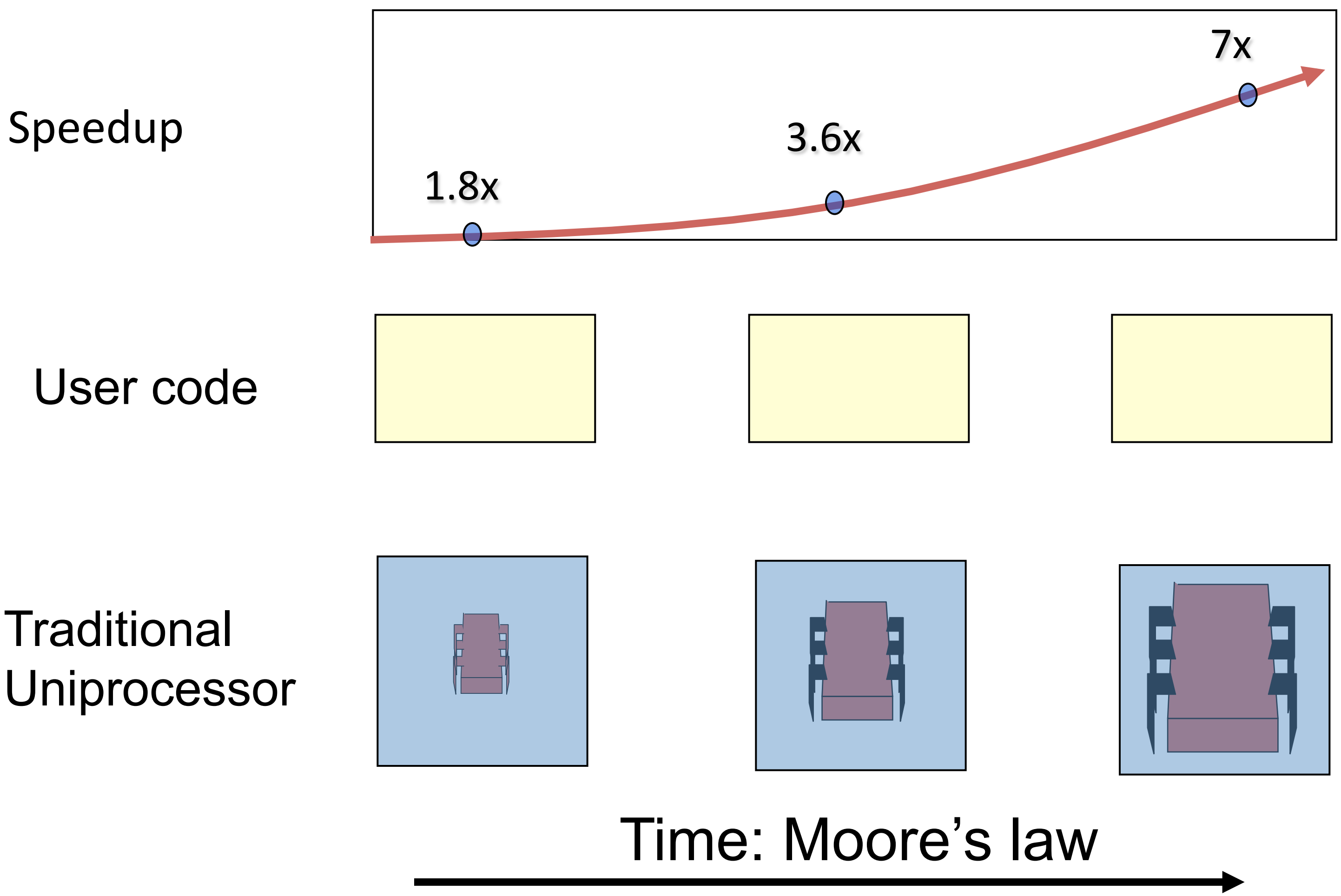
Transactional Memory

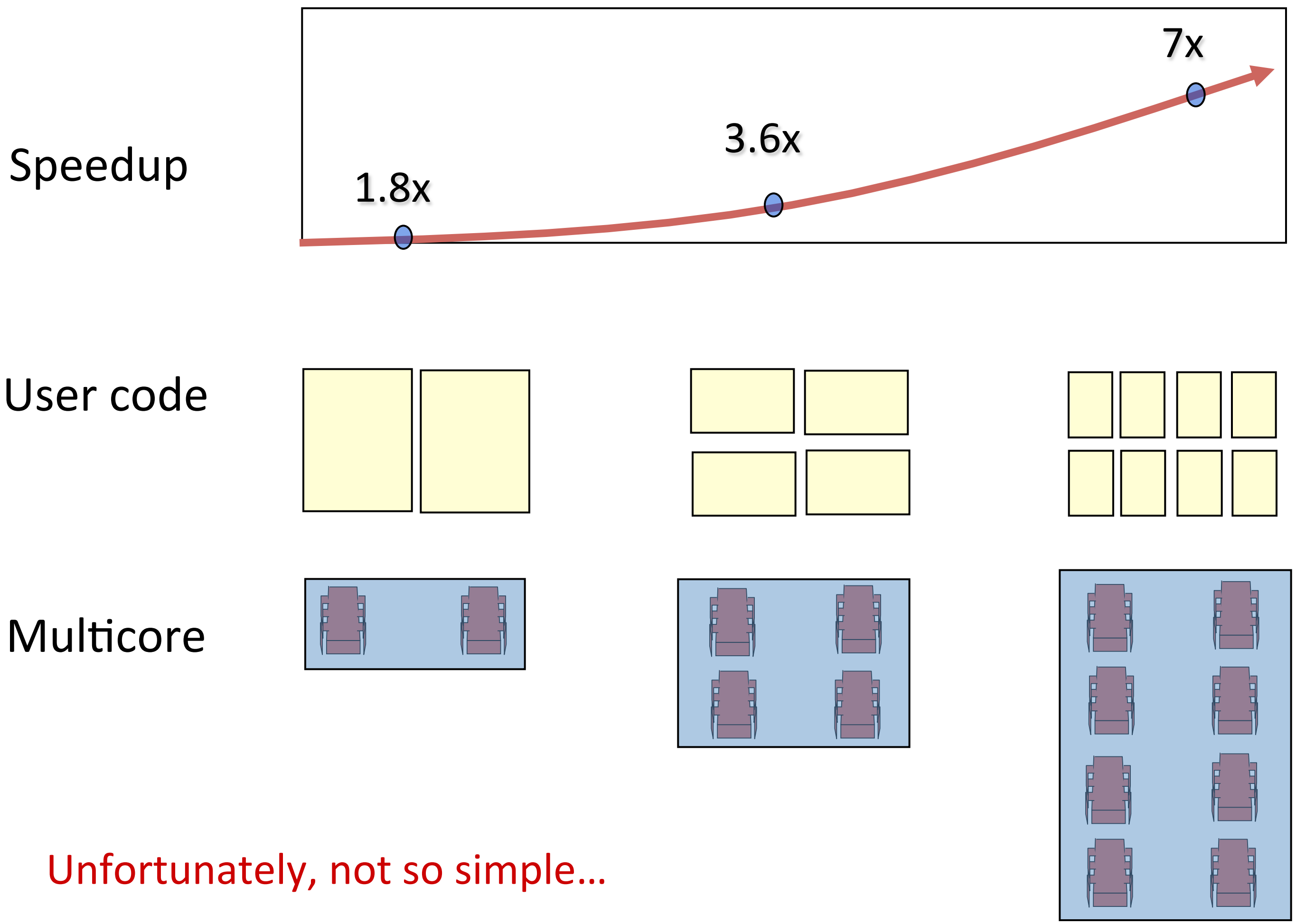
Making Practical

Alexander Matveev
