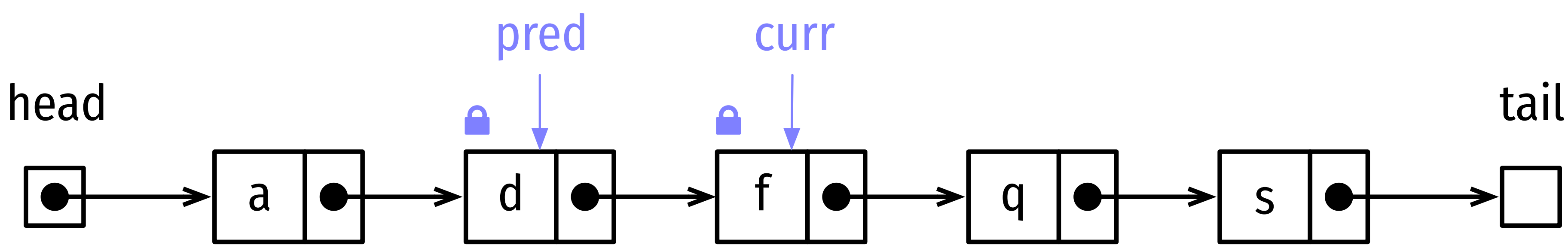
- This strategy to lock is called hand-over-hand locking
  - or lock coupling
- At least one node is always locked
- This prevents interference between goroutines

Hand over hand locking

» Locking two nodes at once prevents goroutines from “overtaking” each other

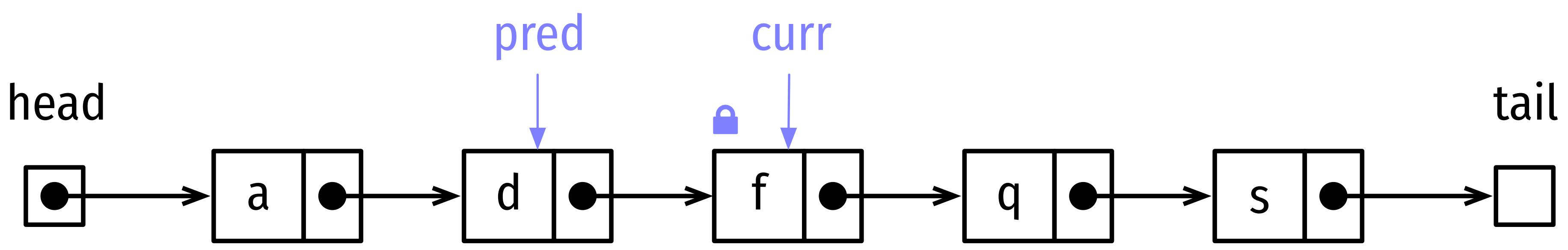
» This helps avoid problems with conflicting operations

