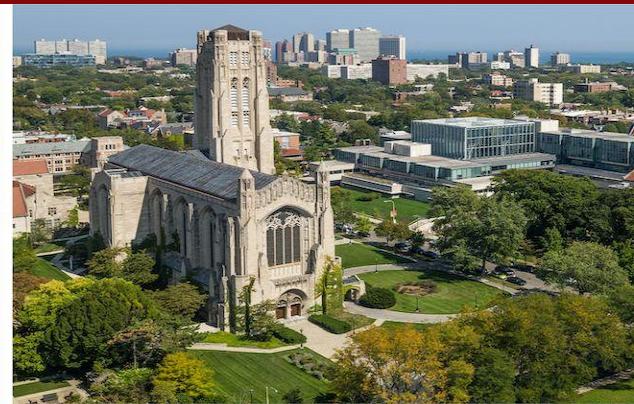


# MPCS 51300 - Parallel Programming

## M1: Introduction to Parallel Programming



Lamont Samuels

# Course Logistics

- All course information is located on the course website

<http://people.cs.uchicago.edu/~lamonts/classes/mpcs52060/index.html>

# Motivation for Parallelism

# Basic Architecture of a Computer

- The modern computers are based on an architecture introduced by John Von Neumann (i.e., Von Neumann architecture). These are the main components:
  - Memory** - a collection of locations, each of which is capable of storing both instructions and data. Every location consists of an address, which is used to access the location, and the contents of the location.
    - A computer can contain various types of memory components (registers, caches, RAM, secondary memory (i.e., hard drive) etc.)
  - Central Processing Unit (CPU)** - is the controller of a computer and is composed of many different parts. However there are two significant parts: a *control unit* and a *arithmetic and logic unit*
    - Control unit** - responsible for deciding which instruction in a program should be executed. (the boss)
    - Arithmetic and logic unit (ALU)** - responsible for executing the actual instructions. (the worker)
- Input devices** - devices that send information to a computer for processing (e.g., keyboard, mouse, etc.)
- Output devices** - devices that display/use the results of processing of tasks by the computer. (e.g., monitor, printer, speakers)

Uniprocessor Computer Architecture

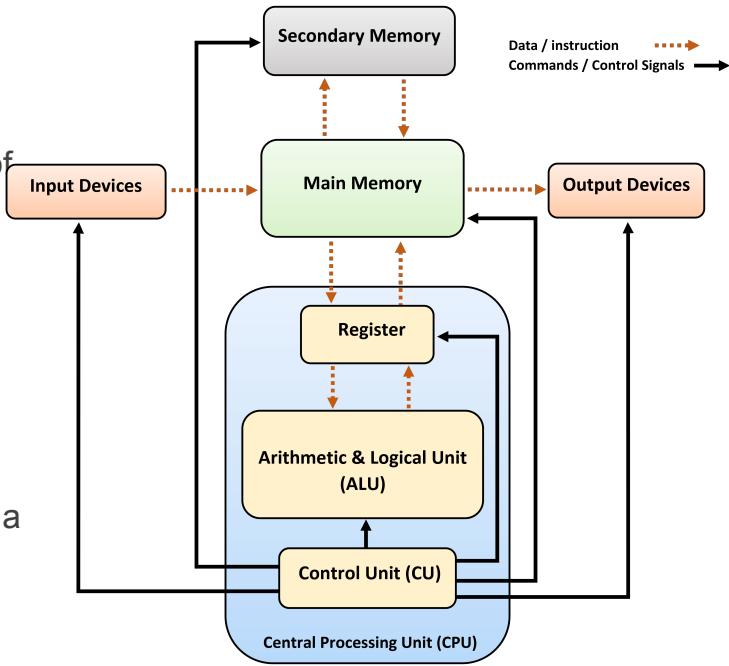


Diagram Source: [https://commons.wikimedia.org/wiki/File:Computer\\_architecture\\_block\\_diagram.png](https://commons.wikimedia.org/wiki/File:Computer_architecture_block_diagram.png)

# Sequential Programs

- As programmers, we are overwhelmingly accustomed to developing **sequential programs**
  - A program solves a problem that is composed of a series of textual statements that specifies their order of execution.
  - Each statement is converted into an instruction that is executed by the CPU.
  - Sequential programs are **deterministic**, which means that every time the program runs the result is the same.

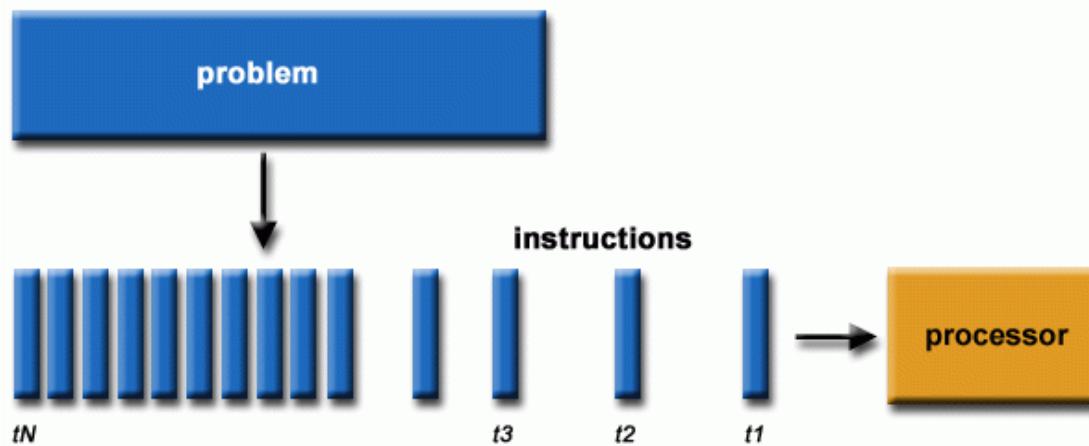
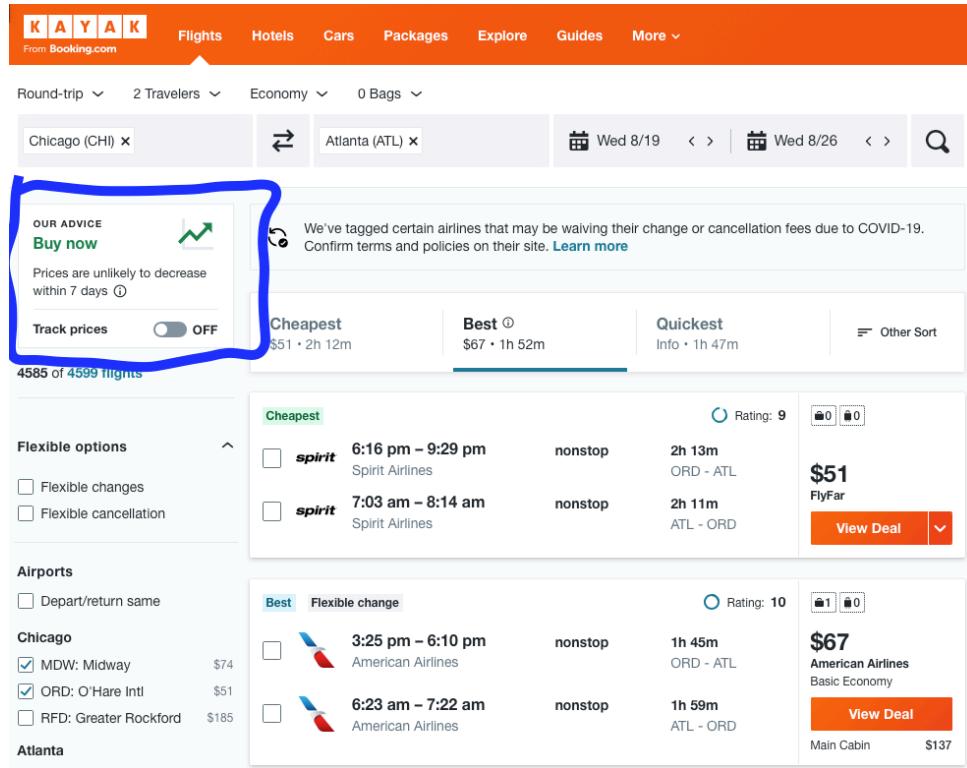


Diagram Source: <https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial>

# Sequential Program Example

- Simple example of serial computation:
  - **Problem:** Develop a program that gathers data about route information from different cities. The overall objective is to use this program to help develop a classifier to tell a user whether to buy a ticket now or later.



# Crawler Components

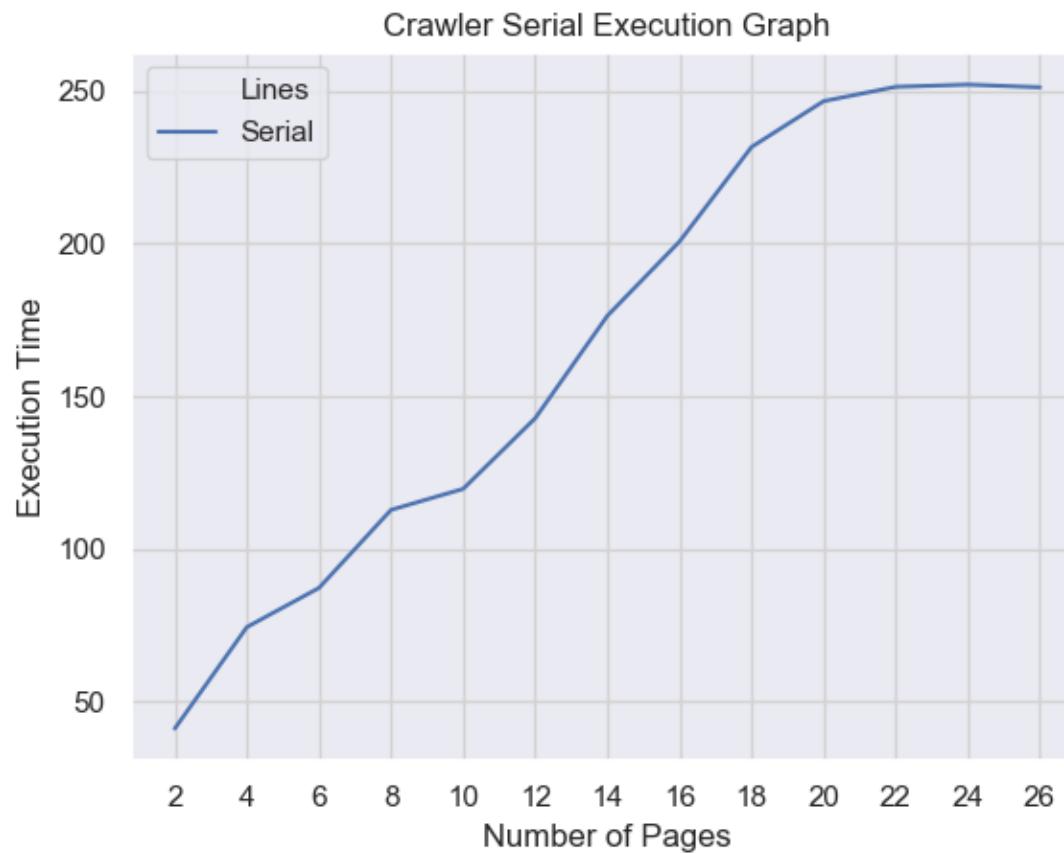
- Components of the route finder application:
  - A web crawler that crawls flight route information for popular cities on Fare Detective (<https://www.faredetective.com/farehistory/>).
  - Program Output:** A CSV file with lowest-cost fare information from the cities listed on the site.

Partial CSV Output

routes		
From	To	Fare
Amsterdam	Dublin	50
Amsterdam	South Hampton	88
Amsterdam	Birmingham	108
Amsterdam	Belfast	128
Amsterdam	Belfast	136
Amsterdam	Aberdeen	137
Amsterdam	Manchester	185
Amsterdam	Venice	190
Amsterdam	Ljubljana	206
Amsterdam	Bristol	209
Amsterdam	Trieste	236
Amsterdam	Gothenburg	296
Amsterdam	Minneapolis	338
Amsterdam	Boston	391
Amsterdam	Montreal	445
Amsterdam	Washington DC	549
Amsterdam	Los Angeles	601
Amsterdam	San Francisco	619
Amsterdam	Vancouver	624
Amsterdam	West Palm Beach	643
Amsterdam	Kuching	664
Amsterdam	Toronto	682
Amsterdam	Shanghai	711
Amsterdam	Houston	718
Amsterdam	Washington DC	724
Amsterdam	Calgary	732
Amsterdam	Tyler	734
Amsterdam	Oakland	865
Amsterdam	Portland	876
Amsterdam	Denpasar Bali	895
Amsterdam	San Jose	908
Amsterdam	Jakarta	941
Amsterdam	Shanghai	949
Amsterdam	Santa Ana	977
Amsterdam	Atlanta	1000
Amsterdam	Portland	1018
Amsterdam	Denver	1026
Amsterdam	Norfolk	1063
Amsterdam	Honolulu	1080

# Crawler Performance

- Execution time (in seconds):



# Is Sequential Programming Good Enough?

- For many years, sequential programs executed faster on newer processors with no modification.
- Since ~2003, single-processor performance improvement has slowed to about 20% per year. **Maybe not?**
- History of Hardware Trends
  - Increase in single processor performance has been driven by increasing the *density of transistors* on the CPU chip.
    - **Transistors** - electronic components on integrated circuits that act as switches in order to construct logical gates. We use logic gates to form logical units capable of arithmetic and complex logical operations.
  - As the size of the transistors decreases, their speed can be increased, and the overall speed of the integrated chip will be increased (i.e., increasing clock rate).

# Moore's Law

- Moore's Law: computing power tends to approximately double every two years.
  - What the law really means is that the number of transistors that can be packed into a given unit of space will roughly double every two years.

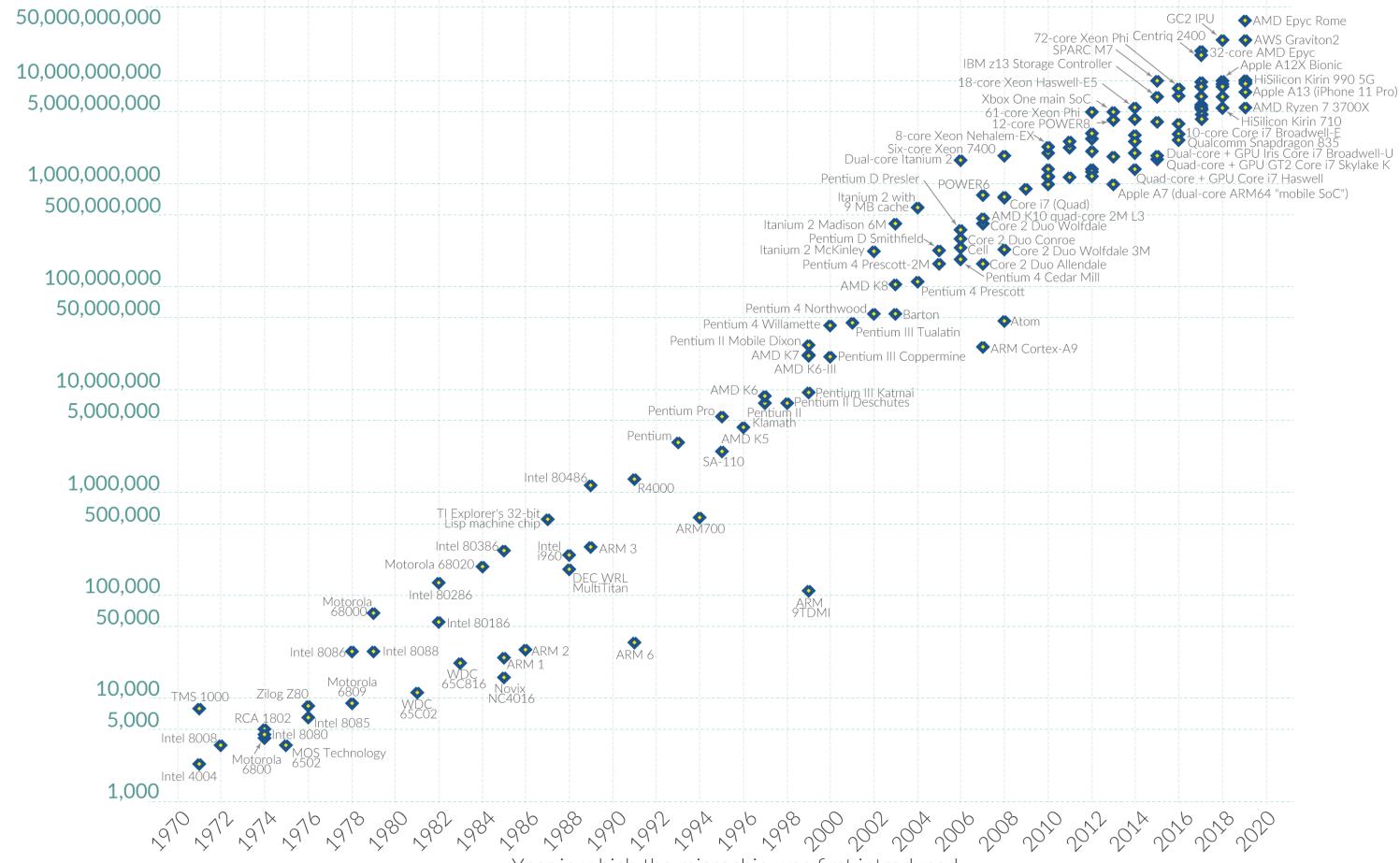
# History of Moore's Law

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World  
in Data

Transistor count

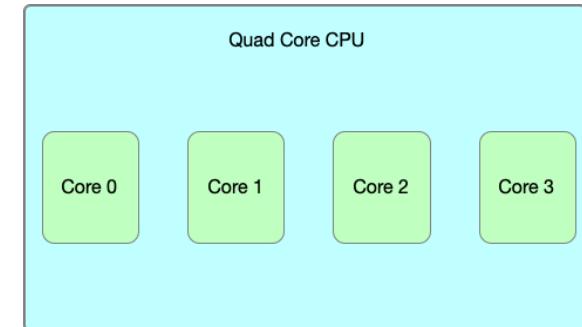
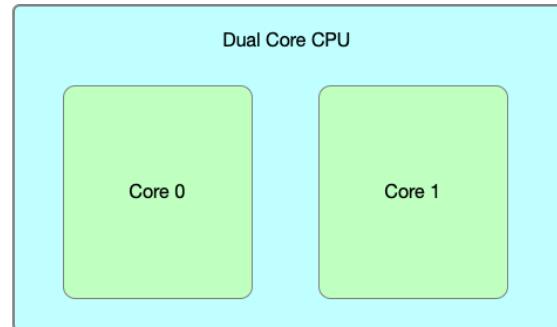
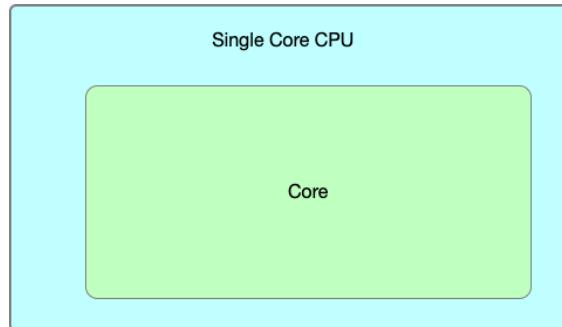


# But wait...There's a Problem!

- Transistors are starting to suffer power consumption and integrity issues:
  - As the speed of transistors increases, their power consumption also increases; therefore this exceeds the power density that can be dealt with by air cooling.
  - Power consumption is dissipated as heat and when an integrated circuit gets hot it becomes unreliable.
  - Transistor gates have become too thin, affecting their structural integrity, which leads to currents starting to leak.
  - Physical manufacturing problems such as quantum tunneling (the inability to keep electrons contained beyond a certain thickness threshold) is also leading to a slow in single processor production
- Overall, it's becoming impossible to continue to increase the speed of integrated circuits (hovering around 3.0GHz-3.7Ghz). Although with overclocking we've seen this rise between 4.0GHz-5.0+GHz.

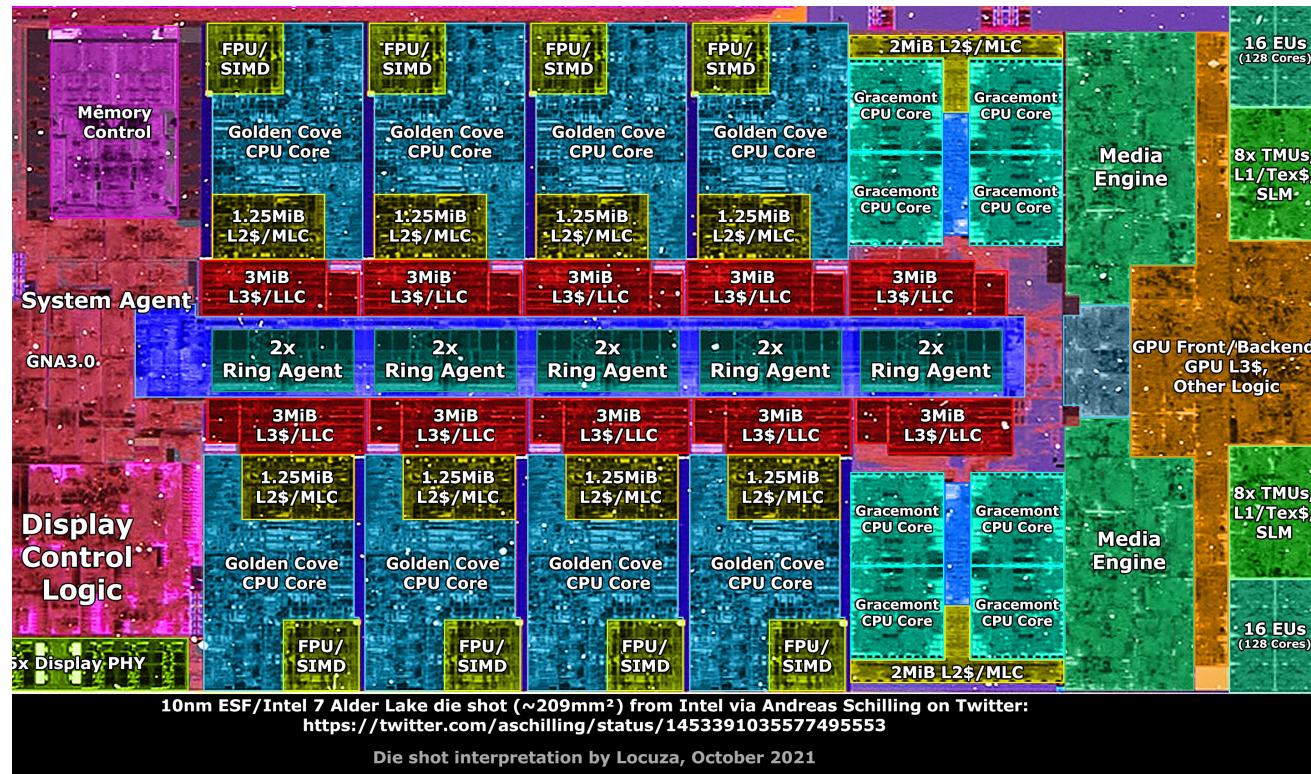
# What's the Fix?

- To manage CPU power dissipation, hardware manufacturers have moved to a **multi-core** chip design
  - Continue to increase transistor density by having multiple and lower clocked processors on one chip (CPU).
- Terminology:
  - **Core** (also known as **processor**) : synonymous with central processing unit (CPU).
  - **Multicore** (also known as **multiprocessor**): more than one core on an integrated circuit.



# Real-World Multicore Architectures: Intel's Alder Lake-S Silicon

- Die shot of the Alder Lake- processors. These are Intel's newest (January 2022) processors used in the Core i9-12900K, Core i7-12700K, and Core i5-12600K, and other Intel chips.
  - 8-6 Performance CPU cores (normal CPU cores that are large, and run at high clock-speed)
  - 8-4 Efficient CPU cores ( small CPU cores that run at reduced clock speed)



# Real-World Multicore Architectures: Apple Silicon

**M1**

- | CPU: 4x Performance Cores, ≤3.2 GHz + 4x Efficiency Cores, ≤2.06 GHz
- | GPU: 8x GPU Cores (1024 EUs)  
24.576 „Threads in-flight“  
~1.27 GHz  
2.6 FP32 TeraFLOPs  
82 GigaTexel/s (64 TMUs)  
41 GigaPixel/s (32 ROPs)
- | Memory: 128-bit LPDDR4X-4266  
68.3 GB/s bandwidth

**M1 Pro**

- | CPU: 8x Performance Cores + 2x Efficiency Cores
- | GPU: 16x GPU Cores (2048 Execution Units)  
49.152 „Threads in-flight“  
~1.27 GHz  
5.2 FP32 TeraFLOPs  
164 GigaTexel/s (128 TMUs)  
82 GigaPixel/s (64 ROPs)
- | Memory: 256-bit LPDDR5-6400  
204.8 GB/s bandwidth

**M1 Max**

- | CPU: 8x Performance Cores + 2x Efficiency Cores
- | GPU: 32x GPU Cores (4096 Execution Units)  
98.304 „Threads in-flight“  
~1.27 GHz  
10.4 FP32 TeraFLOPs  
327 GigaTexel/s (256 TMUs)  
164 GigaPixel/s (128 ROPs)
- | Memory: 512-bit LPDDR5-6400  
409.6 GB/s bandwidth

**PlayStation 5**

- 8x Zen2, ≤3.5 GHz
- 18x Cores (2304 EUs)  
92.160 „Threads“  
~2.23 GHz  
10.28 FP32 TeraFLOPs  
321 GTex/s (144 TMUs)  
143 GPix/s (64 ROPs)
- 256-bit GDDR6-14000  
448 GB/s bandwidth

Sources: M1 CPU clock frequency from Anandtech, chip images from Apple via Anandtech  
Compiled and annotated by Locuza, October 2021

# Programming on Multicore Architectures

- Programming on multicore chips requires programmers to break the problem into discrete parts (i.e., **tasks**) where each part can be ran on a separate core.
  - A task is a unit of work, where in this class, work is a series of program statements. Tasks can potentially be executed in parallel but this is not a requirement.
  - At a high-level, a parallel program solves a problem that consists of multiple tasks running on multiple processors simultaneously .

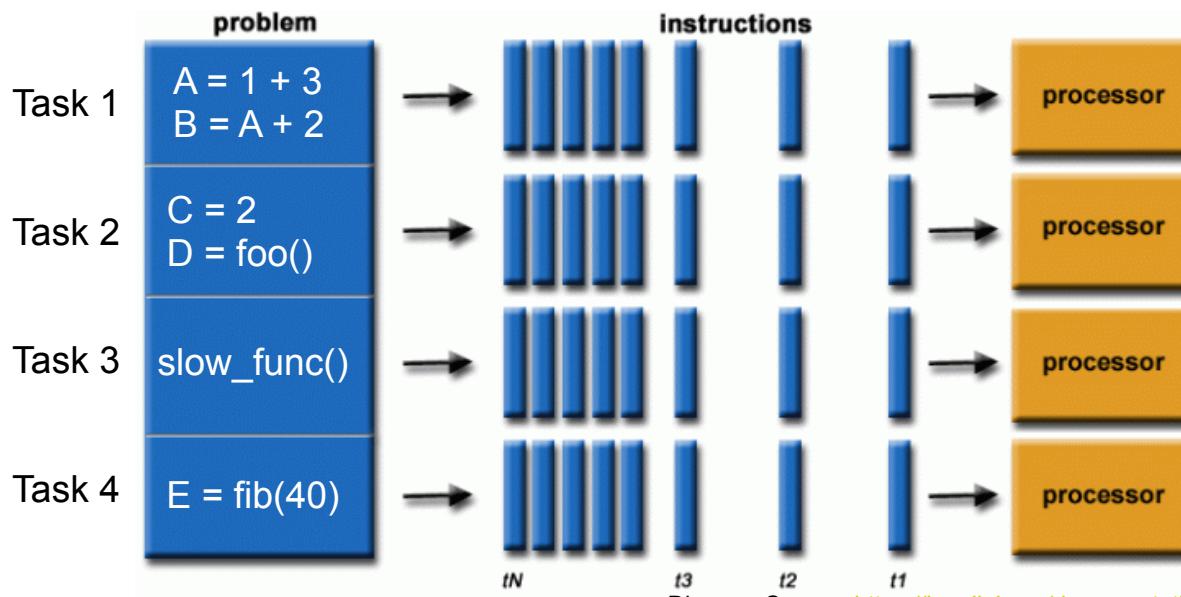


Diagram Source: <https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial>

# Concurrency vs. Parallelism

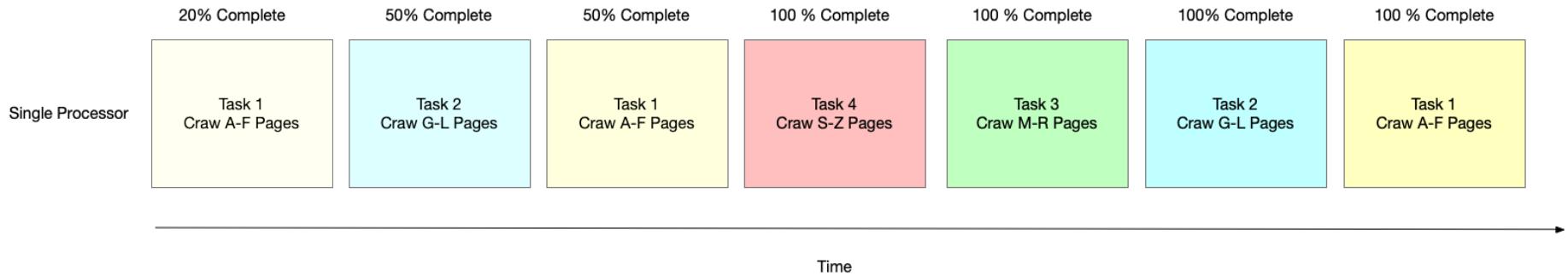
- You may hear the terms **concurrent** and **parallel** referring to the same notion of executing a task at the same time; however, there is a key distinction between the two.
  - Concurrency is about dealing with tasks that are logically happening simultaneously. Two tasks are concurrent if they can be logically active at some point in time.
  - Parallelism is about dealing with tasks that are physically happening simultaneously.
- Concurrency provides a way to structure a solution to a problem that may (but not necessary) have tasks executed in parallel.

# Crawler Example as a Concurrent Program (1)

- One sequential solution could be to use a for-loop along with some code to crawl all 26 pages (A-Z) to produce the CSV file.
- One concurrent solution is to break the problem down into tasks where each task crawls a certain number of pages. The results of each task will be combined together to produce the final CSV file.
  - Task #1 - Crawl Pages A-F (Number of Pages: 6)
  - Task #2 - Crawl Pages G-L (Number of Pages: 6)
  - Task #3 - Crawl Pages M-R (Number of Pages: 6)
  - Task #4 - Crawl Pages S-Z (Number of Pages: 8)
- **This is not the only way to break this problem down into tasks.** I'm just showing you one way. We will discuss various other ways throughout this course.

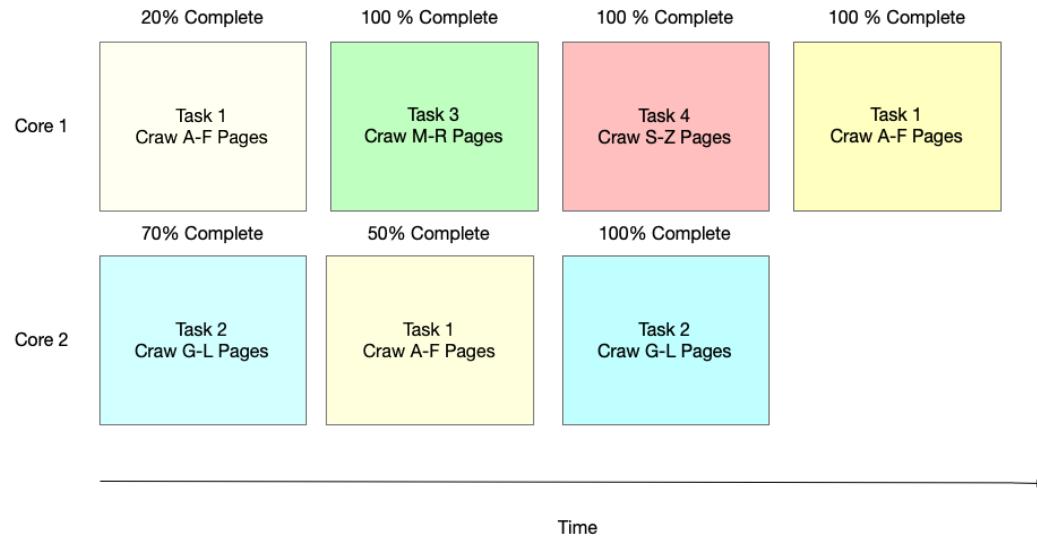
# Crawler Example as a Concurrent Program

## Concurrent Program with no parallelism

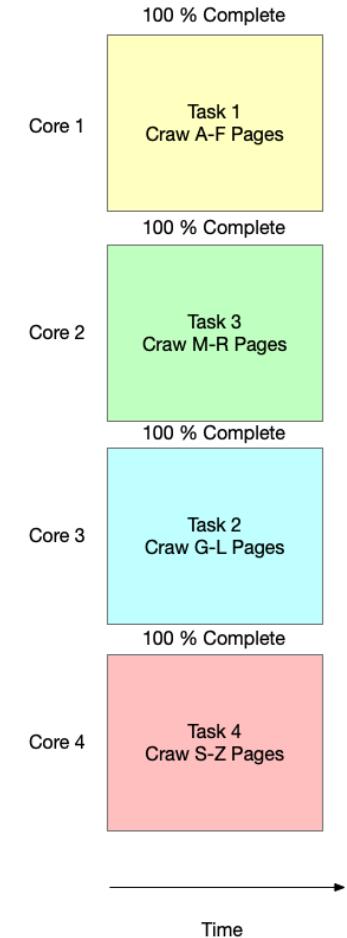


# Crawler Example as a Concurrent Program (with Parallelism)

Parallel Program #1 with Dual Core Chip



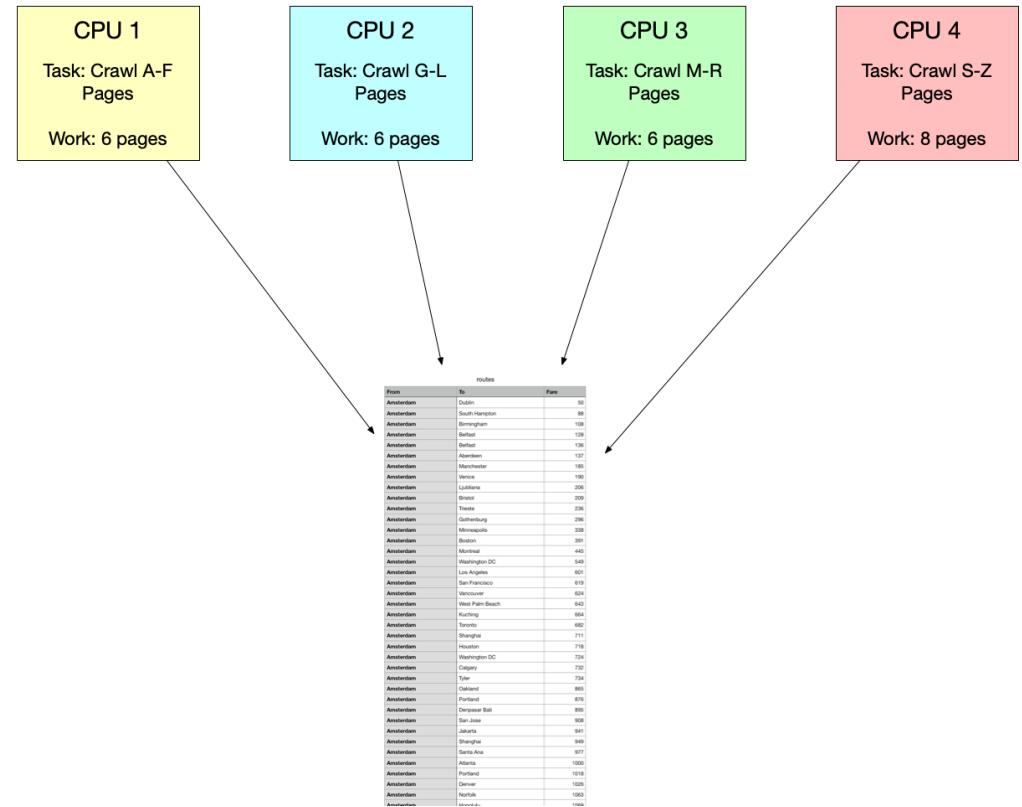
Parallel Program #2 with Quad Core Chip



Each parallel program should outperform the single concurrent and sequential program.

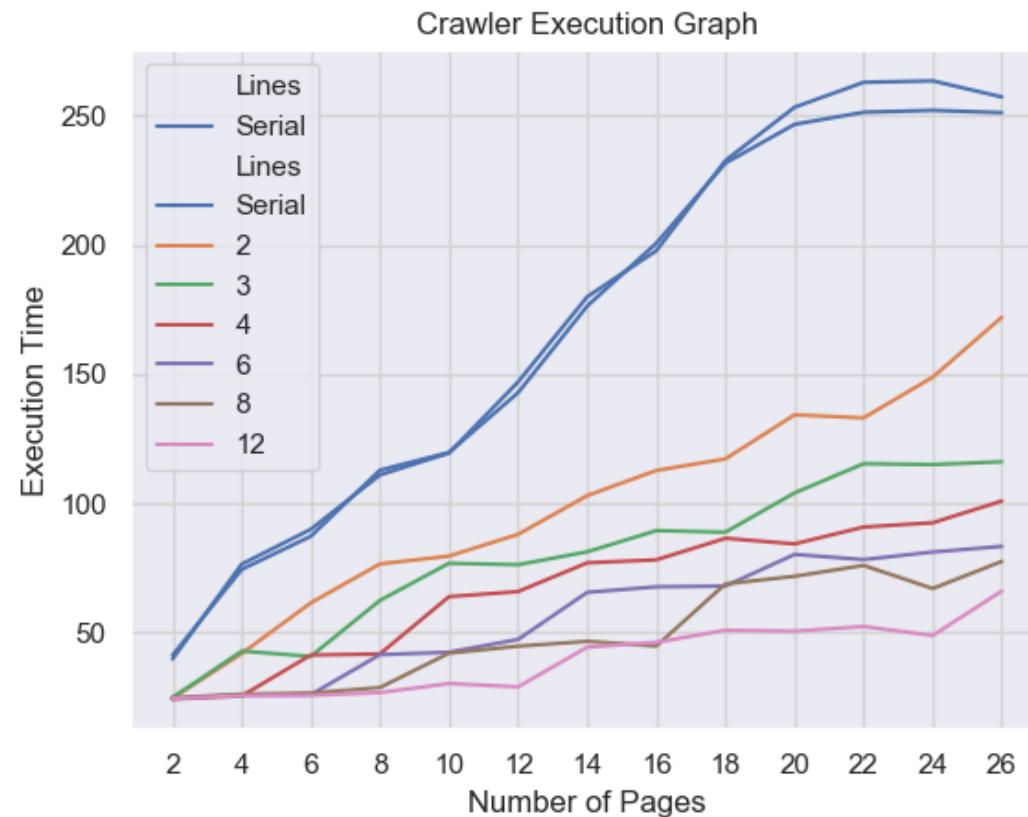
# Crawler Example as a Parallel Program

- Demo: Crawler Program



# Crawler Execution Time

- Execution time (in seconds) with a parallel component for our crawler:



# What will you learn in this class?

- Main objectives:
  - Best practices for designing and implementing parallel programs.
  - Synchronization mechanisms to maintain deterministic results
  - Schemes for processing tasks in parallel: algorithms, patterns and techniques to help with maintaining performance when scaling your application.
  - Understanding parallel architectures for fine tuning performance
- **Overall Goal:** Regardless of your programming language of choice (C, Java, C++, Python, etc.), you should be able to use the techniques, algorithms, patterns, and practices taught in this class and apply them to developing parallel applications in those languages.

# Go Bootcamp

# Design of Go

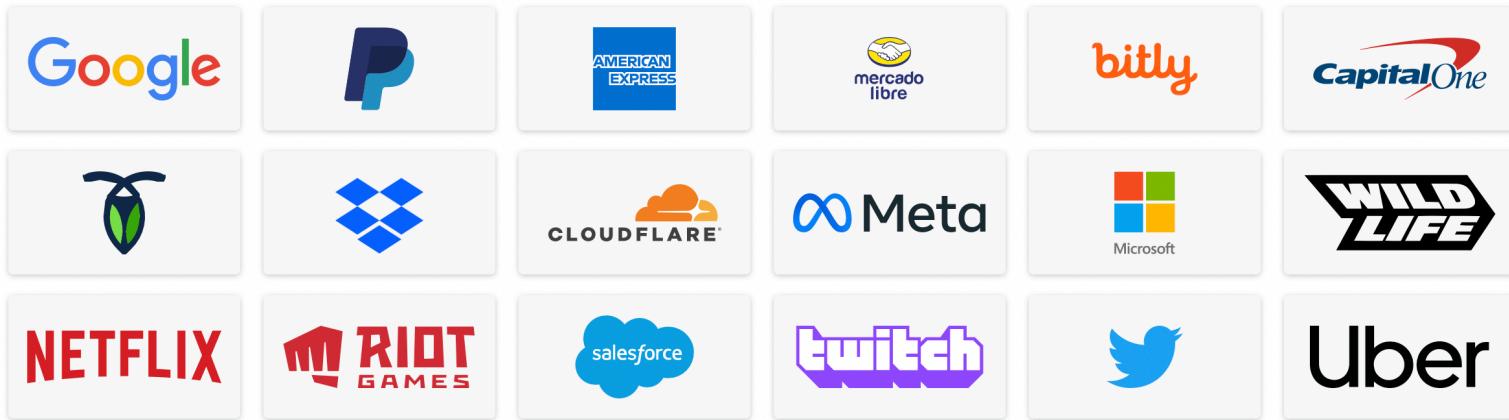
Go was designed by Google (specifically Robert Griesemer, Rob Pike, and Kenneth Thompson in 2007) to solve problems that Google faces.

- Goals
  - Eliminate slowness
  - Inefficiencies
  - Maintain and improve scale
  - Make a concurrent language
- Types of problems Google faces?
  - C++ for servers, plus lots of Java and Python mixed in
  - Large employee base
  - Millions upon Millions of lines of code for various projects
  - Distributed build system
  - Millions of compute cluster machines, needing to perform tasks

# Who uses Go?

## Companies using Go

Organizations in every industry use Go to power their software and services [View all stories](#)



- Diagram Source: <https://go.dev>

# How are we going to learn Go?

- We are actually going to learn Go by going through examples shown on: <https://gobyexample.com>
- Constructs you should know:
  1. Values
  2. Variables
  3. Constants
  4. Importing
  5. Iteration statement: For, Range
  6. Selection statements: If/Else, Switch
  7. Core Data Structures: Arrays, Slices, Maps
  8. Functions
  9. Multiple Return Values
  10. Variadic Functions
  11. Pointers (VERY IMPORTANT!)
  12. Structs
  13. String Formatting and Functions
  14. Generics

# MPCS 52060 - Parallel Programming

## M2: Shared-Memory Architecture



Lamont Samuels

# Shared Memory Systems

The focus of this course will be on implementing parallel programs on **shared-memory** systems

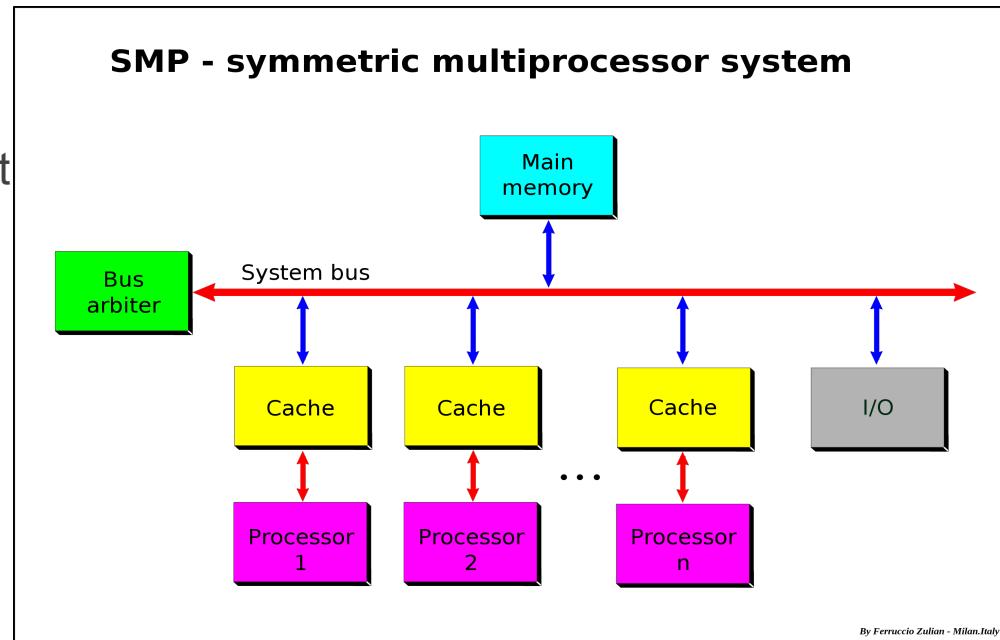
- A shared-memory system consists of at least one multi-core CPU that allows all processors to access memory as a single global address space (also known as global memory).
  - Many of the architectures designed by Intel, AMD, ARM, and etc.
  - Architectures in many modern personal computers and phone.
- Shared memory can be implemented as a set of memory modules (e.g., caches, physical memory, etc.)

# Symmetric Multiprocessing

- For this course, we focus on a specific type of shared memory system known as **uniform memory access (UMA)** systems
  - UMA systems provide each core with equal access and access times to global memory
- Specifically, we look at **symmetric multiprocessing (SMP)** systems that implement a UMA architecture
  - Each SMP processor is identical
  - All processors can access the shared main memory at the same speed.
  - Controlled by a single operating system instance that treats all processors equally (no processor is reserved for a specific purpose).
  - SMP systems are tightly coupled multiprocessor systems,
    - All processors can execute different programs (and with different data) in parallel.
    - All processors share common resources (e.g., memory, I/O device, interrupt system) all on the same system bus.

# Symmetric Multiprocessing

- Processors and memory are linked by a system bus(a broadcast medium that acts like a tiny Ethernet).
- Processors and memory have bus controllers units in charge of sending and listening for messages broadcast on the bus (listening is sometimes called **snooping**).
- SMP Advantage: Most common interconnect architecture used today because it's easy to build.
- SMP Disadvantage: Not scalable to large number of processors because the bus becomes overloaded.
  - See (NUMA in textbook for a potential improvement)



# System Bus for SMPs

- The system bus (also known as interconnect) is a finite resource shared among processors
- Performance in SMP systems is limited by memory bus bandwidth
- Processors can be delayed if others are consuming too much of the interconnect's bandwidth
- SMP configurations do not scale well past 64 processors

# Program Execution on Shared Memory Systems

- The operating system (OS) along with the hardware is responsible for managing the execution of a program.
- An instance of a program running is known as a **process** and contains
  - Its own block of memory, where memory is made up of registers, stack, etc.
  - The program code translated into machine language (i.e., instructions for the processor)
  - Information about the state of the process (e.g., program counter)
  - Security Information
  - Descriptors of resources the OS has allocated to the process
  - A process cannot access the memory of another process.

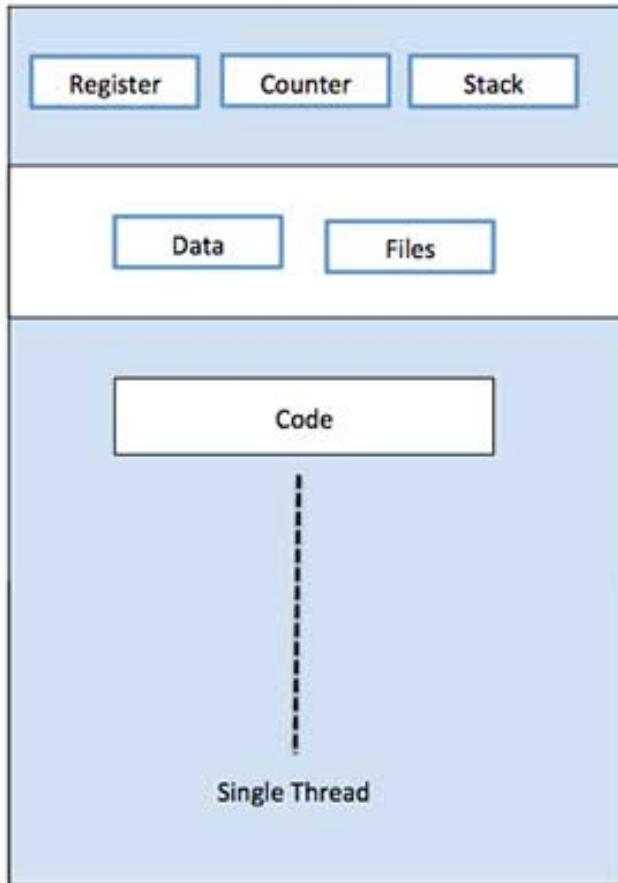
# Program Execution on Shared Memory Systems

- Each process when started will spawn a single **thread**, which is an entity within the process that can be scheduled for execution on a processor by the OS.
- A thread is responsible for executing the instructions of the task assigned to it.
  - Remember a task is a unit of work (i.e., a series of instructions from the program)
  - In a sequential program there is only one thread, known as the **main/primary/heavy** thread and is executing all the code in the program.
- A concurrent program can fork (also known as spawn) additional “light-weight” threads within a given process to execute tasks.
  - All threads in the process can then be executed in parallel on the various processors.