Prof. Nir Shavit
Prof. Yehuda Afek
Tel Aviv University and MIT

Traditional Software Scaling

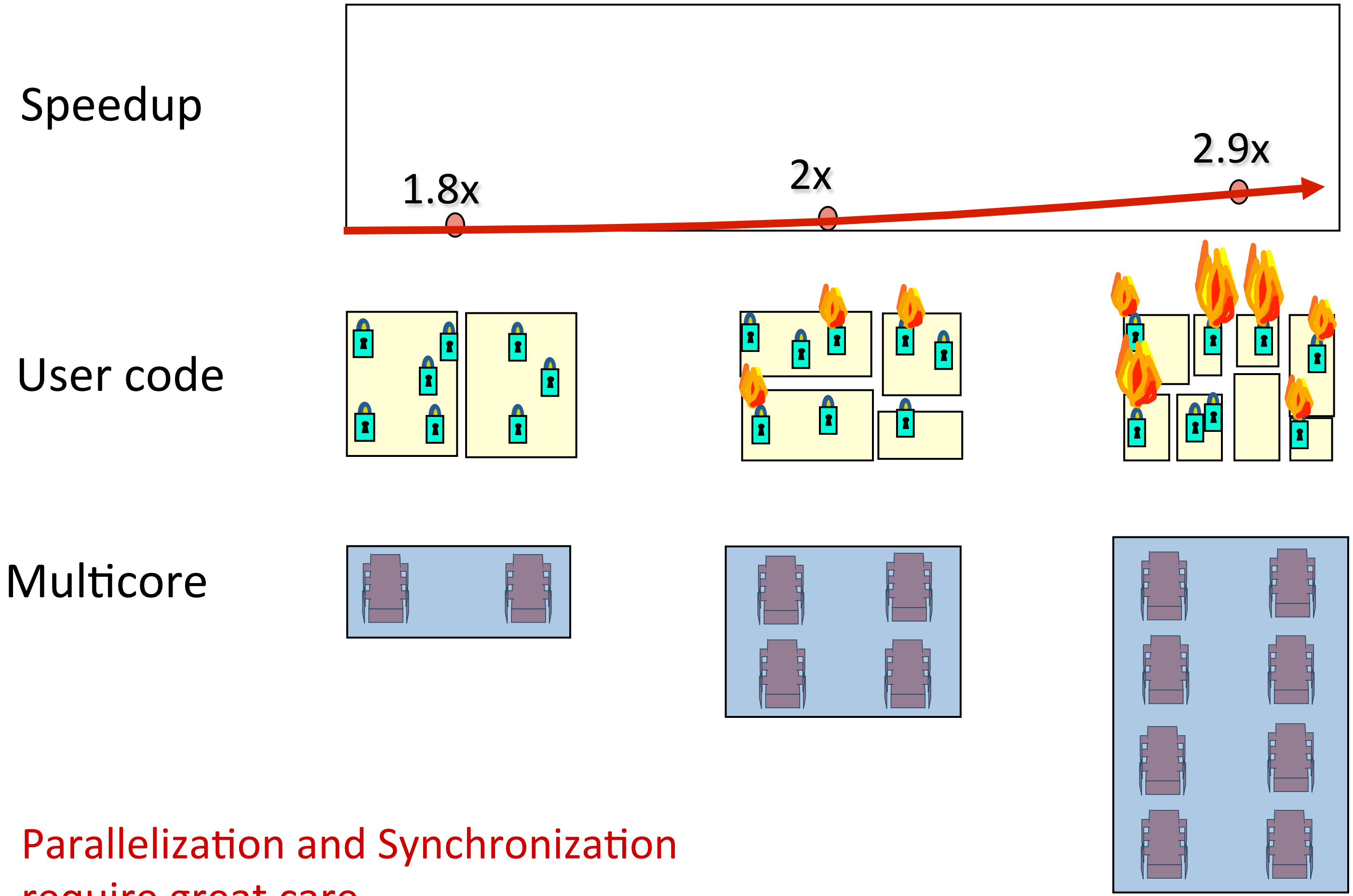


Multicore Software Scaling



Unfortunately, not so simple…

Real-World Multicore Scaling



Parallelization and Synchronization
require great care…

Why?