Hand over hand locking



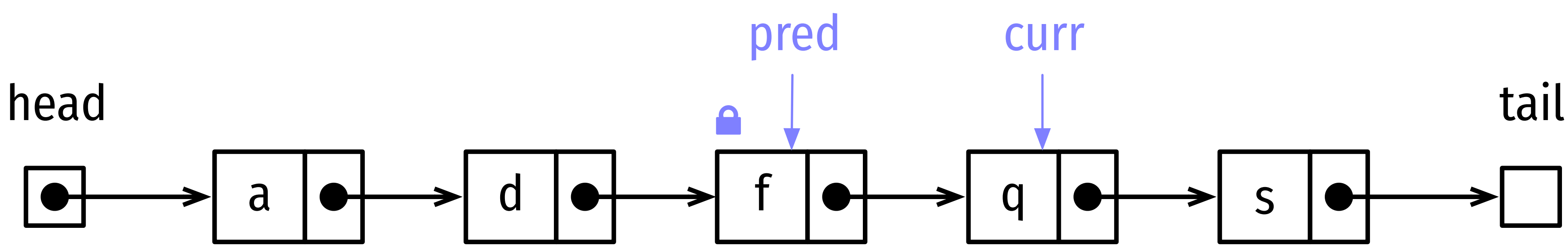
Hand over hand locking

Hand over hand locking



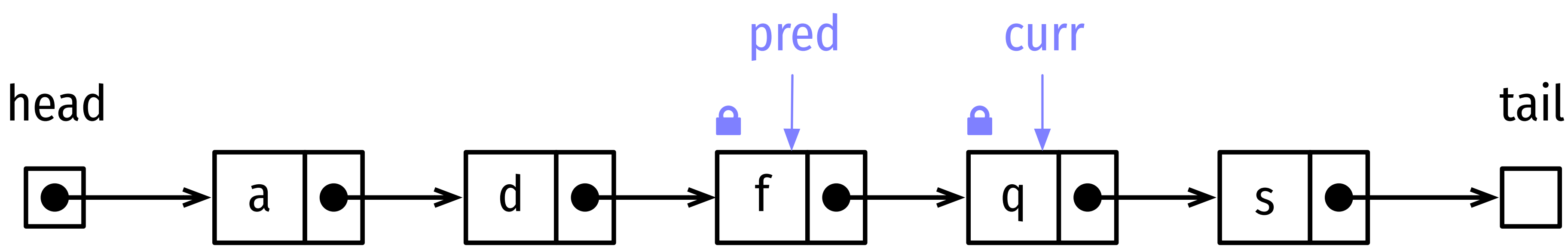
Hand over hand locking

Hand over hand locking



Hand over hand locking

Hand over hand locking



Hand over hand locking

Add

```
func (lst LLFine) Add(k int) bool {
    pred, curr := lst.find(k)
    defer pred.lock.Unlock()
    defer curr.lock.Unlock()
    if curr.key == k {
        return false
    } else {
        n := LLFineNode{key: k, nxt: curr}
        pred.nxt = &n
        return true
    }
}
```

Remove and contains

- Uses the same pattern as add,
- just replace the contents of the try

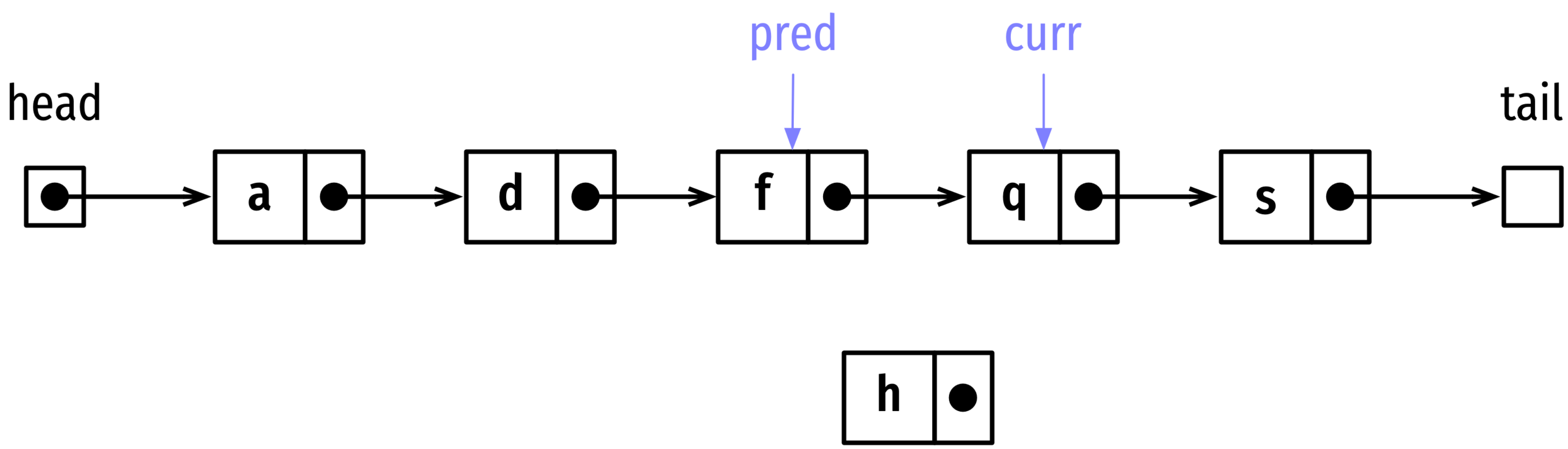
Better?