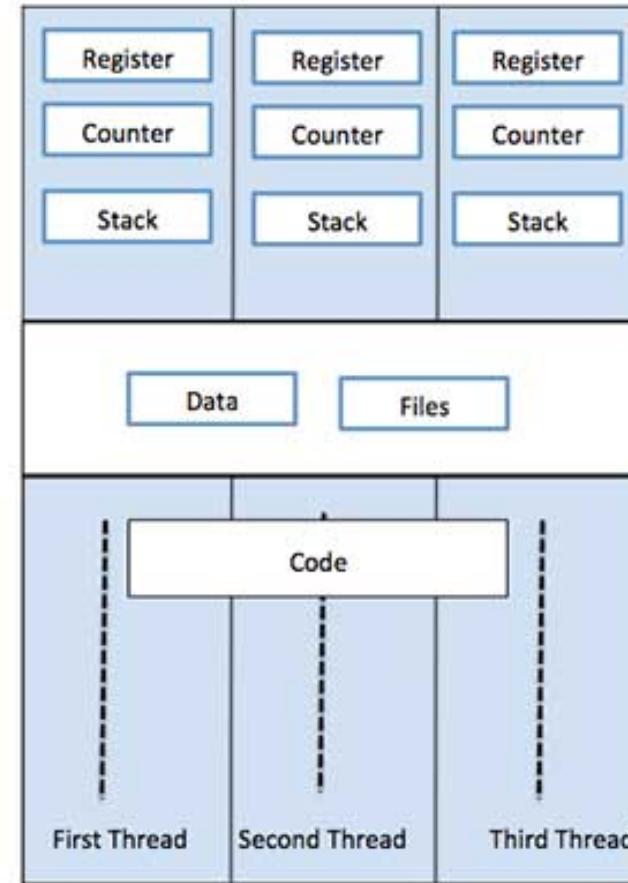
# Threads

- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.
- Have their own stack where they can store local variables, function calls, etc but share the heap where dynamically allocated items are stored with other threads of the same process.
- Some languages and operating systems provide the notion of thread-local storage, where threads can store and retrieve values independent of other threads.

# Thread vs Process



Single Process P with single thread



Single Process P with three threads

Diagram Source: [https://www.tutorialspoint.com/operating\\_system/os\\_multi\\_threading.htm](https://www.tutorialspoint.com/operating_system/os_multi_threading.htm)

# Processes and Processors

The OS gives the illusion that a single processor system is running multiple programs simultaneously.

- Each process takes turns running (i.e., time slice).
- A processor can run a process for a while and then set it aside and run another process (i.e., **context switch**)
- After its time is up, it waits (i.e., blocks) until it has a turn again.

Processor may set aside or descheduled a process for a number of reasons:

- A memory request that will take some time to satisfy
- A process has run long enough (i.e., reached an end to its time slice.). Thus, it's time for another process to begin its time slice.
- Note: When a process is descheduled, it may resume execution on another processor.

# Aside: Simultaneous Multithreading (aka HyperThreading)

- Most modern multicore architectures have **Simultaneous Multithreading(SMT)**:
  - Use several threads to schedule instructions from different threads in the same cycle if necessary.
  - It helps increase the usage of functional units of a processor more effectively.
  - Hardware support for SMT is based on the replication of the chip area used to store the processor state.

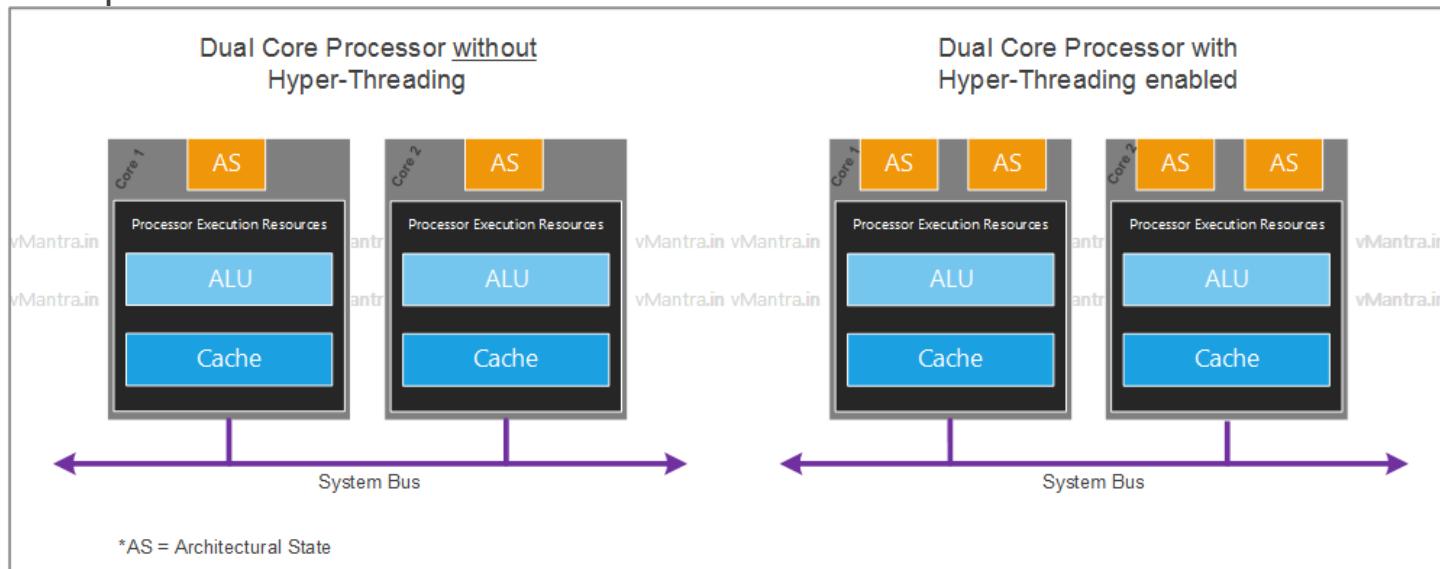


Diagram Source: <https://www.vmantra.in/hyper-threading/>

# Aside: Simultaneous Multithreading (aka HyperThreading)

- The processor appears to the operating system and user programs as a set of logical processors to which processors or threads can be assigned for execution.
- Processes or threads can come from a single or several user programs.
- Number of replications of the processor state determines the number of logical processors.
- These logical processors are also known as **hardware threads**.

# Concurrent Programming on Shared Memory Systems

# Aside: Understanding Pointers

- Before we start talking about concurrent programming in Go, you need to understand the notion of a pointer.
  - A pointer is just an object that stores a memory address.
  - It will be important to understand because this will be the way we allow “threads” in Go to communicate initially when writing parallel programs.
- Demo: m2/pointers directory

# Structuring Parallel Programs

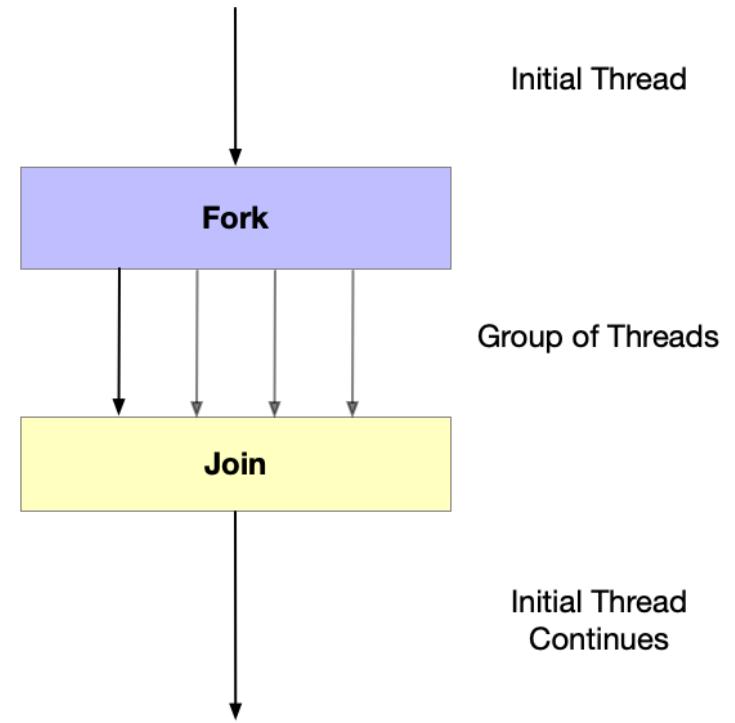
- There are many factors to consider when structuring and designing parallel programs (which we will learn throughout this course) but to start here are a few important ones to consider
  - **Task Decomposition** - how to divide up a problem into subproblems (i.e., tasks) to be executed concurrently.
  - **Distribution of tasks** - how to structure parallel programs to efficiently assign tasks to threads and how this affects the scheduling of threads by the OS.
    - We will take a look at various ways to structure parallel programs based on how tasks are generated via an algorithm.
  - **Synchronization** - how tasks/threads communicate and coordinate in order to obtain deterministic results
  - **Data dependencies** - where one task's output is required to be the input for another task.
  - **Scalability** - as the program is given more compute resources (i.e., better parallel hardware) its performance also increases.

# Task Decomposition and Granularity

- When thinking about breaking a problem down into tasks you need to consider the **granularity** (or **grain size**) of a task
  - Granularity - a measure of the amount of work (or computation) which is performed by that task
  - Also can be thought as qualitative measure of the ratio of computation (time to complete the task) to communication (time needed to exchange data between processors for a task).
  - We will talk more about granularity later in the course.
- For now, it's best on shared memory systems to have many small tasks that are evenly distributed across all processors.

# Fork-join Pattern

- New parallel control flows are created by spawning (i.e., also known as **forking**) new threads and are terminated at a **join**, where the parallel control flows meet and a single flow continues onwards.
  - Basis for many of the parallel patterns we will see throughout this class.
  - Each spawned thread can then work on one or more tasks.



# Basic Fork-join Pattern

- Generic pseudocode

Non-recursive

```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solveTask(part)
        join all subtasks spawned in previous loop
        return combined results
```

Recursive  
(Divide and  
conquer algorithms)

```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solve(part)
        join all subtasks spawned in previous loop
        return combined results
```

Pseudocode Source: [Doug Lea](#) (2000). [A Java fork/join framework](#). ACM Conference on Java.

# Forking in Go

- In Go, each concurrently executing activity/task is called a **goroutine**:
  - A goroutine is similar to a thread in other languages/operating systems but it's not actually a thread.
  - At program startup, the only goroutine is the one that calls the main function, which is known as the **main goroutine**.
- New goroutines are created by the **go** statement:
  - Is an ordinary function/method call prefixed by the keyword **go**.
  - Causes the function being called to execute concurrently by a goroutine
  - The goroutine forking the new goroutine returns immediately after the go statement

`f() // call f(); wait for it to return`

`go f() // create a new goroutine that falls f(); don't wait`

# Fork-join Pattern

- In Go, using a Waitgroup defined in the sync package allows us to implement the fork-join pattern
  - Methods:

```
type WaitGroup
```

```
func (wg *WaitGroup) Add(delta int)
func (wg *WaitGroup) Done()
func (wg *WaitGroup) Wait() // join point
```

- As stated in the documentation
  - “A *WaitGroup* waits for a collection of goroutines to finish. The main goroutine calls *Add* to set the number of goroutines to wait for. Then each of the goroutines runs and calls *Done* when finished. At the same time, *Wait* can be used to block until all goroutines have finished.”

# Demo: Fork-join in Go

## m2/accounts0

# Race Conditions

- Shared-memory systems use **shared variables/resources** (i.e., memory locations) which can be accessed by all processors.
- Communication and cooperation between the processors is organized by writing and reading shared variables that are stored in memory.
- A **race condition** is non-deterministic behavior in a parallel program when the result of an operation depends on the interleaving of certain individual operations.
  - Code with a race condition can run deterministically sometimes and fail other times.
  - Hard to reproduce and diagnose because they can appear frequently.
  - Only present sometimes under heavy load or when using certain compilers, platforms, architectures
  - A specific type of race condition is a **data race**, which is a situation when at least two threads access a shared variable at the same time. One thread must try to modify the variable.
    - This is normally due to not using proper synchronization when accessing the shared variable.
    - Races can also happen with files and I/O (e.g., printing to the screen) too.

# Race Conditions and Critical Sections

- Examples of race conditions

Thread 1	Thread 2
$a := x$	$b := x$
$a += 1$	$b += 1$
$x = a$	$x = b$

Data race to update the value of  $x$

Thread 1	Thread 2
$x = 1$	$y = 1$
$a = y$	$b = x$

Race not explained for serial interleaving  
Assume  $x$  and  $y$  are initially zero.

- To ensure determinism and to avoid race conditions, you need to determine **critical sections** in your code:
  - A critical section is a block of code where potentially more than one thread can execute the code at the same time. This potentially where shared resources are accessed/modified.
  - The execution of code in a critical section should, effectively, be executed as serial code.
  - Eventually, another thread should be able to access this section once one thread has completed the critical section.

# Synchronization using Atomics

- Synchronization is needed when dependencies exist between parallel tasks and/or to handle race conditions
- Many of the low-level synchronization primitives (e.g., locks, monitors, etc.) are built off of specialized hardware primitives/instructions (also known as **atomic** operations):
  - On a shared memory system, an operation is consider **atomic** if it completes in a single step relative to other threads.
  - No other thread can observe the modification to that shared variable half-way through its operation.
  - Provided in Go sync package: **import** “sync/atomic”

# **Demo: Fork-join in Go with correct synchronization**

## **m2/accounts1**

# Problems with Atomic Operations

- Best practices is to use atomic operations sparingly because:
  - Most atomic operations are implemented using CAS or (LL/SC), which take significant more cycles to complete than a simple load or store instruction.
  - Causes a memory fence, which forces the write-back buffer to be sent to main memory. This process can then stall other processors from reading/writing to main memory.
  - Prevents out-of-order execution and various compiler optimizations.
  - Cost to performance varies depending on architectures, program design, etc.
  - Adds more hardware complexity.
- In general, best practice for all synchronization is to use them only at that point where it is necessary.
  - Too little synchronization leads to non-deterministic behavior.
  - Too much synchronization can limit scaling and/or lead to deadlock (next week).

# Instruction Level Parallelism in Hardware

# Processor Cycle

- For multicore architectures, the basic unit of time is a **cycle**:
  - The time it takes a processor to fetch and execute a single instruction
  - As technology advances, cycle times change
    - 1980: 10 million cycles/sec
    - 2005: A 3 GHz processor does 3 billion cycles/sec
    - Some instructions take one cycle, and some may take hundreds.

# Instruction Level Parallelism

- Modern hardware architectures have a multi-core architecture, where both serial and parallel programs have benefited from **instruction level parallelism(ILP)**:
  - Simultaneous execution of a sequence of instructions
- There exists various techniques to exploit ILP (here are a few)
  - **Instruction Pipelining** - Uses functional units like ALUS (arithmetic logical units), FPUs(floating point unit), load/store units, or branch units to execute independent instructions in parallel by different functional units.
  - **Out-of-order execution** - instructions execute in any order that does not violate data dependencies
  - **Speculative execution & Branch prediction** - Processors can execute instructions speculatively before branches or data have been computed.
- Processors that allows this instruction level parallelism are known as **superscalar processors**.

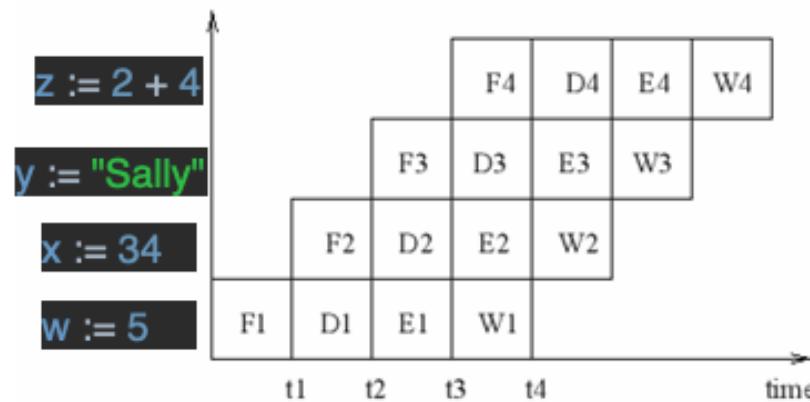
All of these techniques discussed are handled by compilers and/or hardware, which make sequential and parallel programs more efficient and have better performance.

# Instruction Pipelining

Simple example of pipeline stages:

- fetch: Retrieve the next instruction to be executed from memory
- decode: decode the instruction fetched in step (1).
- execute: load the operands specified and execute the instruction
- write back: write the result into the target location.

```
func main() {  
    w := 5  
    x := 34  
    y := "Sally"  
    z := 2 + 4  
    //...  
}
```



- Main benefit: Different pipeline stages can operate in parallel, if there are no control or data dependencies between the instructions to be executed.

# Improving Data Access Performance

# Processors & Memory

On architectural principle drives everything else: processors and main memory are far apart.

- Takes a long time to read a value from memory
- Takes a long time for a processor to write a value to memory
- Takes a longer time for the processor to verify that the value written is installed in memory.
- The relative cost of instructions such as memory access changes slowly when expressed in terms of cycles.
- **Analogy:** Accessing memory is more like mailing a letter than making a phone call.

Memory access time has a large influence on program performance. The objective of architecture trends over the years have been to reduce memory access **latency**:

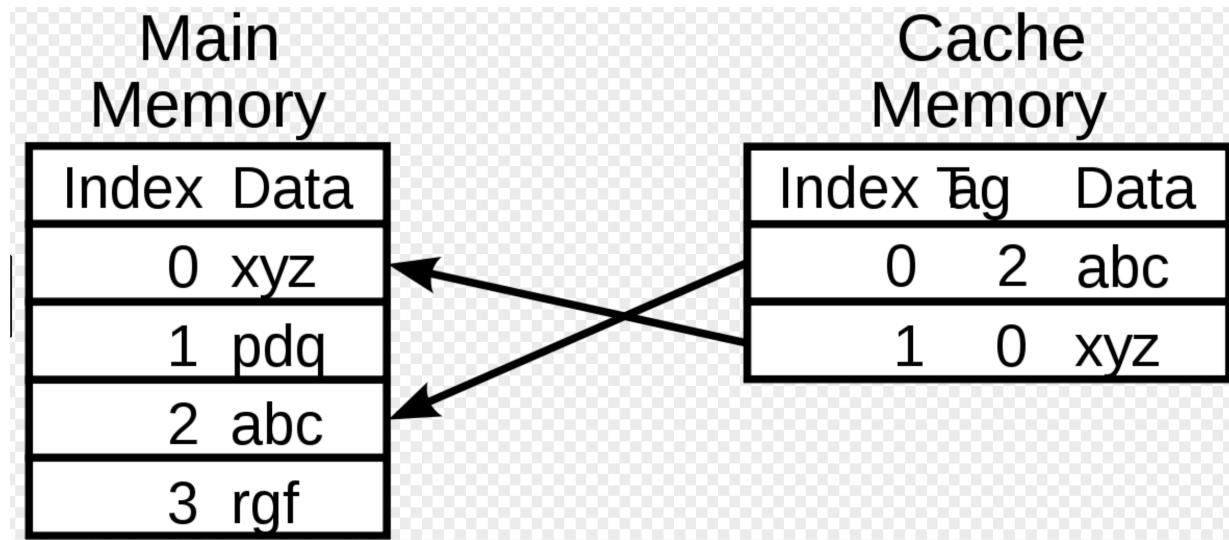
- The total time that elapses until a memory access operation has been completely terminated.

# Processors & Memory

- To help alleviate these memory latency issues, memory inside modern computers is actually a hierarchy of components that store data.
- Ranges from very few registers, to one or more levels of small, fast caches to relatively slow main memory.
- **Understanding how these levels interact is essential to understanding the actual performance of many concurrent algorithms.**

# Caches

- On modern architectures, processors can waste hundreds of CPU cycles waiting to access main memory.
- We can alleviate problem by using **caches**:
  - A collection of memory locations that can be accessed in less time than some other memory locations.
  - A cache is typically located on the same chip as the processors
  - **Cache line**: fixed-size block of data that also contains metadata (e.g., tag, index)
  - Cache lines are normally 64 or 128 bytes



# Principle of locality

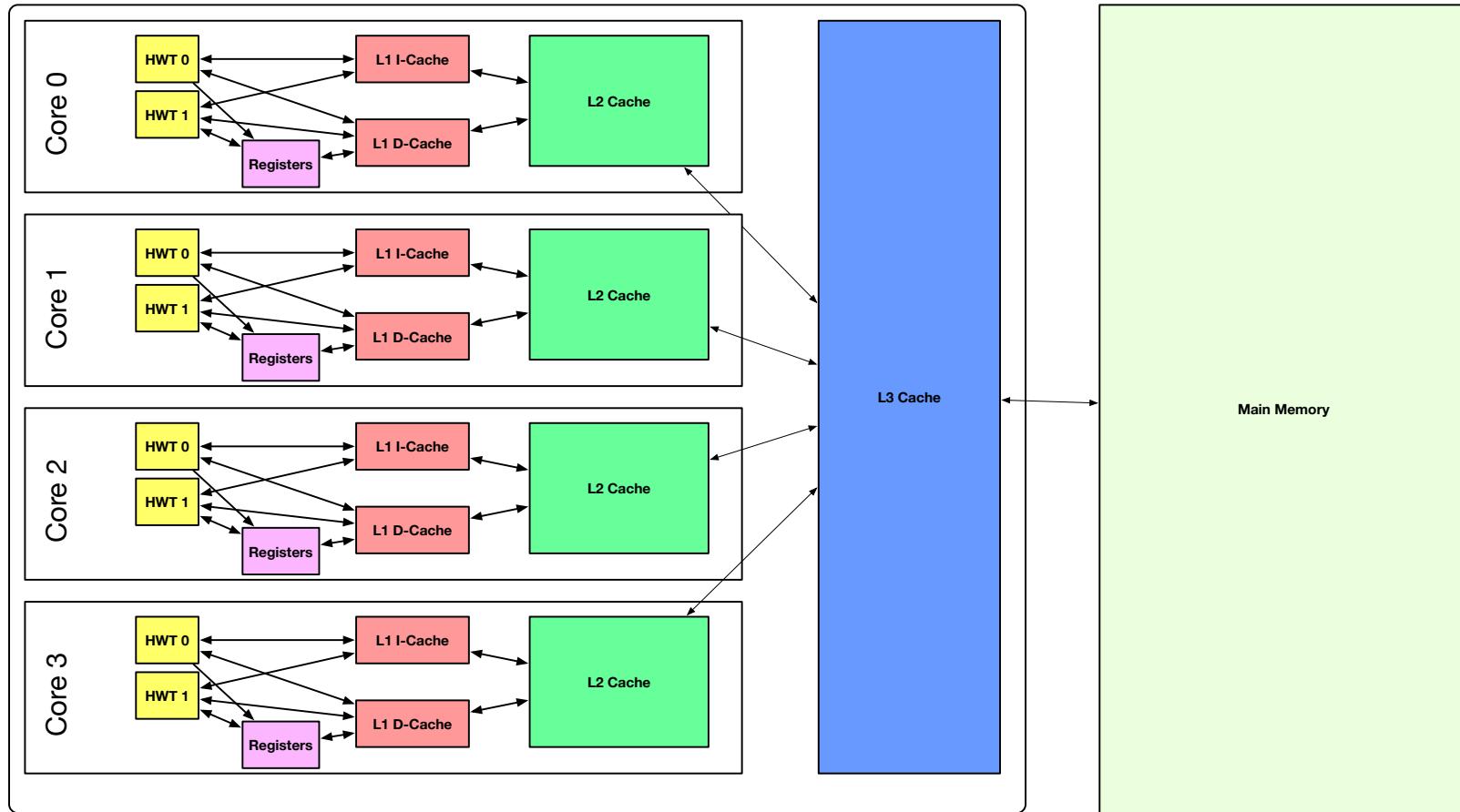
- Caches are effective because most programs display a high degree of **locality**:
  - Accessing one location is followed by an access of a nearby location.
  - Types of locality
    - **Spatial locality** – accessing a nearby location.
    - **Temporal locality** – accessing in the near future.

# Cache Levels

- Most processors have two levels of caches, called L1, L2, and L3:
  - L1 - typically resides on the same chip as the processor and takes 1-2 cycles to access
  - L2 - may reside either on or off-chip, and takes about 10 cycles to access
  - L3 - normally is off chip but is accessible within about 30 cycles
- Note: These times vary from platform to platform. All these are significantly faster to access than main memory, which can be 100s of cycles.

# Core i7-9xx Cache Hierarchy

Core i7 Multiprocessor



L1 i-cache and d-cache: 32KB, 8-way, **Access:** 4 cycles

Cache-Line (Block) size: 64 bytes  
(All caches)

L2 cache: 256KB, 8-way, **Access:** 11 cycles

L3 cache: 8MB, 16-way, **Access:** 30-40 cycles

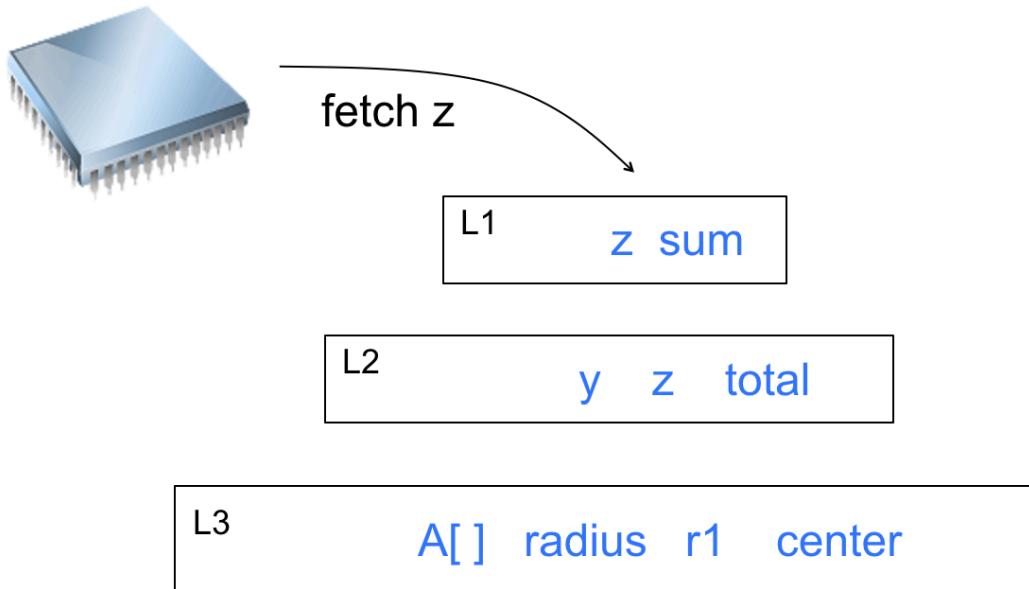
# Principle of locality

- What could the caches levels look like for this program?

```
package main
import "fmt"
func main() {
    var z [1000]int
    var sum int
    for i := 0; i < 1000; i++ {
        sum += z[i]
    }
    fmt.Printf("%v\n", sum)
}
```

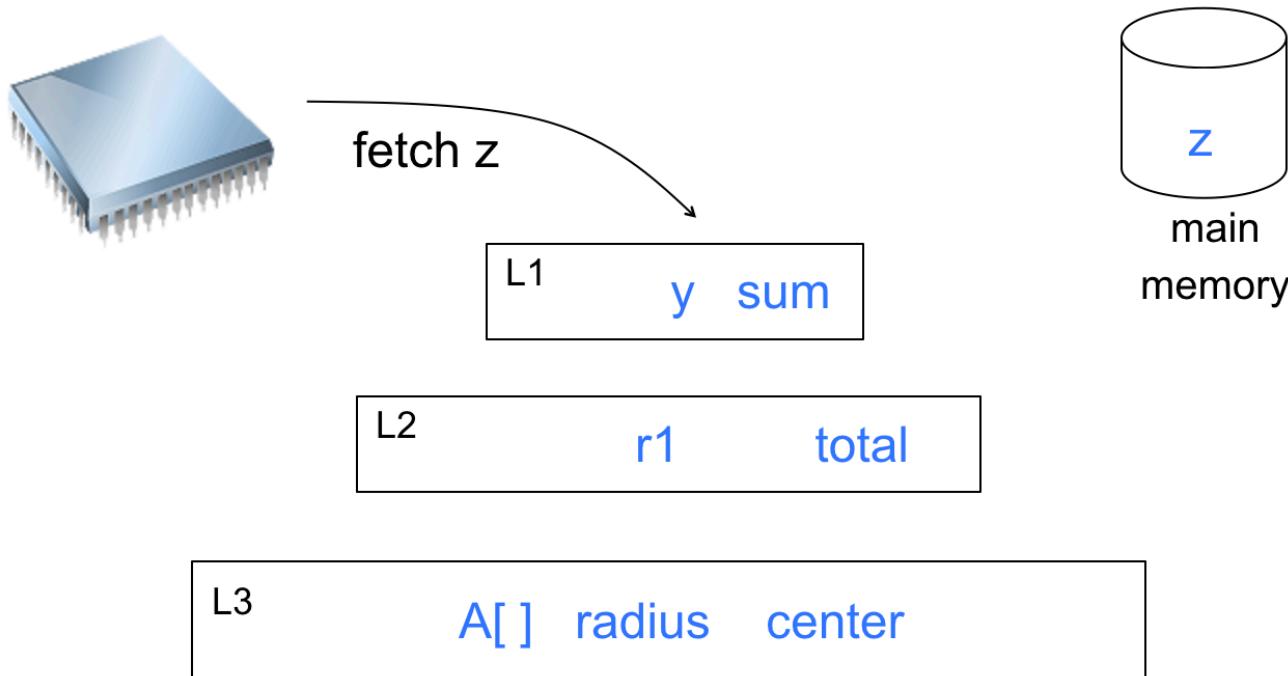
# Cache hit

- When a processor attempts to read a value from a given memory address, it first checks the cache(s).
- We call it a **cache hit** if the value for the processor is located in one of the cache levels.



# Cache Miss

- Otherwise, a **cache miss** is when a value is not in the cache and the process is required to go to main memory.



# When a Cache Becomes Full...

- Caches are expensive to build and therefore significantly smaller than main memory.
- Need to make room for new entry when the cache is full by evicting an existing entry:
  - Discarding a entry if it has not been modified
  - Writing it back to main memory if it has been modified
- Need a **replacement policy**(determines which cache line to replace to make room for a new location).
  - Usually some kind of least recently used heuristic.

# Cache Coherence

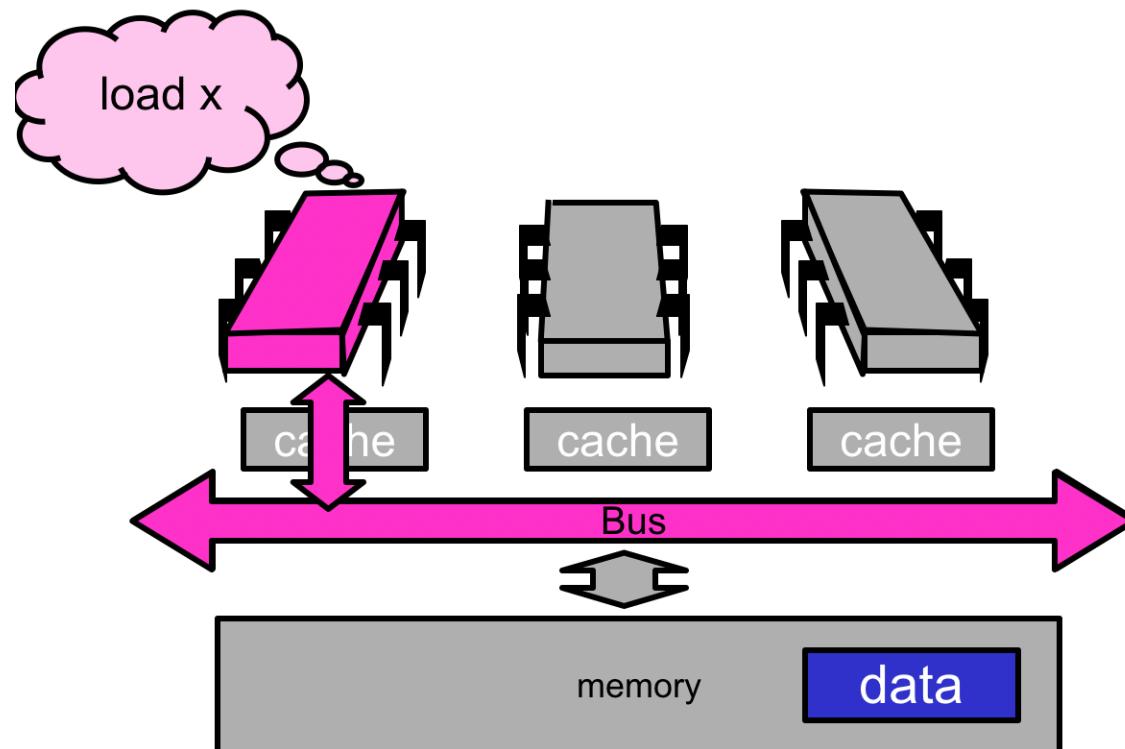
- Processor **A** and Processor **B** both cache an address  $x$
- Processor **A** writes to  $x$
- This operation updates the cache
- How does Processor **B** find out about the update?
- A cache coherence protocols provides a specification on how to keep caches in sync with each other. Many cache coherence protocols in the literature.

# MESI

- MESI (pronounced “messy”) is one of the most commonly used cache coherence protocols. Provides four states that a cache line can be in:
  - **Modified(M)** - Modified, A processor has modified cached data, must write back to memory
  - **Exclusive(E)** - Not modified, Only one processor has a copy of a main memory data in a cache line.
  - **Shared(S)** - Shared Not modified, a piece of data from main memory may be in different caches.
  - **Invalid(I)** - Cache line contents not meaningful

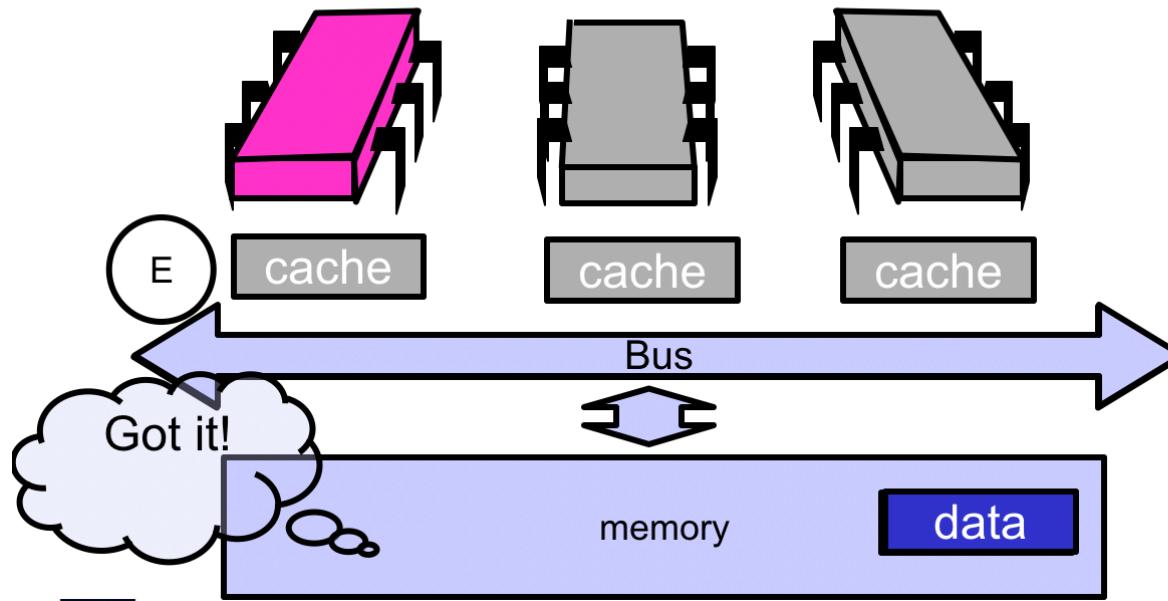
# MESI Example

- A processor issues load request at address x from main memory.



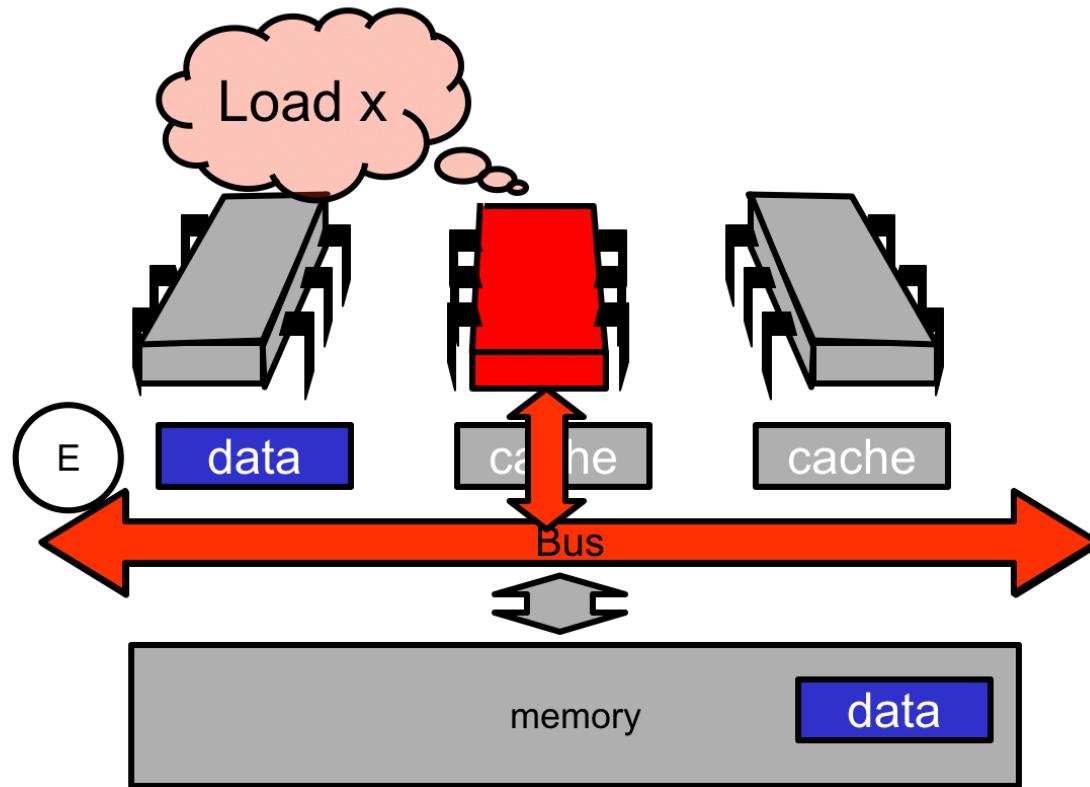
# MESI Example

- Main memory responds with the requested data and the processor places the data in a cache line. The data is exclusive to the cache of the requesting processor.



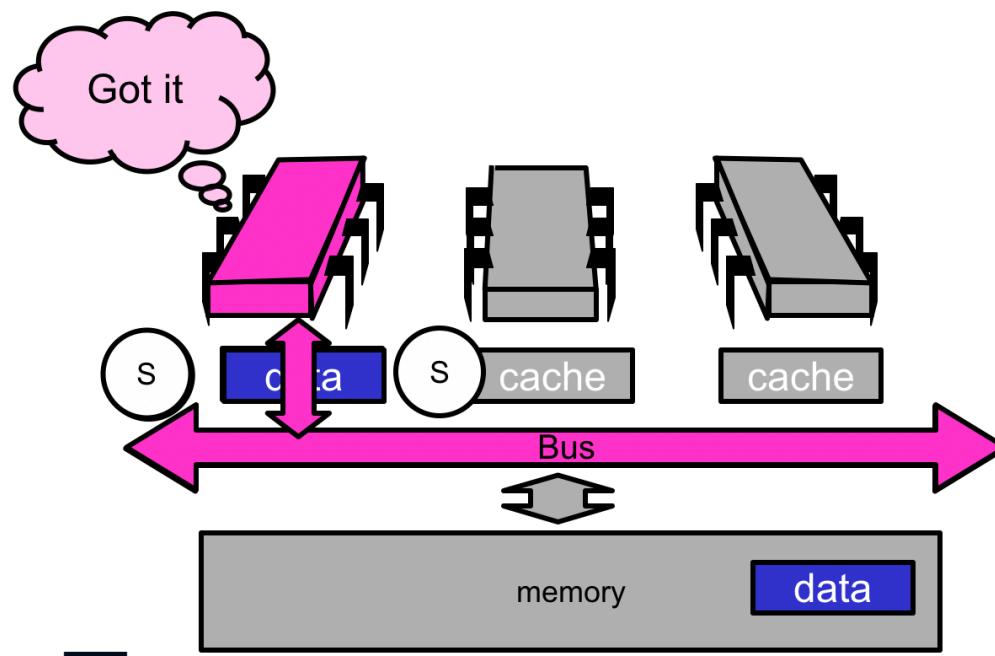
# MESI Example

- Another processor requests the data from the same address  $x$  from main memory.



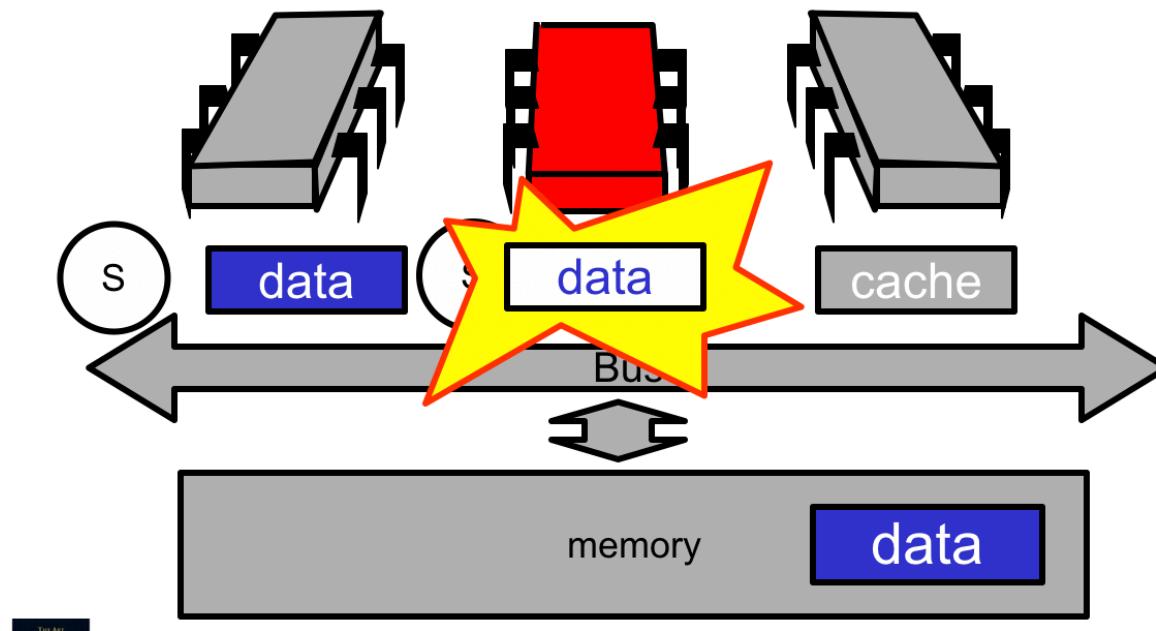
# MESI Example

- Processors communicate with each other to update their states to shared since both contain data from the same memory address.



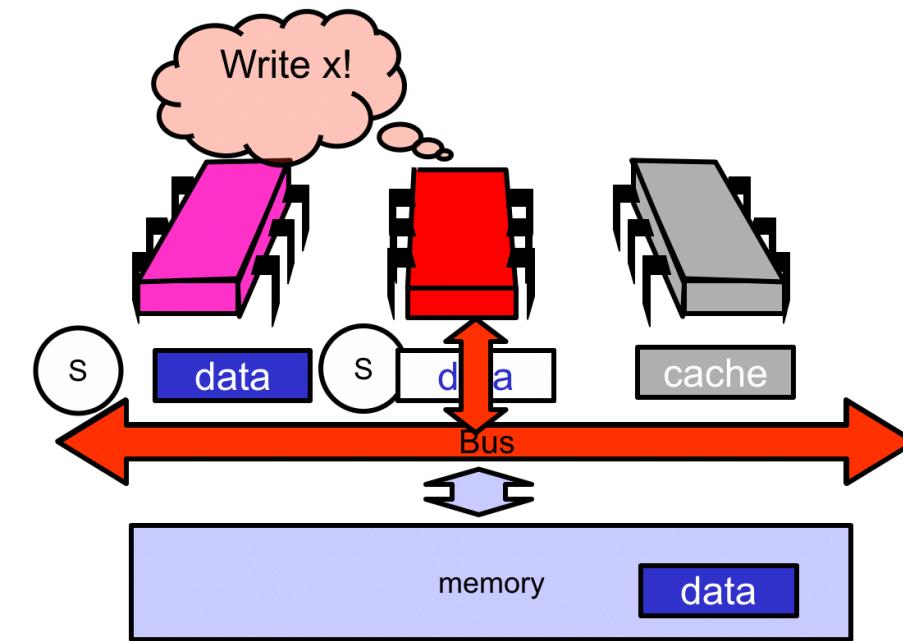
# MESI Example

- A processor modifies the data for that memory address in its cache. Other processors need to be notified about this update.



# MESI Example

- All other processors are given the updated data and depending on the write scheme for the cache the data is either immediately sent to main memory or held for a period of time before being sent to it.



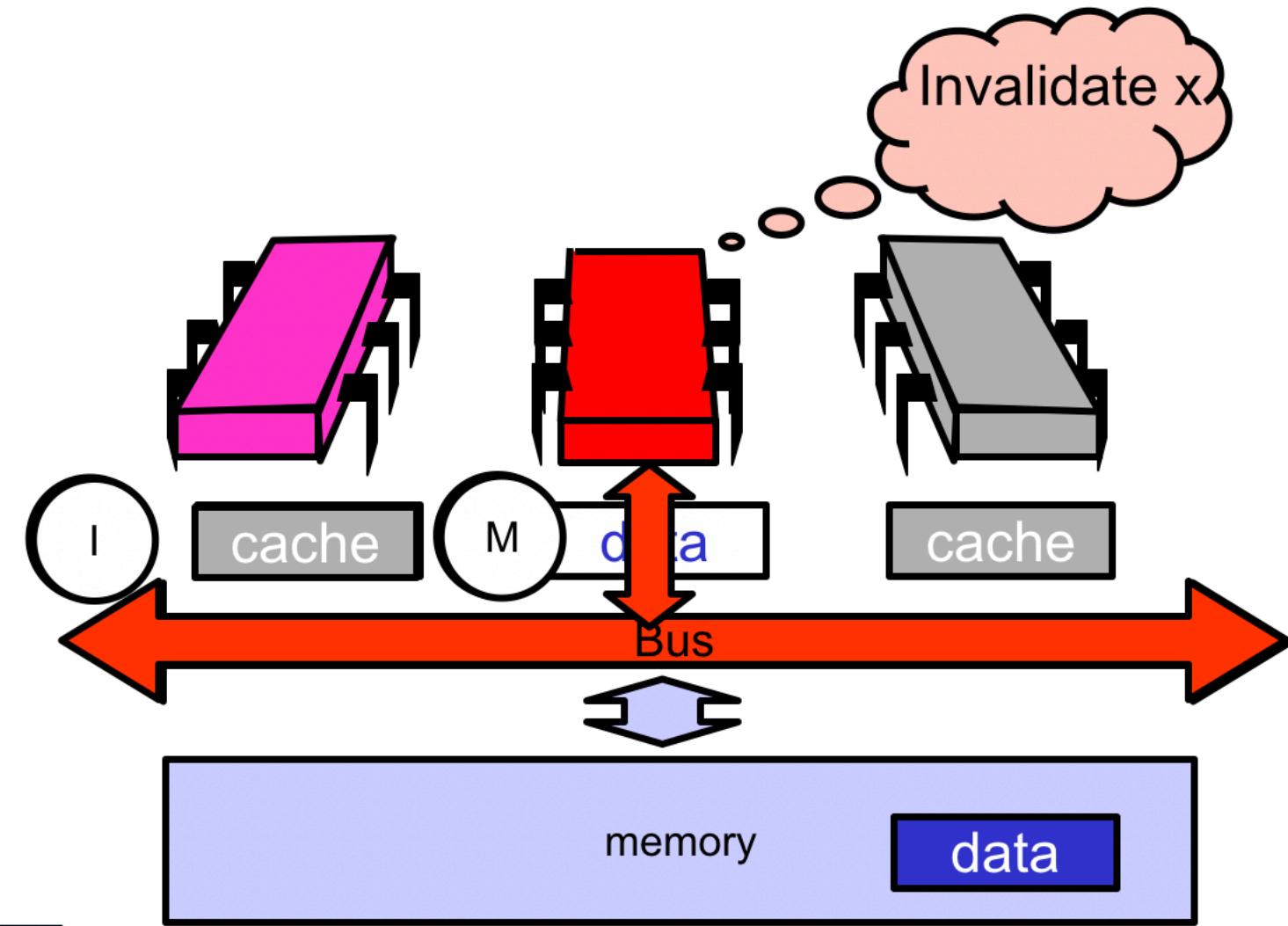
# Write-Through Caches

- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
    - For example, loop indices ...
- Hardly used in practice due to the bus traffic problem

# Write-Back Caches

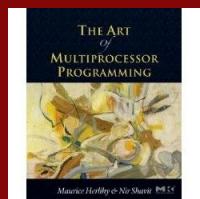
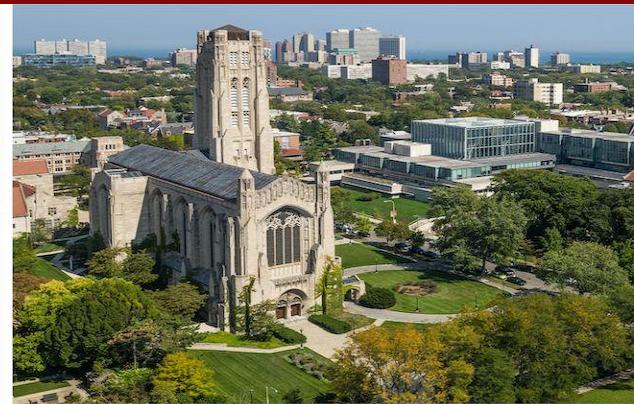
- Immediately broadcast changes
- Caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.
- The dirty data is usually held in a **write-buffer** that will eventually write its contents back to main memory periodically.