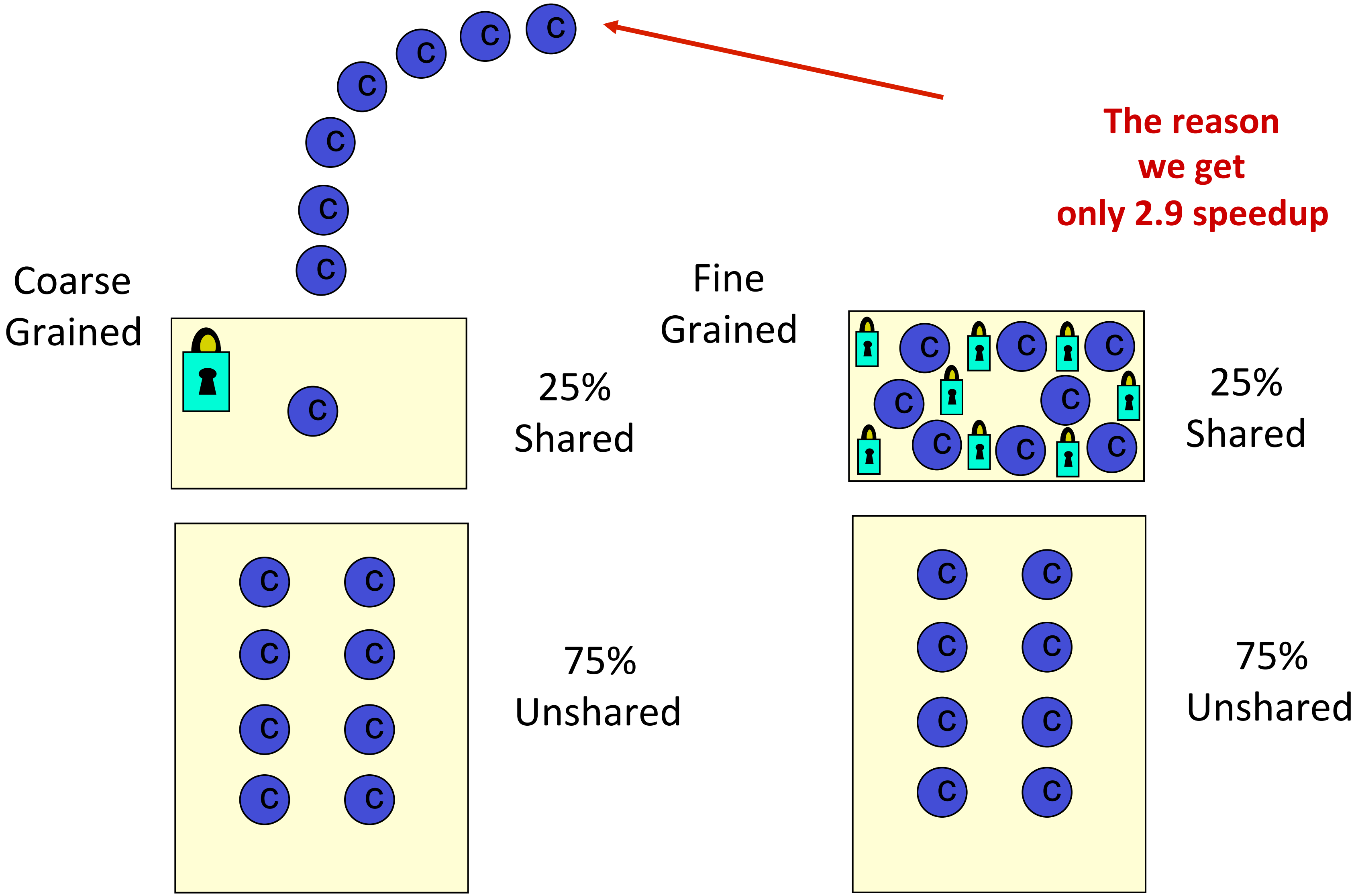
Amdahl's Law:

Speedup = 1/(ParallelPart/N + SequentialPart)

Pay for N = 8 cores
SequentialPart = 25%

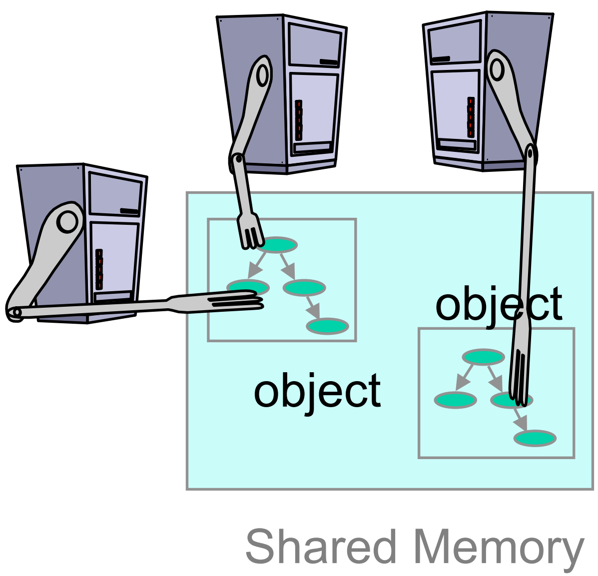
Effect of 25% becomes more accute as num of cores grows
2.3/4, 2.9/8, 3.4/16, 3.7/32........4/ ∞