- Threads that operation on different parts of the list may be able to operate in parallel
- But threads can still block each other (no overtaking)
- Can be quite slow, many lock operations

Optimistic locking

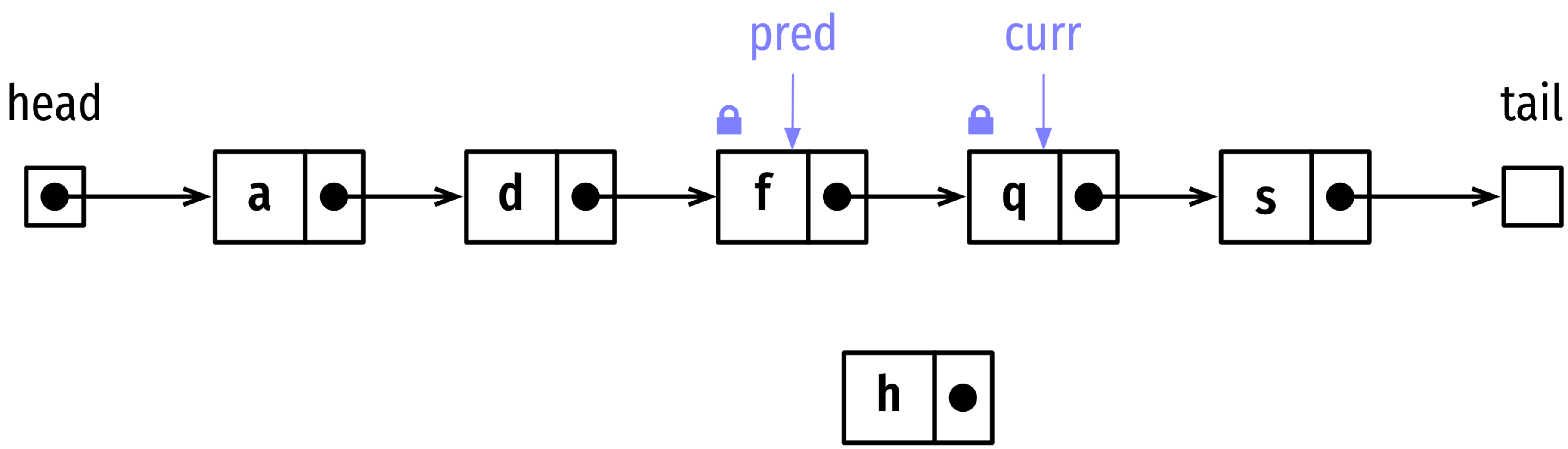
- Is there really no way to safely do find without locking?
- Think back to our previous discussion on optimistic vs pessimistic
- So, validate after finding?