# MESI - Invalidate



# MPCS 52060 - Parallel Programming

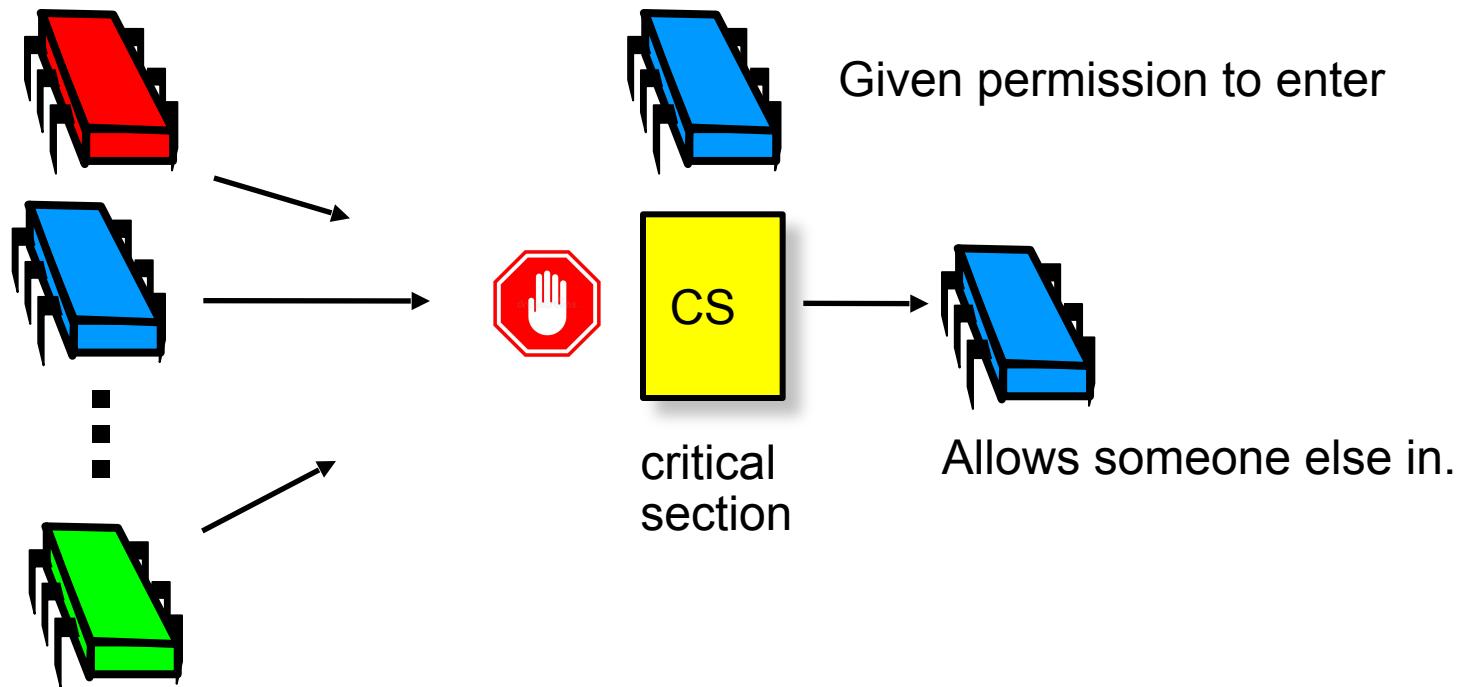
## M3: Principles of Mutual Exclusion



Original slides from “The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit” With modifications by Lamont Samuels

# M3 Objective

- How we are going to ensure determinism when threads want to enter in the critical section?



# Principles of Mutual Exclusion

# Formalizing Concurrent Computation

Two types of formal properties in asynchronous computation:

- **Safety Properties:**
  - Nothing bad happens ever
  - For example - a traffic light never displays green in all directions, even if power fails.
- **Liveness Properties:**
  - States that a particular “good” thing will happen.
  - For example - a red light will eventually turn green.

# Formalizing Critical Sections

- Synchronization primitives need to adhere to the following properties and principles about critical sections in order to be correct:
  - **Mutual Exclusion Property:**
    - Critical sections of different threads do not overlap. Only one thread is executing a critical section at a time
    - Guarantees that a computation's results are correct.
    - This is a safety property.
  - **Deadlock-freedom property:**
    - If multiple threads simultaneously request to enter a critical section, then it must allow one to proceed
    - Threads outside the critical section have no say in which thread can proceed into the critical section, only those currently waiting have influence.
    - It implies the system never “freezes”.
    - This is a liveness property.

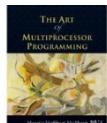
# Formalizing Critical Sections

- **Starvation-freedom property:**
  - Every thread that attempts to acquire the lock eventually succeeds.
  - Many mutual exclusive algorithms in practice are not starvation free because its less likely starvation will occur in those algorithms.
  - There is no guarantee on how long thread will wait to acquire the lock.
  - Also known as lockout freedom or bounded-waiting
  - This is a liveness property.
- **Fairness Principle:**
  - A thread who just left the critical section cannot immediately re-enter the critical section if other threads have already requested to enter the critical section.
  - Some algorithms place bounds on how long a thread can wait.

# Implementing Locks

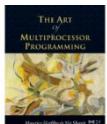
## Two-Thread vs $n$ -Thread Solutions

- 2-thread solutions first
  - Illustrate most basic ideas
  - Fits on one slide
- Then  $n$ -thread solution
- You will never use these protocols
  - Get over it
- You are advised to understand them
  - The same issues show up everywhere
  - Except hidden and more complex



# Two-Thread Conventions

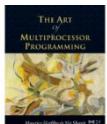
```
class ... implements Lock {  
    ...  
    // thread-local index, 0 or 1  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        ...  
    }  
}
```



# Two-Thread Conventions

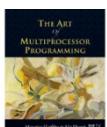
```
class ... implements Lock {  
    ...  
    // thread-local index, 0 or 1  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        ...  
    }  
}
```

Henceforth: i is current  
thread, j is other thread



# LockOne

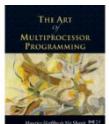
```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
  
    public void unlock() {  
        int i = ThreadID.get();  
        flag[i] = false;  
    }  
}
```



# LockOne

```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

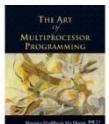
Each thread has flag



# LockOne

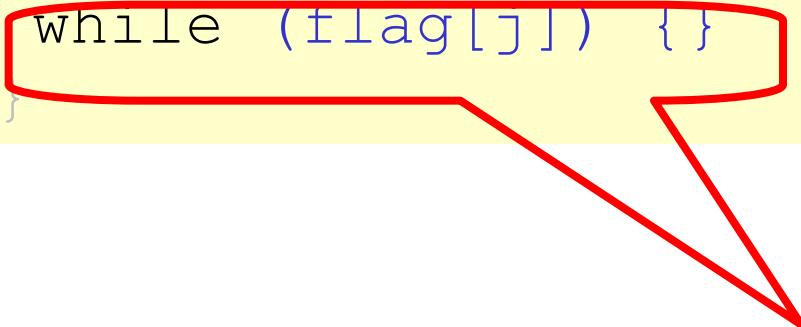
```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (!flag[j]) {}  
    }  
}
```

Set my flag

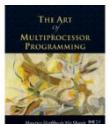


# LockOne

```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```



Wait for other flag to become  
false

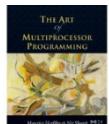


# Deadlock Freedom

- LockOne Fails deadlock-freedom
  - Concurrent execution can deadlock

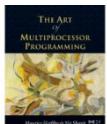
```
flag[i] = true;      flag[j] = true;  
while (flag[j]) {}  while (flag[i]) {}
```

- Sequential executions OK



# LockTwo

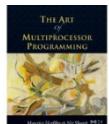
```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```



# LockTwo

```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i; // Red box highlights this line  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

Let other go first

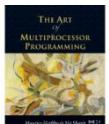


# LockTwo

```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
}
```

Wait for  
permission

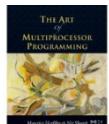
```
    public void unlock() {}  
}
```



# LockTwo

```
public class Lock2 implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
    public void unlock() {}  
}
```

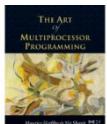
Nothing to do



# LockTwo Claims

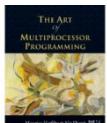
- Satisfies mutual exclusion
  - If thread **i** in critical section
  - Then **victim == j**
  - Cannot be both 0 and 1
- Not deadlock free
  - Sequential execution deadlocks
  - Concurrent execution does not

```
public void LockTwo() {  
    victim = i;  
    while (victim == i) {};  
}
```



# Peterson's Algorithm

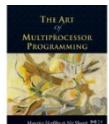
```
public void lock() {  
    int i = ThreadID.get();  
    int j = 1 - i;  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};  
}  
public void unlock() {  
    flag[i] = false;  
}
```



# Peterson's Algorithm

```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};  
}  
  
public void unlock() {  
    flag[i] = false;  
}
```

Announce I'm  
interested

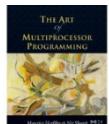


# Peterson's Algorithm

```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};  
}  
public void unlock() {  
    flag[i] = false;  
}
```

Announce I'm interested

Defer to other



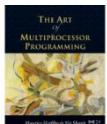
# Peterson's Algorithm

```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};  
}  
  
public void unlock() {  
    flag[i] = false;  
}
```

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim



# Peterson's Algorithm

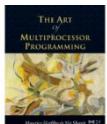
```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};  
}  
  
public void unlock() {  
    flag[i] = false;  
}
```

Announce I'm interested

Defer to other

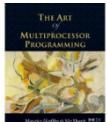
Wait while other interested & I'm the victim

No longer interested



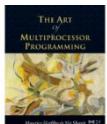
# Peterson's Algorithm

- Satisfies mutual exclusion & deadlock freedom properties
  - Uses both lock-one and lock-two strategies
- Downside: only works for two threads



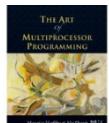
# Bakery Algorithm

- N-threaded locking algorithm
- Provides First-Come-First-Served
- How?
  - Take a “number”
  - Wait until lower numbers have been served
- Lexicographic order
  - $(a,i) > (b,j)$ 
    - If  $a > b$ , or  $a = b$  and  $i > j$



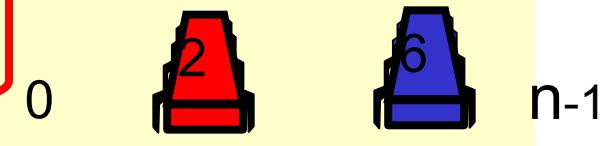
# Bakery Algorithm

```
class Bakery implements Lock {  
    boolean[] flag;  
    Label[] label;  
    public Bakery (int n) {  
        flag = new boolean[n];  
        label = new Label[n];  
        for (int i = 0; i < n; i++) {  
            flag[i] = false; label[i] = 0;  
        }  
    }  
    ...
```



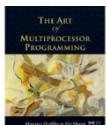
# Bakery Algorithm

```
class Bakery implements Lock {  
    boolean[] flag;  
    Label[] label;  
    public Bakery (int n) {  
        flag = new boolean[n];  
        label = new Label[n];  
        for (int i = 0; i < n; i++) {  
            flag[i] = false; label[i] = 0;  
        }  
    }  
    ...  
}
```



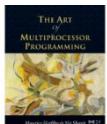
f	f	t	f	f	t	f	f
0	0	4	0	0	5	0	0

CS



# Bakery Algorithm

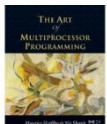
```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while (exists k | flag[k]  
               && (label[i], i) > (label[k], k));  
    }  
}
```



# Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1]) + 1;  
        while (exists k : flag[k]  
                && (label[i], i) > (label[k], k));  
    }  
}
```

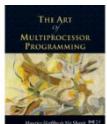
I'm interested



# Bakery Algorithm

Take increasing  
label (read labels in  
some arbitrary  
order)

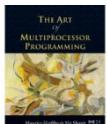
```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1]) + 1;  
        while (exists k such that flag[k]  
               && (label[i], i) > (label[k], k));  
    }  
}
```



# Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1]) + 1;  
        while (!k flag[k]  
               && (label[i], i) > (label[k], k));  
    }  
}
```

Someone is interested

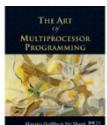


# Bakery Algorithm

```
class Bakery implements Lock {  
    boolean flag[n];  
    int label[n];  
  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while (exists k flag[k]  
               && (label[i], i) > (label[k], k));  
    }  
}
```

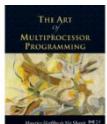
Someone is  
interested ...

... whose  $(label, i)$  in  
lexicographic order is lower



# Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
  
    public void unlock() {  
        flag[i] = false;  
    }  
}
```

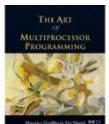


# Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void unlock() {  
        flag[i] = false;  
    }  
}
```

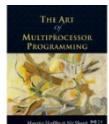
No longer interested

labels are always increasing



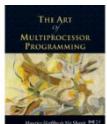
# Bakery Algorithm

- Has no deadlock and adheres to mutual exclusion property
- There is always one thread with earliest label
- Is there a problem with the labeling portion of the algorithm?



# Deep Philosophical Question

- The Bakery Algorithm is
  - Succinct,
  - Elegant, and
  - Fair.
- Q: So why isn't it practical?
- A: Well, you have to read  $N$  distinct variables



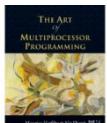
# Practical Implementation of Locks

# Mutex in Go

- Go has a package called sync that provides basic synchronization primitives such as mutual exclusion locks.
  - The sync package Go's provides mutual exclusion with `sync.Mutex` and its two methods:
  - `m.Lock()`: locks m. If the lock is already in use, the calling goroutine blocks until the mutex is available.
  - `m.Unlock()`: Unlock unlocks m. It is a run-time error if m is not locked on entry to Unlock.
- Now let's take a look at how locks are implemented behind the scenes.

# What Should you do if you can't get a lock?

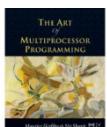
- Keep trying
  - “spin” or “busy-wait”
    - When spinning, a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
    - Can be very wasteful of CPU cycles.
    - Can also be unreliable if compiler optimization is turned on.
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor



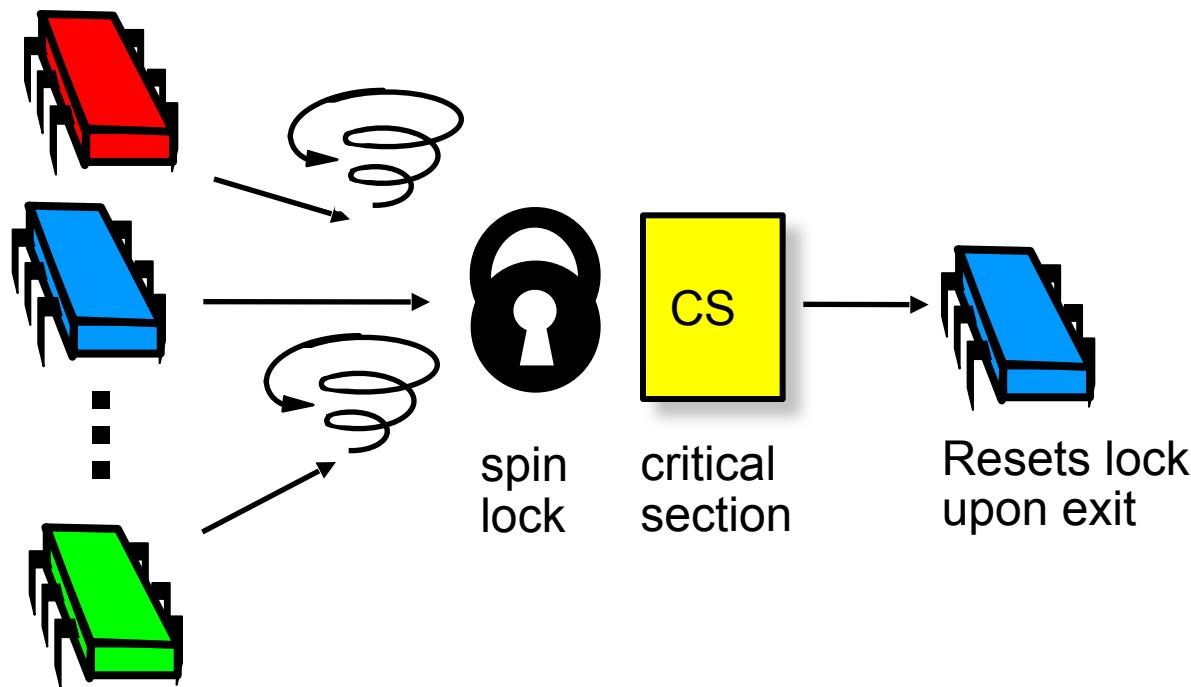
# What Should you do if you can't get a lock?

- Keep trying
  - “spin” or “busy-wait”
    - When spinning, a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
    - Can be very wasteful of CPU cycles.
    - Can also be unreliable if compiler optimization is turned on.
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

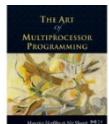
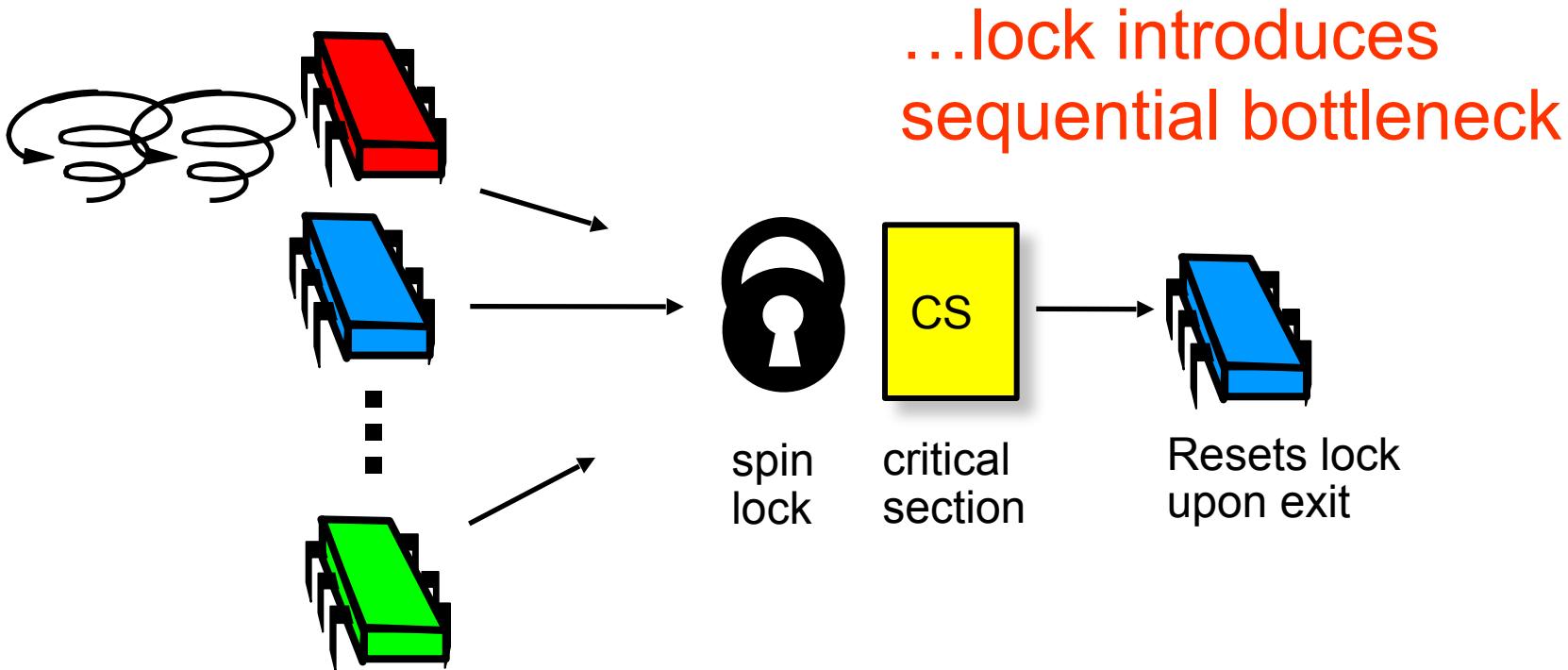
our focus



# Basic Spin-Lock

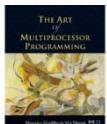
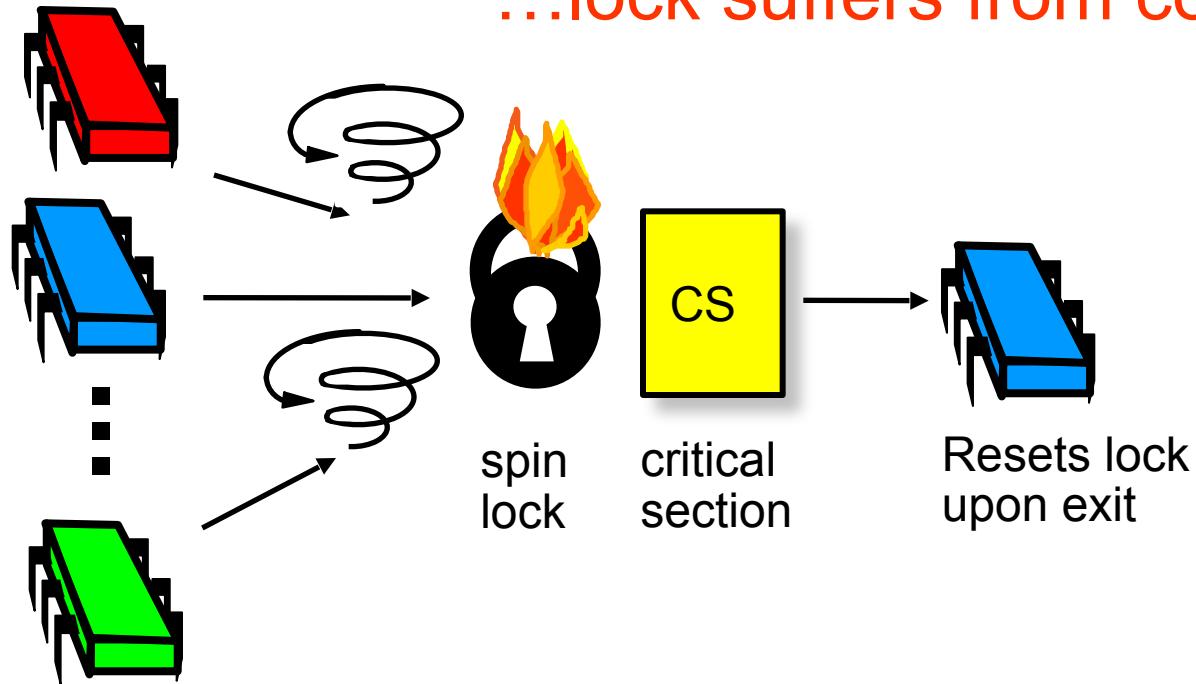


# Basic Spin-Lock



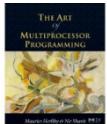
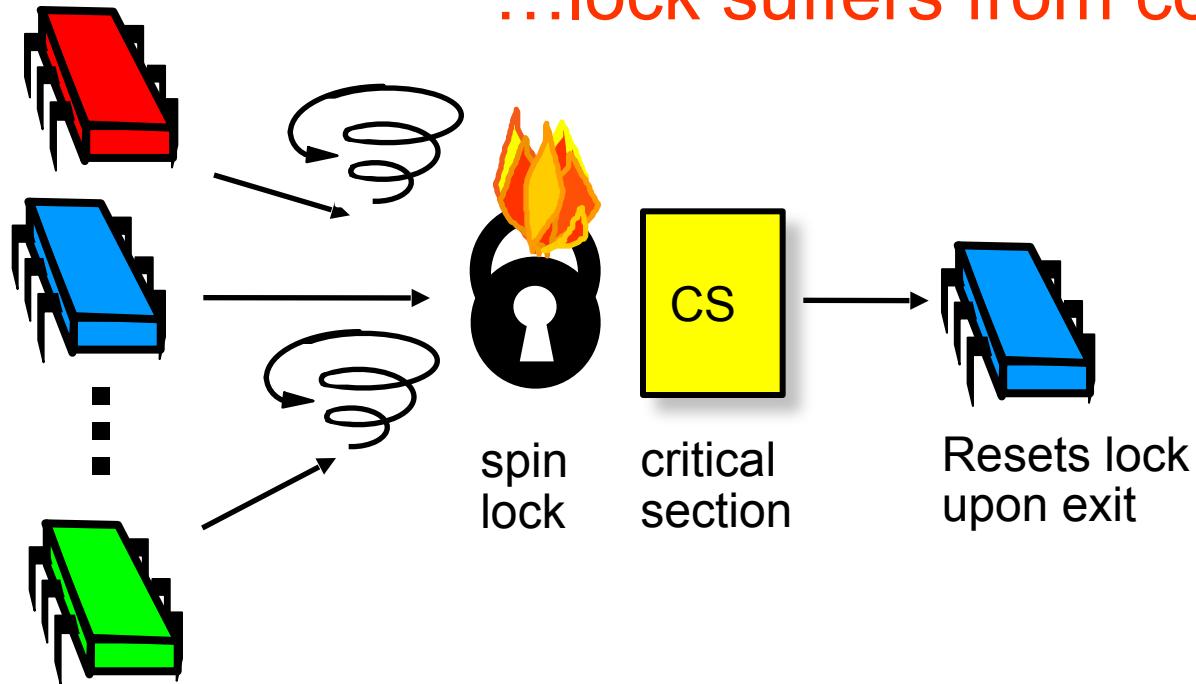
# Basic Spin-Lock

...lock suffers from contention



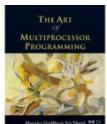
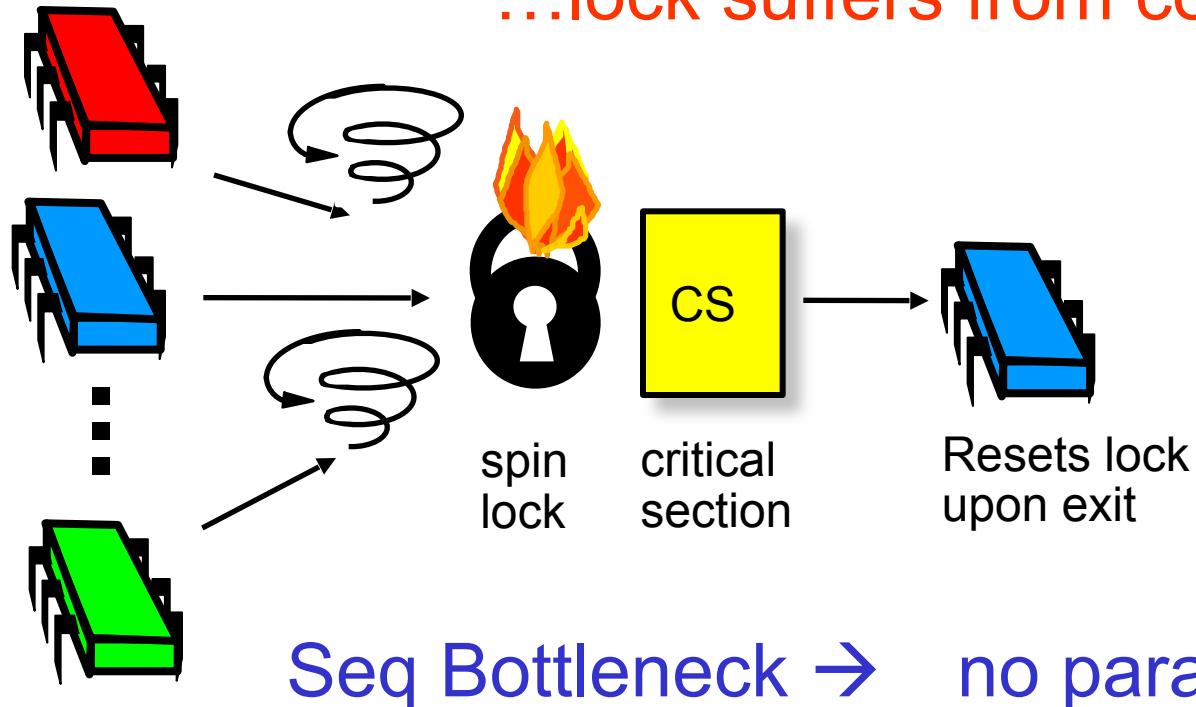
# Basic Spin-Lock

...lock suffers from contention



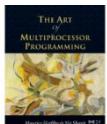
# Basic Spin-Lock

...lock suffers from contention



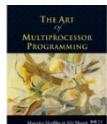
# Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap **true** with current value
  - Return value tells if prior value was **true** or **false**
- Can reset just by writing **false**
- TAS aka “getAndSet”



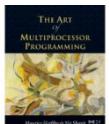
# Test-and-Set Locks

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false



# Test-and-Set

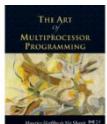
```
public class AtomicBoolean {  
    boolean value;  
  
    public synchronized boolean  
        getAndSet(boolean newValue) {  
        boolean prior = value;  
        value = newValue;  
        return prior;  
    }  
}
```



# Test-and-Set

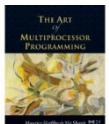
```
public class AtomicBoolean {  
    boolean value;  
  
    public synchronized boolean  
    getAndSet(boolean newValue) {  
        boolean prior = value;  
        value = newValue;  
        return prior;  
    }  
}
```

Swap old and new  
values



# Test-and-Set

```
AtomicBoolean lock  
= new AtomicBoolean(false)  
...  
boolean prior = lock.getAndSet(true)
```

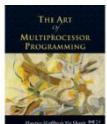


# Test-and-Set

```
AtomicBoolean lock  
= new AtomicBoolean(false)
```

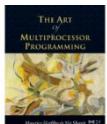
```
boolean prior = lock.getAndSet(true)
```

Swapping in true is called “test-and-set” or TAS



# Test-and-set Lock

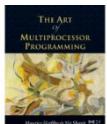
```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    } }
```



# Test-and-set Lock

```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.  
    } }
```

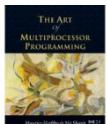
Lock state is AtomicBoolean



# Test-and-set Lock

```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    } } 
```

Keep trying until lock acquired



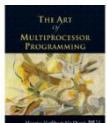
# Test-and-set Lock

```
class TASLock {
    AtomicBoolean state;
    new AtomicInteger(0);
}

void lock() {
    while (state.getAndSet(true)) { }
}

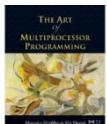
void unlock() {
    state.set(false);
}
```

Release lock by resetting  
state to false



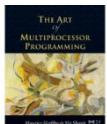
# Space Complexity

- TAS spin-lock has small “footprint”
- $N$  thread spin-lock uses  $O(1)$  space
- As opposed to  $O(n)$  Peterson/Bakery

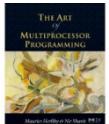
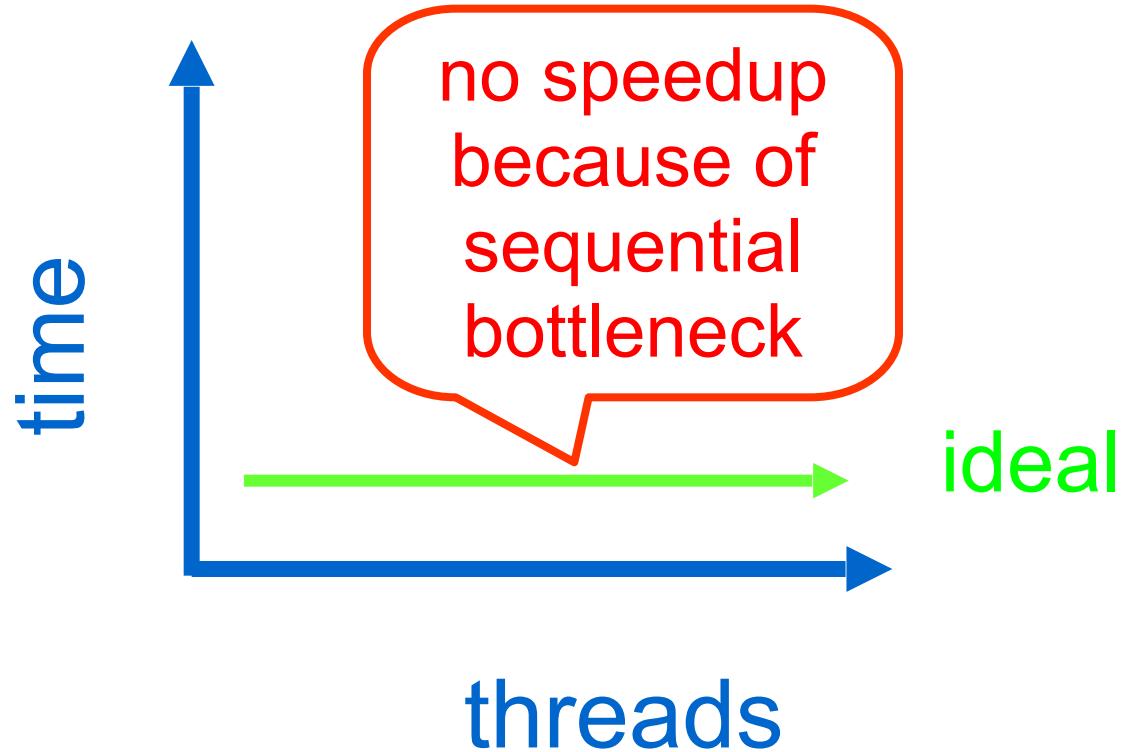


# Performance

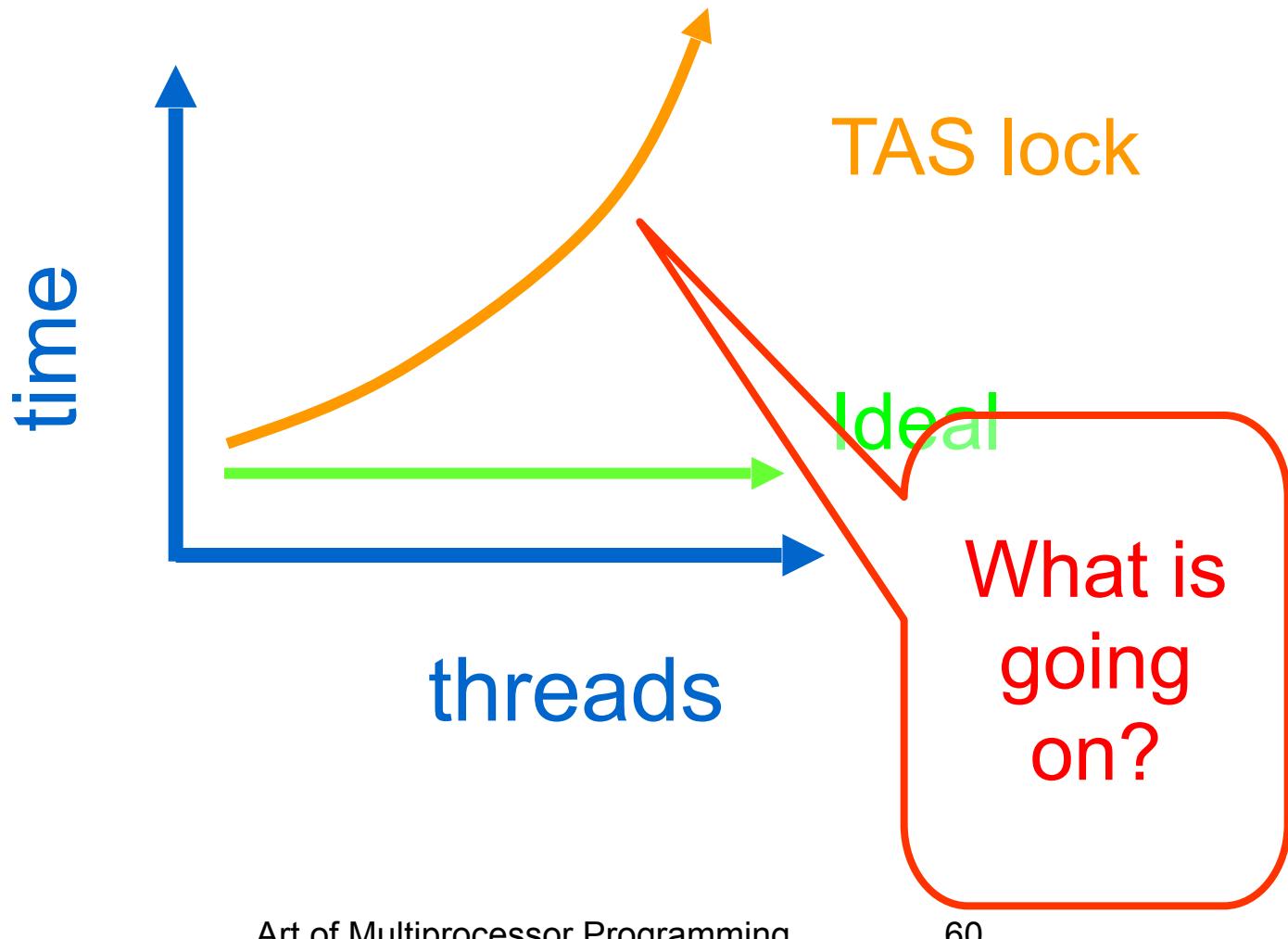
- Experiment
  - $n$  threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?



# Graph

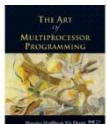


# Mystery #1



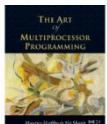
# Test-and-Test-and-Set Locks

- Lurking stage
  - Wait until lock “looks” free
  - Spin while read returns true (lock taken)
- Pouncing state
  - As soon as lock “looks” available
  - Read returns false (lock free)
  - Call TAS to acquire lock
  - If TAS loses, back to lurking



# Test-and-test-and-set Lock

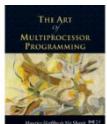
```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
        }  
    }  
}
```



# Test-and-test-and-set Lock

```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
        }  
    }  
}
```

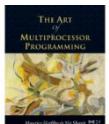
Wait until lock looks free



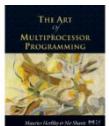
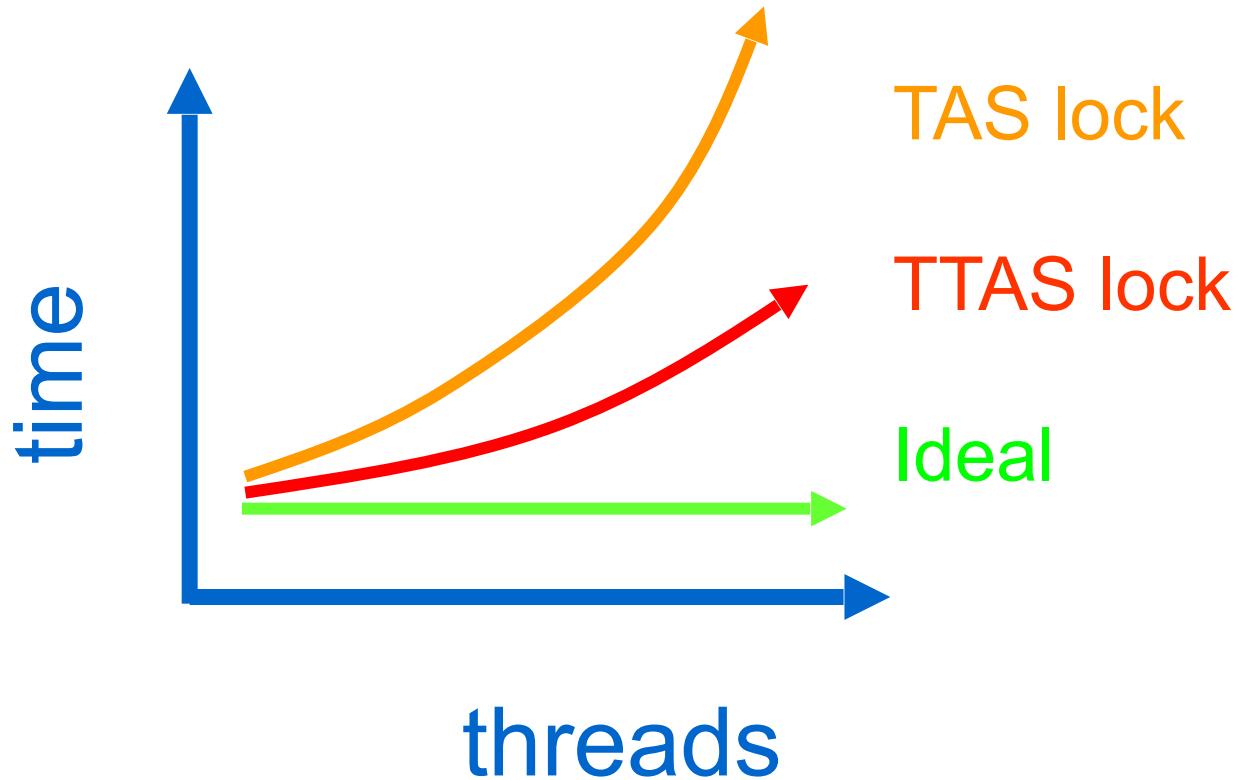
# Test-and-test-and-set Lock

```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
        }  
    }  
}
```

Then try to  
acquire it

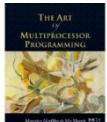


# Mystery #2



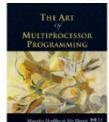
# Mystery

- Both
  - TAS and TTAS
  - Do the same thing (in our model)
- Except that
  - TTAS performs much better than TAS
  - Neither approaches ideal

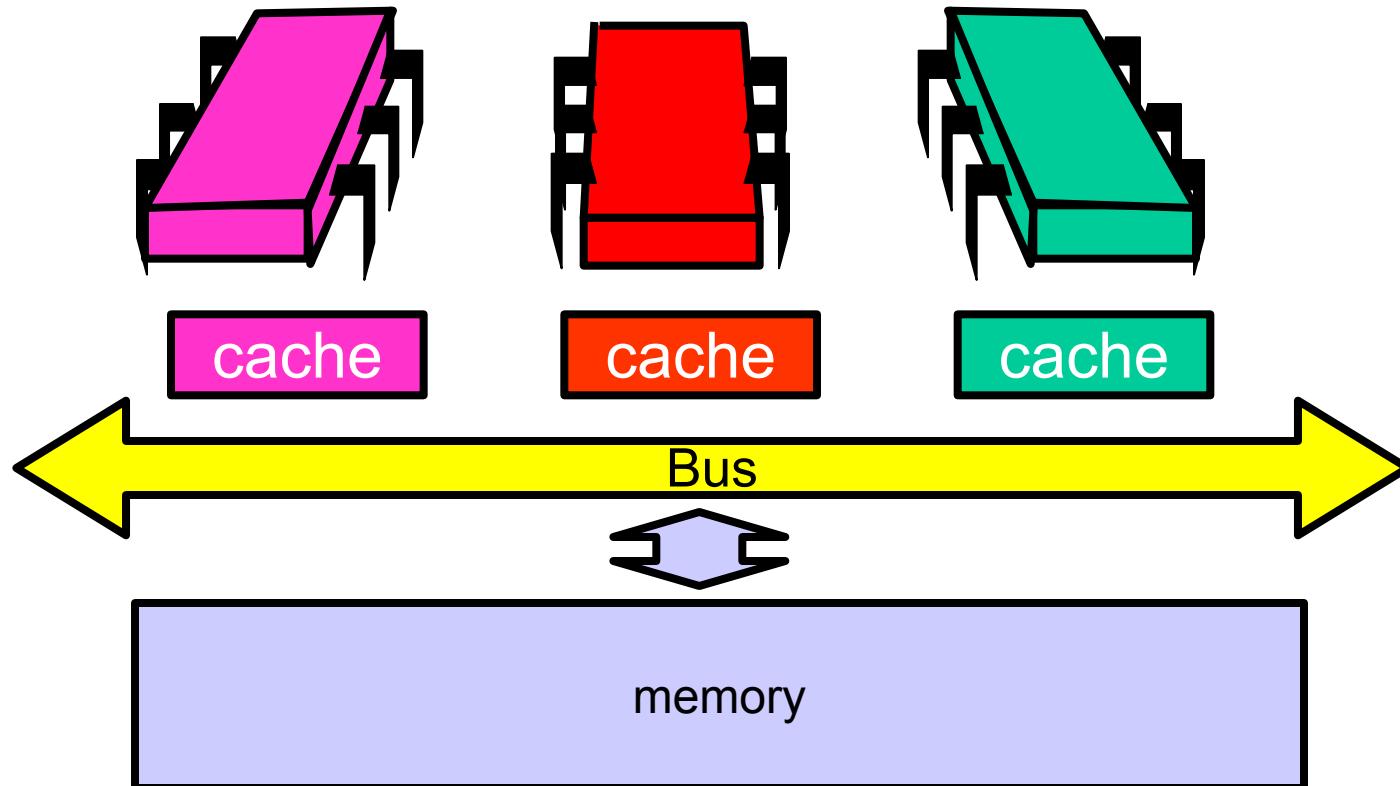


# Opinion

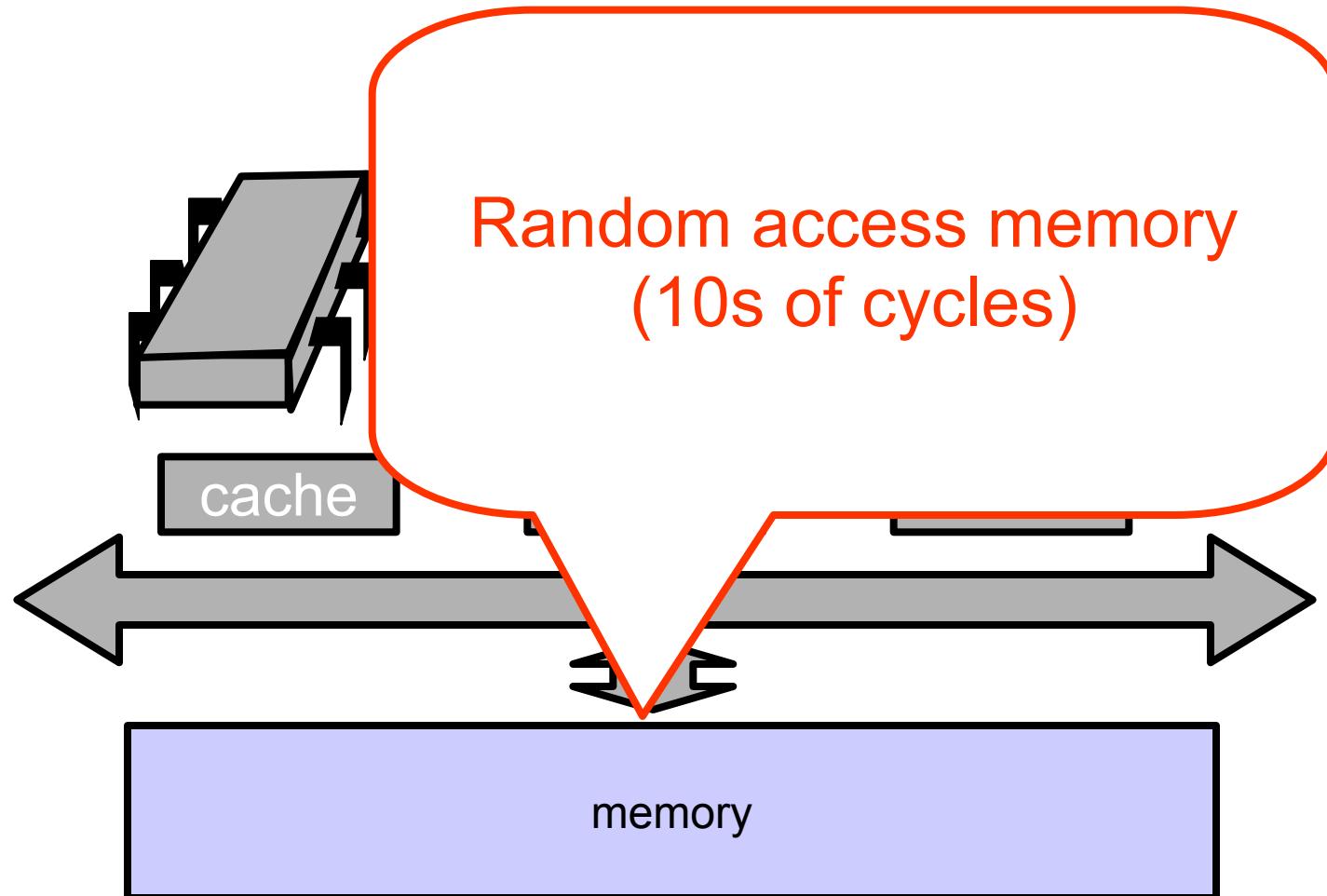
- Our memory abstraction is broken
- TAS & TTAS methods
  - Are provably the same (in our model)
  - Except they aren't (in field tests)
- Need a more detailed model ...