Shared Data Structures

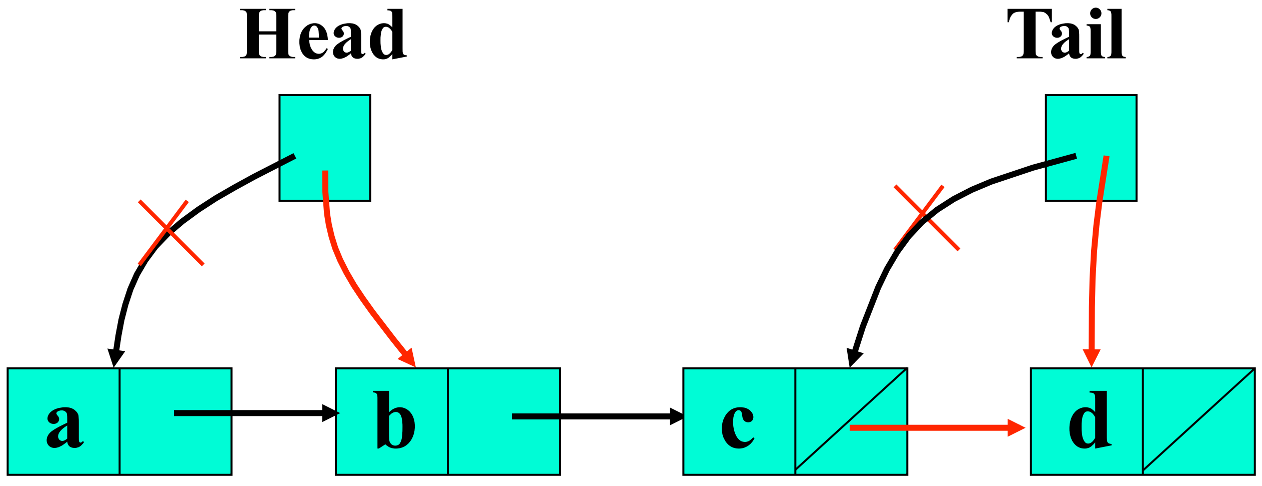


Concurrent Programming

How do we lower the granularity of synchronization without making the concurrent programmer's life unbearable?!