Optimistic locking



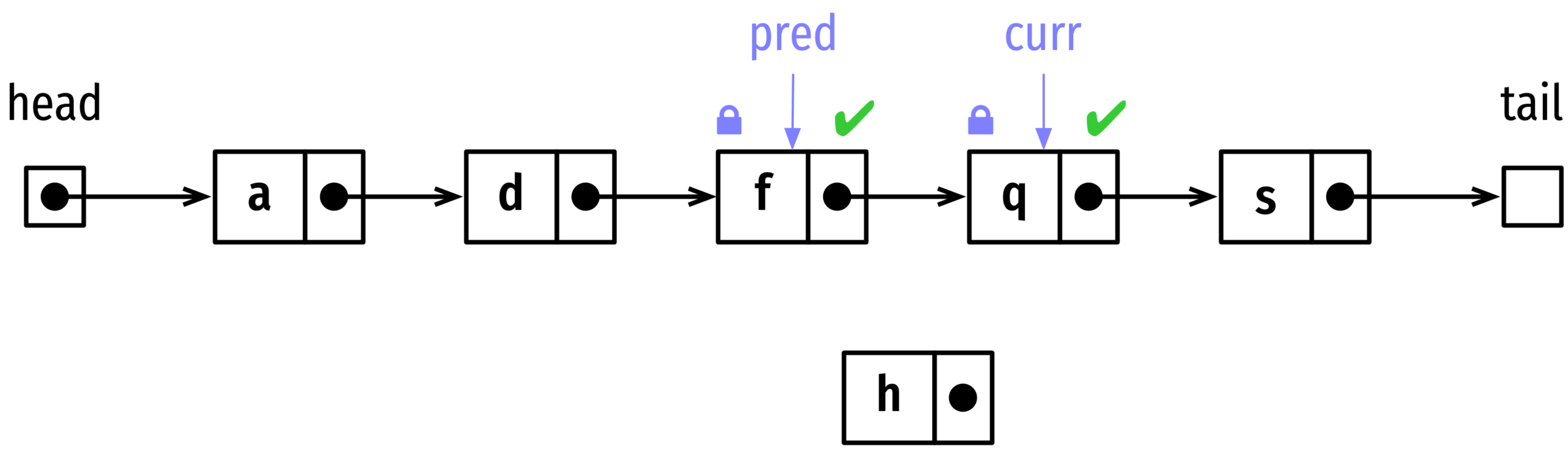
Optimistic locking: find

Optimistic locking



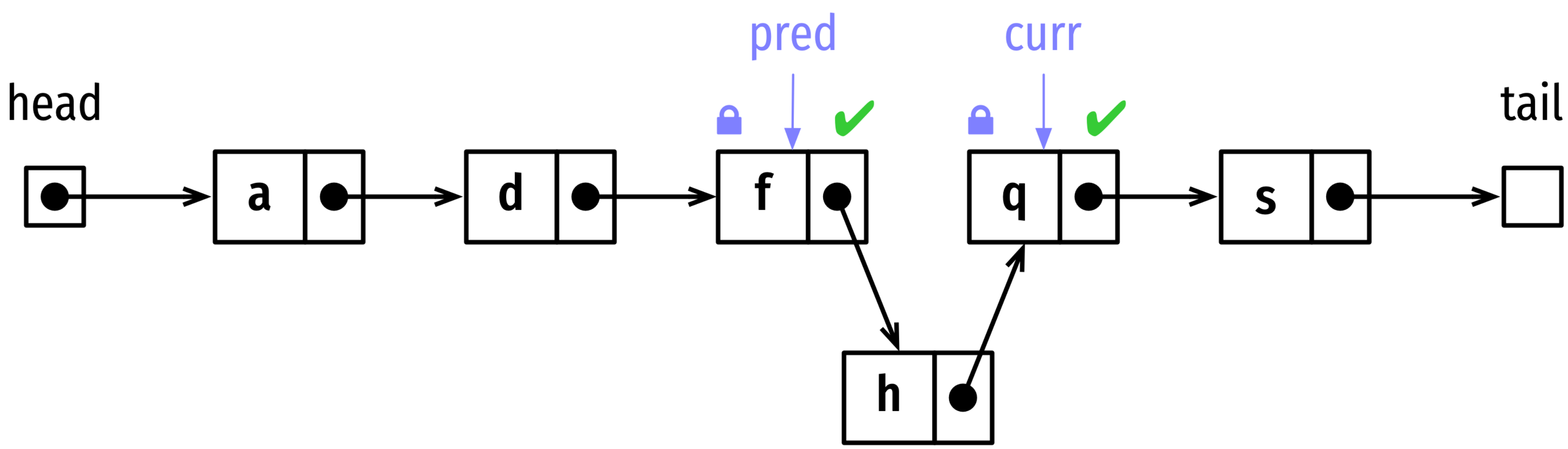
Optimistic locking: lock

Optimistic locking



Optimistic locking: validate

Optimistic locking



Optimistic locking: add

Aside: Java Memory Model (JMM)

- Defines how memory should work in Java
- General rule, sequential consistency
- Yes, Java, but there is a point!