# Bus-Based Architectures



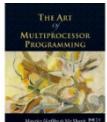
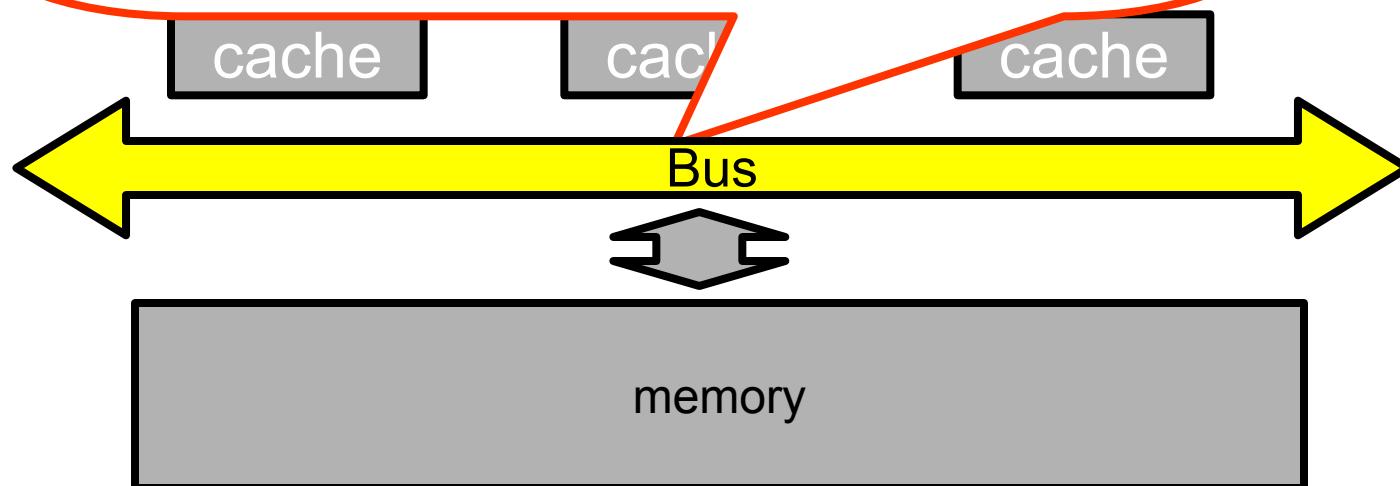
# Bus-Based Architectures



# Bus-Based Architectures

## Shared Bus

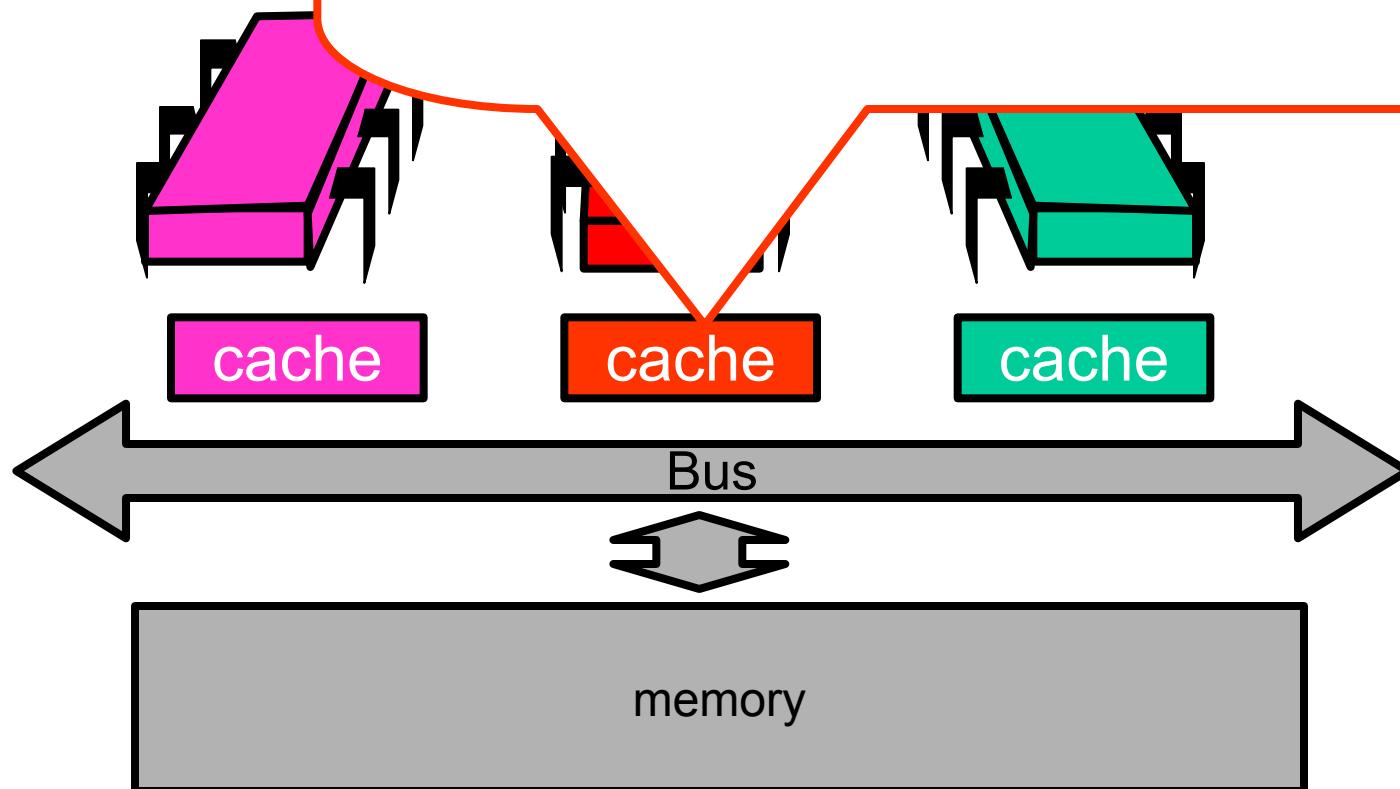
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”



Bus-

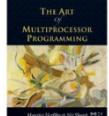
## Per-Processor Caches

- Small
- Fast: 1 or 2 cycles
- Address & state information



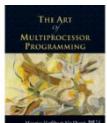
# Mutual Exclusion

- What do we want to optimize?
  - Bus bandwidth used by spinning threads
  - Release/Acquire latency
  - Acquire latency for idle lock



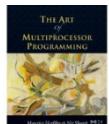
# Simple TASLock

- TAS invalidates cache lines
- Spinners
  - Miss in cache
  - Go to bus
- Thread wants to release lock
  - delayed behind spinners

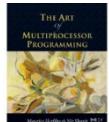
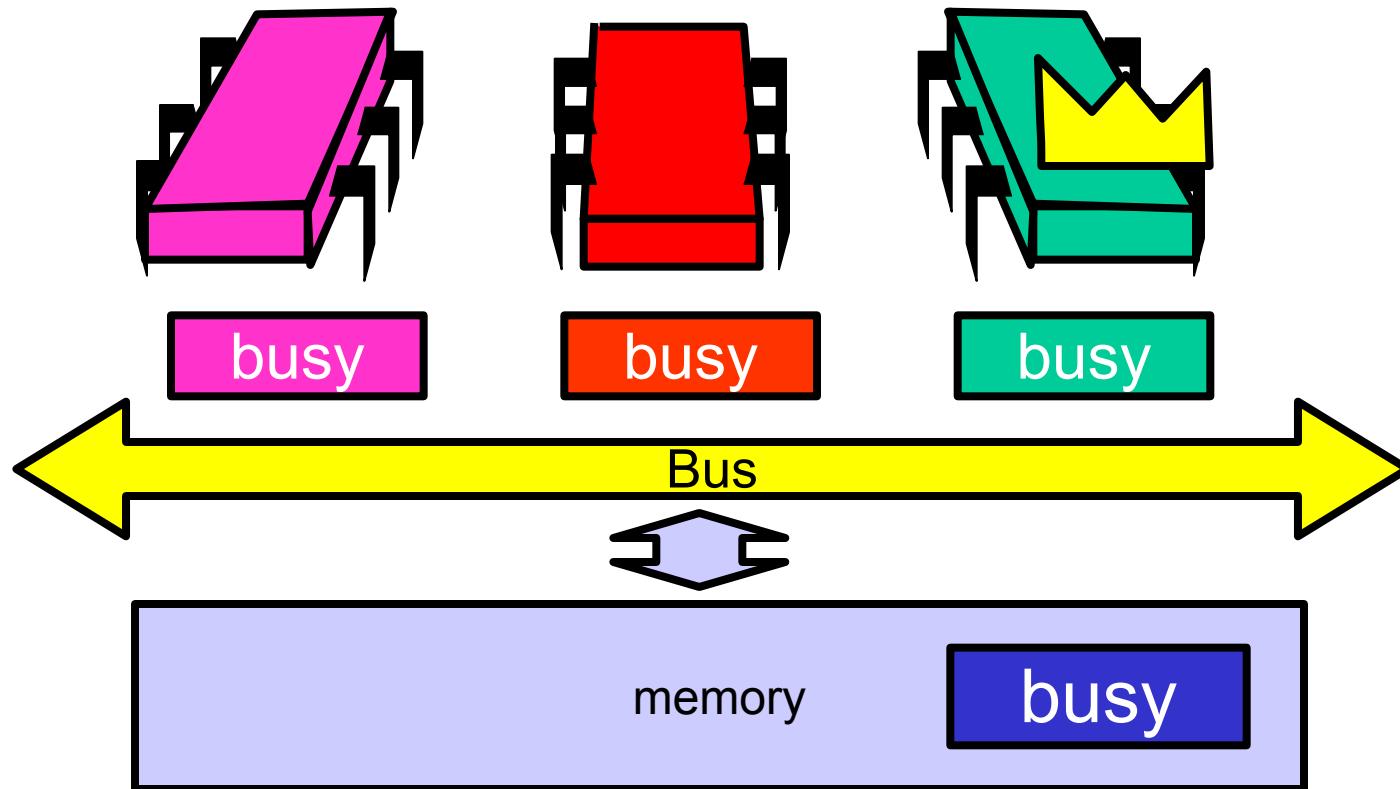


# Test-and-test-and-set

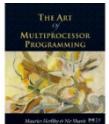
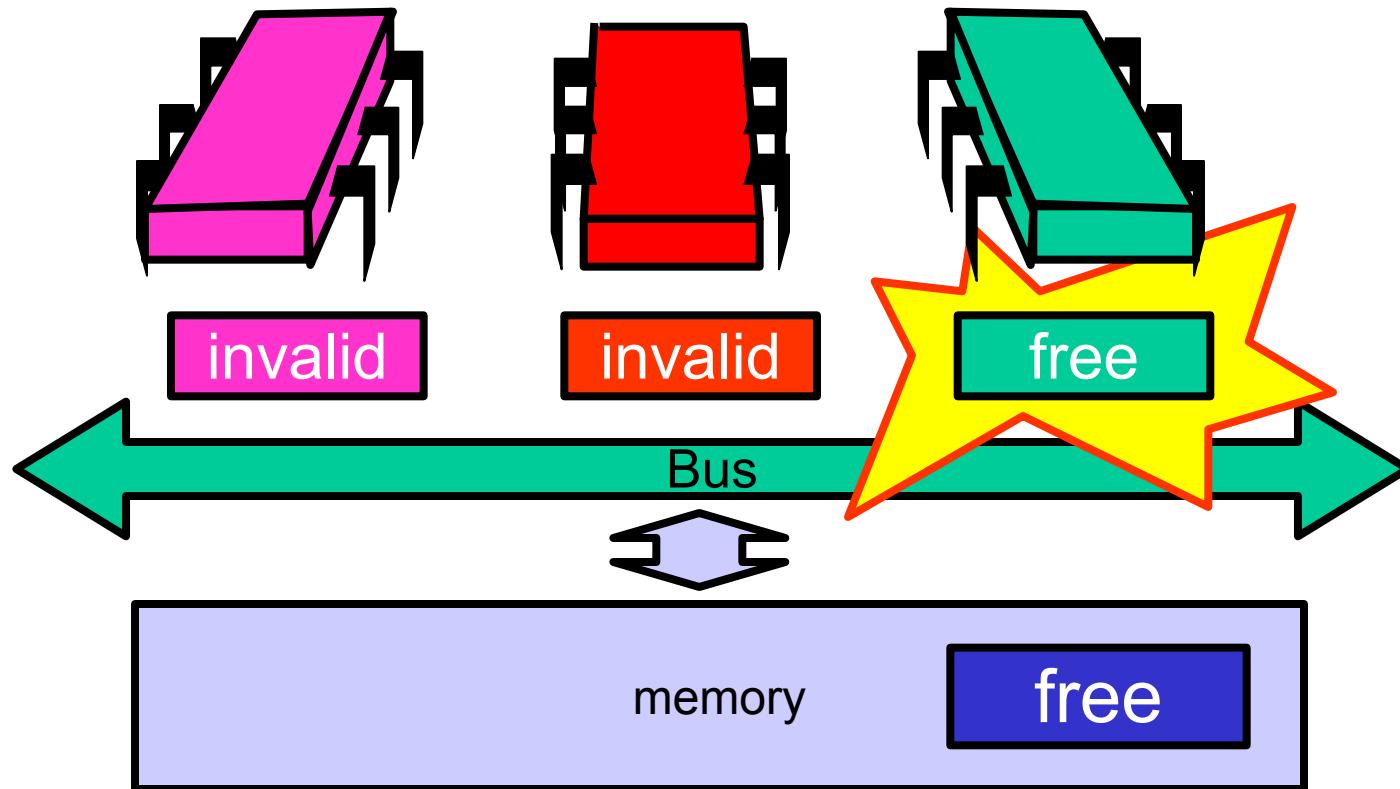
- Wait until lock “looks” free
  - Spin on local cache
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm ...



# Local Spinning while Lock is Busy

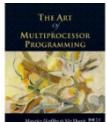
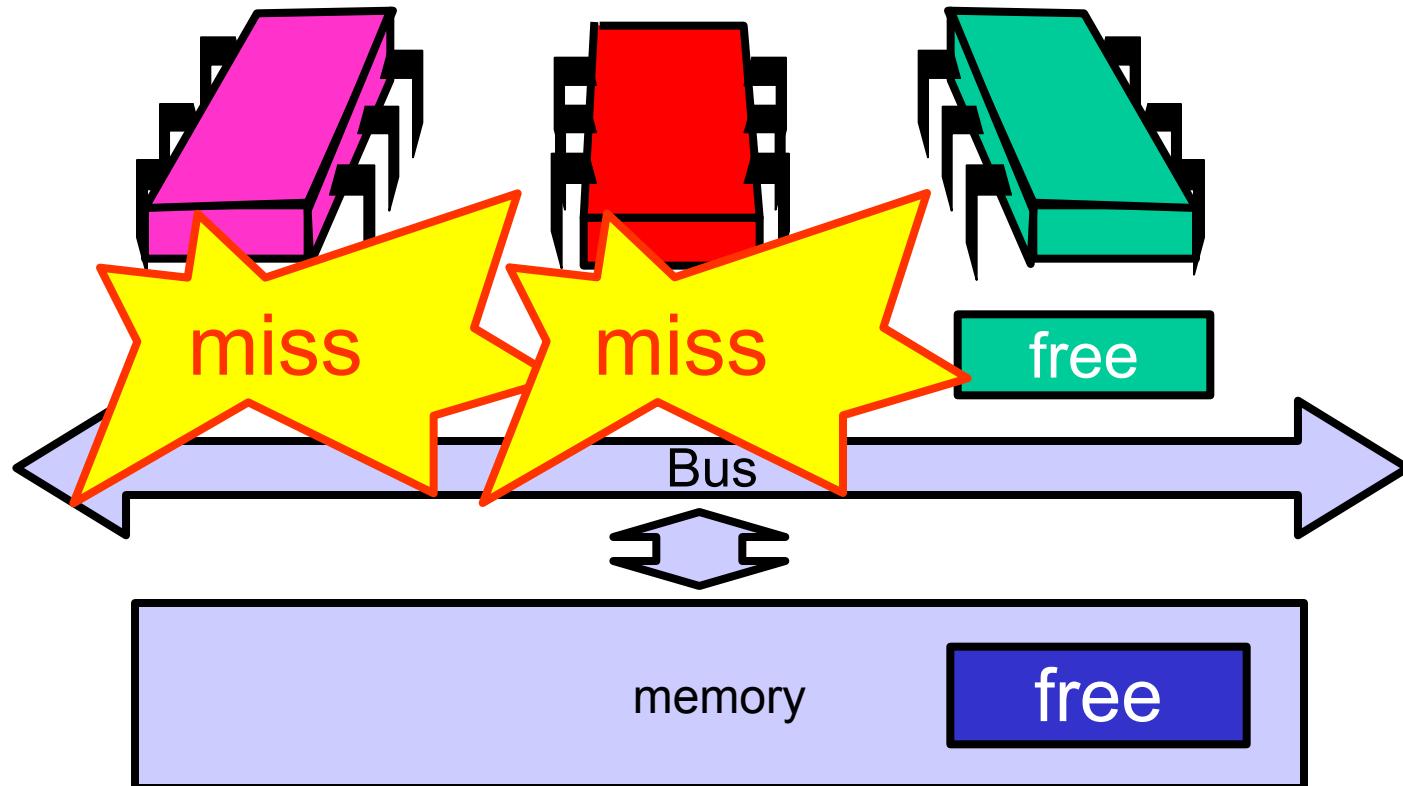


# On Release



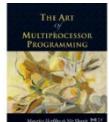
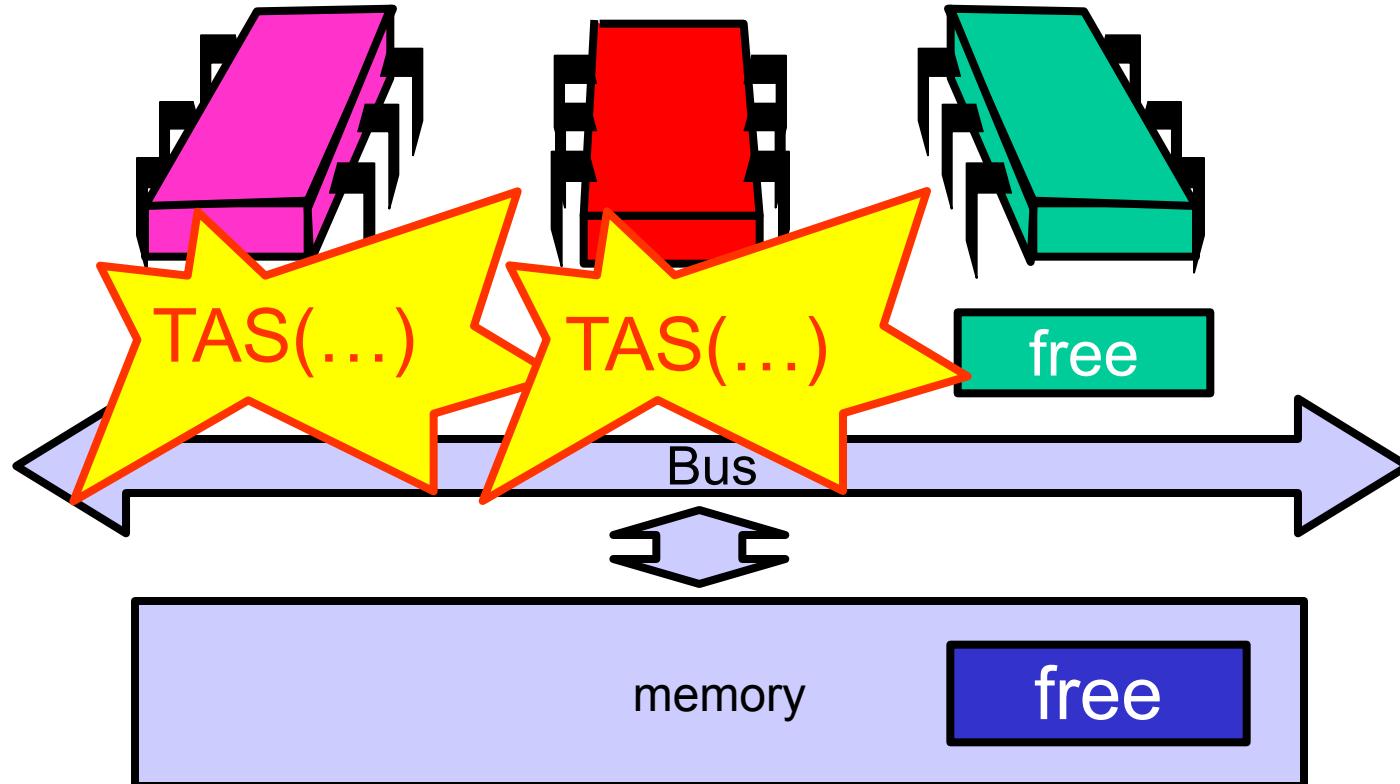
# On Release

Everyone misses,  
rereads



# On Release

## Everyone tries TAS

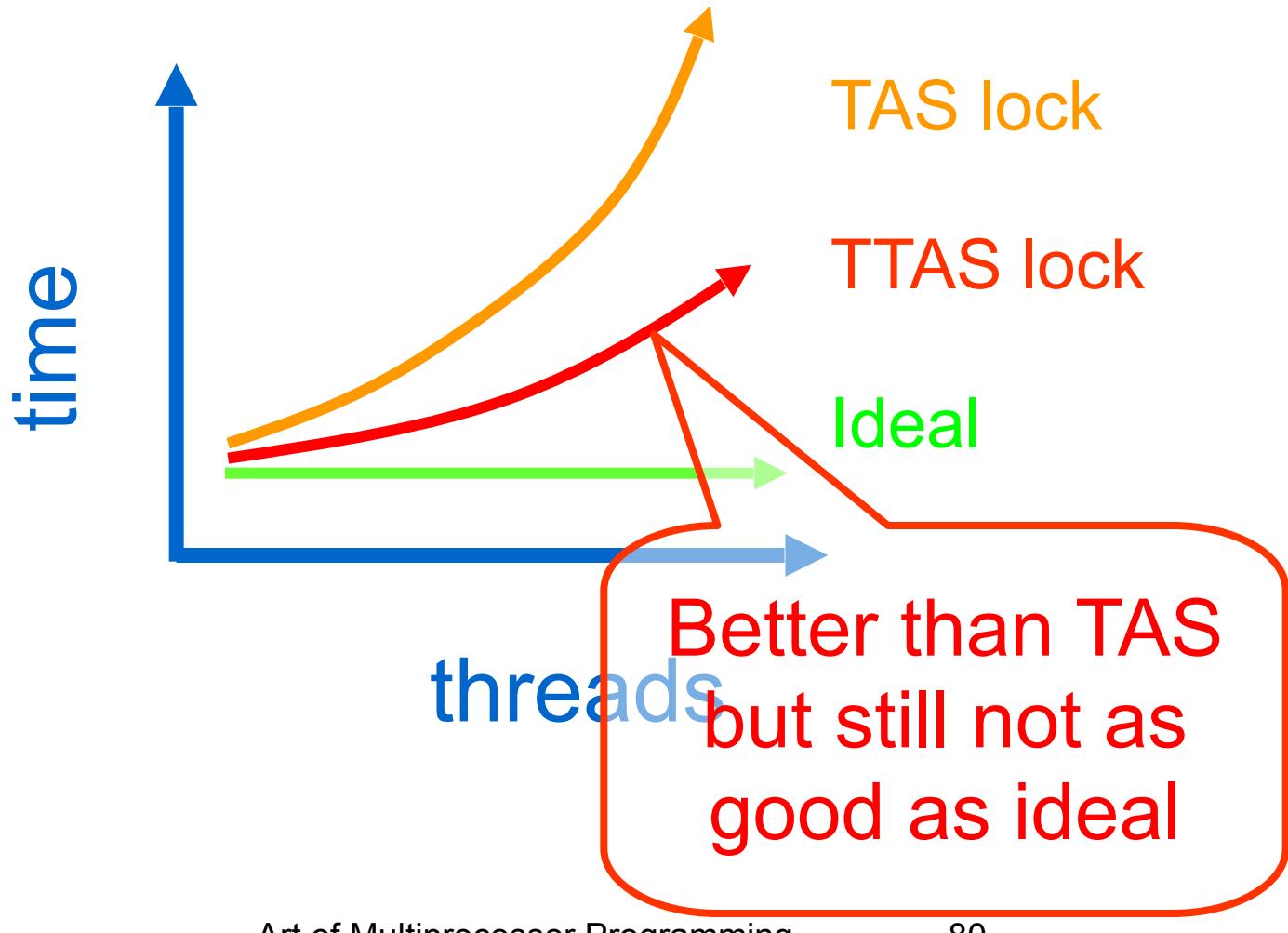


# Problems

- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches

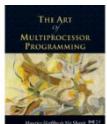
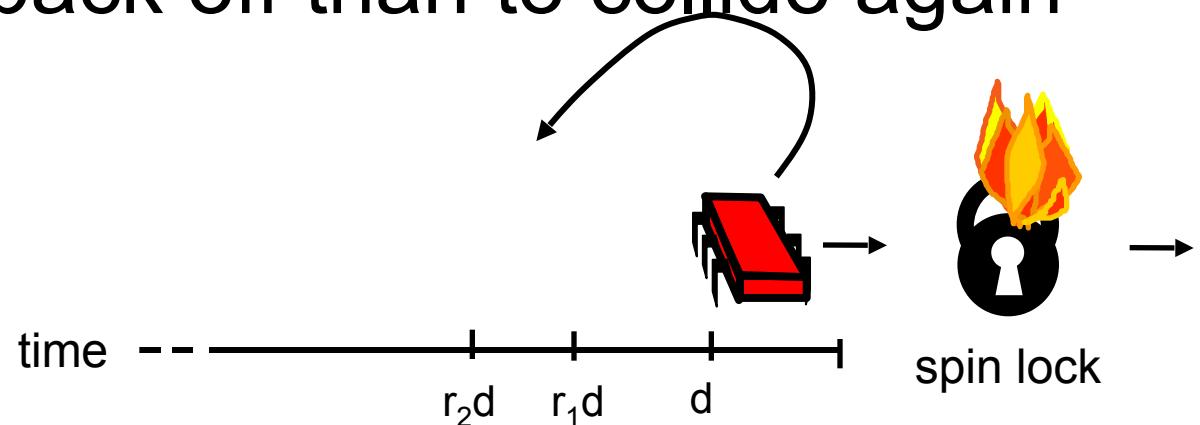


# Mystery Explained

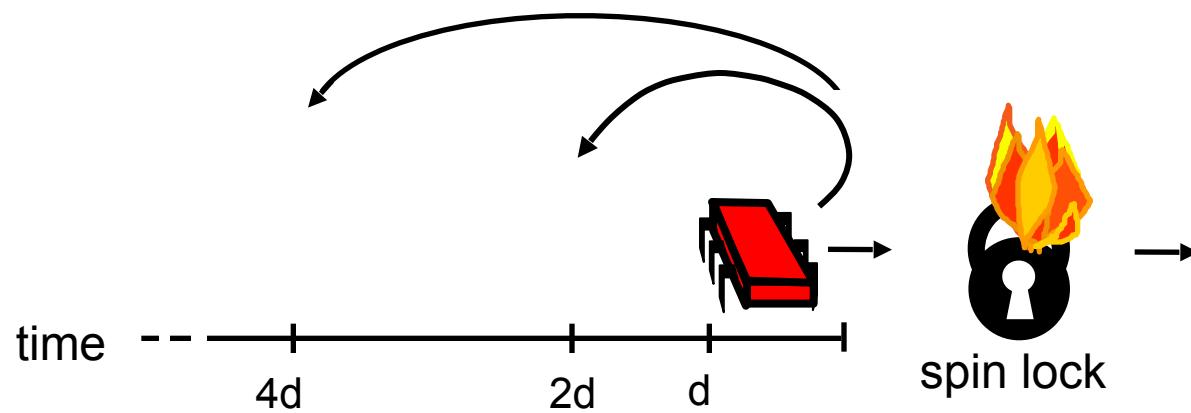


# Solution: Introduce Delay

- If the lock looks free
  - But I fail to get it
- There must be contention
  - Better to back off than to collide again



# Dynamic Example: Exponential Backoff

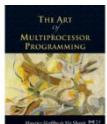


If I fail to get lock

- wait random duration before retry
- Each subsequent failure doubles expected wait

# Exponential Backoff Lock

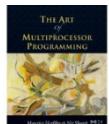
```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        } } }
```



# Exponential Backoff Lock

```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        } } }
```

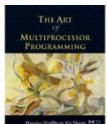
Fix minimum delay



# Exponential Backoff Lock

```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        } } }
```

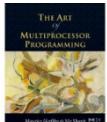
Wait until lock looks free



# Exponential Backoff Lock

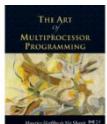
```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        } } }
```

If we win, return



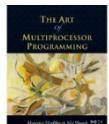
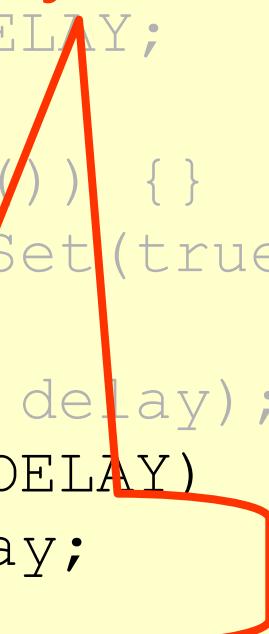
# Exponential Backoff Lock

```
public class Backoff implements lock {  
    public Back off for random duration  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get() == true) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        } } }
```

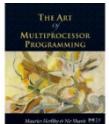
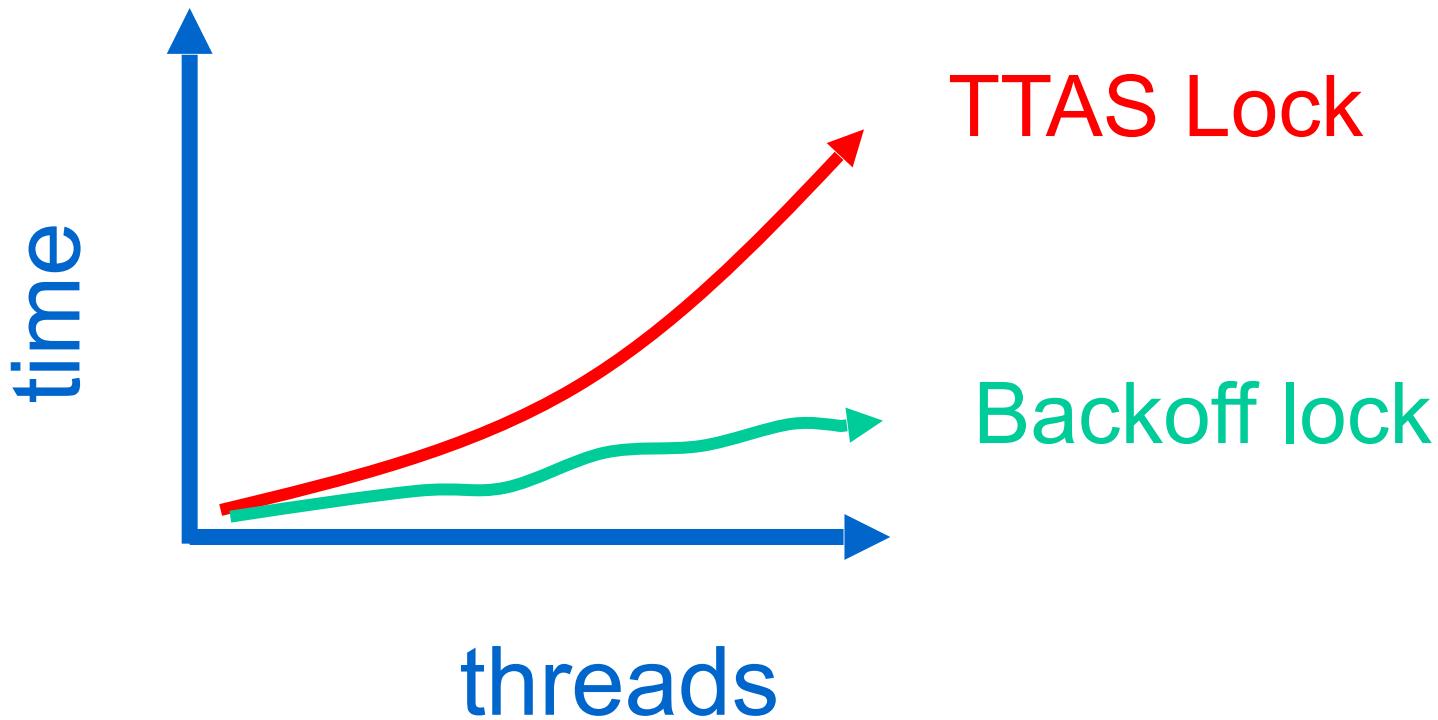


# Exponential Backoff Lock

```
public class Backoff implements lock {  
    public Double max delay, within reason  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get() == true) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        } } }  
}
```

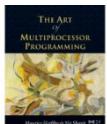


# Spin-Waiting Overhead



# Backoff: Other Issues

- Good
  - Easy to implement
  - Beats TTAS lock
- Bad
  - Must choose parameters carefully
  - Not portable across platforms



# Parallel Performance

# Performance Theory

In this course, the main goal for parallelization is to improve performance. But what does “performance” mean? Usually it’s one of the following

- Reducing the **latency**, which is the total time for a program/task to compute a single result.
- Increasing **throughput**, which is the rate at which results are computed.
- Reducing the power consumption of a computation
- Also the distinct of reducing costs or adding resources to meet a deadline.

All of these are valid interpretations of “performance”.

# Speedup

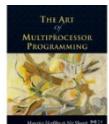
- One important metric related to performance and parallelism is **speedup**
  - A ratio between the latency of solving a task with one processing unit versus solving the same problem with multiple processing units in parallel.
  - **Linear speedup** - a algorithm runs  $P$  times faster on  $P$  processors
    - Very rare in practice due to the extra work disturbing tasks to processors and coordinating them. (Amdahl's Law)

# Amdahl's Law

- Limit on speedup - Amdahl's law sometimes called strong scalability which considers speedup as n-threads vary the problem size stays the same.

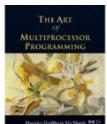
Speedup =

$$\frac{1\text{-thread execution time}}{n\text{-thread execution time}}$$



# Amdahl's Law

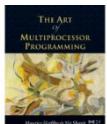
**Speedup =** 
$$\frac{1}{1 - p + \frac{p}{n}}$$



# Amdahl's Law

Speedup =  $\frac{1}{1 - p + \frac{p}{n}}$

Parallel fraction



# Amdahl's Law

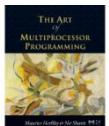
**Sequential fraction**

**Parallel fraction**

**Speedup =**

$$\frac{1}{(1 - p) + \frac{p}{n}}$$

The diagram illustrates the formula for Amdahl's Law. It features a horizontal line segment with a break. On the left side, a red bracket labeled "Sequential fraction" points to the first term "1 - p". On the right side, another red bracket labeled "Parallel fraction" points to the second term "p/n". The terms are separated by a plus sign.



# Amdahl's Law

**Sequential fraction**

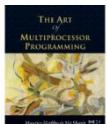
**Parallel fraction**

**Speedup =**

$$\frac{1}{1 - p + \frac{p}{n}}$$

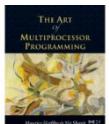
**Number of threads**

The diagram illustrates the components of the Amdahl's Law formula. The formula is  $\frac{1}{1 - p + \frac{p}{n}}$ . The term  $\frac{1}{1-p}$  is labeled "Sequential fraction" with a red line. The term  $\frac{p}{n}$  is labeled "Parallel fraction" with a red line. The term  $1$  is labeled "Speedup =".



# Example

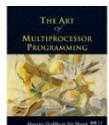
- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?



# Example

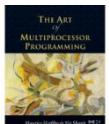
- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}$$



# Example

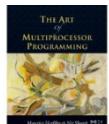
- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?



# Example

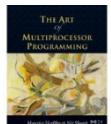
- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}$$



# Example

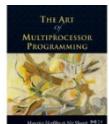
- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?



# Example

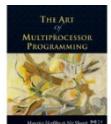
- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$



# Example

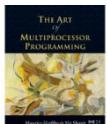
- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?



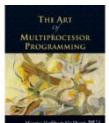
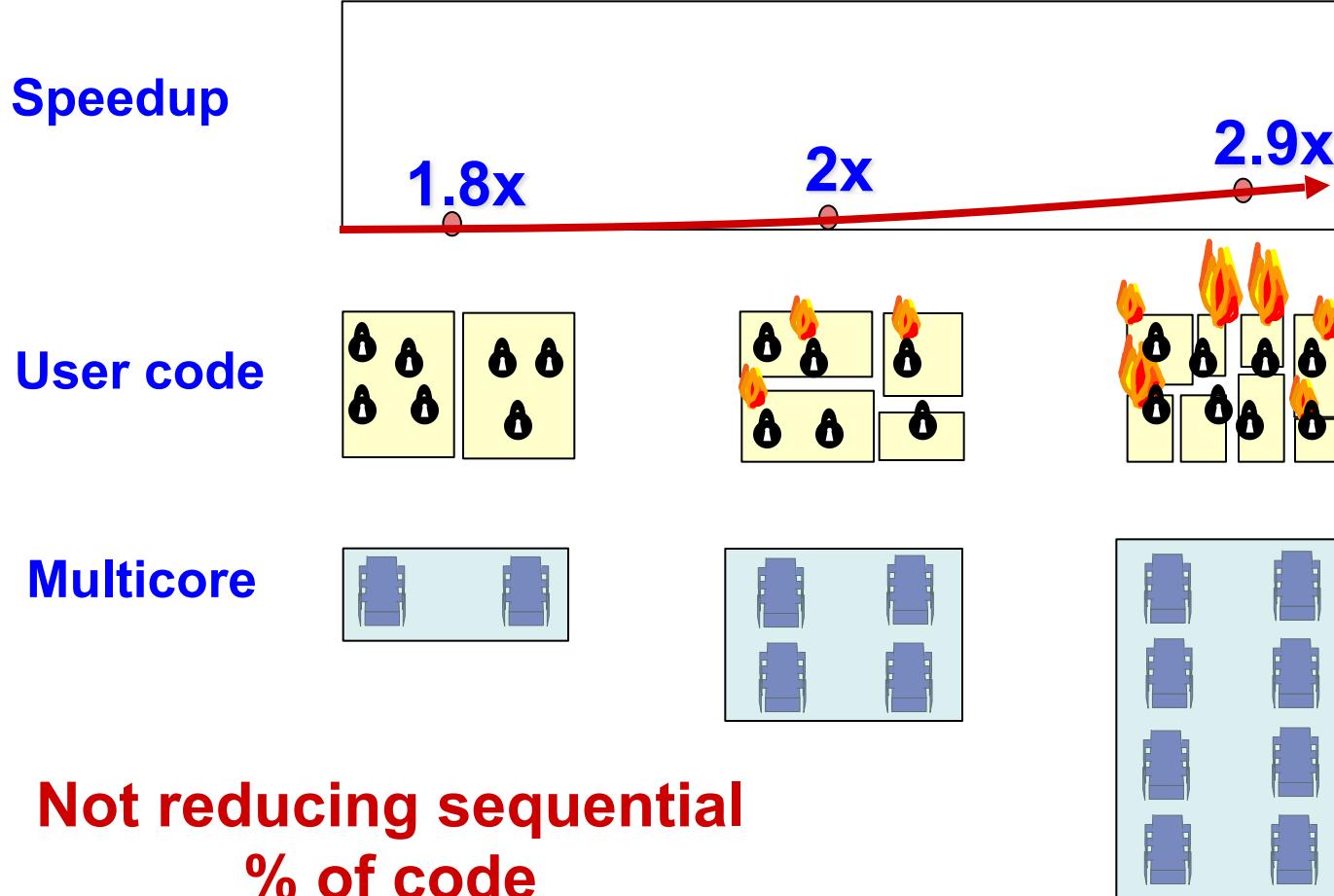
# Example

- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}$$



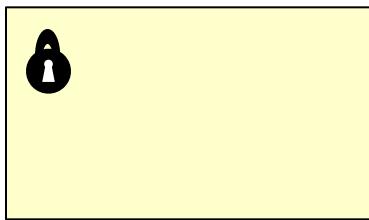
# Back to Real-World Multicore Scaling



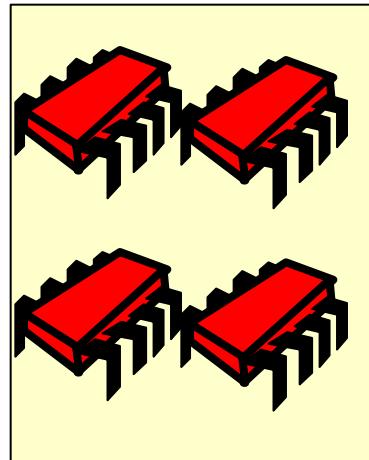
Art of Multiprocessor Programming

# Shared Data Structures

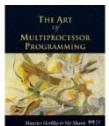
Coarse  
Grained



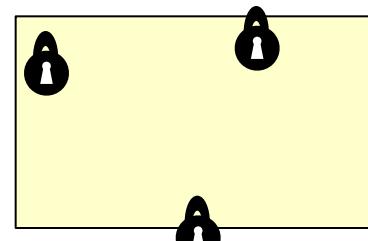
25%  
Shared



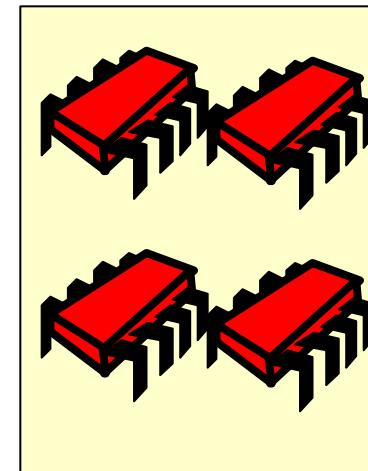
75%  
Unshared



Fine  
Grained

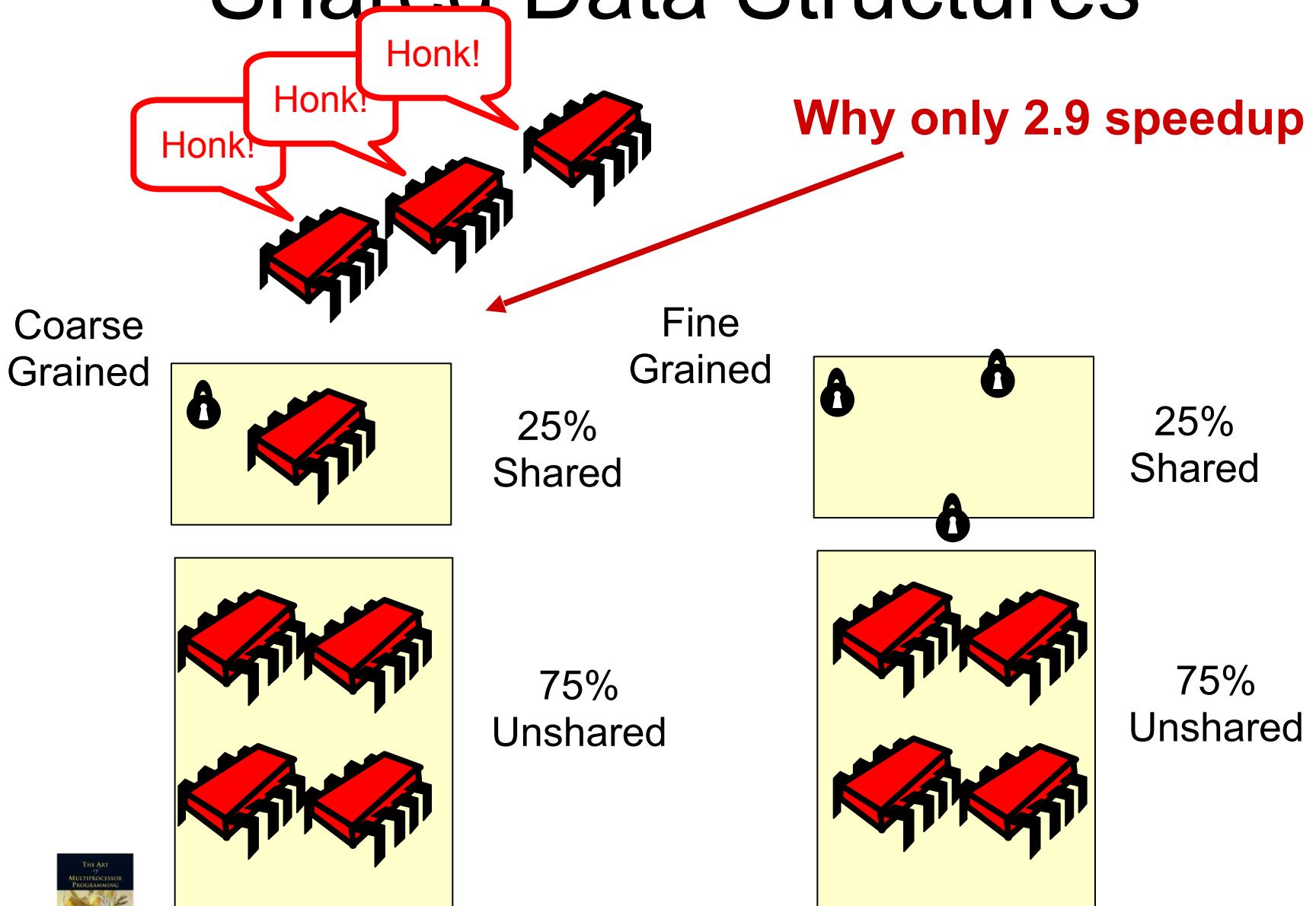


25%  
Shared

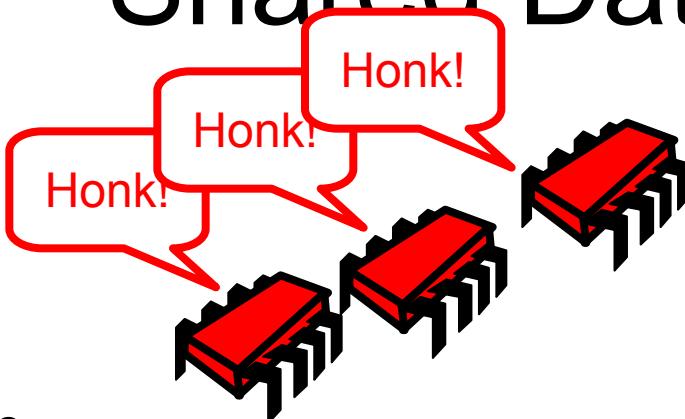


75%  
Unshared

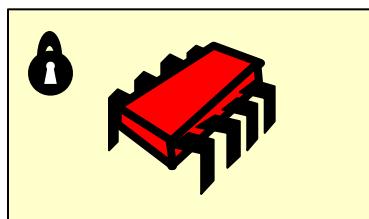
# Shared Data Structures



# Shared Data Structures

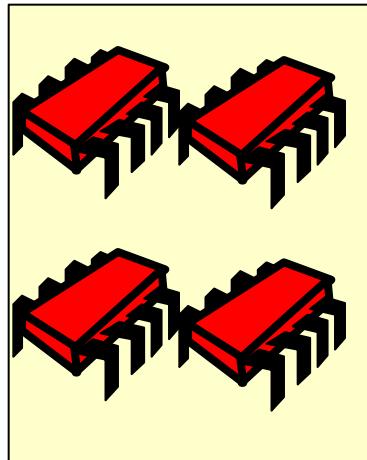


Coarse  
Grained



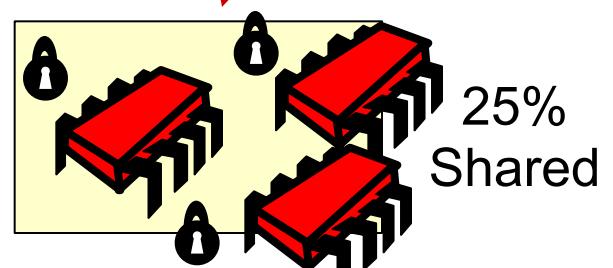
Fine  
Grained

25%  
Shared

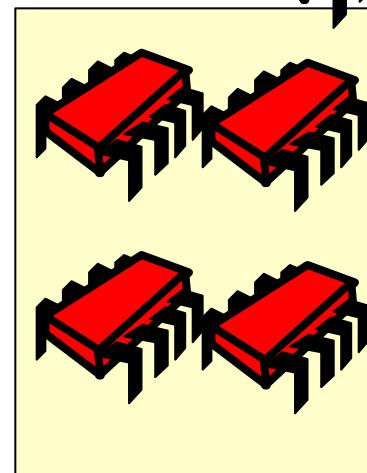


75%  
Unshared

**Why fine-grained parallelism matters**



25%  
Shared



75%  
Unshared

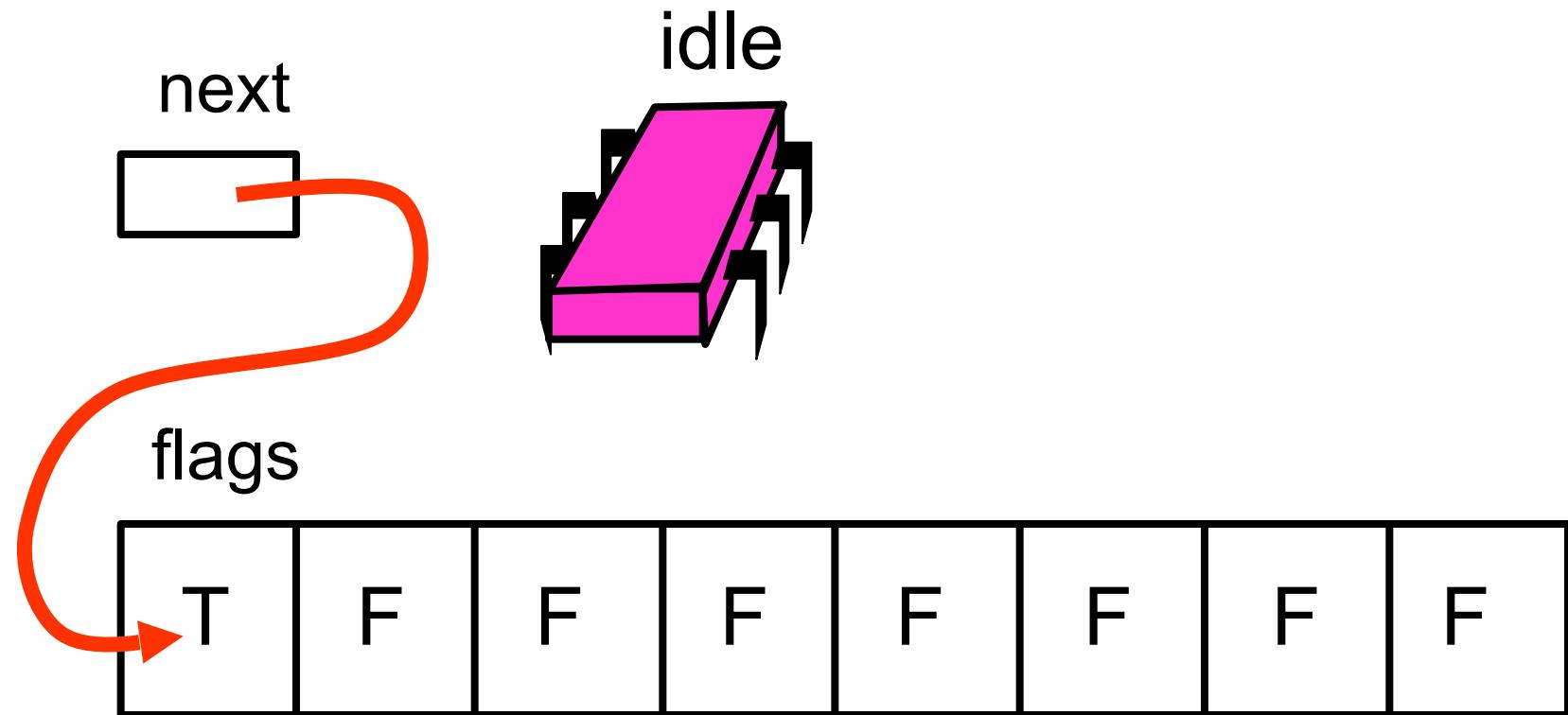
# Fairness and Efficient Locks

# Idea

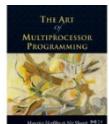
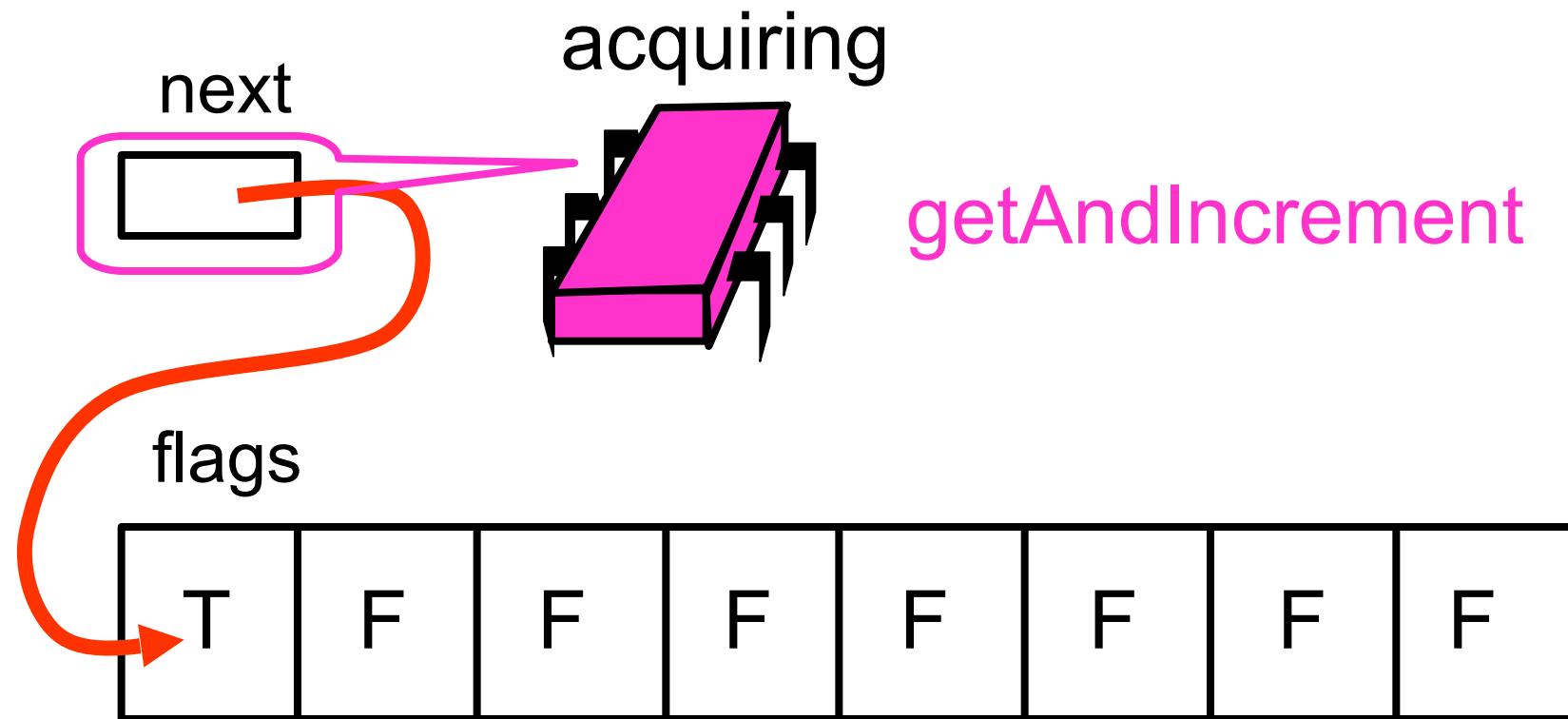
- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others
  - Fairness when it comes to who should get lock next based on who arrived first.



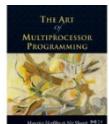
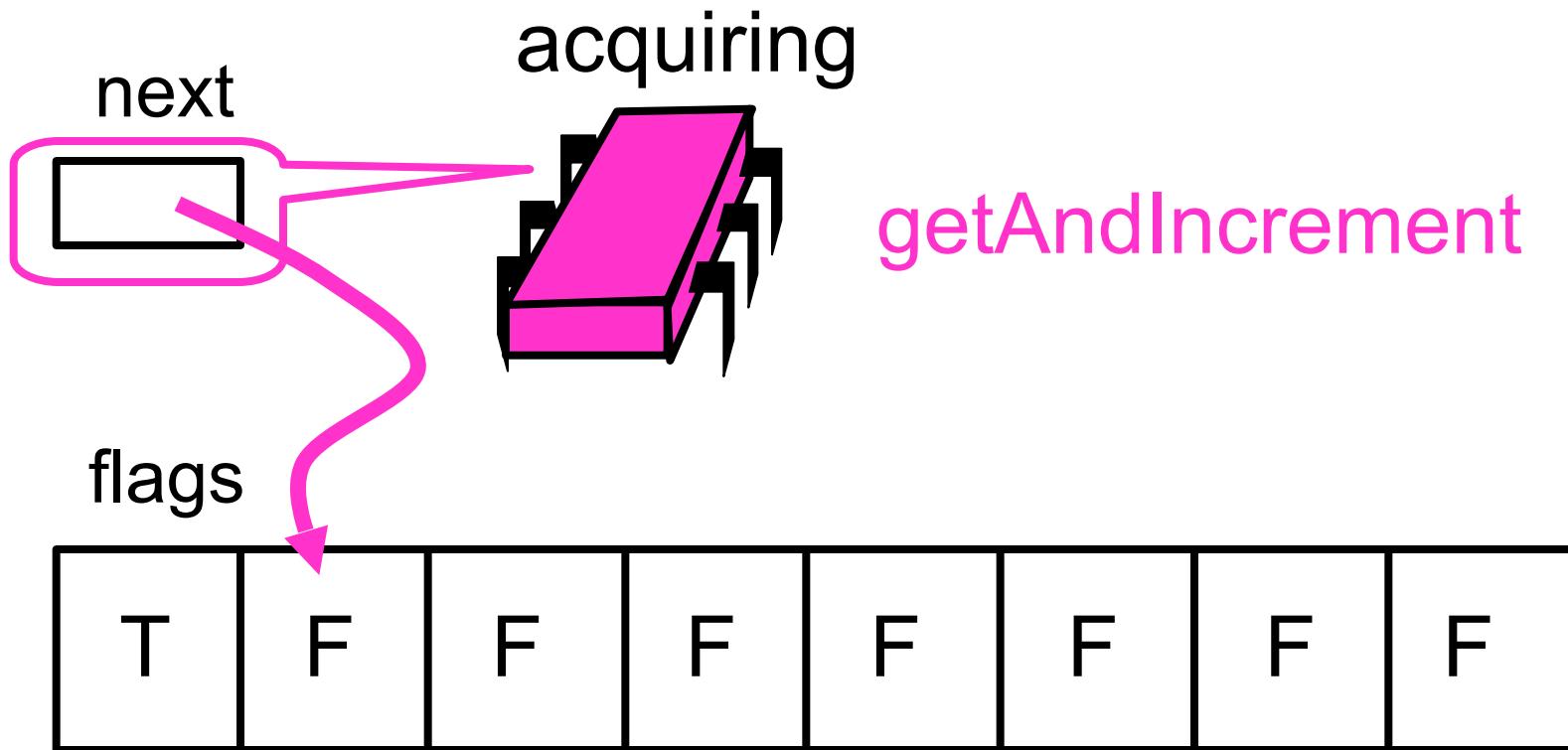
# Anderson Queue Lock



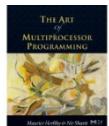
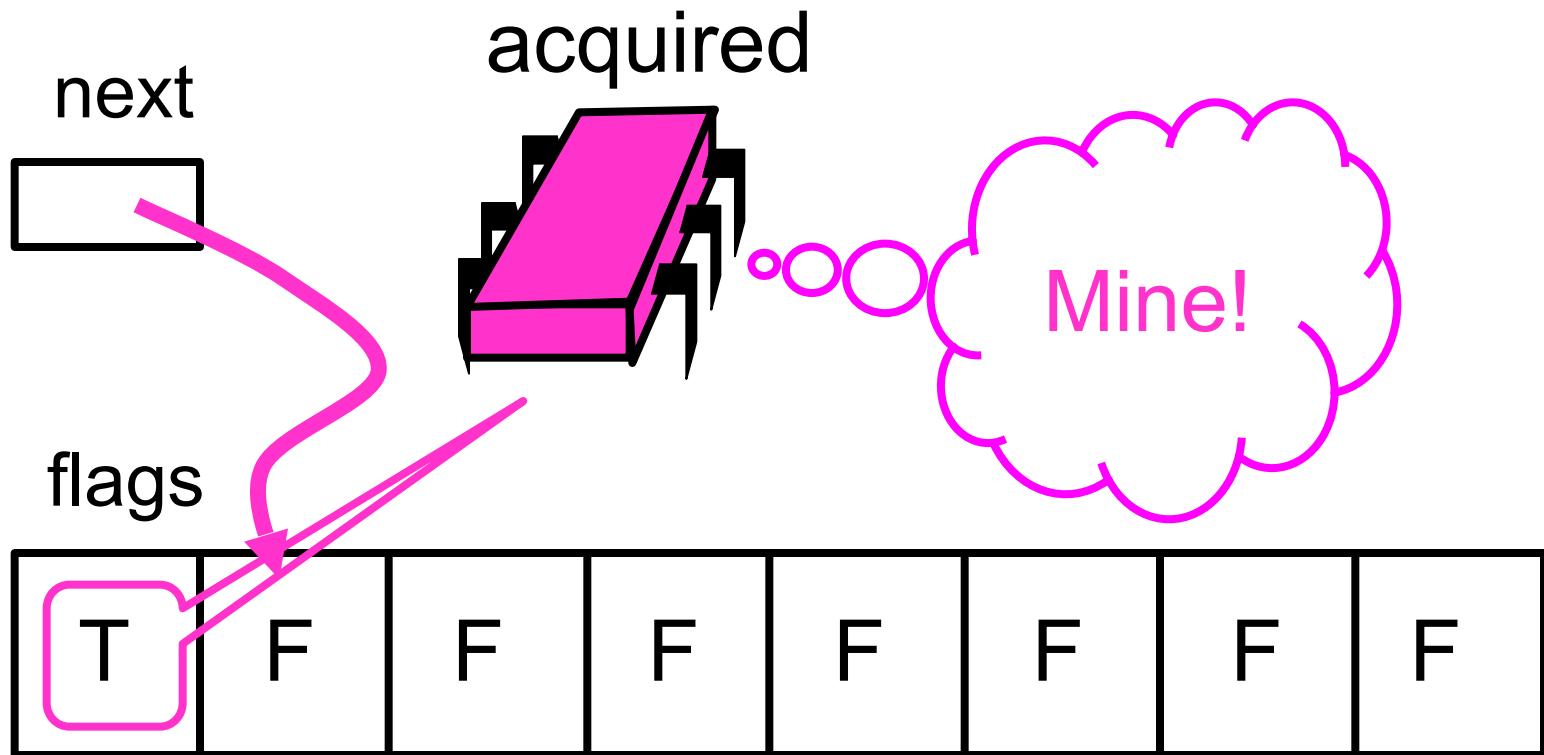
# Anderson Queue Lock



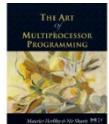
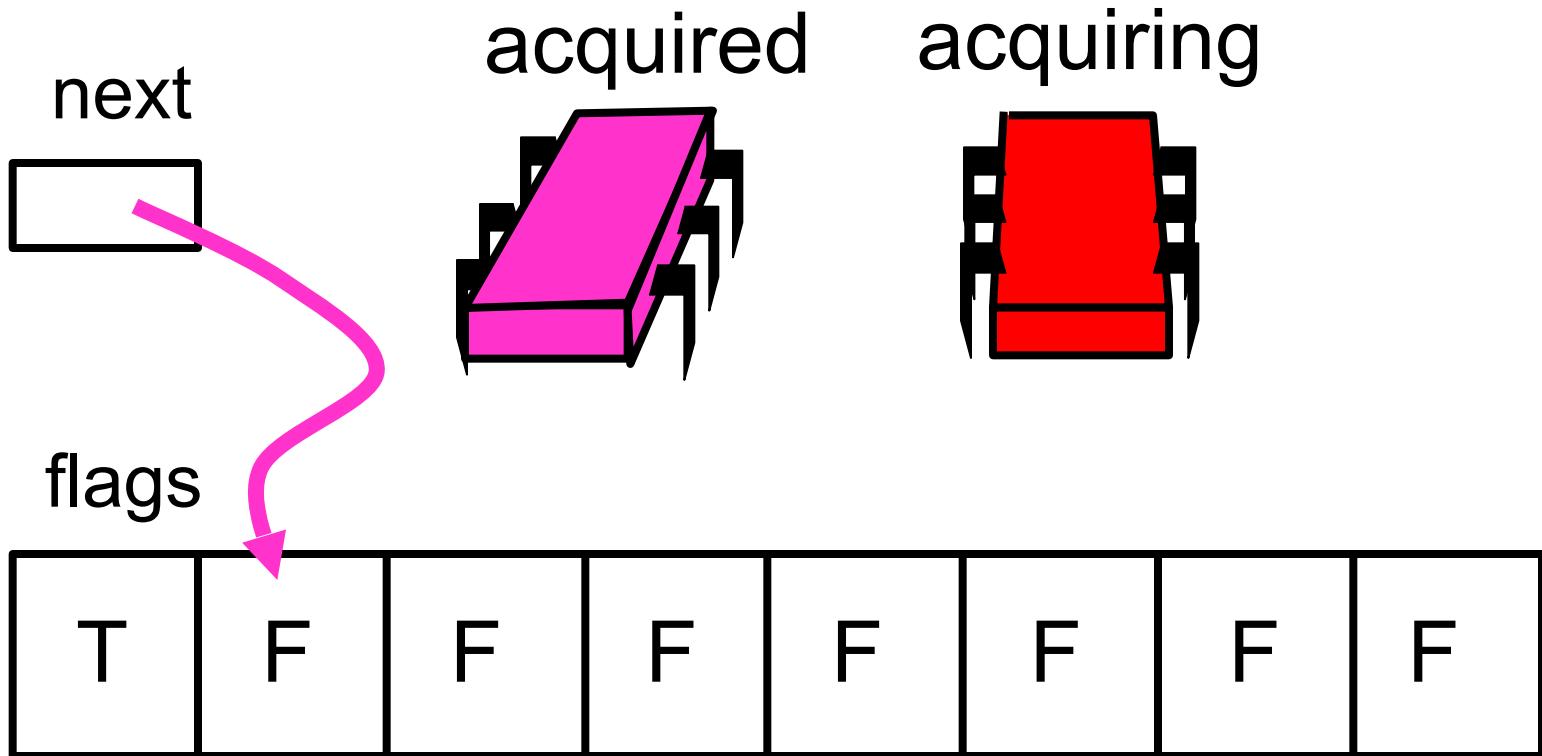
# Anderson Queue Lock



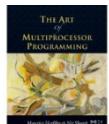
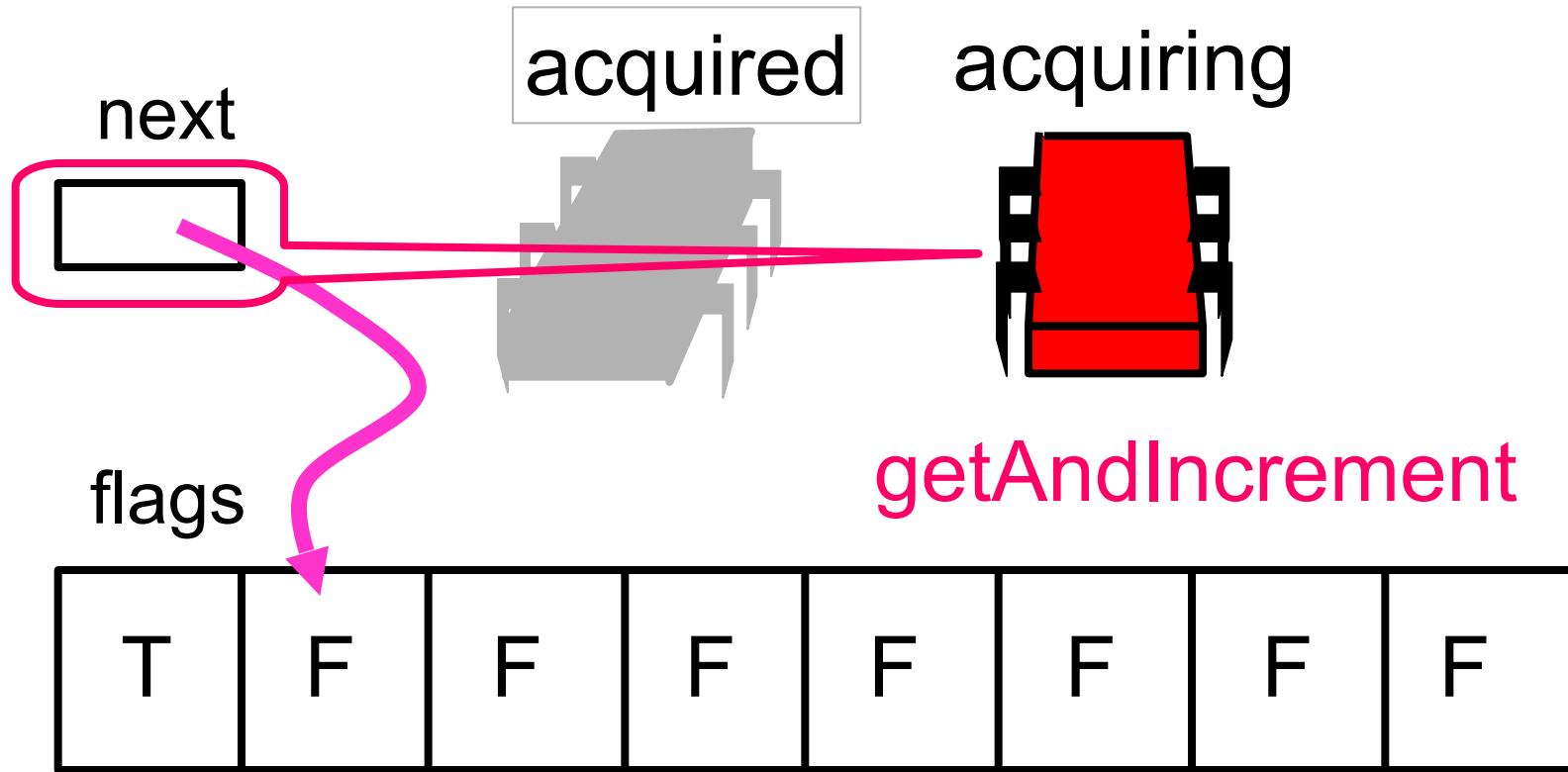
# Anderson Queue Lock



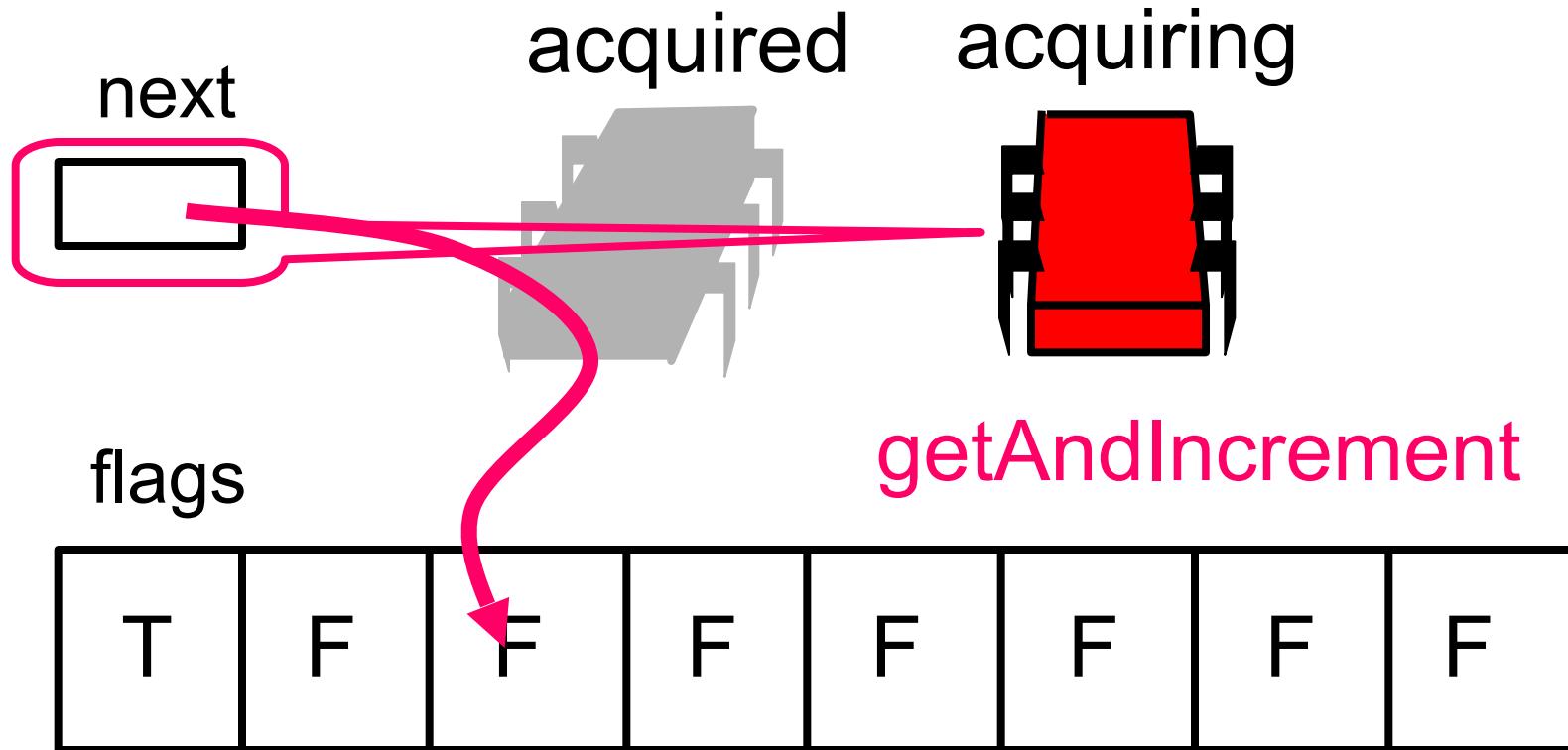
# Anderson Queue Lock



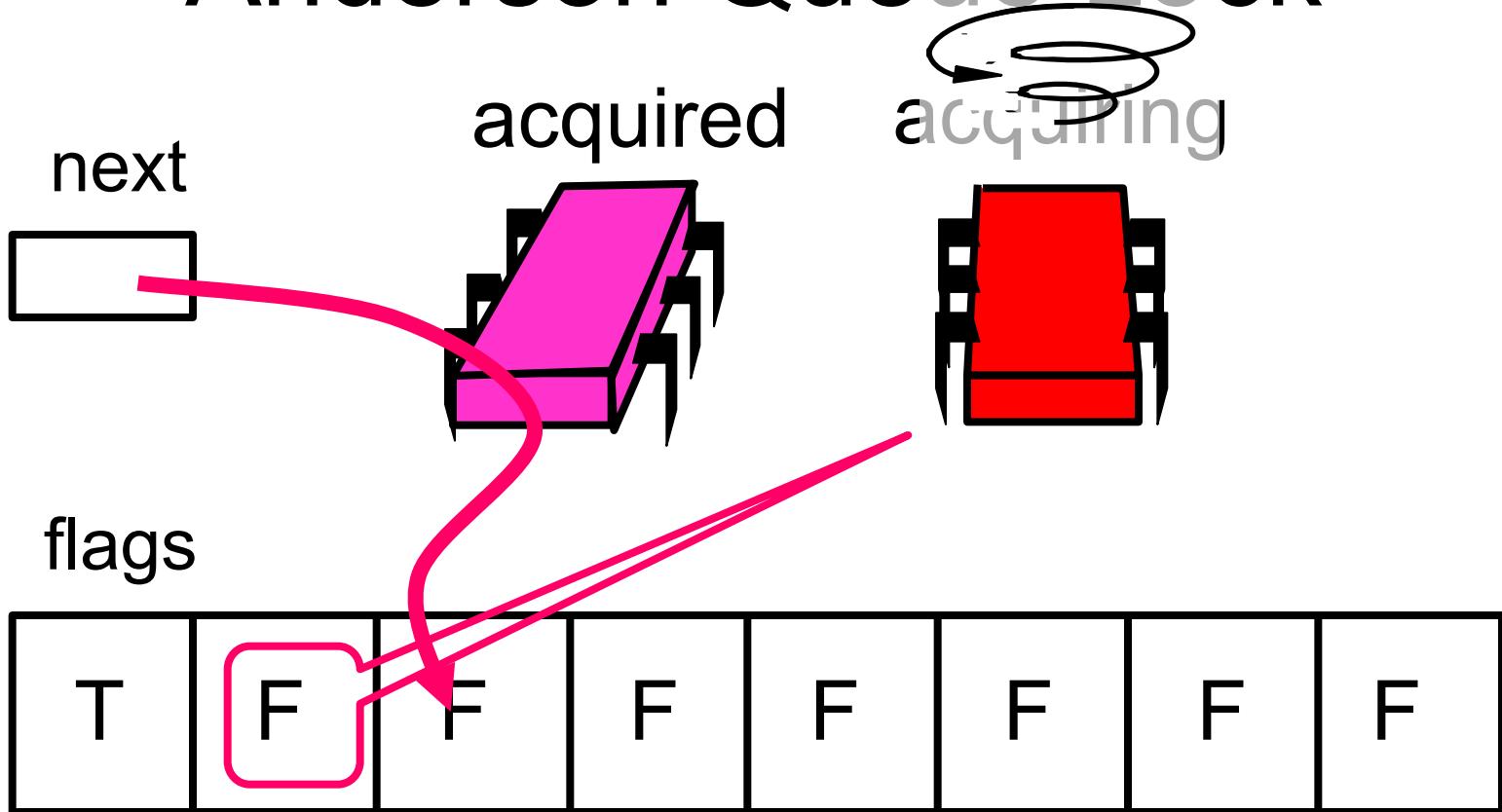
# Anderson Queue Lock



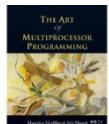
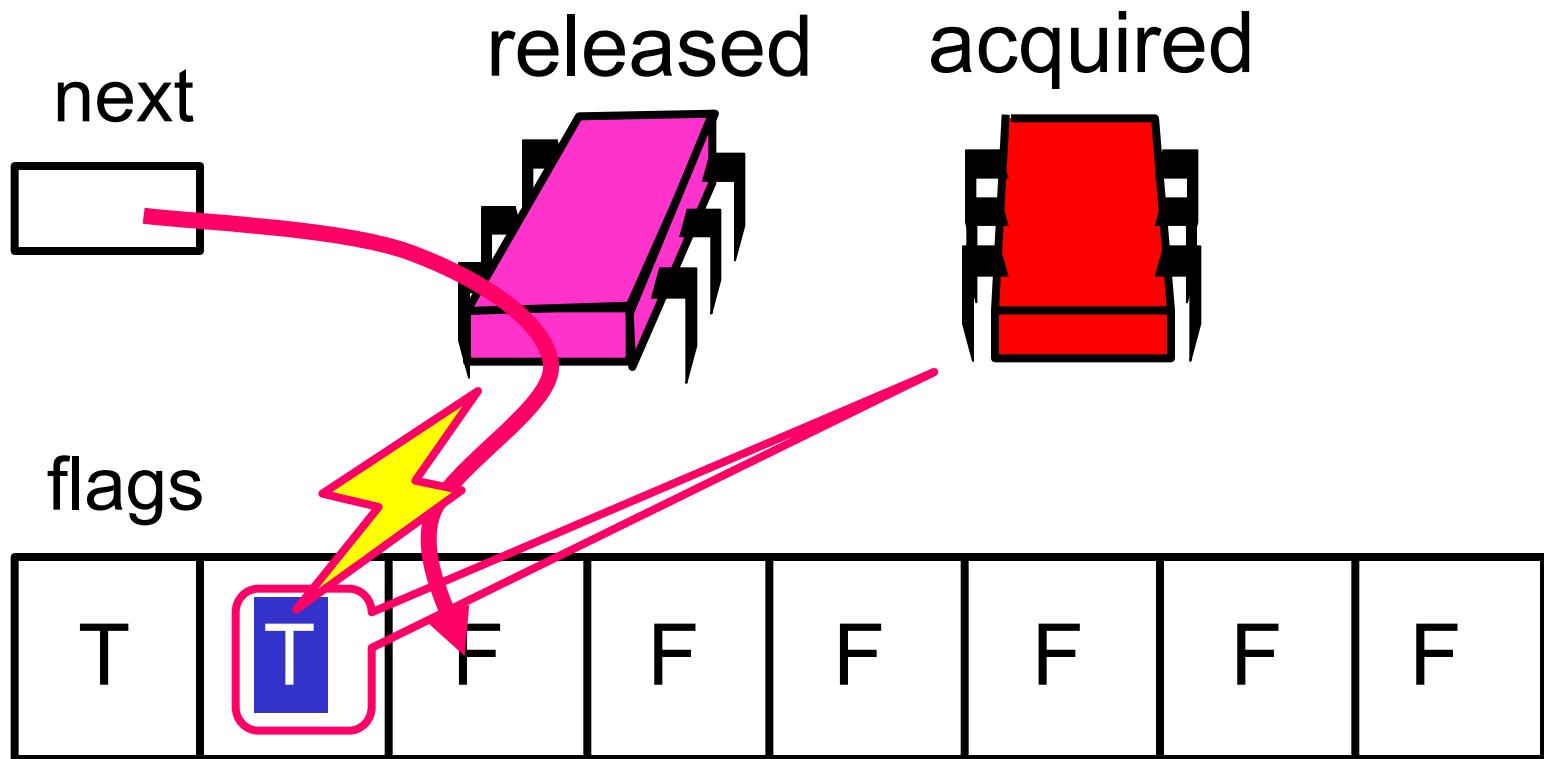
# Anderson Queue Lock



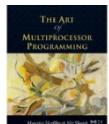
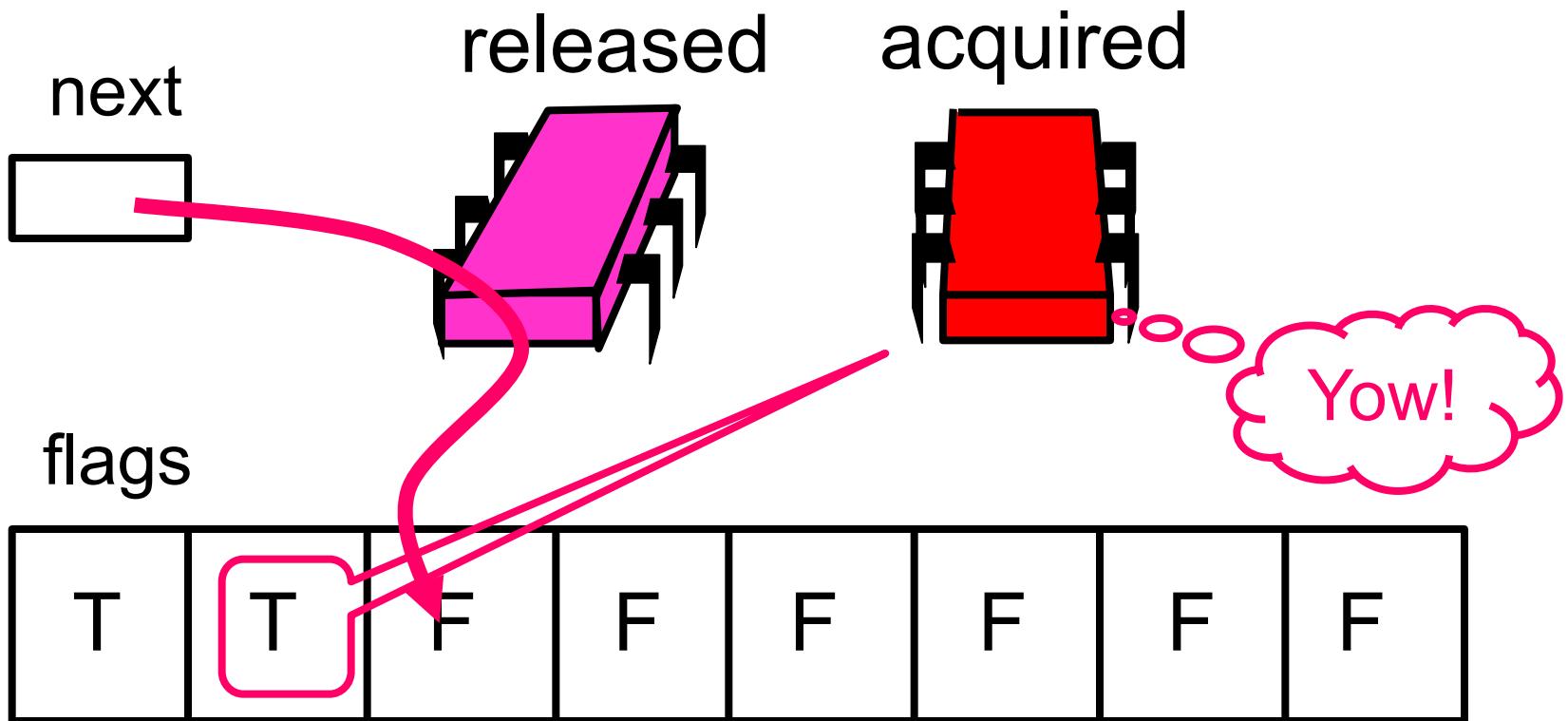
# Anderson Queue Lock



# Anderson Queue Lock

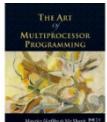


# Anderson Queue Lock



# Anderson Queue Lock

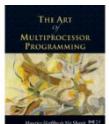
```
class ALock implements Lock {  
    boolean[] flags={true, false, ..., false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```



# Anderson Queue Lock

```
class ALock implements Lock {  
    boolean[] flags={true, false, ..., false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

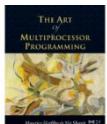
One flag per thread



# Anderson Queue Lock

```
class ALock implements Lock {  
    boolean[] flags = {true, false, ..., false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

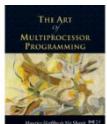
Next flag to use



# Anderson Queue Lock

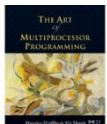
```
class ALock implements Lock {  
    boolean[] flags={true, false, ..., false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

Thread-local variable



# Anderson Queue Lock

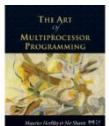
```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```



# Anderson Queue Lock

```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```

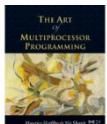
Take next slot



# Anderson Queue Lock

```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```

Spin until told to go

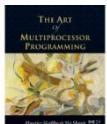


# Anderson Queue Lock

```
public lock() {  
    myslot = next.getAndIncrement();  
    while (!flags[myslot % n]) {};  
    flags[myslot % n] = false;  
}
```

```
public unlock() {  
    flags[(myslot+1) % n] = true;  
}
```

Prepare slot for re-use



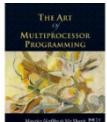
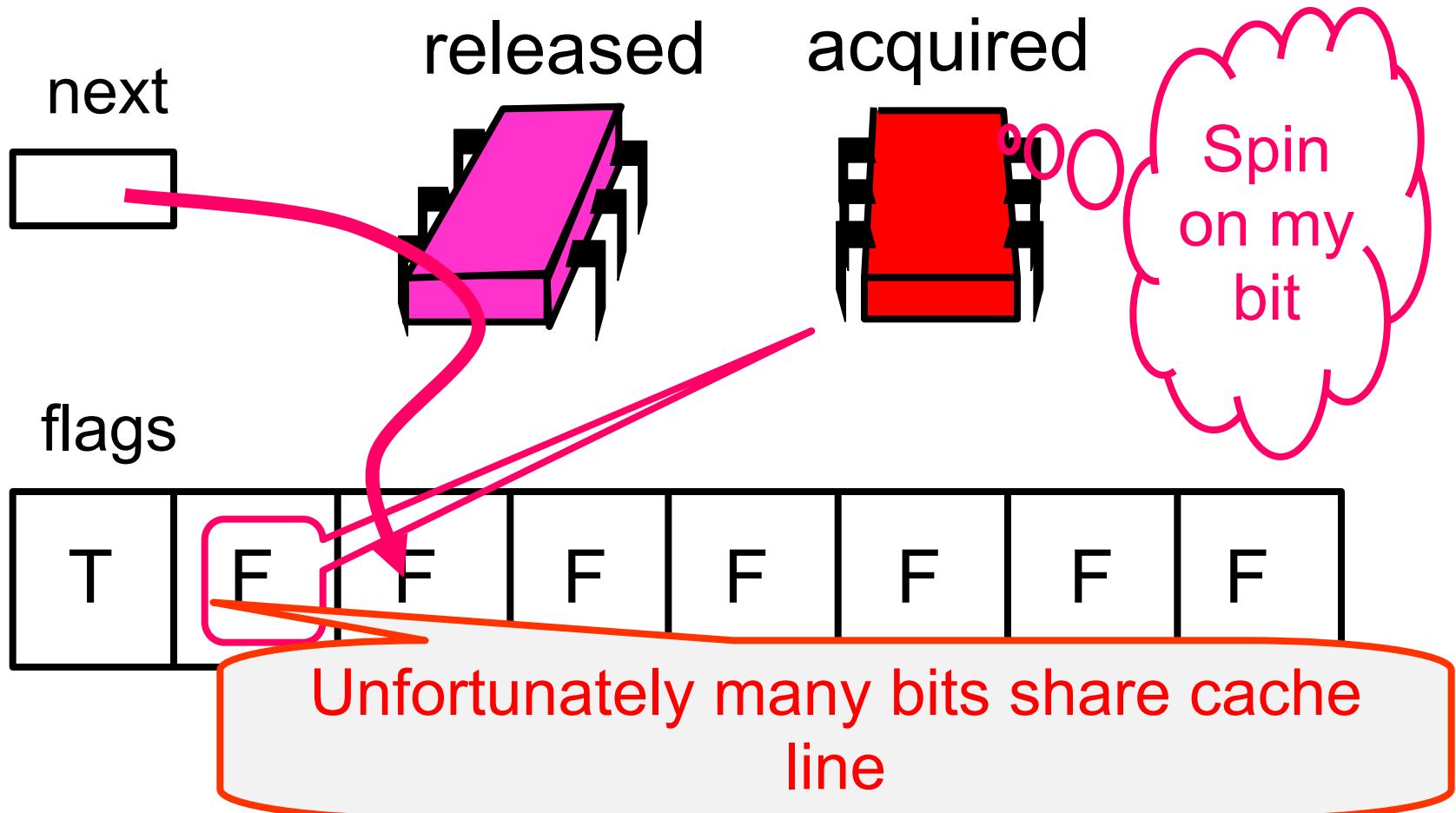
# Anderson Queue Lock

```
public lock() { Tell next thread to go
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) { };
    flags[mySlot % n] = false;
}