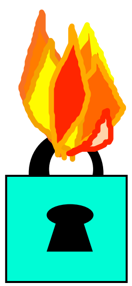
A FIFO Queue



`Dequeue() => a`

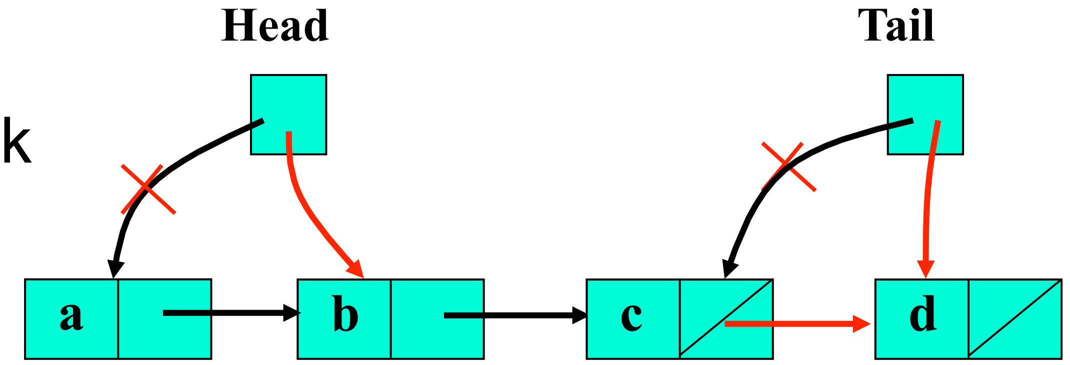
`Enqueue(d)`

A Concurrent FIFO Queue



Object lock

Simple Code, easy to prove correct

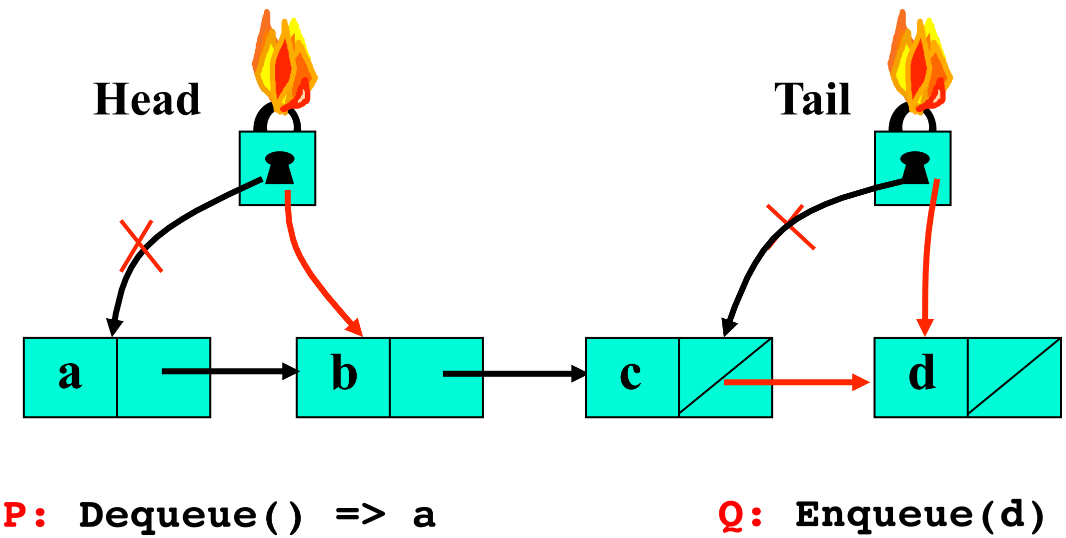


`P: Dequeue() => a`   `Q: Enqueue(d)`

Contention and sequential bottleneck

Fine Grain Locks

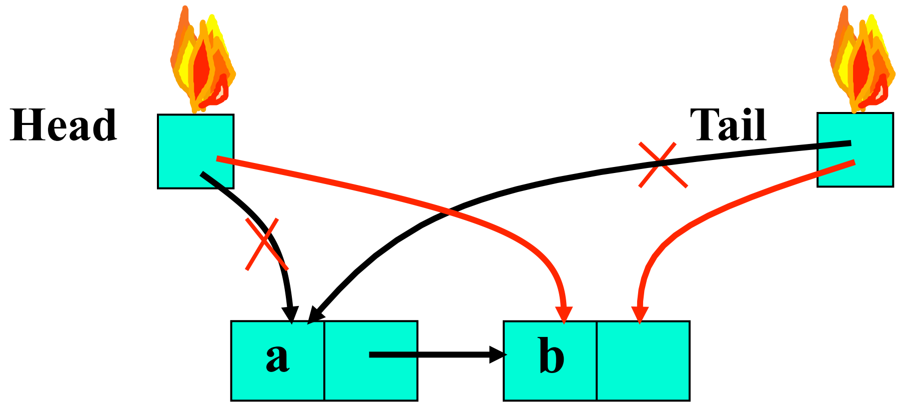
Finer Granularity, More Complex Code



Verification nightmare: worry about deadlock, livelock…

Fine Grain Locks

Complex boundary cases: empty queue, last item

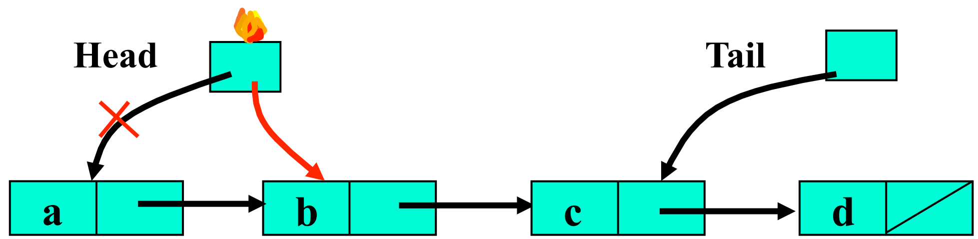


P: Dequeue() => a **Q: Enqueue(b)**

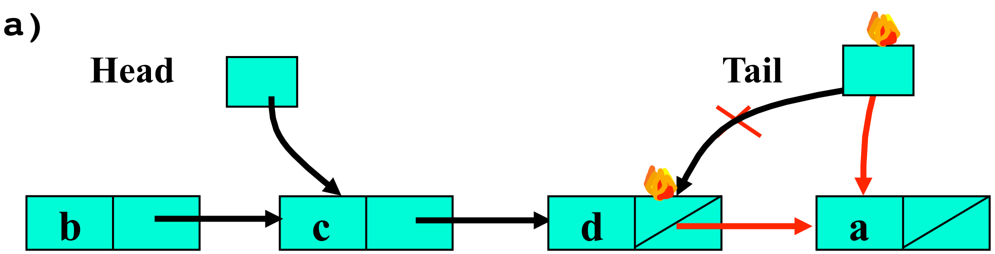