Remember sequential consistency

```python
a = 5
b = 7
c = 9
d = 11
e = 13

a = b * 2 + 1
c = d // 3 * 2
a += 7

print(f'{a=}, {c=}')
```

```
a=22, c=6
```

Optimization and instruction

- e is not used, so it can be thrown away
- a and c can be computed in any order, there are no dependencies
- a is assigned twice, keep in register or cache

> Compilers and CPUs use many tricks to increase performance!

Problematic in MP systems!

```go
var a, b int

func f() {
    a = 1
    b = 2
}

func g() {
    print(b)
    print(a)
}

func main() {
    go f()
    g()
}
```

JMM again

- Defined in terms of actions
  - E.g., reads, writes, locks, unlocks, …
- A partial ordering, happens-before, on all actions within a program
- Each action in a thread happens-before every action in that thread that comes later in the program order
- An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock

Problematic in MP systems!

```go
var a, b int

func f() {
    a = 1
    b = 2
}

func g() {
    print(b)
    print(a)
}

func main() {
    go f()
    g()
}
```

JMM again

» A write to a volatile field happens-before every subsequent read of that same field

» Reads and writes of atomic variables have the same memory semantics as volatile variables

» Synchronization actions are totally ordered

» Go atomic variables work like Java volatile

Atomic pointers

```go
type LLANode struct {
	lock sync.Mutex
	key int
	nxt atomic.Pointer[LLANode]
}
```

Atomic pointers

```go
type LLFine struct {
    head, tail atomic.Pointer[LLANode]
}
```

Optimistic set

So, we should

1. Find the position
2. Lock pred and curr
3. Validate while locked
    1. If valid, perform operation
    2. If invalid, repeat from #1

Works well if validation is successful most of the time

Add

```go
func (lst LLFine) Add(k int) bool {
	for {
		pred, curr := lst.find(k)
		pred.lock.Lock(); curr.lock.Lock()

		if lst.valid(pred, curr) {
			n := LLANode{key: k}
			n.nxt.Store(curr)
			pred.nxt.Store(&n)
			pred.lock.Unlock();curr.lock.Unlock()
			return true
		}
		pred.lock.Unlock(); curr.lock.Unlock()
	}
}
```

Validate

```go
func (lst LLFine) valid(pred, curr *LLANode) bool {
    n := lst.head.Load()
    for n.key <= pred.key {
        if n == pred {
            return pred.nxt.Load() == curr
        }
        n = n.nxt.Load()
    }
    return false
}
```

Validate is safe

- Fails if pred or curr is removed
- Fails if a node is added between pred and curr
- Succeeds in all other cases
- Also when modified during validation
    - Since can only modify before or after

Better?

- Threads working on different parts of the list can still operate in parallel
- If validate succeeds most of the time, much less locking
- For any benefit, the iteration without locking must be significantly faster than with locking
  - We might have to do it multiple times

Danger

- Optimistic set suffers from starvation
- There is no guarantee that the operation will succeed, ever...
- Imagine that the nodes involved keep being changed by other threads
  - Stuck in a find/validate loop

Can we do better?

- In many applications, contains is the most commonly used operation
- With previous approaches, all methods require the same locking
- Can we make contains lock-free? If so, is there any benefit?

Lock-free contains

- Remove will interfere with a lock-free contains
- If contains does not try to lock, it will not know if a node is being removed
- Can we make remove atomic?
  - Or at least an indication of remove?