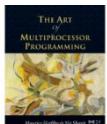
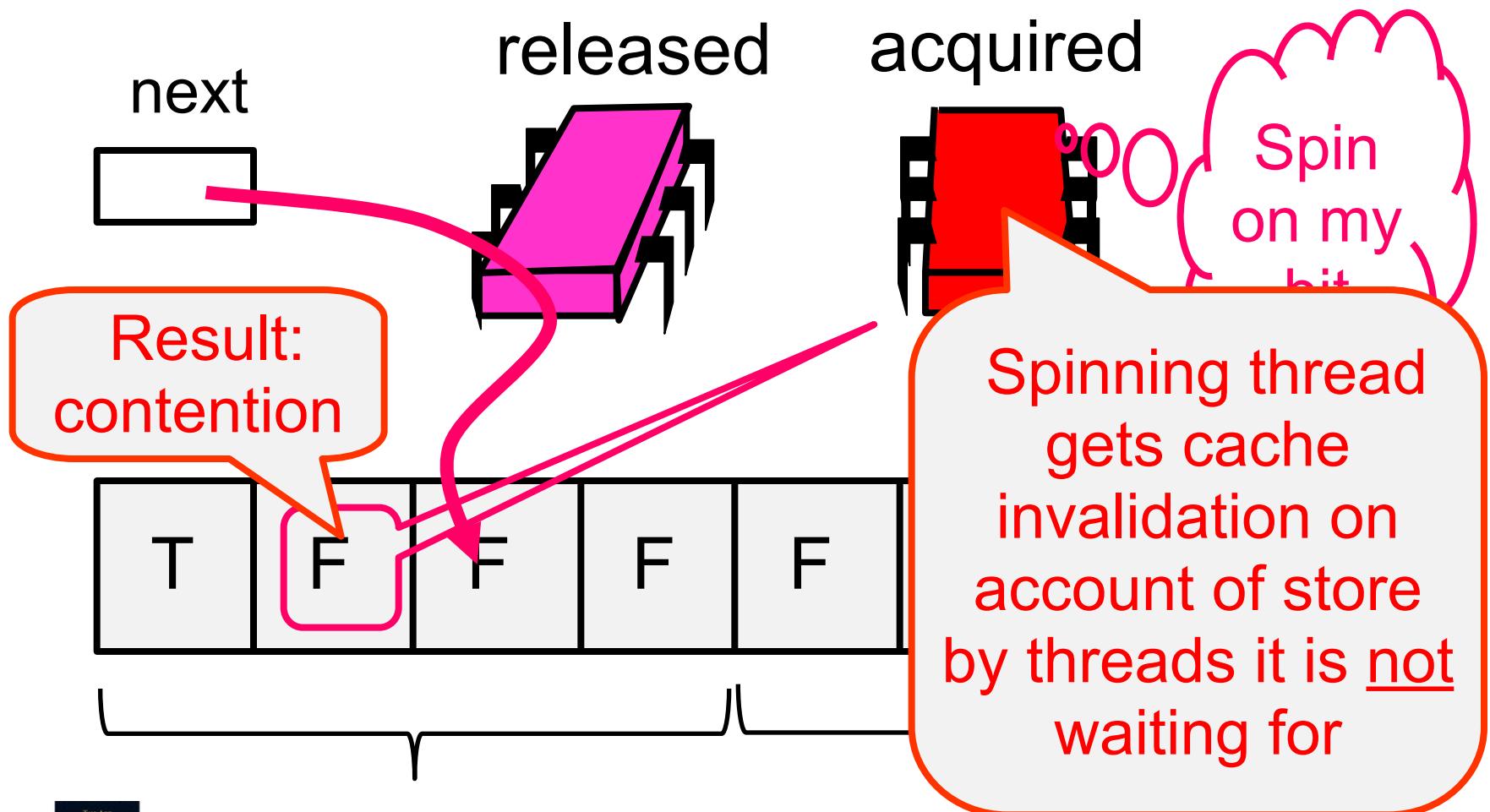
public unlock() {
    flags[(mySlot+1) % n] = true;
}
```



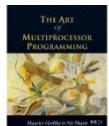
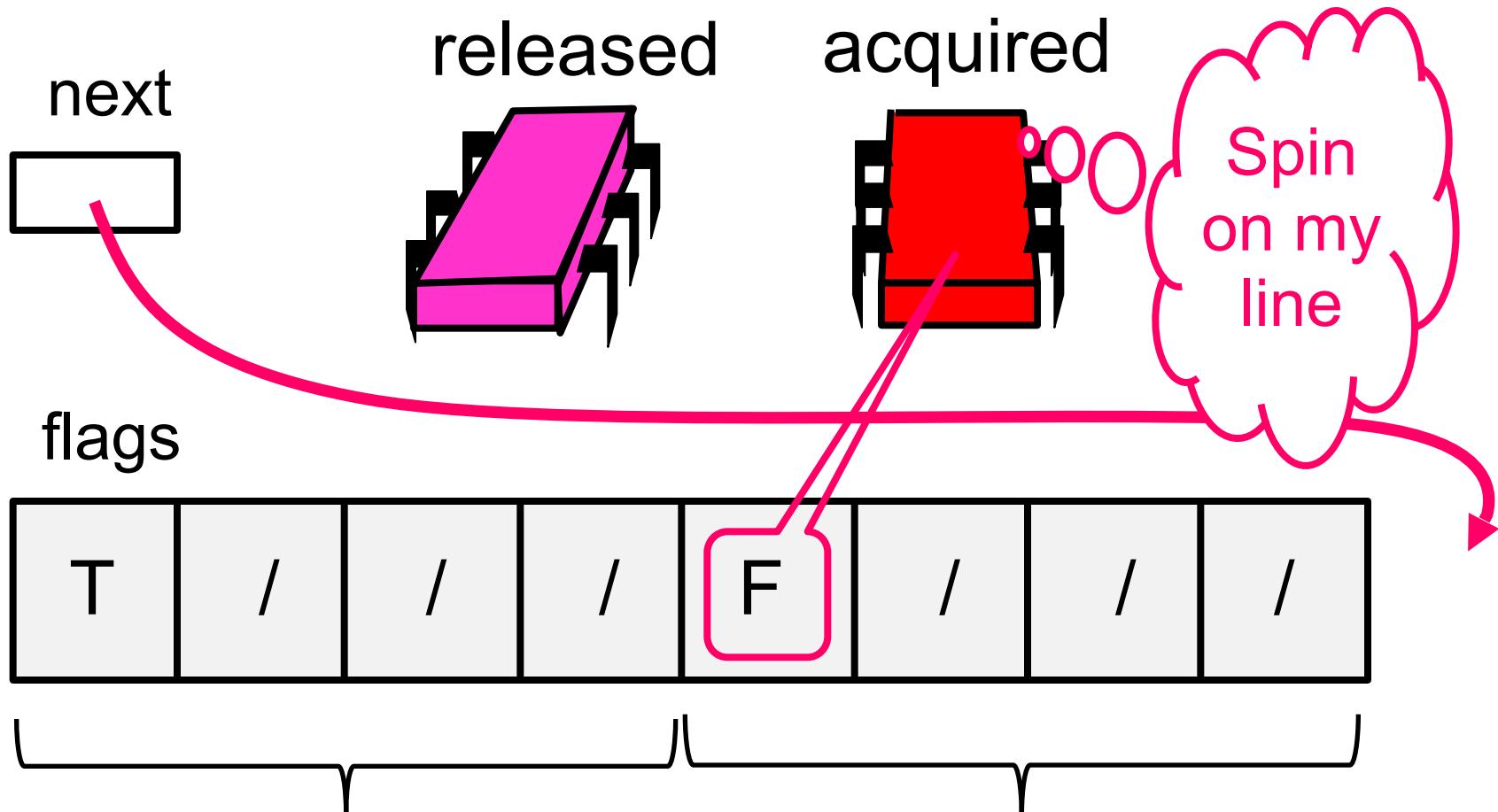
# Local Spinning



# False Sharing

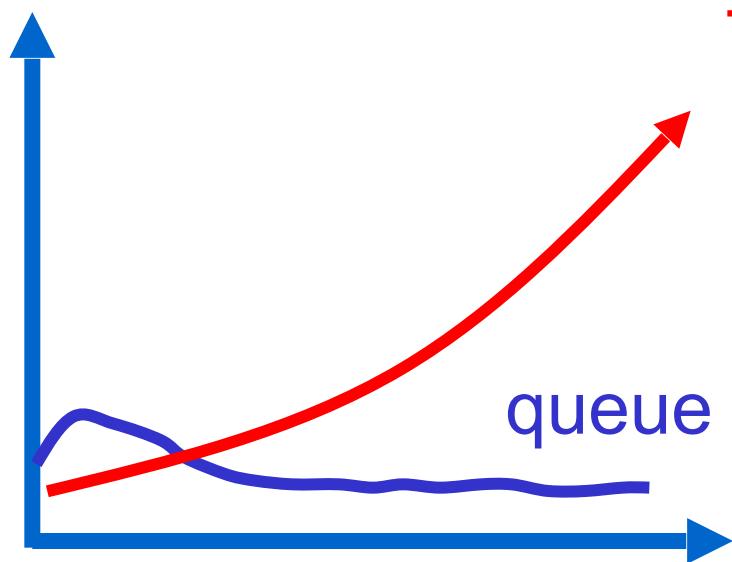


# The Solution: Padding



Line 1 Art of Multiprocessor Programming Line 2<sub>34</sub>

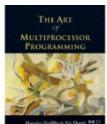
# Performance



TTAS

queue

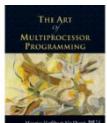
- Shorter handover than backoff
- Curve is practically flat
- Scalable performance



# Anderson Queue Lock

## Good

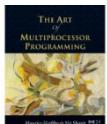
- First truly scalable lock
- Simple, easy to implement
- Back to FIFO order (like Bakery)



# Anderson Queue Lock

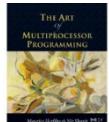
## Bad

- Space hog...
- One bit per thread → one cache line per thread
  - What if unknown number of threads?
  - What if small number of actual contenders?



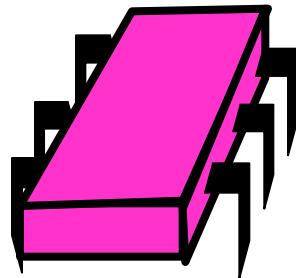
# CLH Lock

- FIFO order
- Small, constant-size overhead per thread

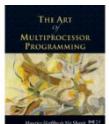
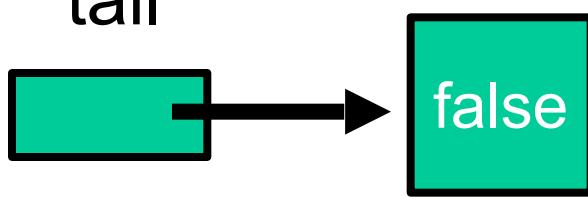


# Initially

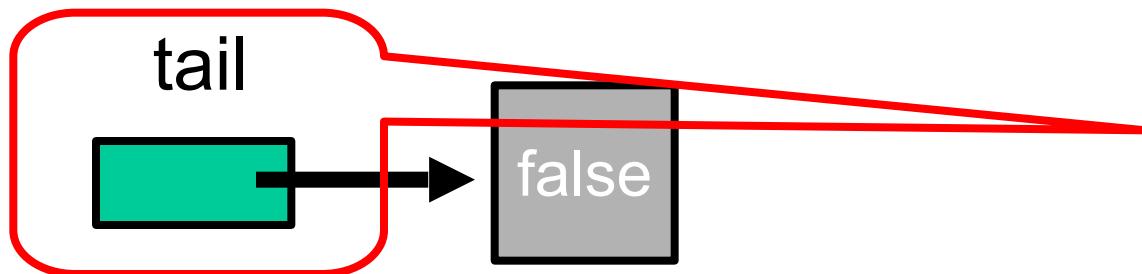
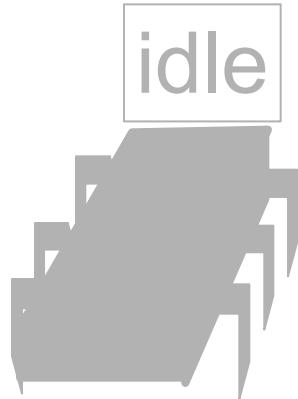
idle



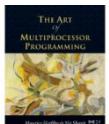
tail



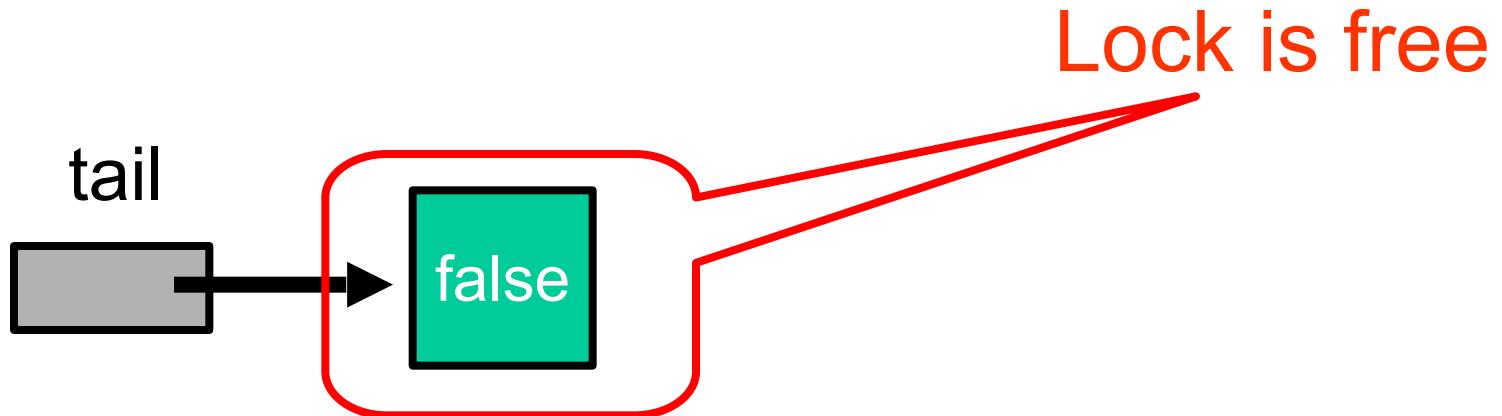
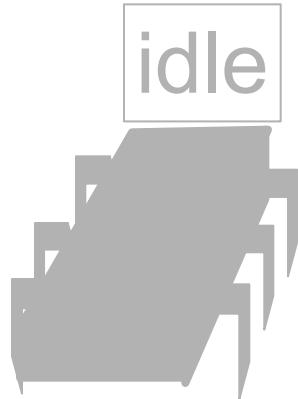
# Initially



Queue tail

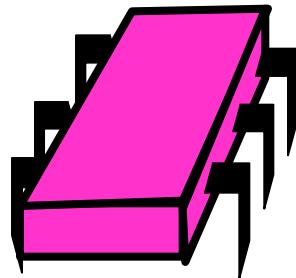


# Initially

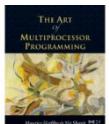
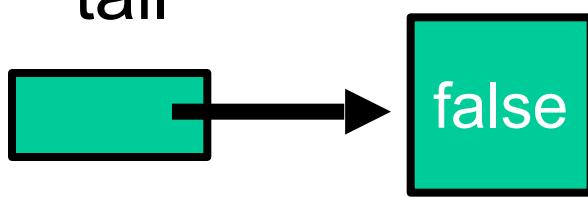


# Initially

idle

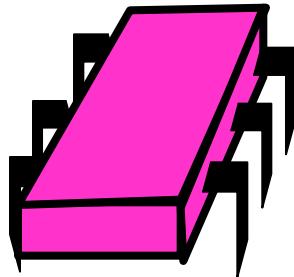


tail

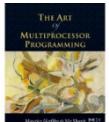
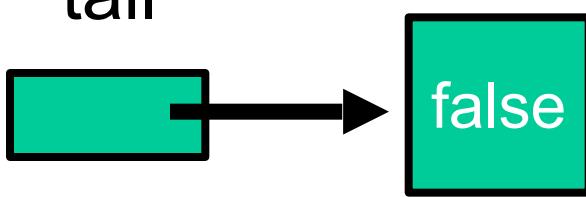


# Purple Wants the Lock

## acquiring

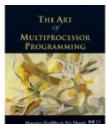
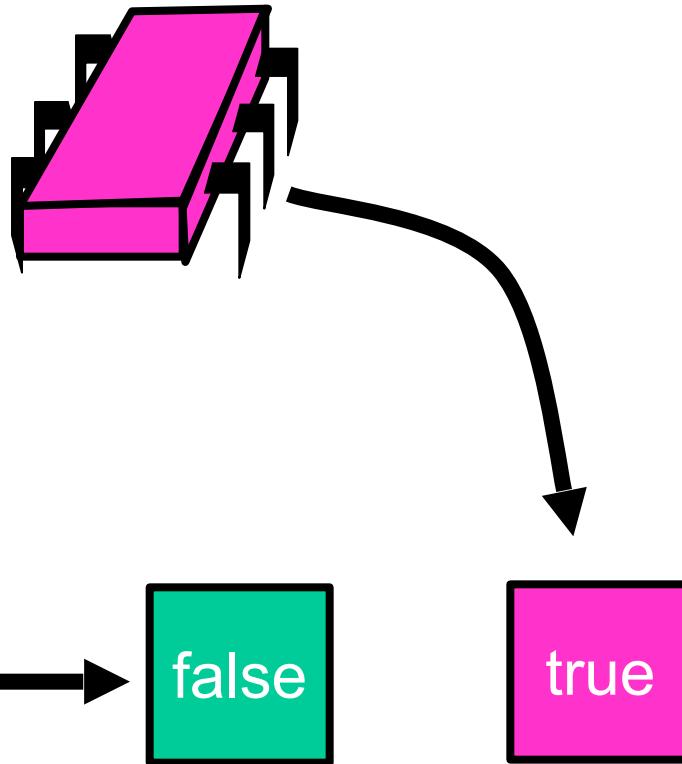


tail



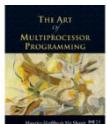
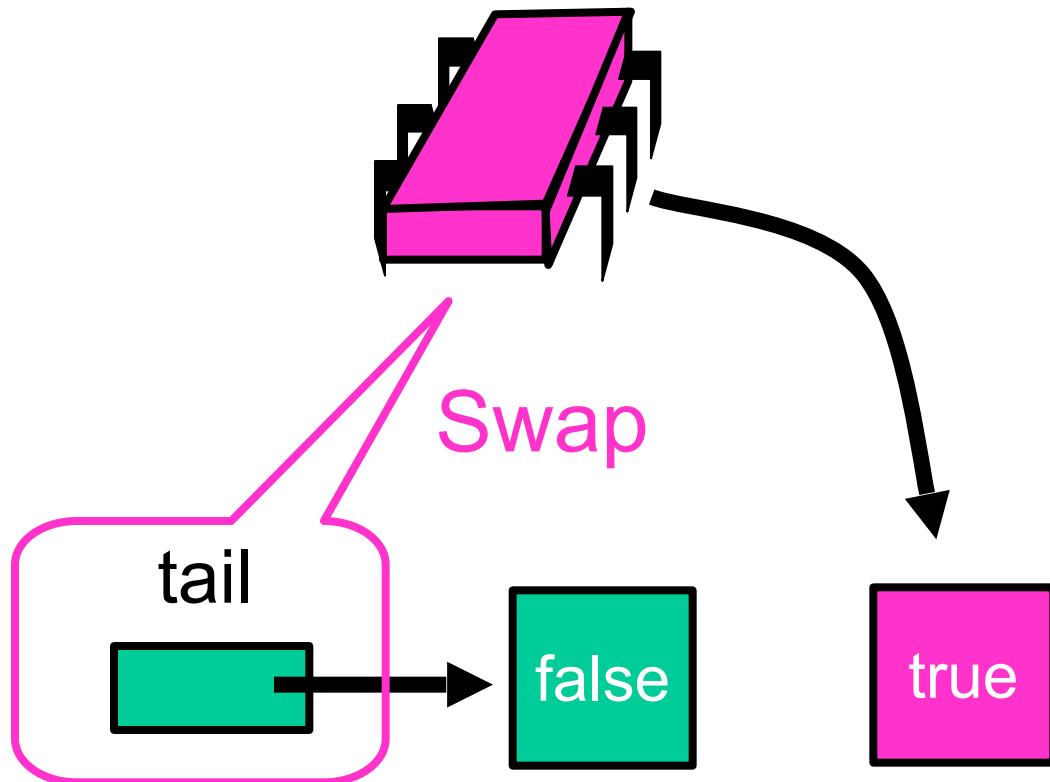
# Purple Wants the Lock

acquiring



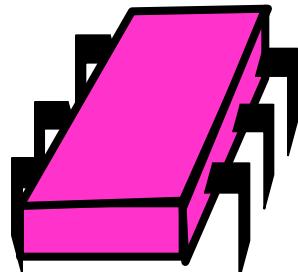
# Purple Wants the Lock

acquiring

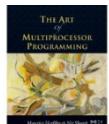
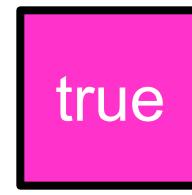
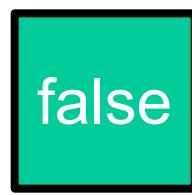


# Purple Has the Lock

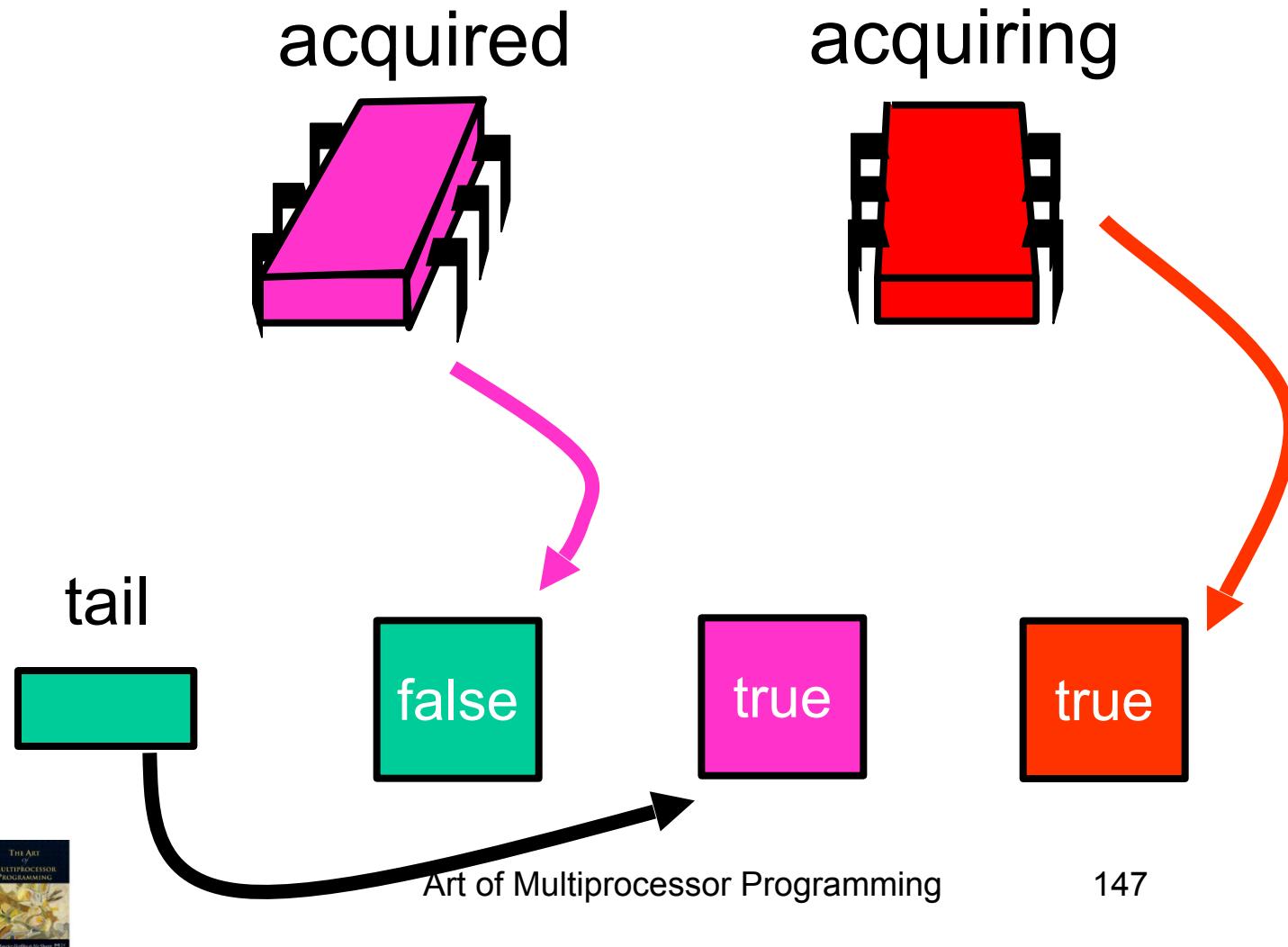
acquired



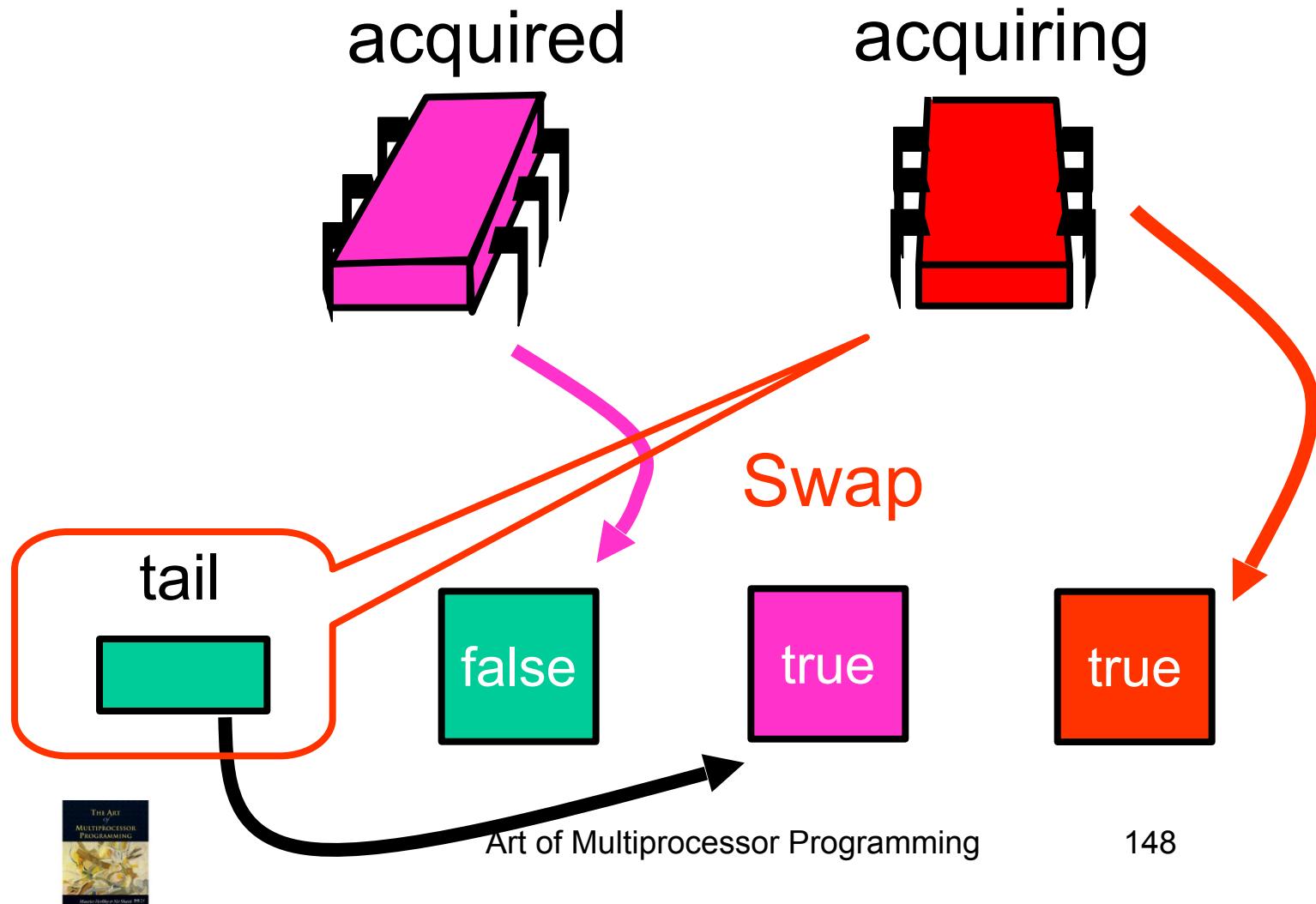
tail



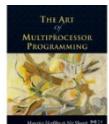
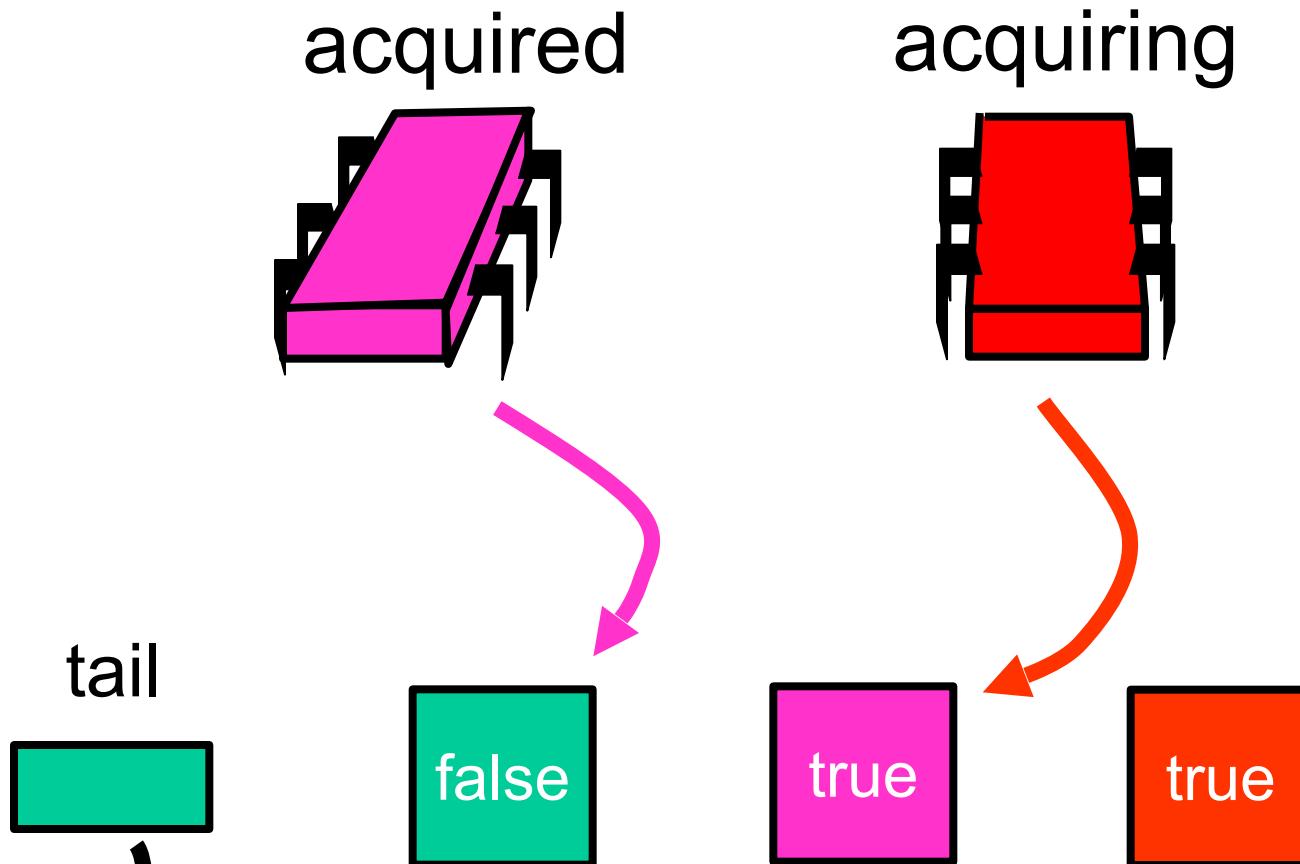
# Red Wants the Lock



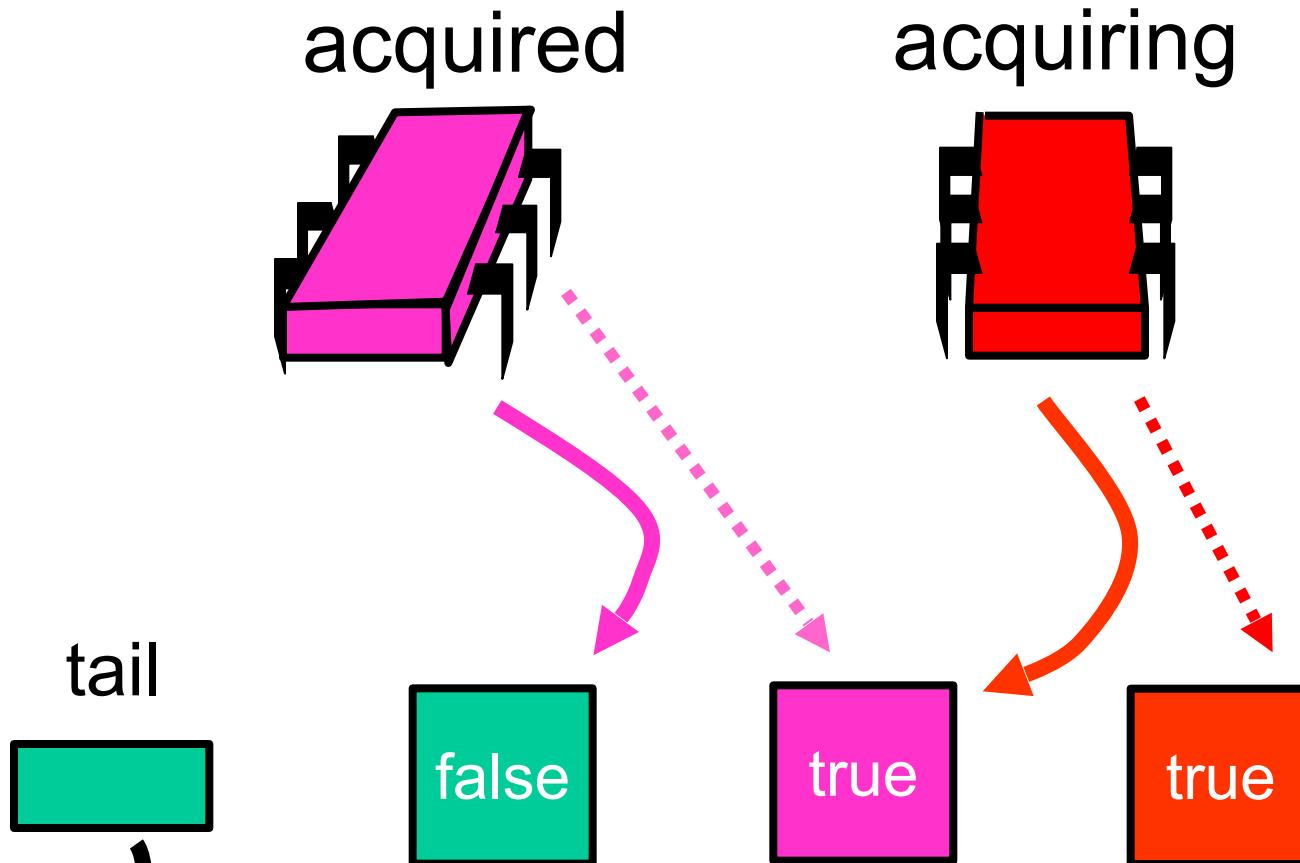
# Red Wants the Lock



# Red Wants the Lock



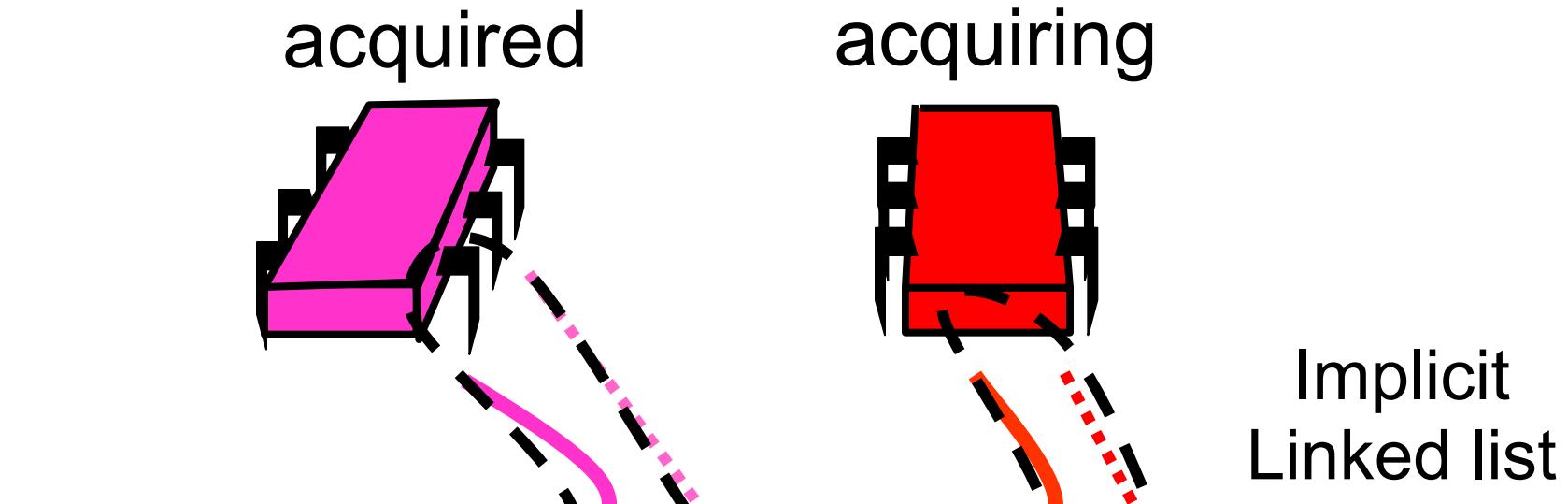
# Red Wants the Lock



Art of Multiprocessor Programming

150

# Red Wants the Lock



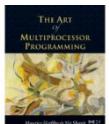
Implicit  
Linked list

tail

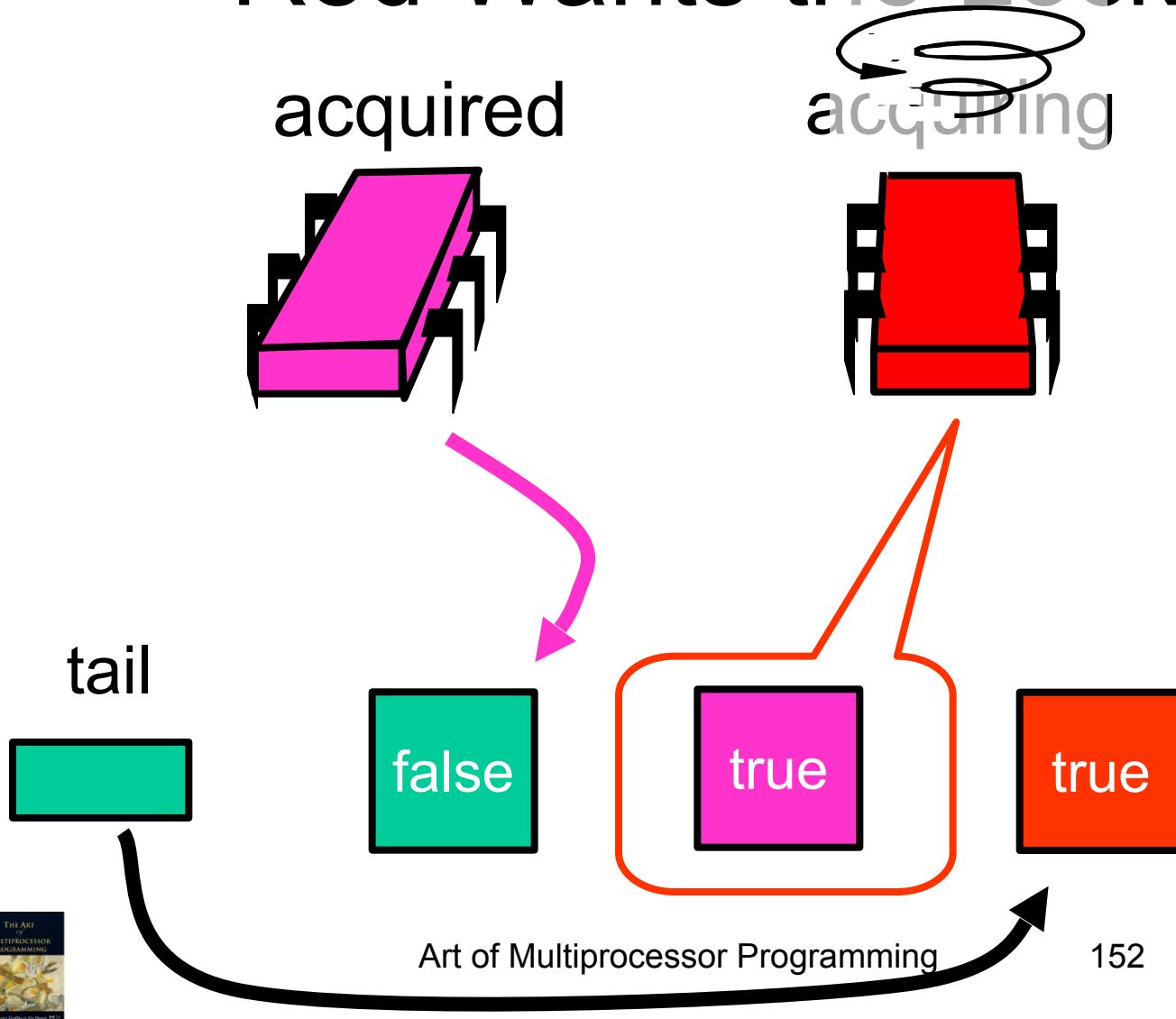
false

true

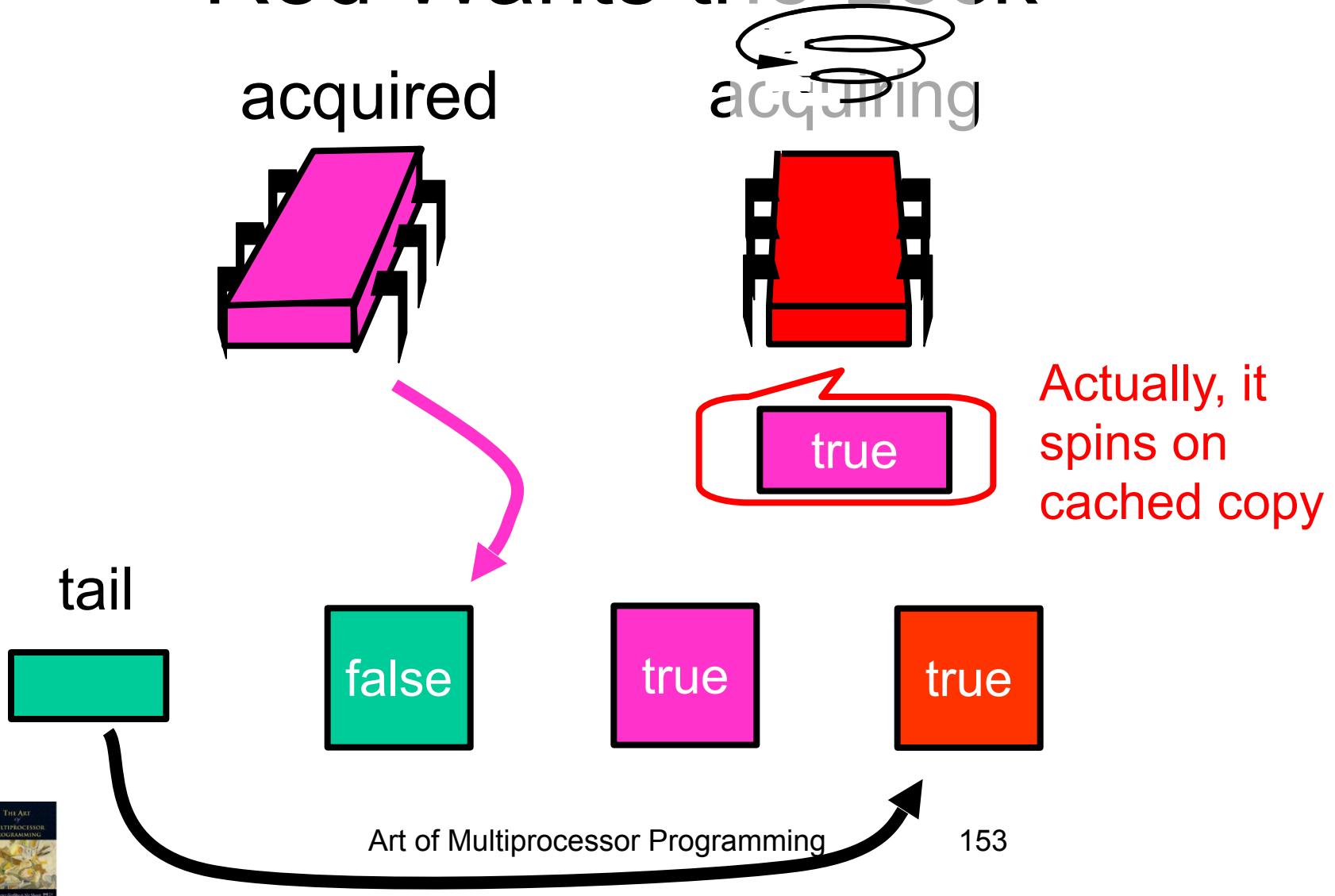
true



# Red Wants the Lock

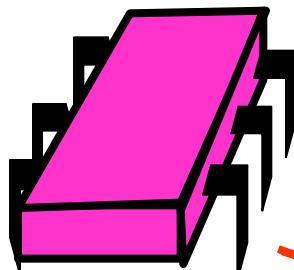


# Red Wants the Lock

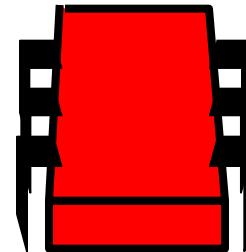


# Purple Releases

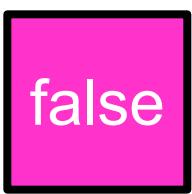
release



acquiring

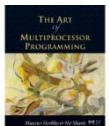


tail



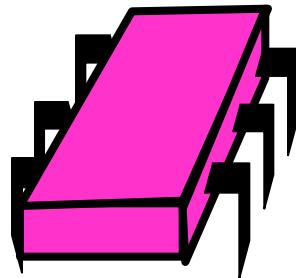
false

Bingo!

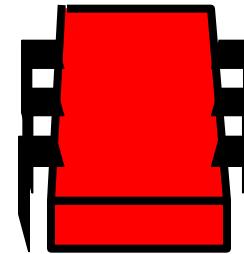


# Purple Releases

released



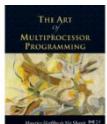
acquired



tail



true

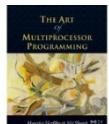


Art of Multiprocessor Programming

155

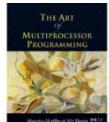
# Space Usage

- Let
  - $L$  = number of locks
  - $N$  = number of threads
- ALock
  - $O(LN)$
- CLH lock
  - $O(L+N)$



# CLH Queue Lock

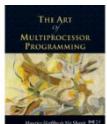
```
class Qnode {  
    AtomicBoolean locked =  
        new AtomicBoolean(true);  
}
```



# CLH Queue Lock

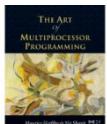
```
class Qnode {  
    AtomicBoolean locked =  
        new AtomicBoolean(true);  
}
```

Not released yet



# CLH Queue Lock

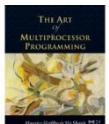
```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    } }
```



# CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    } }
```

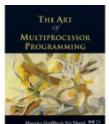
Queue tail



# CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    } }
```

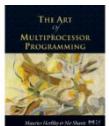
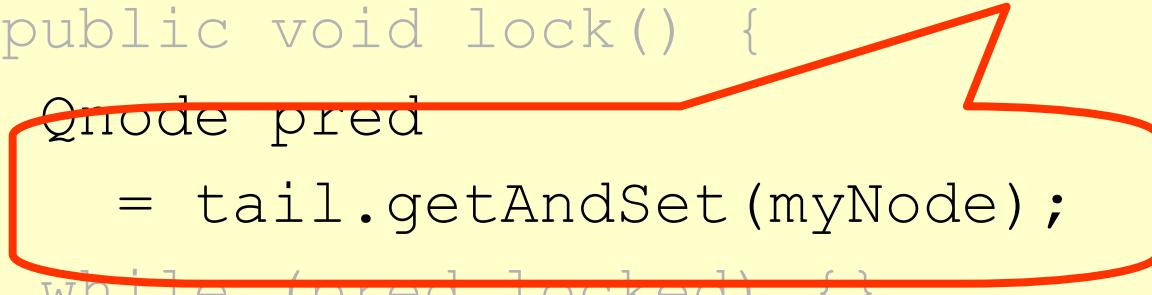
Thread-local Qnode



# CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
        = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    } }
```

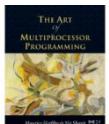
Swap in my node



# CLH Queue Lock

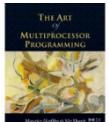
```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
        = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

Spin until predecessor  
releases lock



# CLH Queue Lock

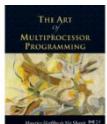
```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```



# CLH Queue Lock

```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

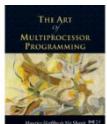
Notify successor



# CLH Queue Lock