Worry how to acquire multiple locks

Real Applications

Complex: Move data atomically between structures




```
P: Dequeue(Q1,a)
   Enqueue(Q2,a)
```



More than twice the worry…

Transactional Memory

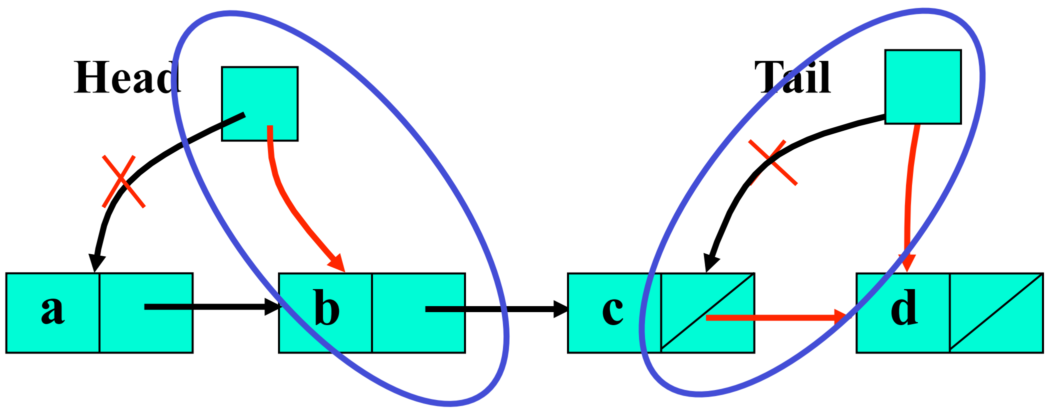
[HerlihyMoss93]

The End of Locks

“The BlueGene/Q processors that will power the 20 petaflops Sequoia supercomputer being built by IBM for Lawrence Livermore National Labs will be the first commercial processors to include hardware support for transactional memory.

Promise of Transactional Memory

Great Performance, Simple Code



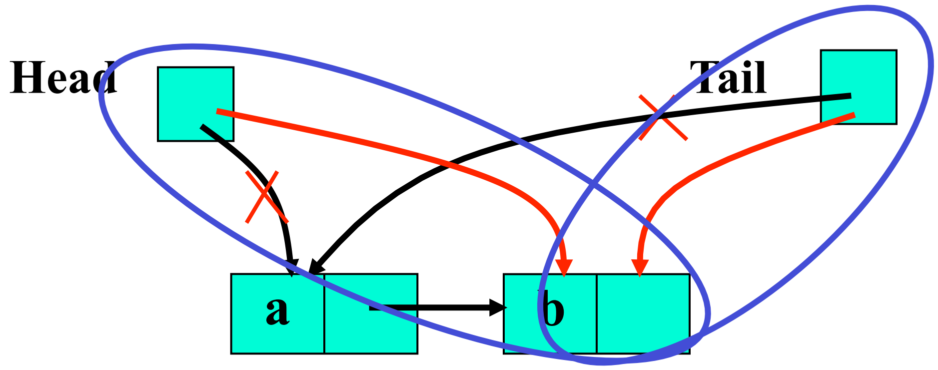
P: Dequeue() => a

Q: Enqueue(d)