A “lazy” node type

```go
type LLANode struct {
  valid atomic.Bool
    lock atomic.Mutex
    key int
    nxt atomic.Pointer[LLANode]
}
```

New valid

```go
func (lst LLLazy) valid(pred, curr *LLLANode) bool {
  p := pred.Load()
  c := curr.Load()

  return p.valid && c.valid && p.nxt == c
}
```

New valid

- Constant time
- Add same as in previous implementation, but uses new valid

New contains

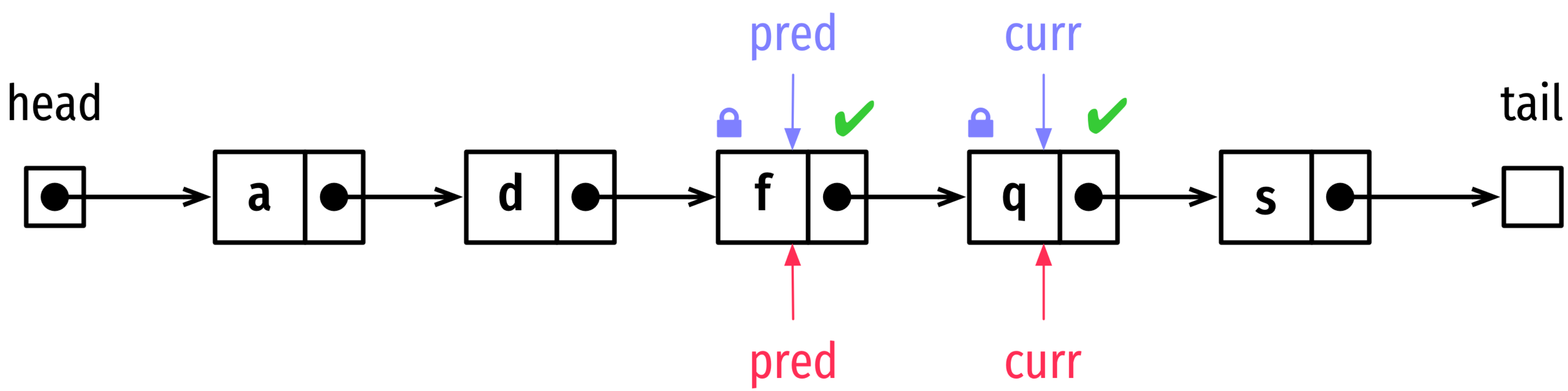
```
func (lst LazyLL) Contains(k int) bool {
    pred, curr := lst.find(k)
  p := pred.Load()
  c := curr.Load()

    return p.valid && c.key == k
}
```

Remove

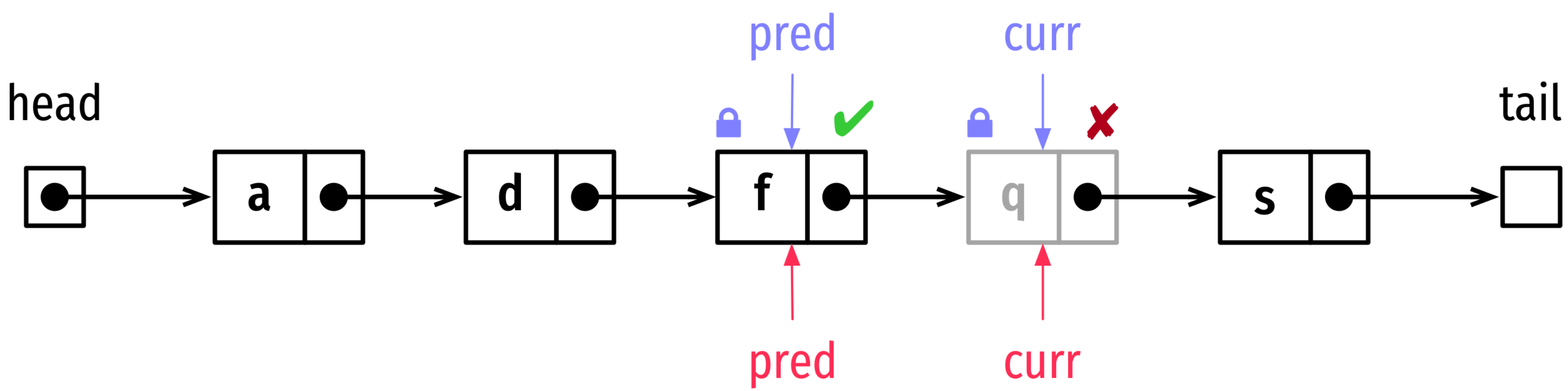
- Now in two steps
  1. Mark the node to be removed as invalid
  2. Update the list to skip over the node to be removed
- Lazy because 1 and 2 can be done at different times
- Logical remove (#1) is sufficient for a node not to be considered by, e.g., contains