```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

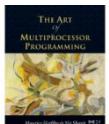
Recycle  
predecessor's node



# CLH Queue Lock

```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

(we don't actually reuse myNode. Code in book shows how it's done.)



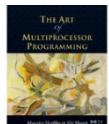
# CLH Lock

- Good
  - Lock release affects predecessor only
  - Small, constant-sized space
- Bad
  - Doesn't work for uncached NUMA architectures



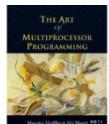
# CLH Lock

- Each thread spins on predecessor's memory
- Could be far away ...



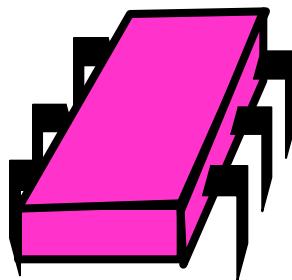
# MCS Lock

- FIFO order
- Spin on local memory only
- Small, Constant-size overhead

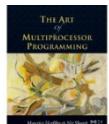
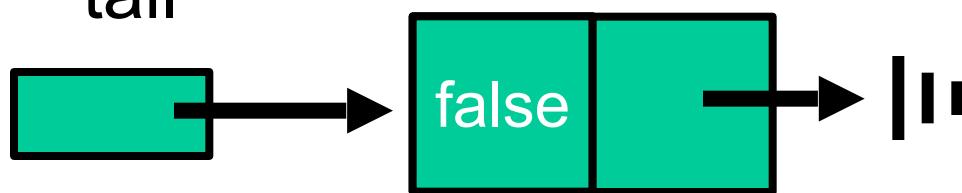


# Initially

idle

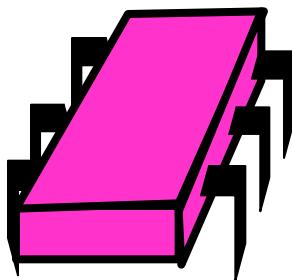


tail



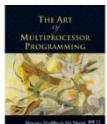
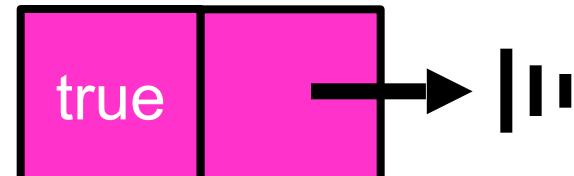
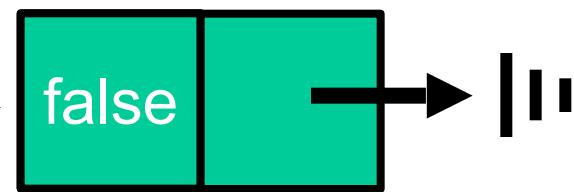
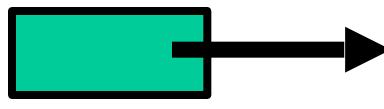
# Acquiring

acquiring

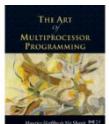
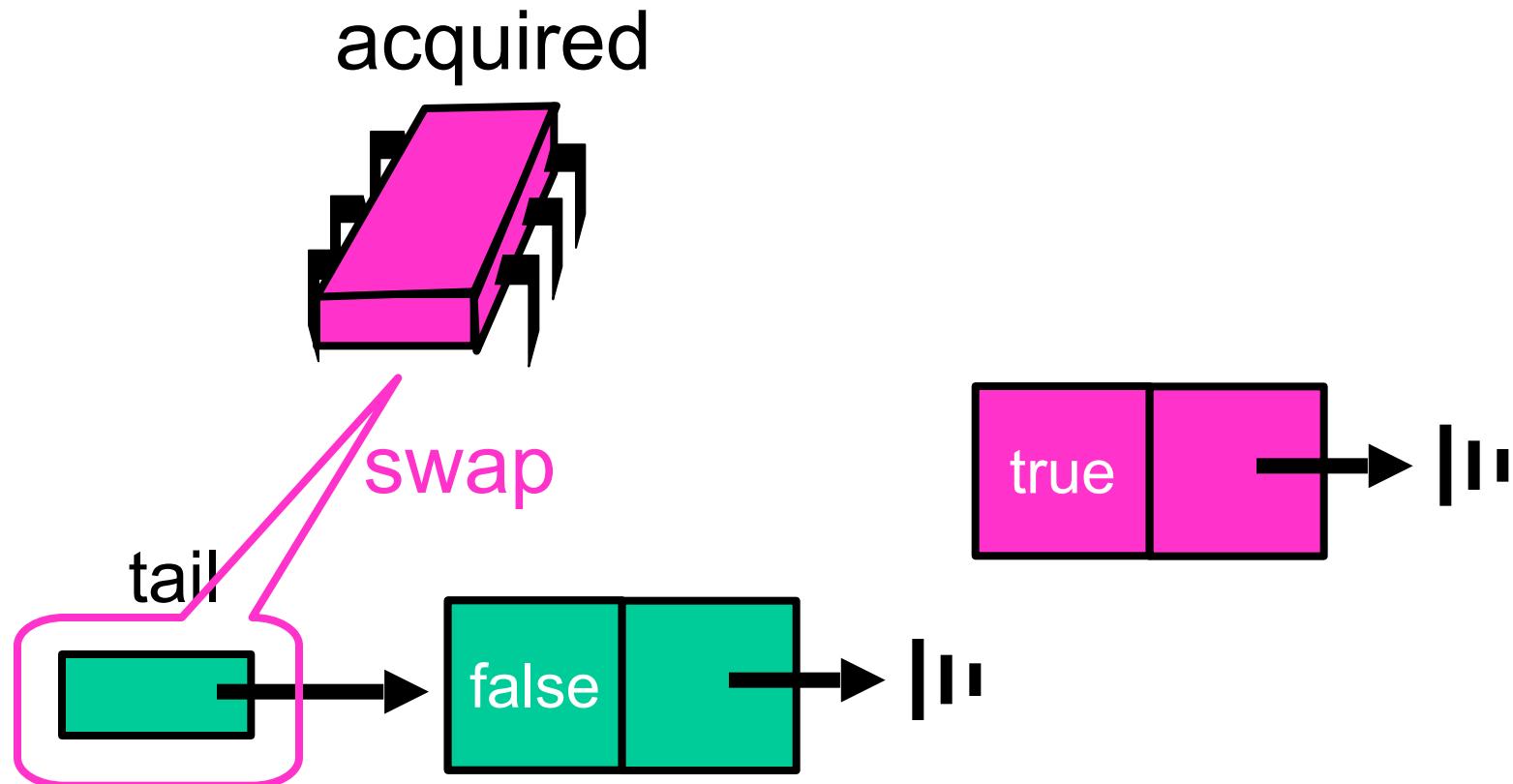


(allocate Qnode)

tail

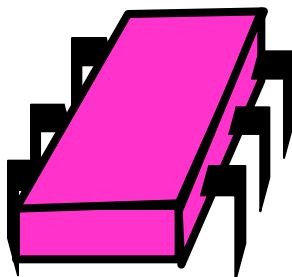


# Acquiring

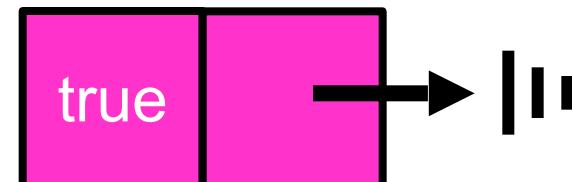
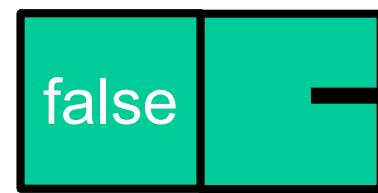


# Acquiring

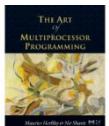
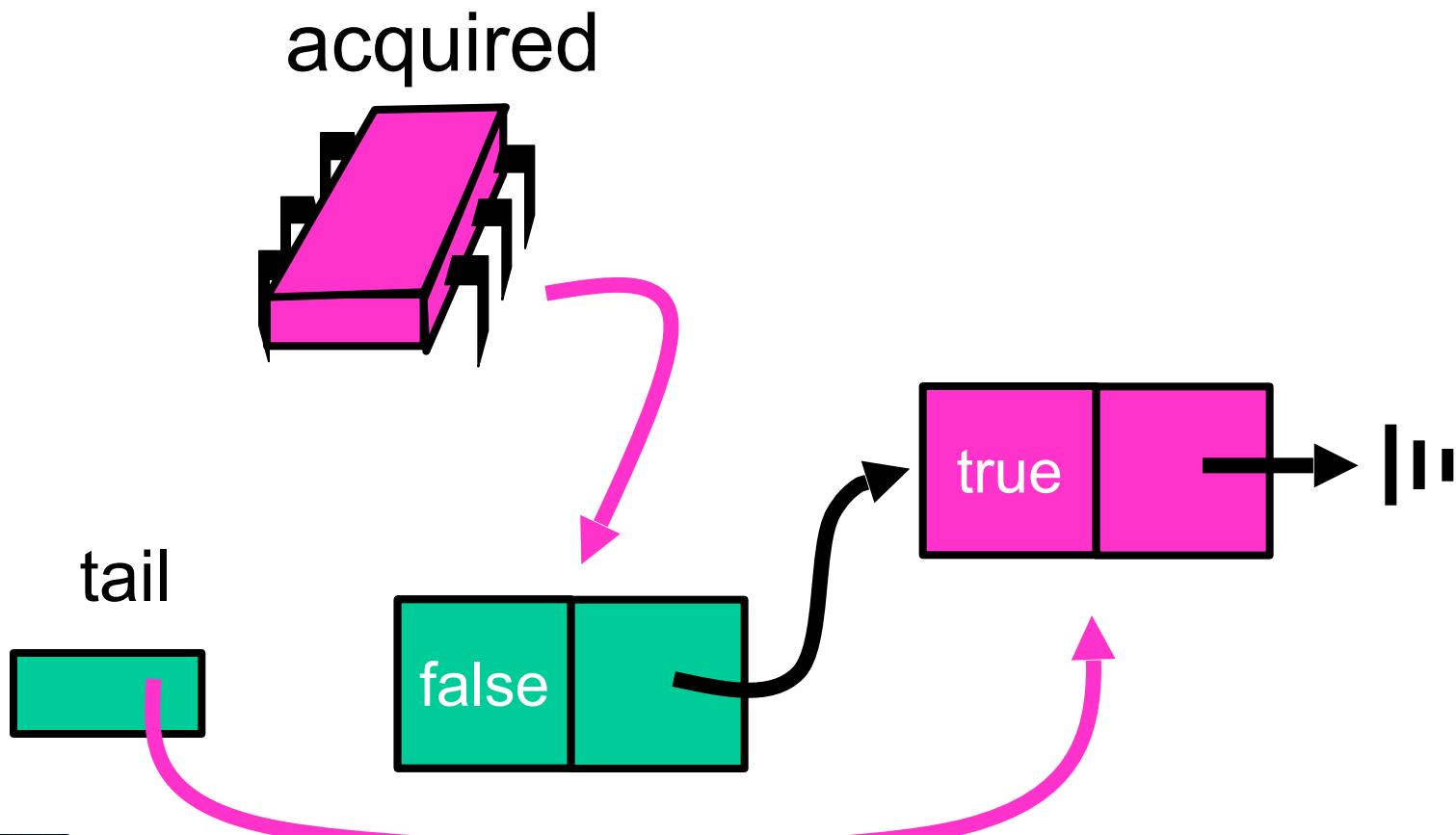
acquired



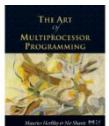
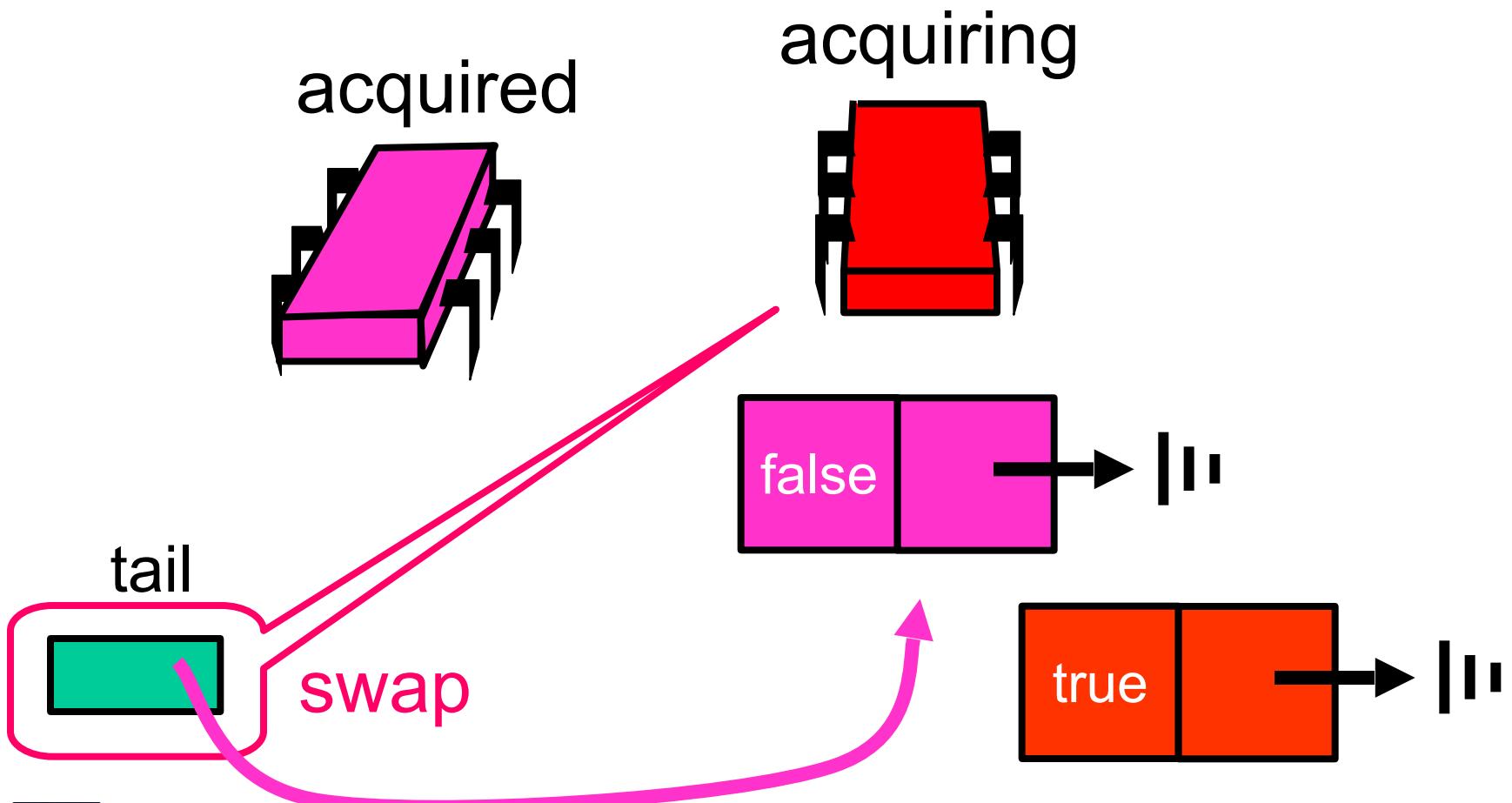
tail



# Acquired

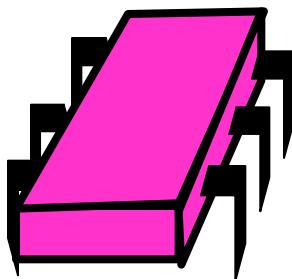


# Acquiring

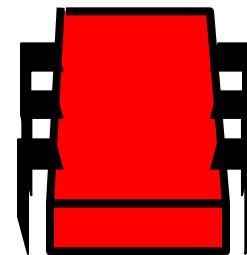


# Acquiring

acquired



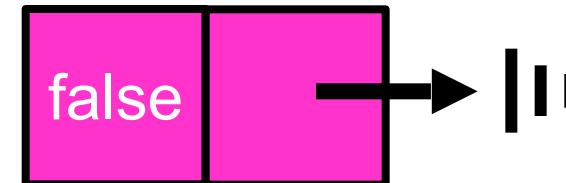
acquiring



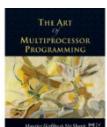
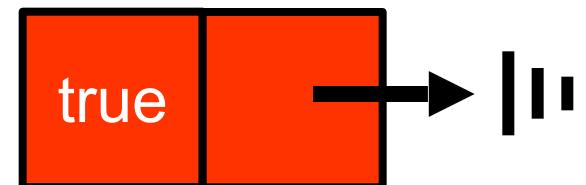
tail



false

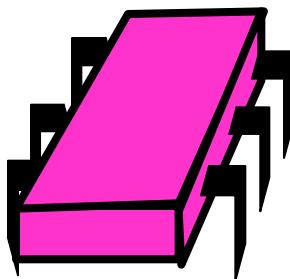


true

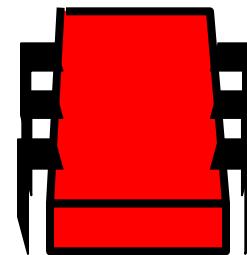


# Acquiring

acquired



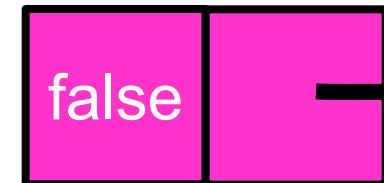
acquiring



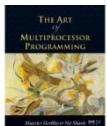
tail



false



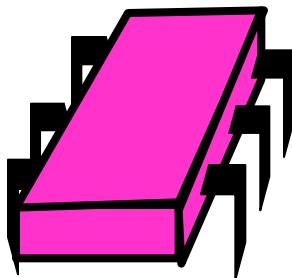
true



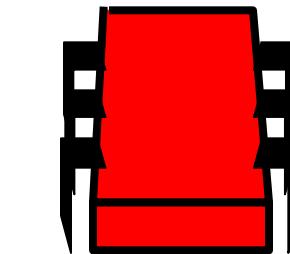
# Acquiring

acquiring

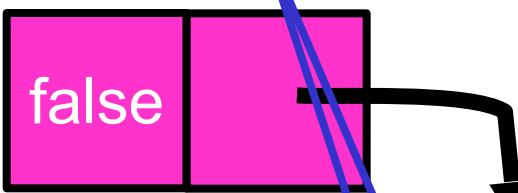
acquired



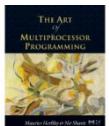
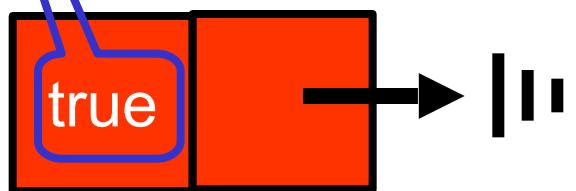
tail

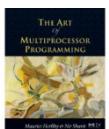
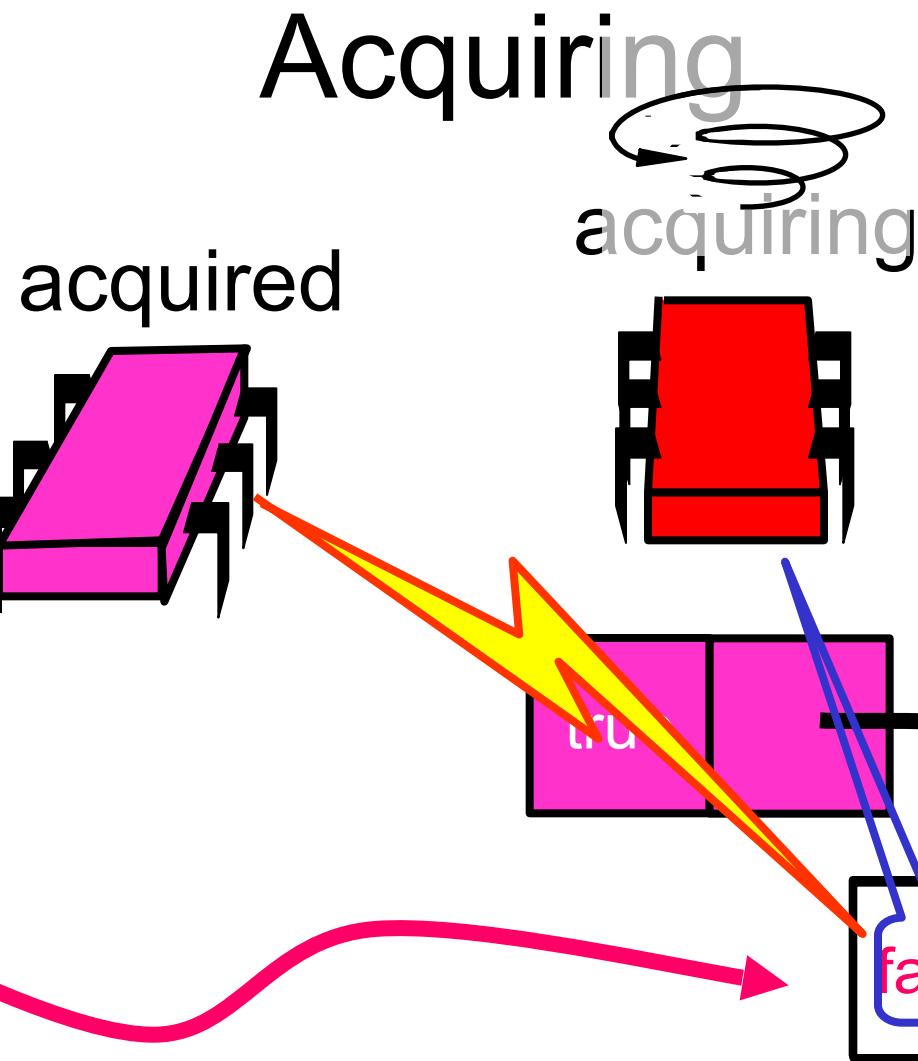


false



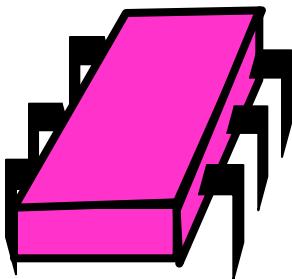
true





# Acquiring

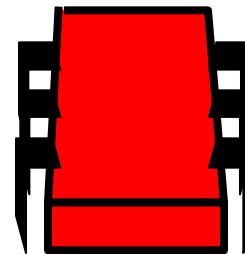
acquired



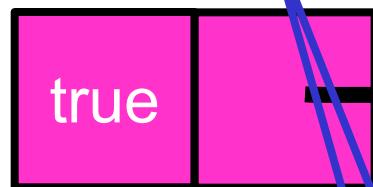
tail



acquiring



true

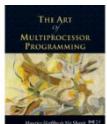


Yes!



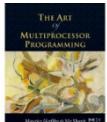
# MCS Queue Lock

```
class Qnode {  
    boolean locked = false;  
    qnode next = null;  
}
```



# MCS Queue Lock

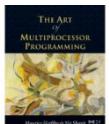
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        } } }
```



# MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        } } }
```

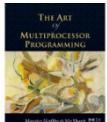
Make a  
QNode



# MCS Queue Lock

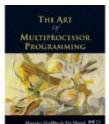
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        } } }
```

**add my Node to  
the tail of queue**



# MCS Queue Lock

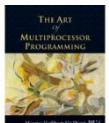
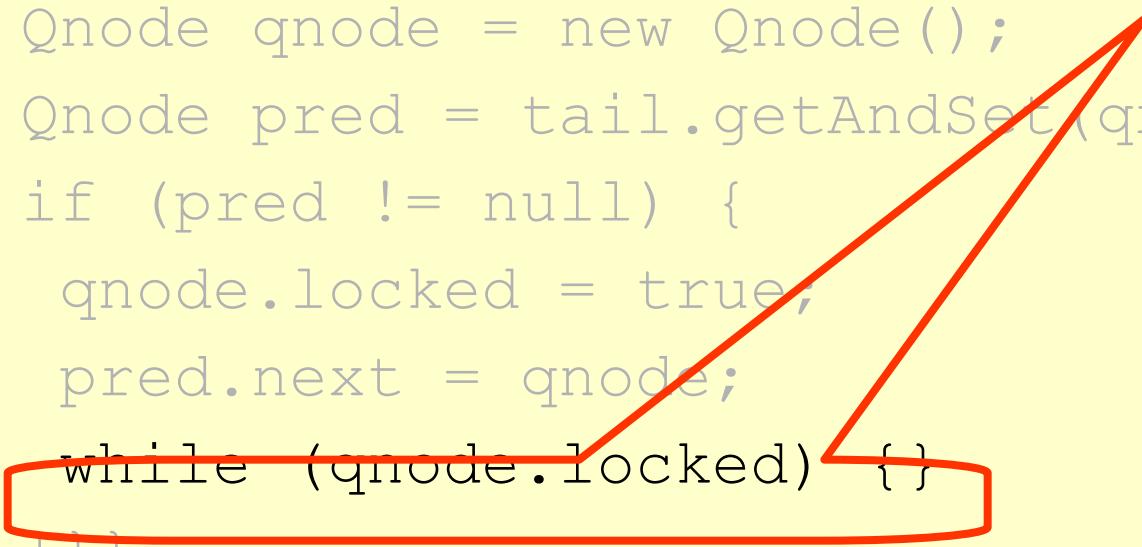
```
class MCSLock implements Lock {  
    AtomicReference tail;  
  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        } } }  
Fix if queue was  
non-empty
```



# MCS Queue Lock

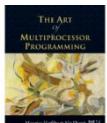
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        }  
    }  
}
```

Wait until  
unlocked



# MCS Queue Unlock

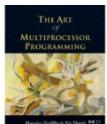
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) { }  
        }  
        qnode.next.locked = false;  
    } }
```



# MCS Queue Lock

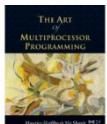
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null))  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    } }
```

Missing  
successor?



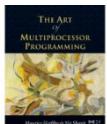
# MCS Queue Lock

```
( If really no successor,           k {  
    return;  
    if (qnode.next == null) {  
        if (tail.CAS(qnode, null)  
            return;  
        while (qnode.next == null) { }  
    }  
    qnode.next.locked = false;  
} }
```



# MCS Queue Lock

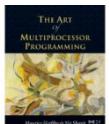
( Otherwise wait for successor to catch up ) {  
 if (qnode.next == null) {  
 if (tail.CAS(qnode, null)  
 return;  
 while (qnode.next == null) {}  
 }  
 qnode.next.locked = false;  
} }



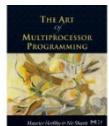
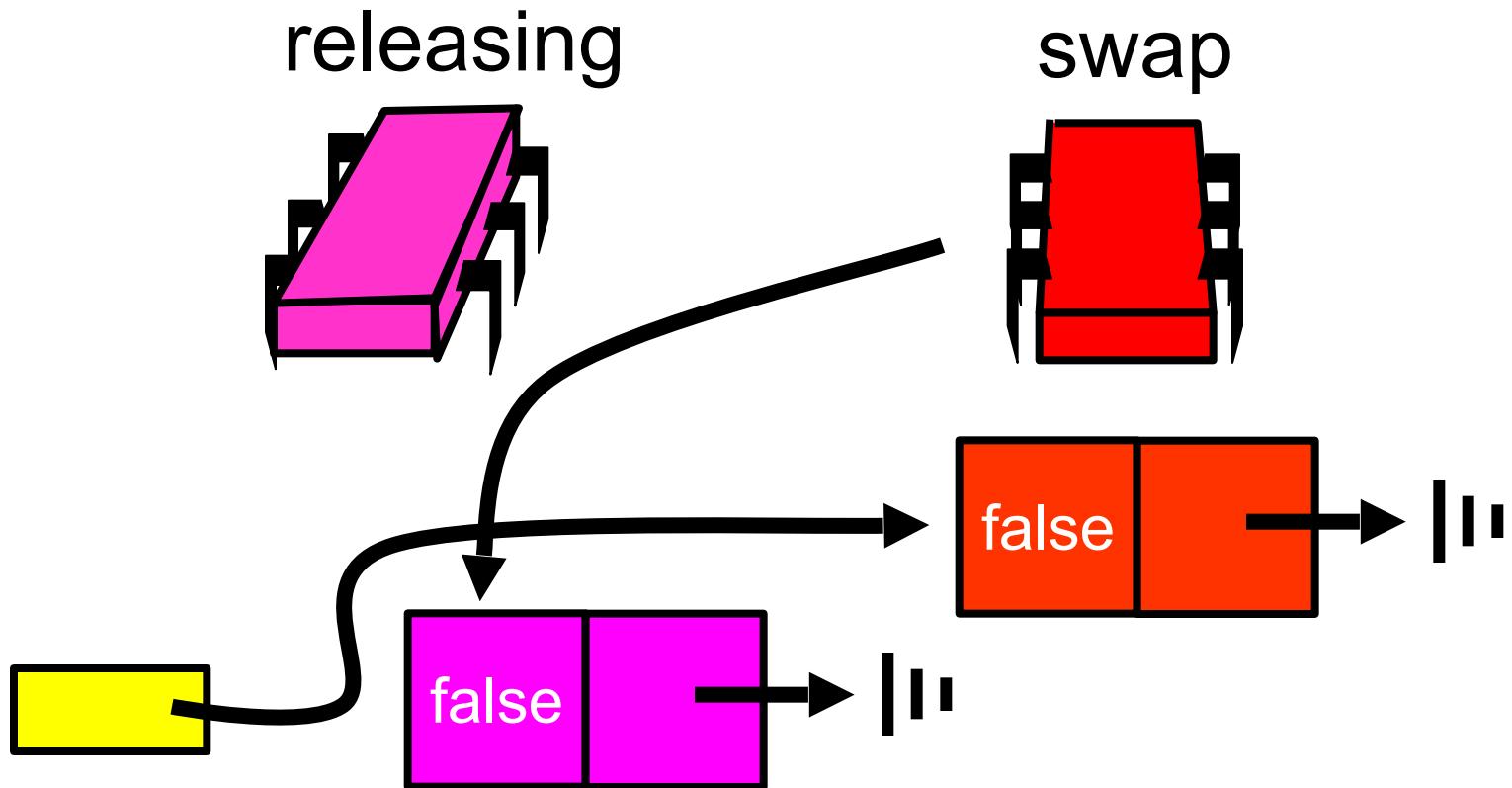
# MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference<QNode> tail = new AtomicReference<QNode>(...);  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null))  
                return;  
            while (qnode.next == null) { }  
        }  
        qnode.next.locked = false;  
    }  
}
```

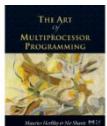
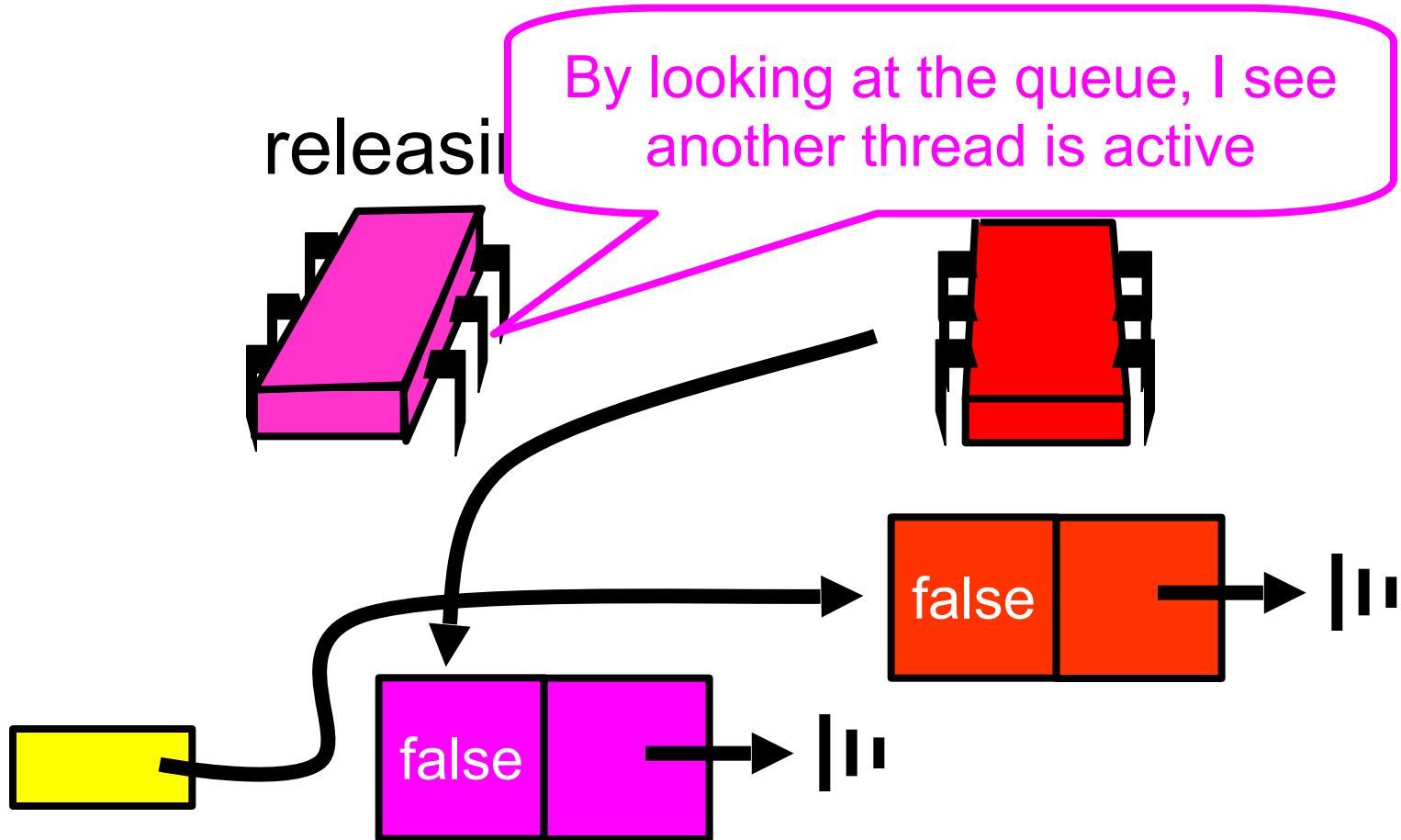
Pass lock to successor



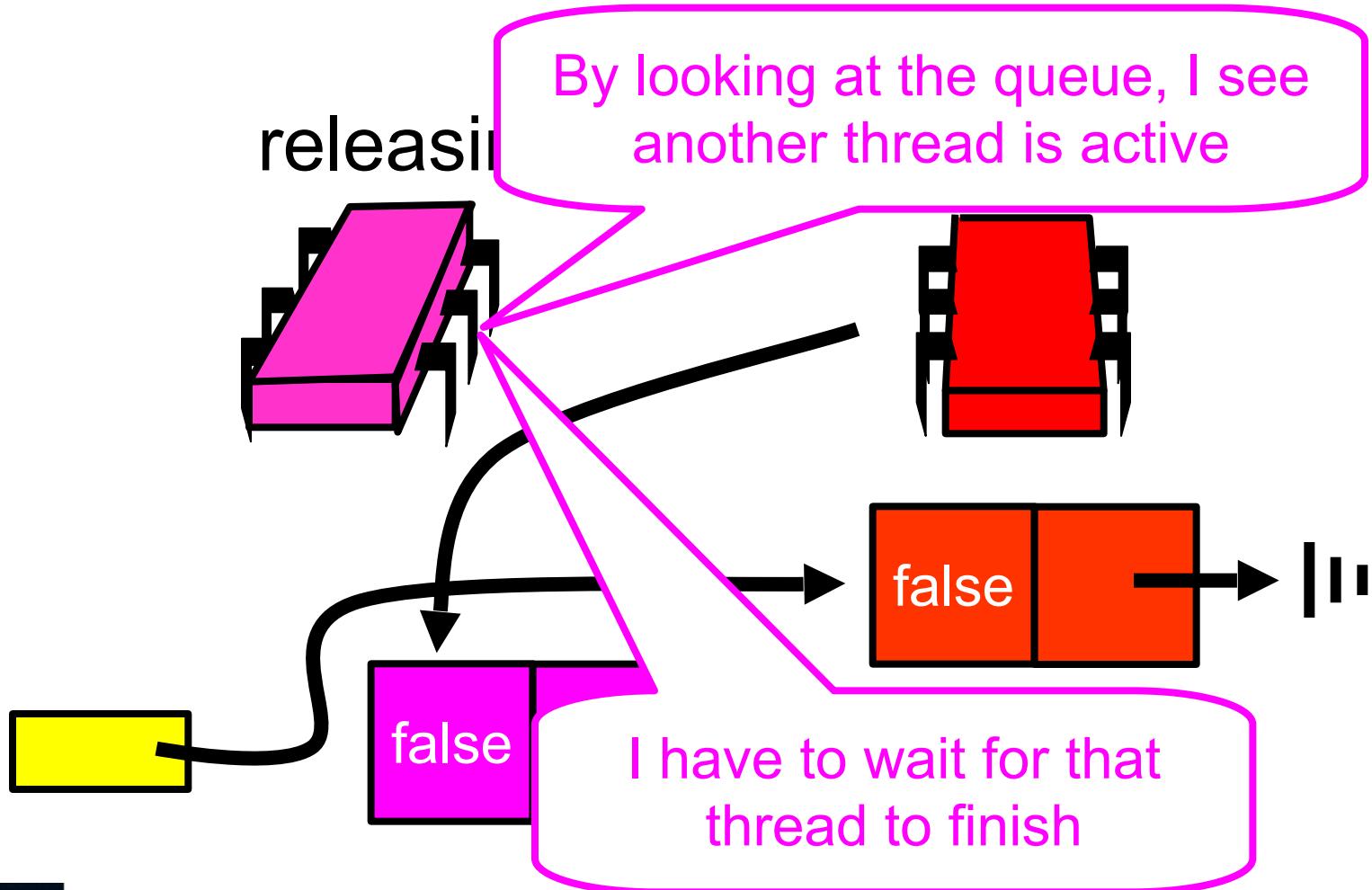
# Purple Release



# Purple Release

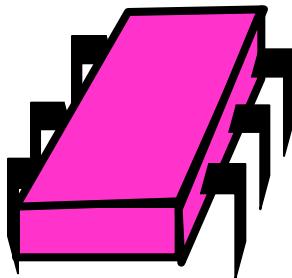


# Purple Release

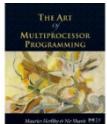
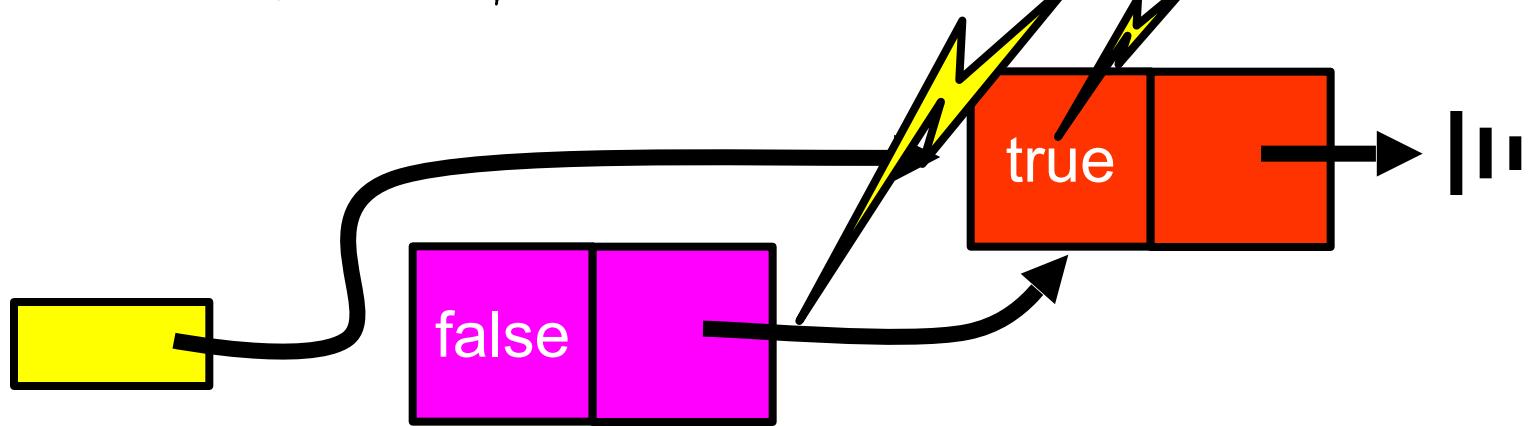
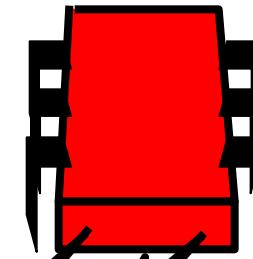


# Purple Release

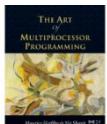
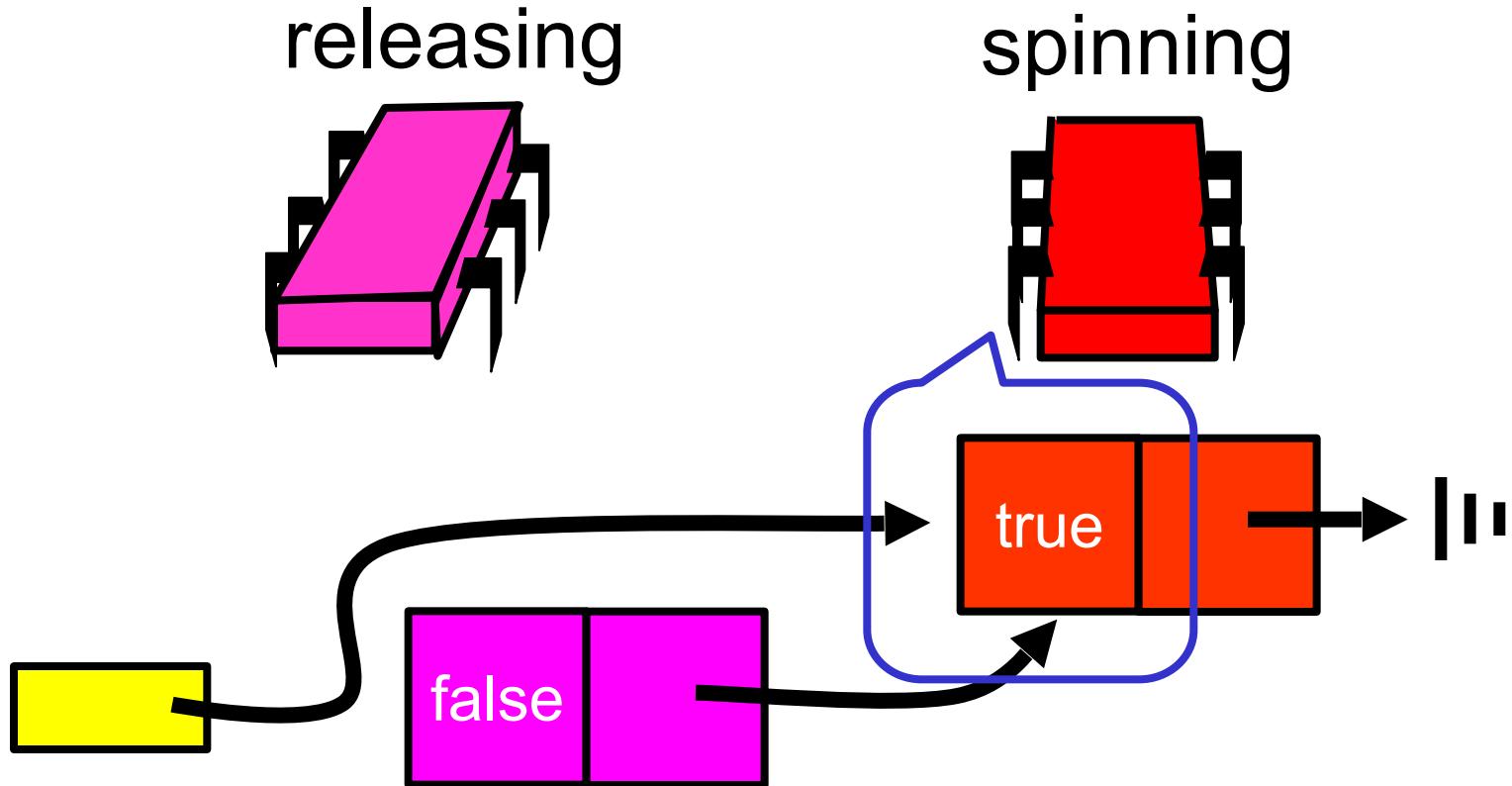
releasing



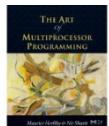
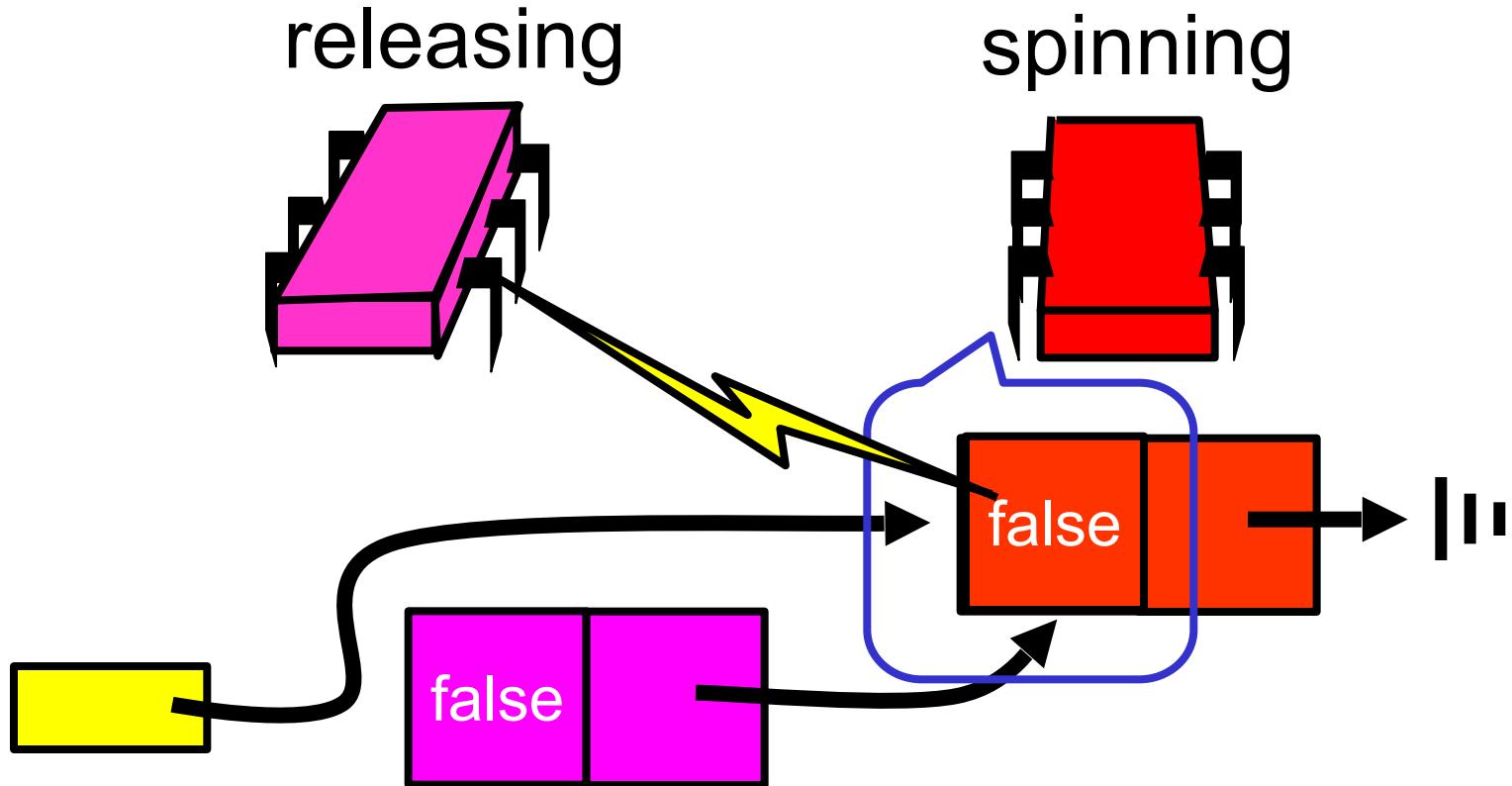
prepare to spin



# Purple Release

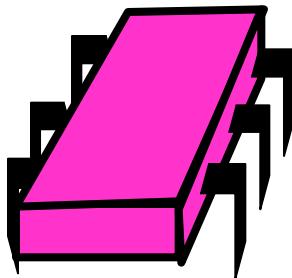


# Purple Release

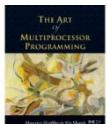
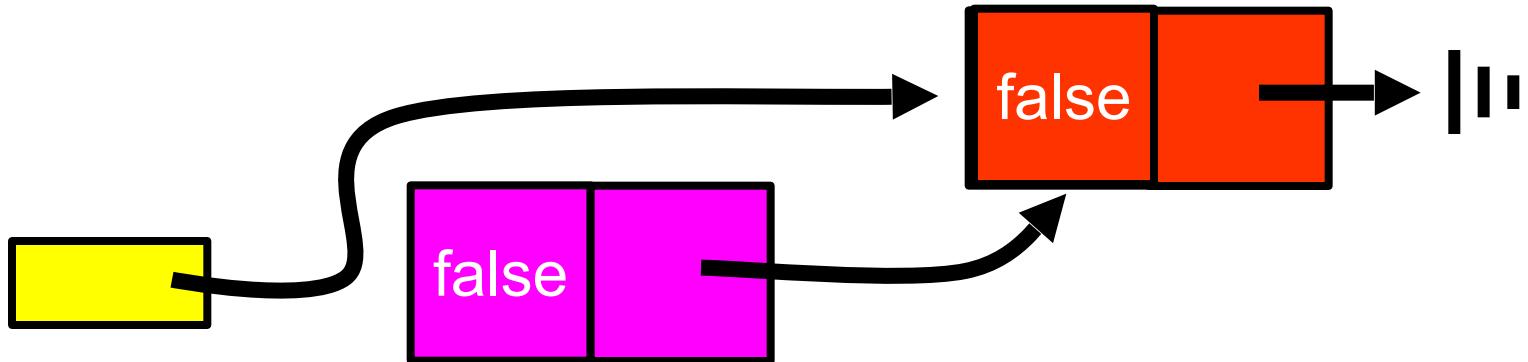
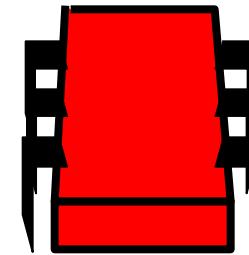


# Purple Release

releasing

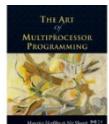


Acquired lock



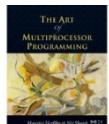
# Abortable Locks

- What if you want to give up waiting for a lock?
- For example
  - Timeout
  - Database transaction aborted by user



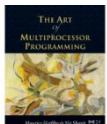
# Back-off Lock

- Aborting is trivial
  - Just return from lock() call
- Extra benefit:
  - No cleaning up
  - Wait-free
  - Immediate return

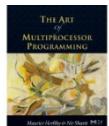
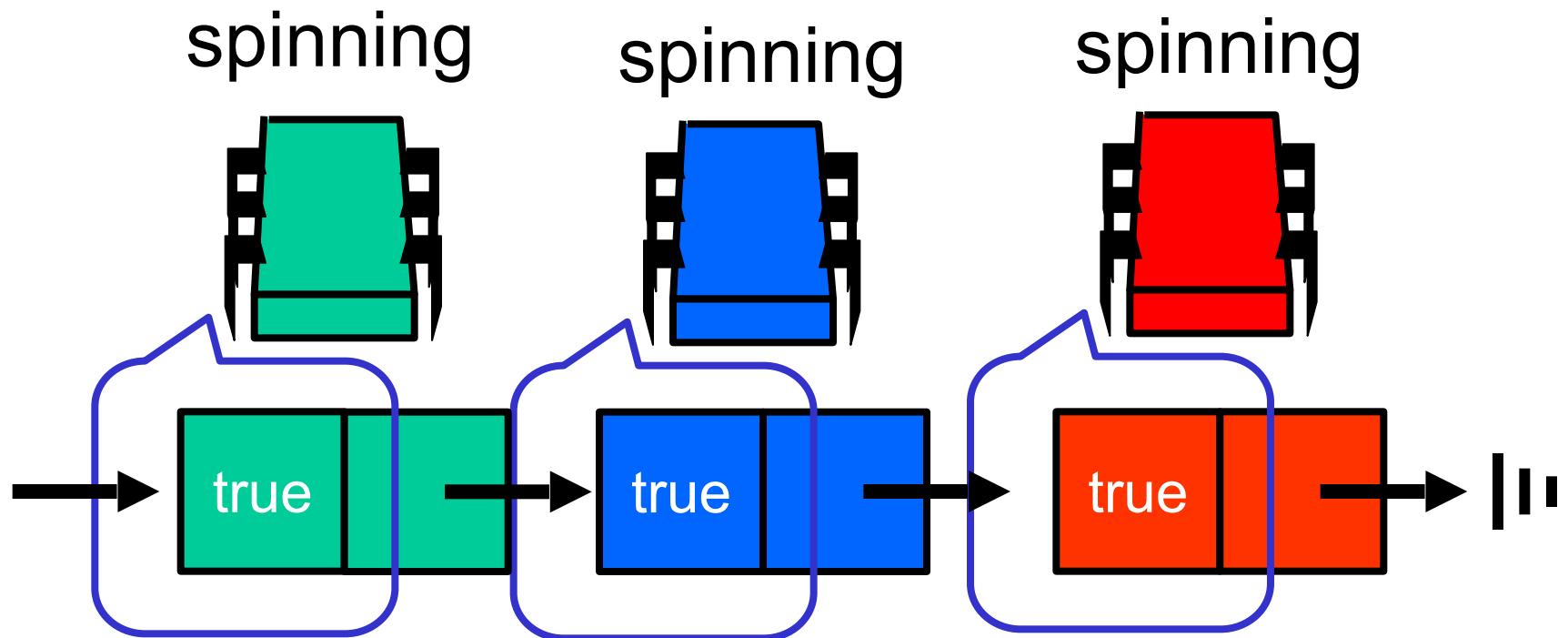