Don't worry about deadlock, livelock, subtle bugs, etc…

Promise of Transactional Memory

Don't worry which locks need to cover which variables when…

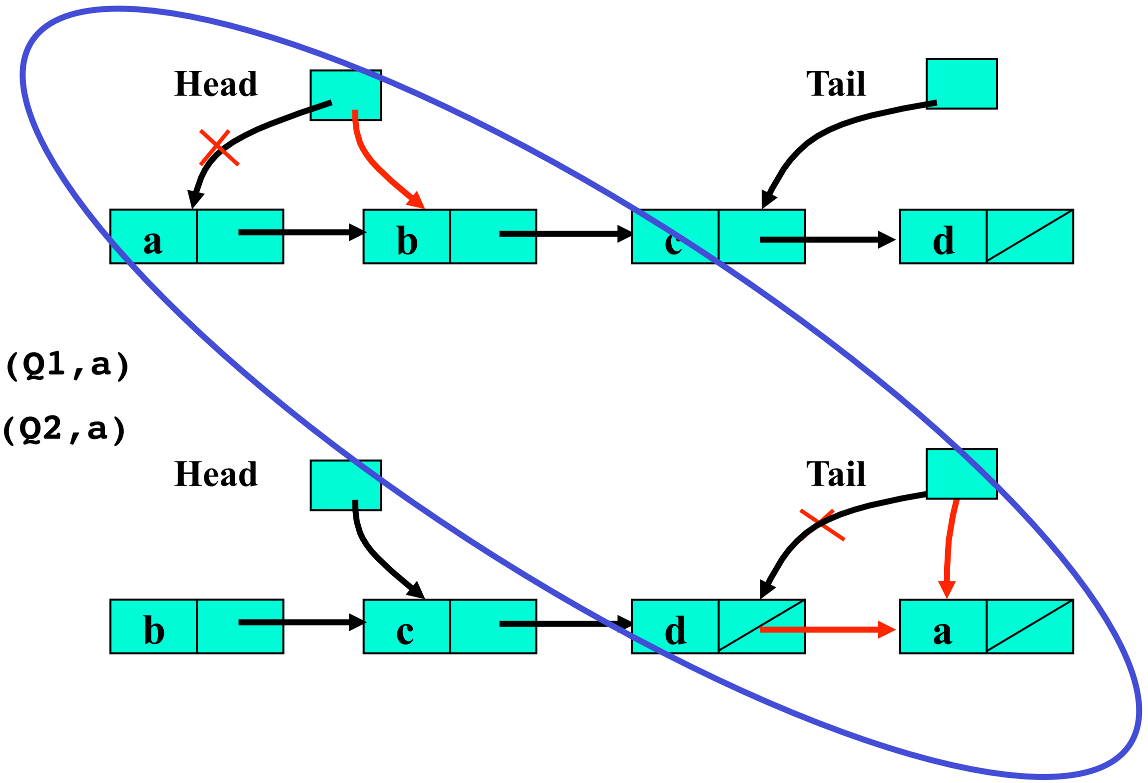


P: `Dequeue() => a` **Q:** `Enqueue(d)`

TM deals with boundary cases under the hood

For Real Applications

Will be easy to modify multiple structures atomically



P: `Dequeue(Q1,a)`

`Enqueue(Q2,a)`

Using Transactional Memory

```
enqueue (Q, newnode) {
  Q.tail-> next = newnode
  Q.tail = newnode
}
```

Using Transactional Memory

```
enqueue (Q, newnode) {
atomic{
   Q.tail-> next = newnode
   Q.tail = newnode
   }
}
```

Transactions Will Solve Many of Locks' Problems

No need to think what needs to be locked, what not, and at what granularity

No worry about deadlocks and livelo[illegible]

No need to think about rea[illegible]g

Can compose c[illegible]t objects in a way that is safe and s[illegible]able

But there are problems!

The Problems

Aborts

- On concurrent conflict the transaction must abort, and restart its execution
- State must be saved on start, and restored on restart
- Hard to debug

Privatization

- Shared Object → Private Object

Our Proposal

New Transactional Memory without Aborts

- Every transaction executed only once
- Limits concurrency significantly
- Surprisingly, works good on many standard benchmarks

New Privatization Technique

- New transaction type: private transaction
- Efficient and Scalable quiescence mechanism for privatization