Remove



Lazy locking

Remove



Lazy locking

Remove

```go
func (lst LLLazy) Remove(k int) bool {
    for {
        pred, curr := lst.find(k)
        pred.lock.Lock(); curr.lock.Lock() // Why not defer unlock here?

        if lst.valid(pred, curr) {
            if curr.key != k {
                pred.lock.Unlock();curr.lock.Unlock()
                return false
            } else {
                curr.valid.Store(false)
                pred.nxt.Store(curr.nxt.Load())
                pred.lock.Unlock();curr.lock.Unlock()
                return true
            }
        }
    pred.lock.Unlock();curr.lock.Unlock()
    }
}
```

Better?

- Validation in constant time
- Contains is now wait-free, can traverse list once without any locking
- Physical removal can be batched (similar to garbage collection)
- Add and remove still requires locking

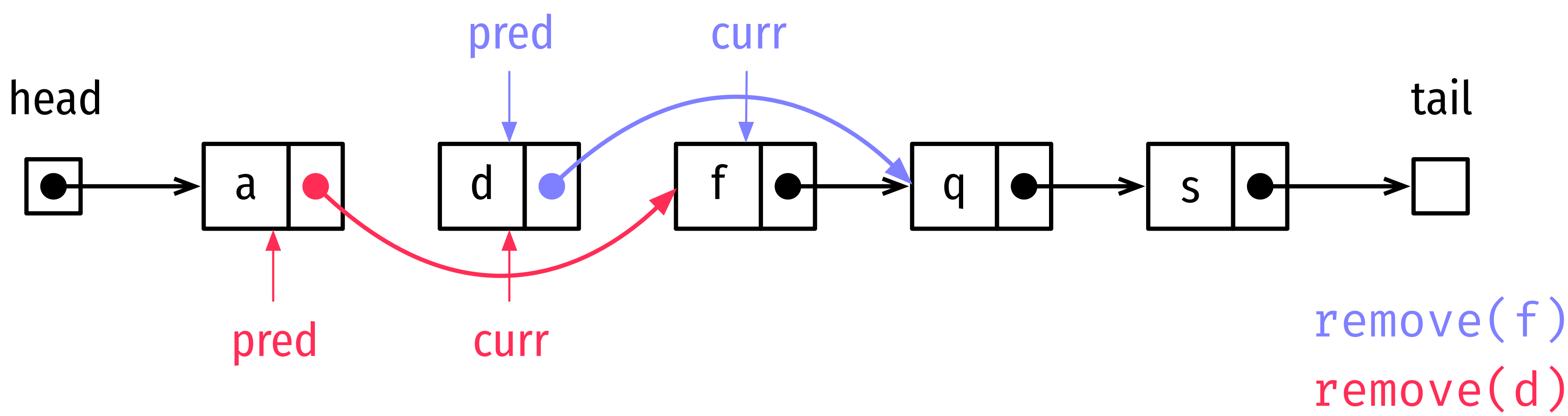
What about atomics?

- Would remove the need for locks
- CAS on atomic.Pointer
- Can we use it for `nxt`?
- Repeat if CAS fails

AtomicReference

```
func (lst LLLazy) Remove(k int) bool {
  var done bool
    for !done {
        pred, curr := lst.find(k)
    if curr.key >= k {
      return false
    } else {
      done = pred.nxt.CompareAndSwap(pred.nxt, curr.nxt)
    }
    }
  return true
}
```

AtomicReference



Using an AtomicReference for nxt is not enough. Can miss removals.

Atomic reference

- Our first, naive attempt does not work
- No surprise, we know that modifications need to control both pred and curr

Second try: Lazy

- We used a valid/invalid flag in the lazy set
- We can reuse this approach
- But we need atomic access to both the flag and nxt
- Atomic values
- Left as an exercise

Next time

» Parallel algorithms