# Queue Locks

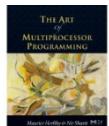
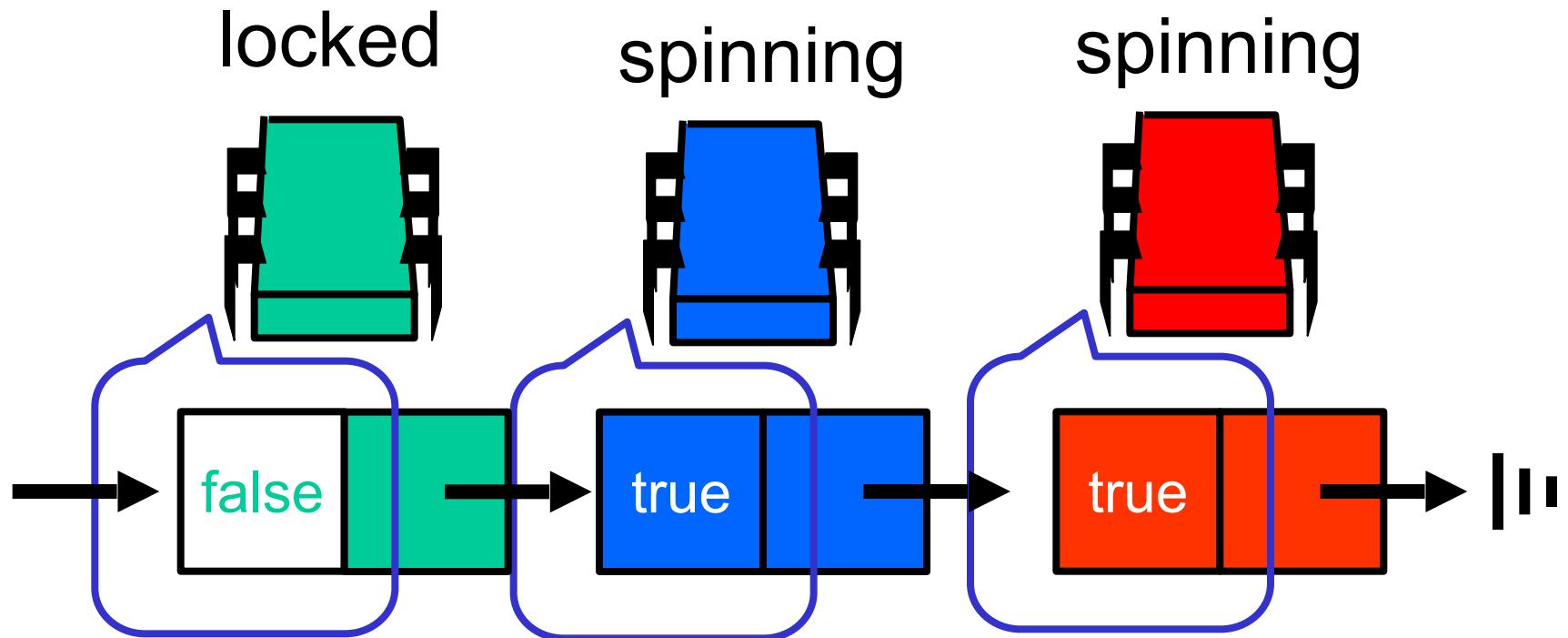
- Can't just quit
  - Thread in line behind will starve
- Need a graceful way out



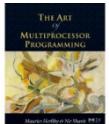
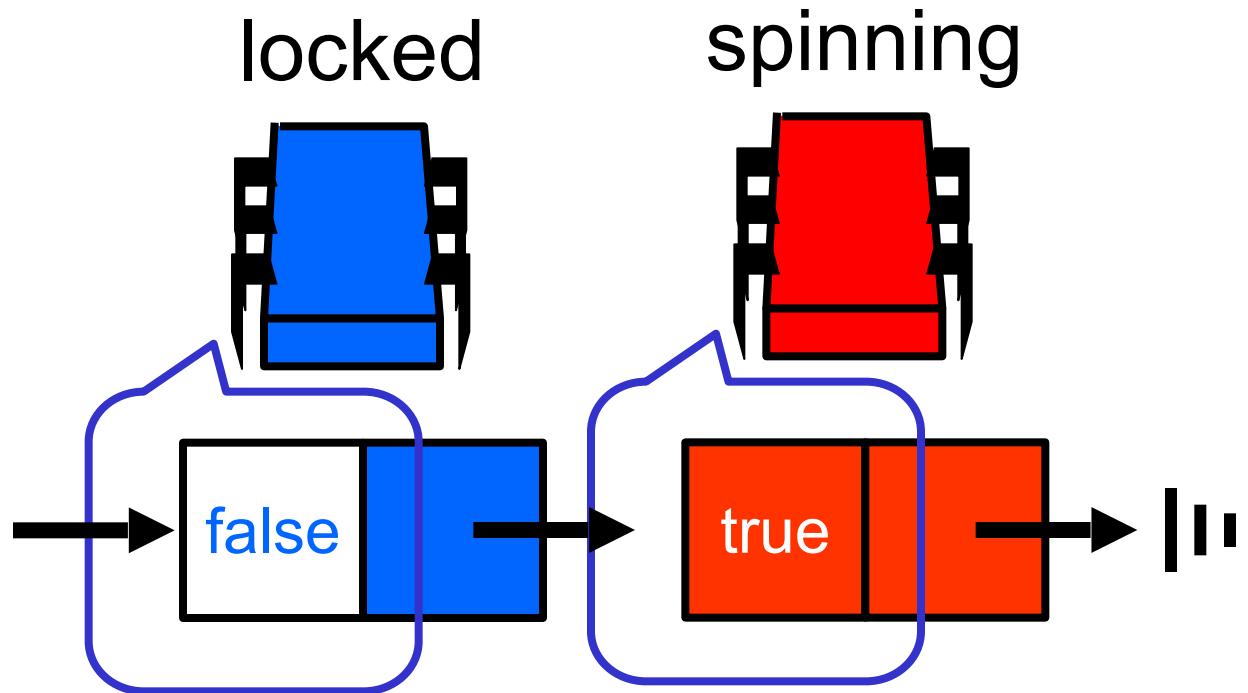
# Queue Locks



# Queue Locks

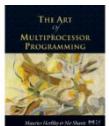
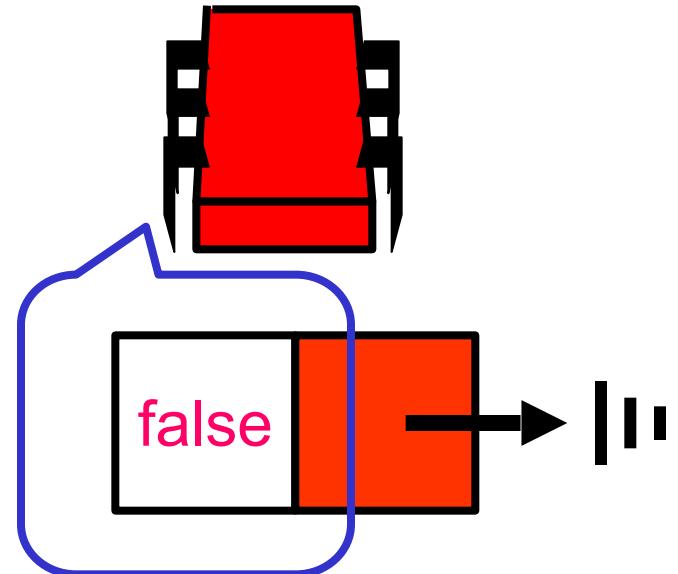


# Queue Locks

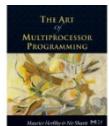
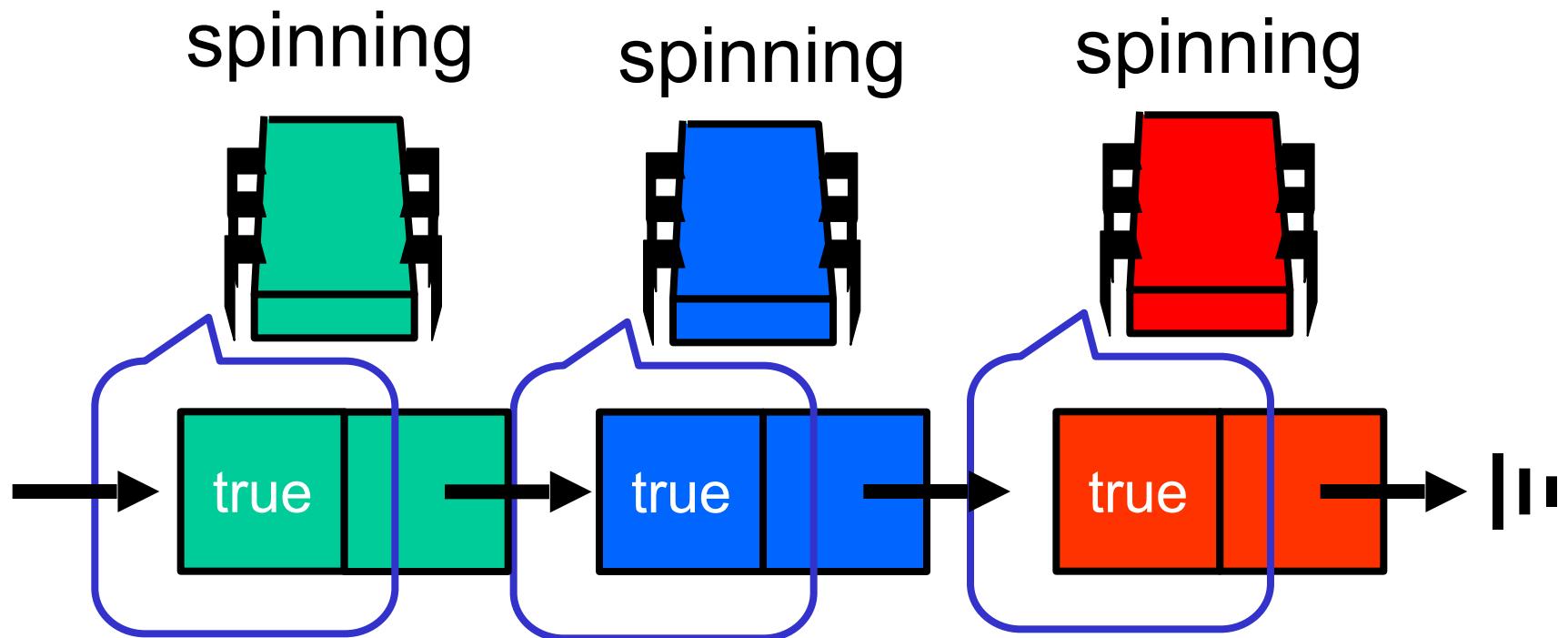


# Queue Locks

locked

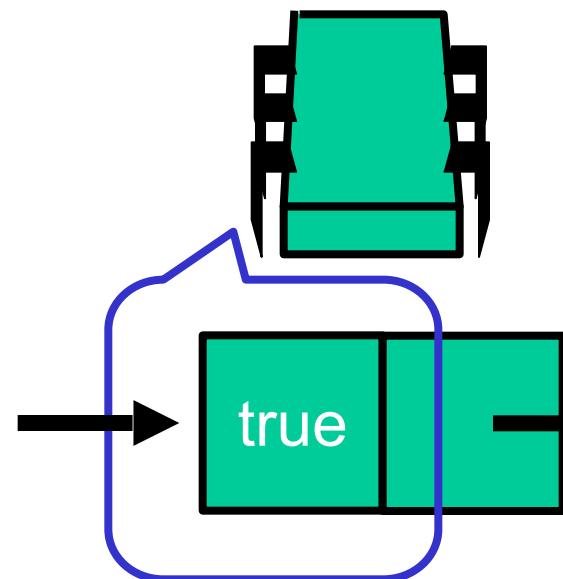


# Queue Locks

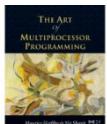
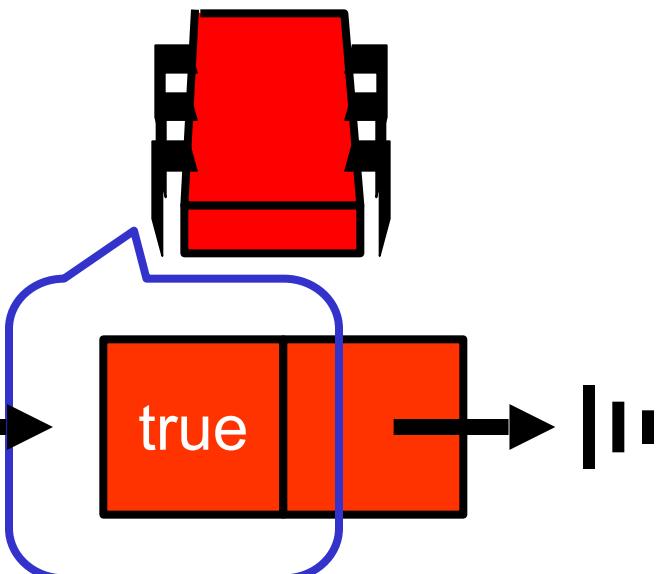


# Queue Locks

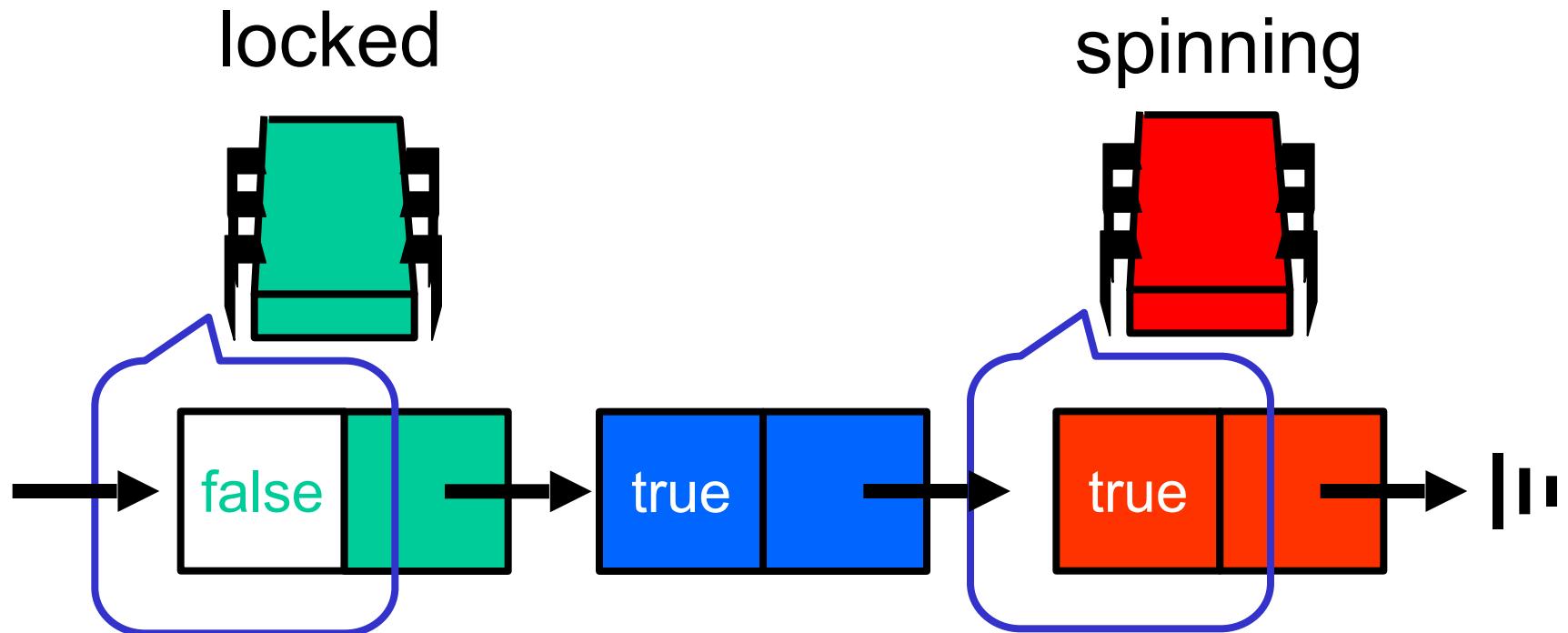
spinning



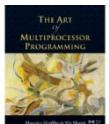
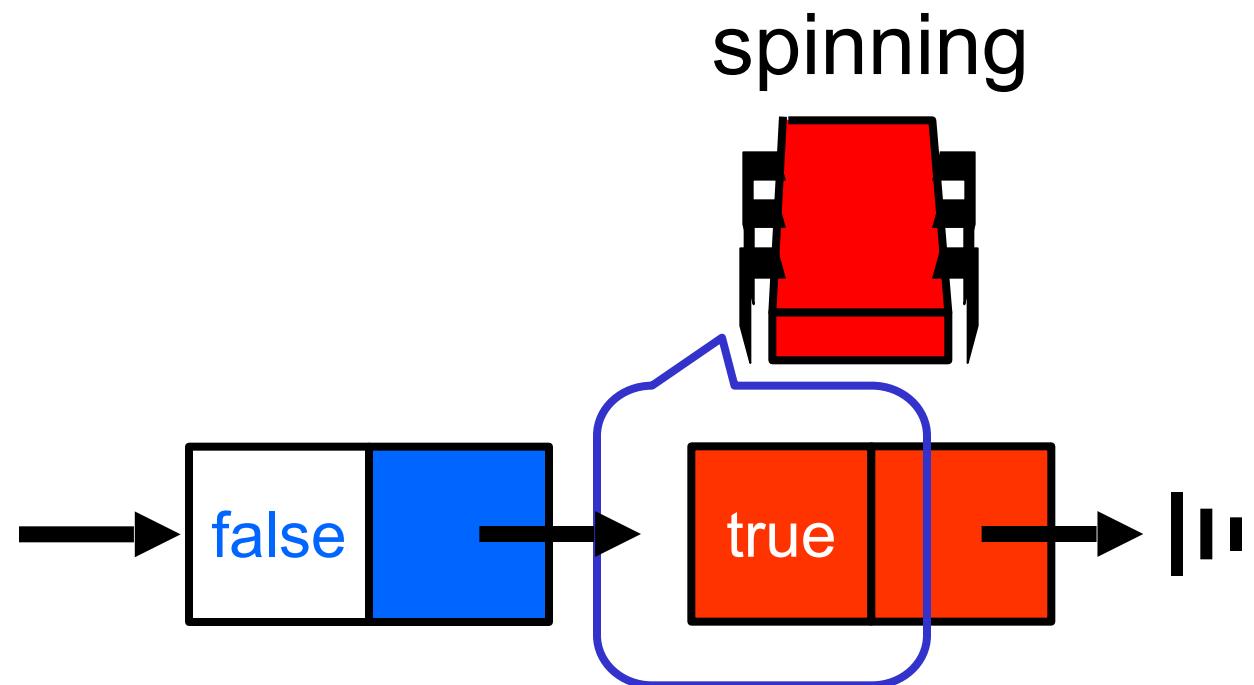
spinning



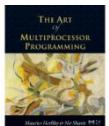
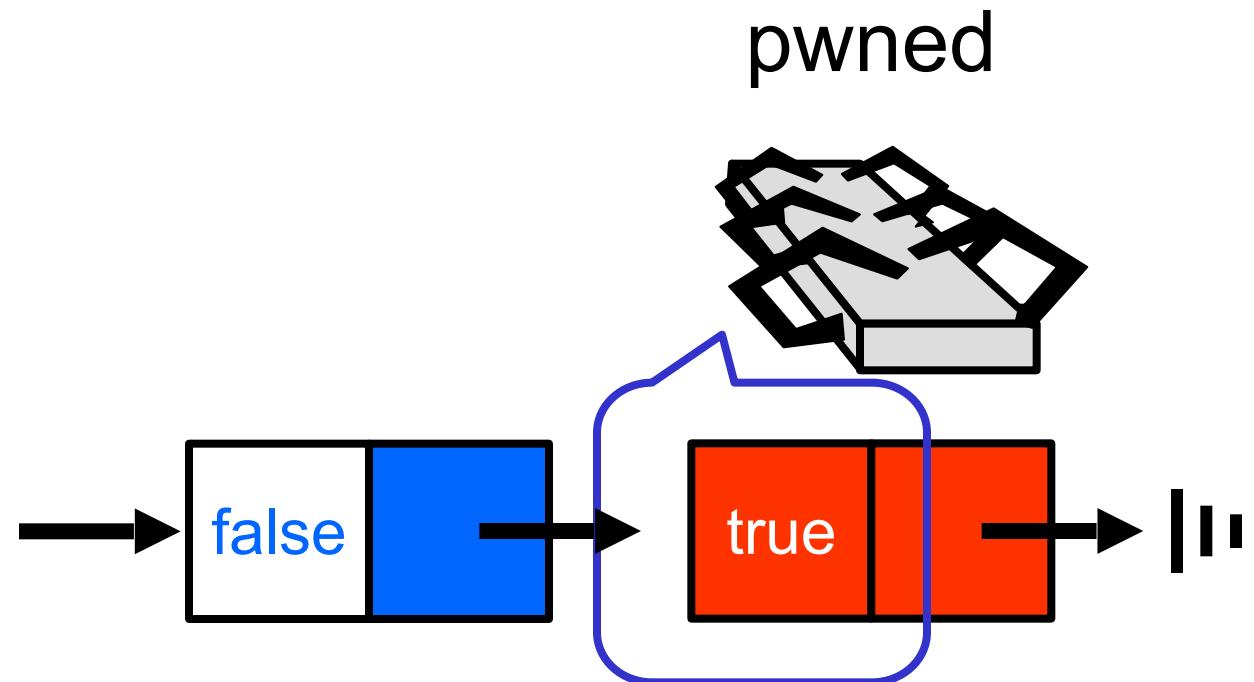
# Queue Locks



# Queue Locks

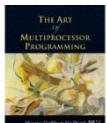


# Queue Locks



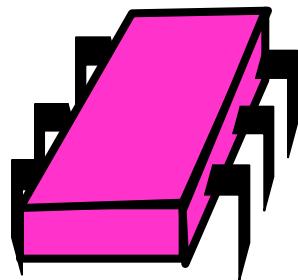
# Abortable CLH Lock

- When a thread gives up
  - Removing node in a wait-free (non-locking) way is hard
- Idea:
  - let successor deal with it.



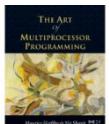
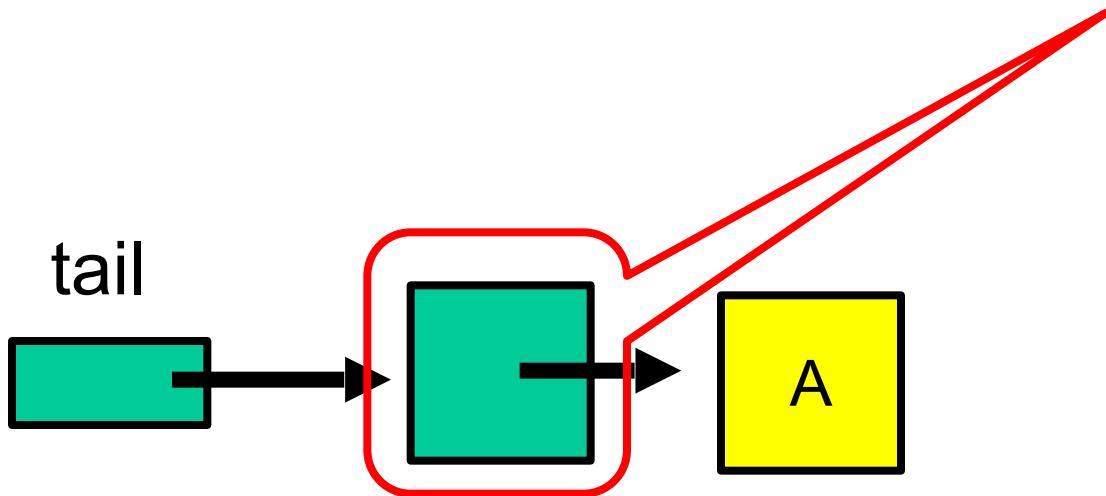
# Initially

idle

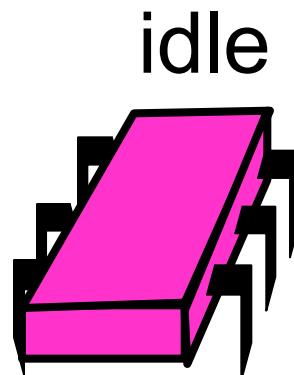


Pointer to  
predecessor (or  
null)

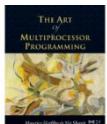
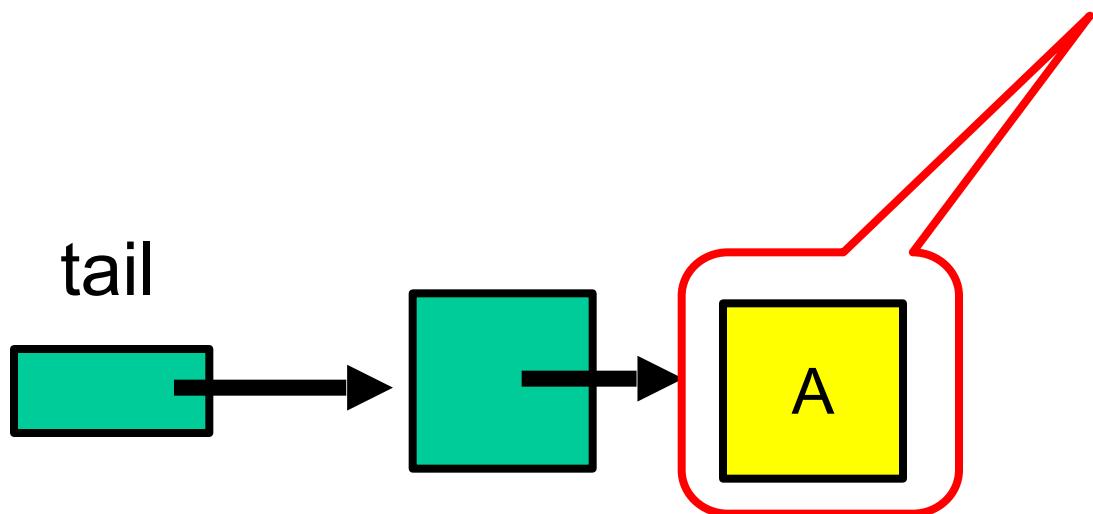
tail



# Initially

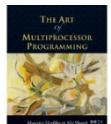
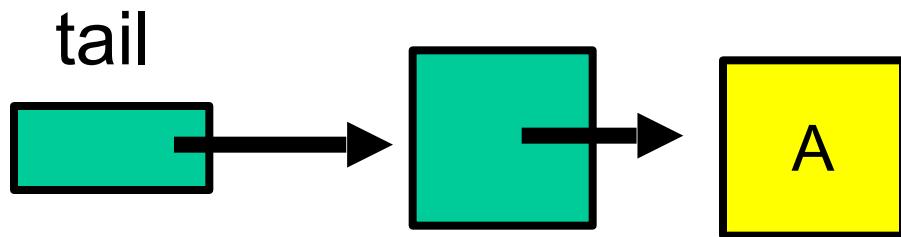
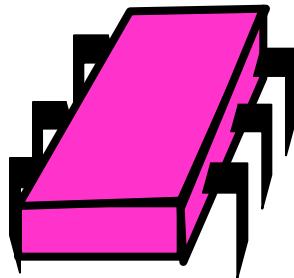


Distinguished  
available node  
means lock is  
free



# Acquiring

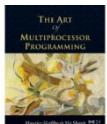
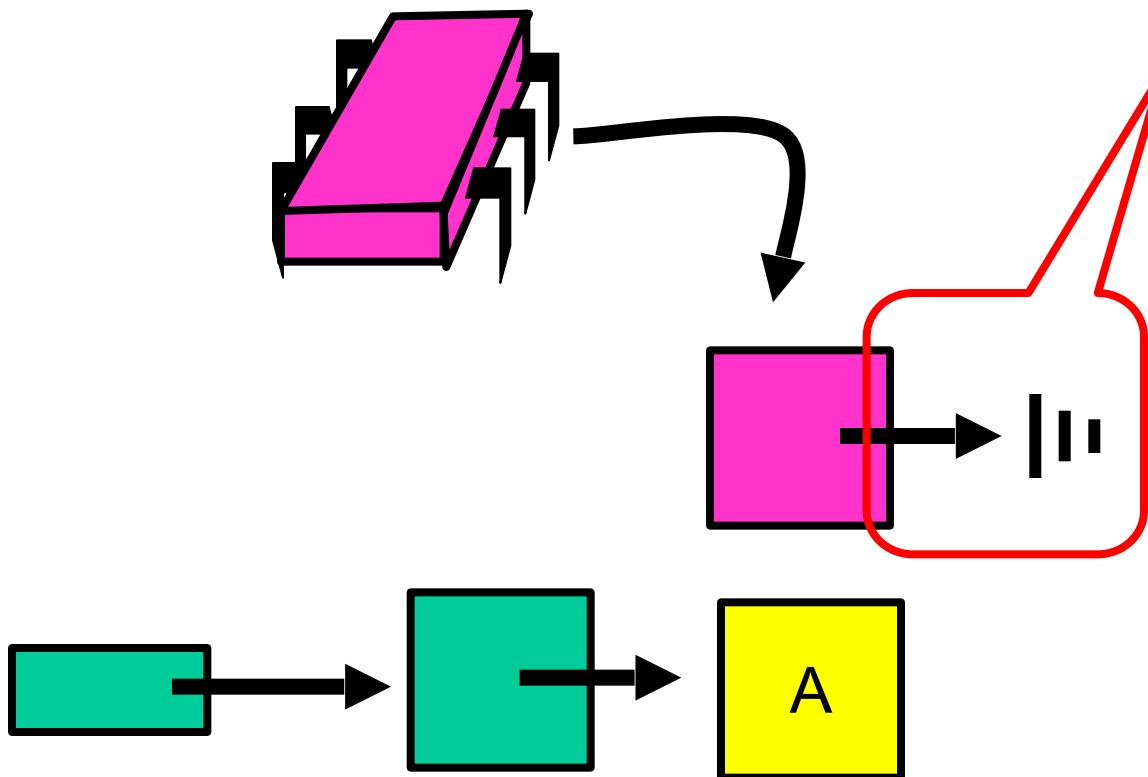
## acquiring



# Acquiring

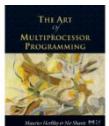
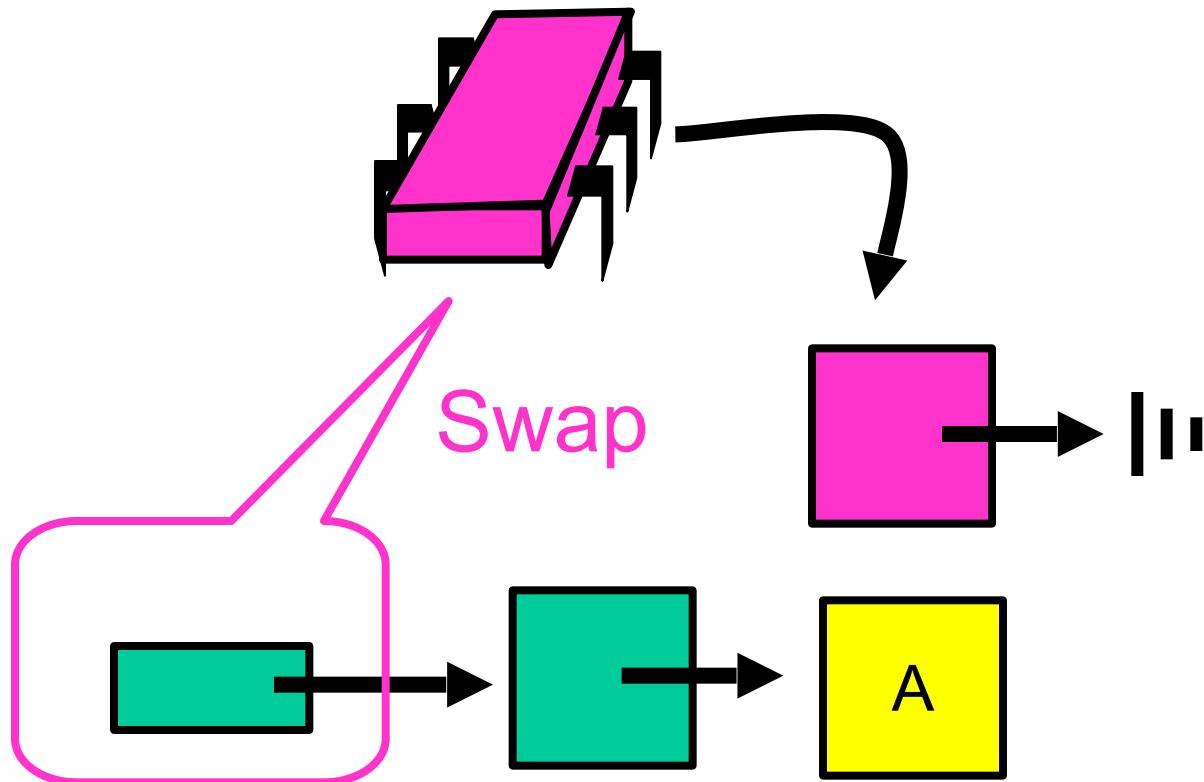
acquiring

Null predecessor  
means lock not  
released or aborted



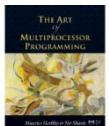
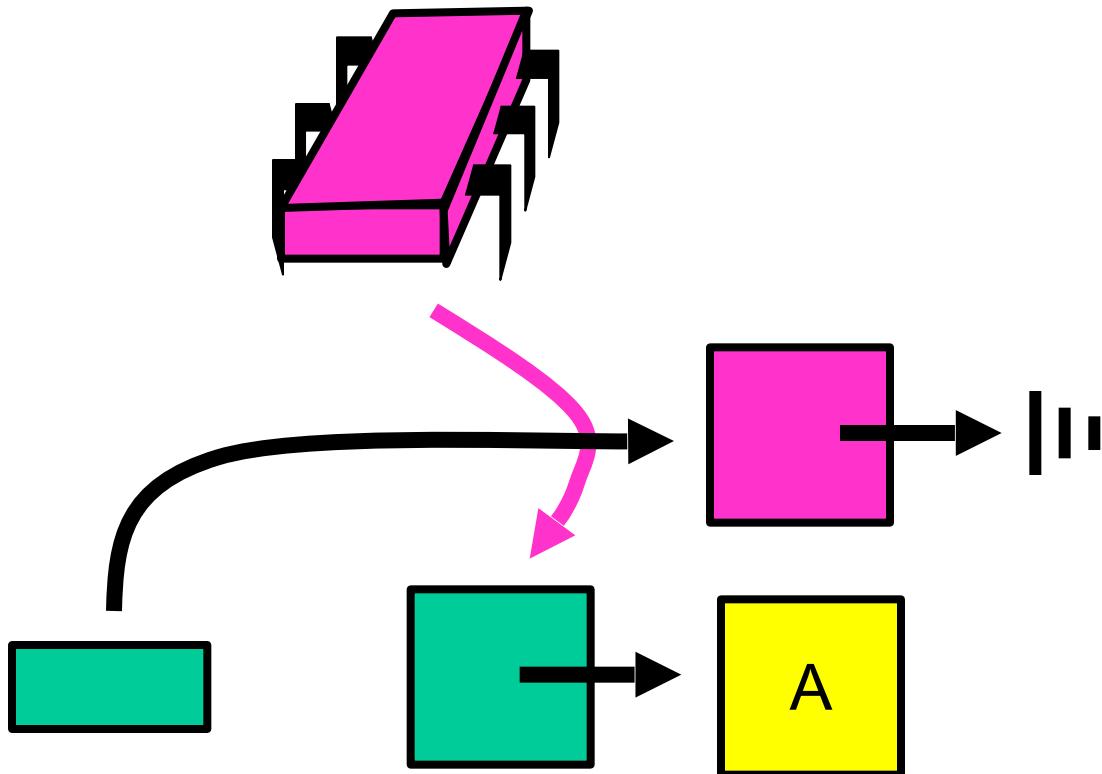
# Acquiring

## acquiring



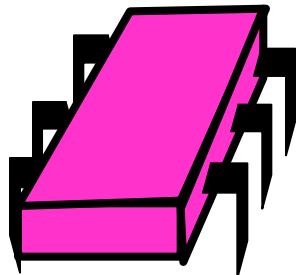
# Acquiring

## acquiring

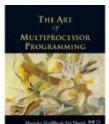
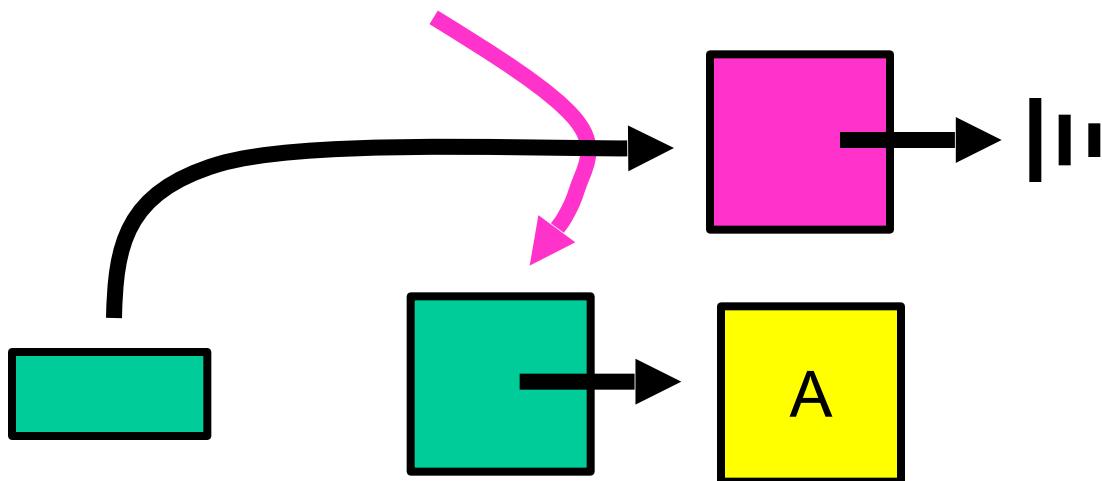


# Acquired

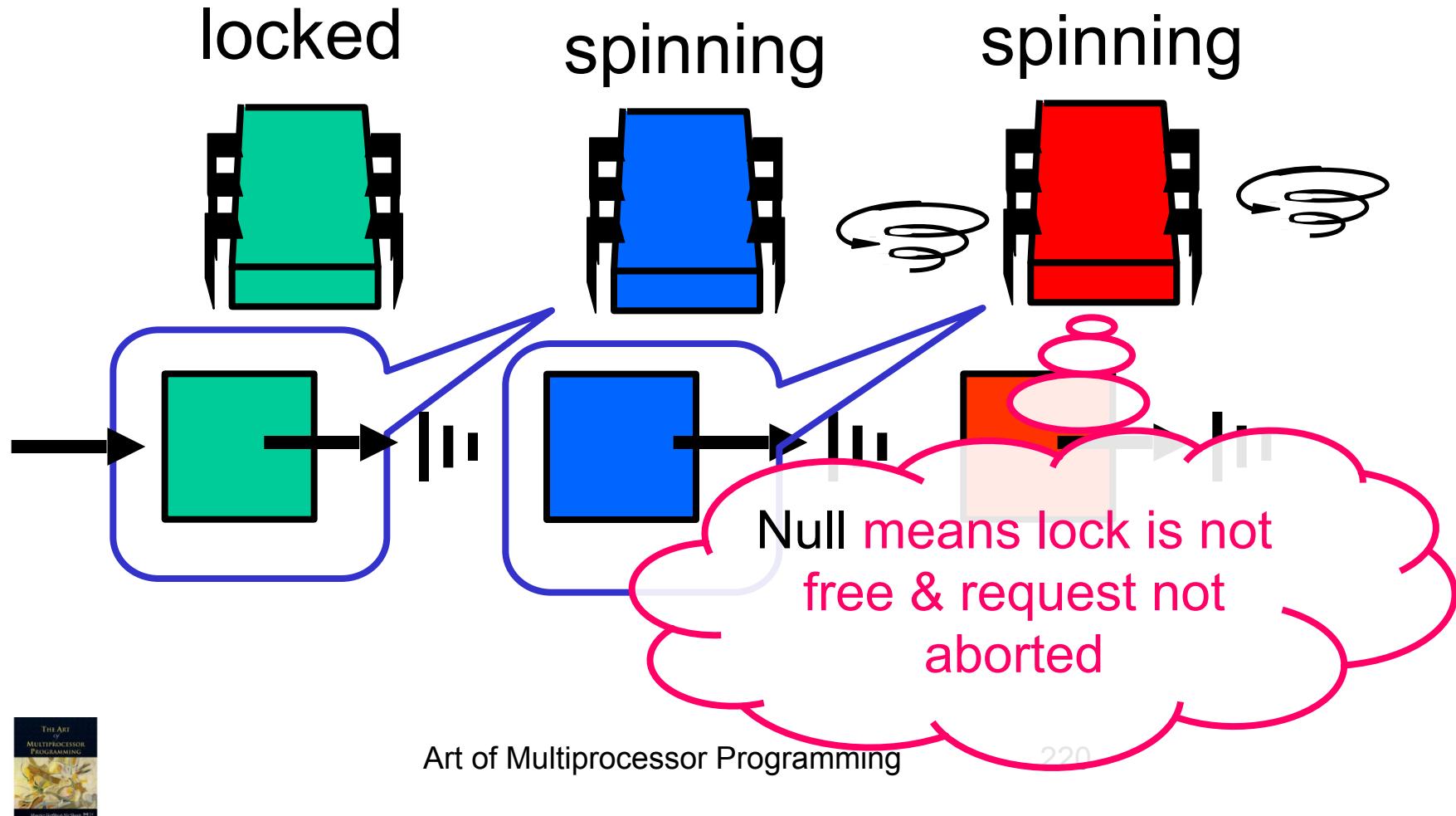
locked



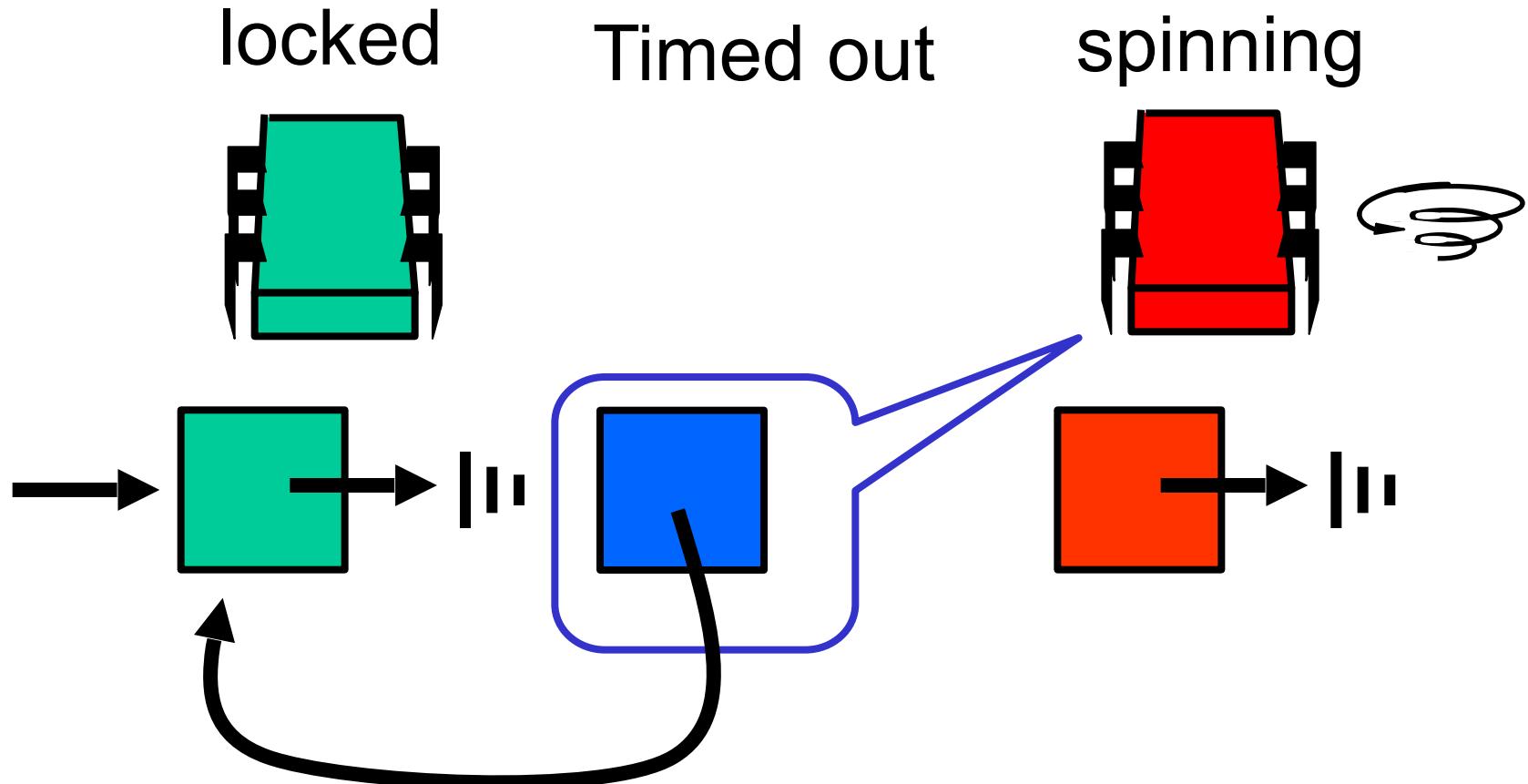
Reference to  
AVAILABLE means  
lock is free.



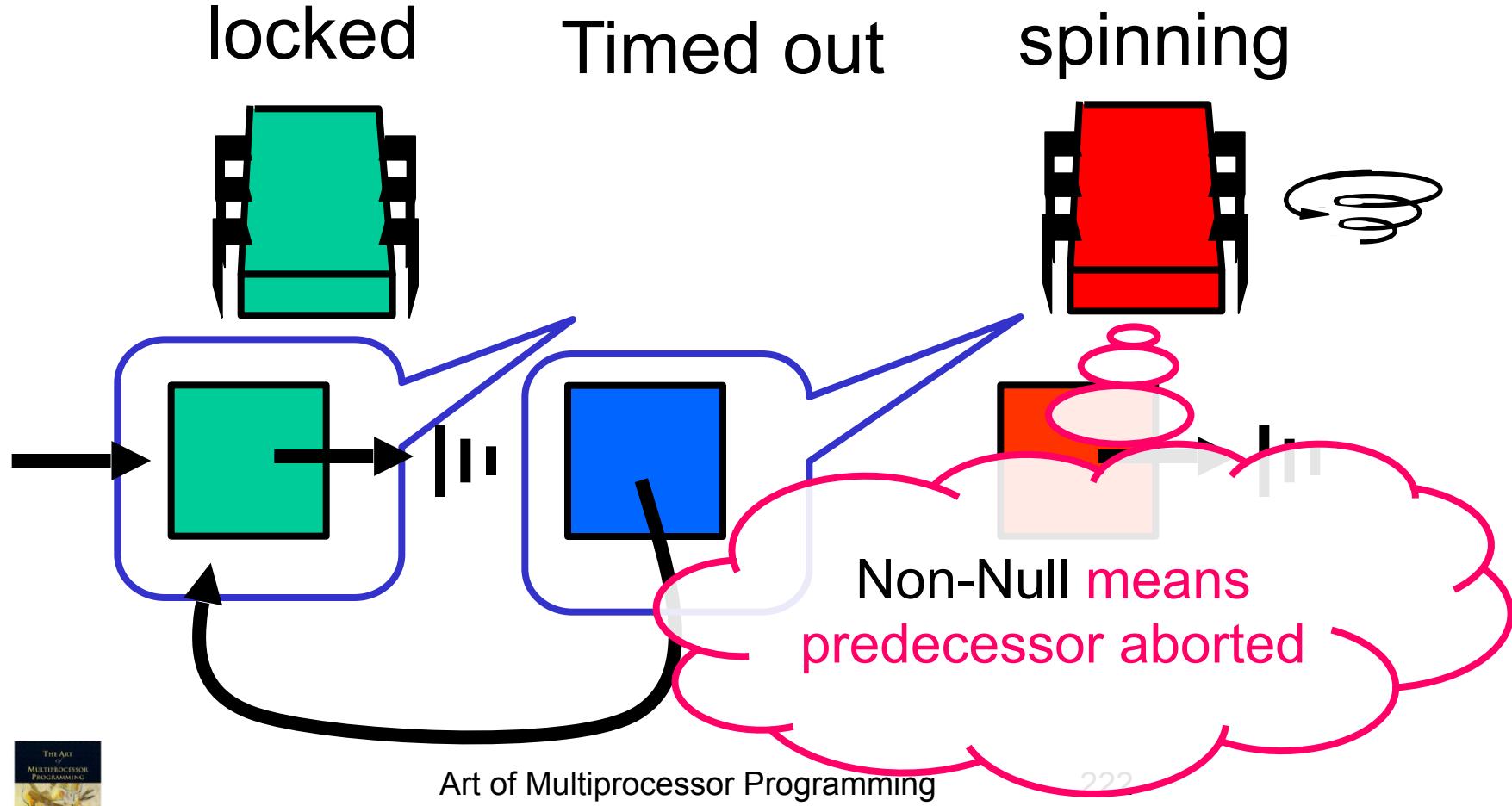
# Normal Case



# One Thread Aborts

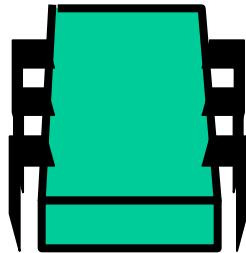


# Successor Notices

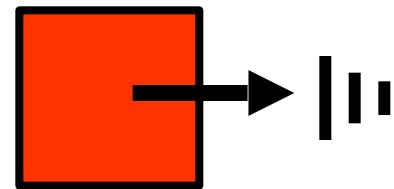
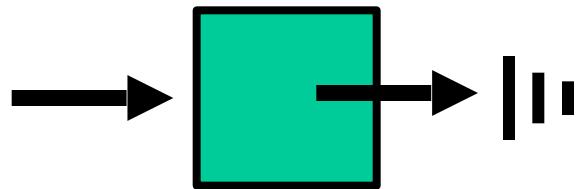
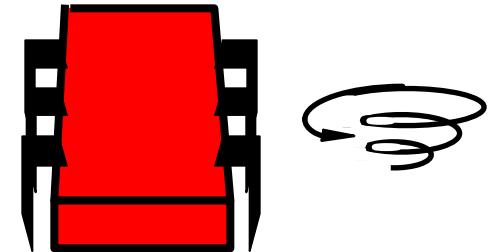


# Recycle Predecessor's Node

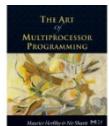
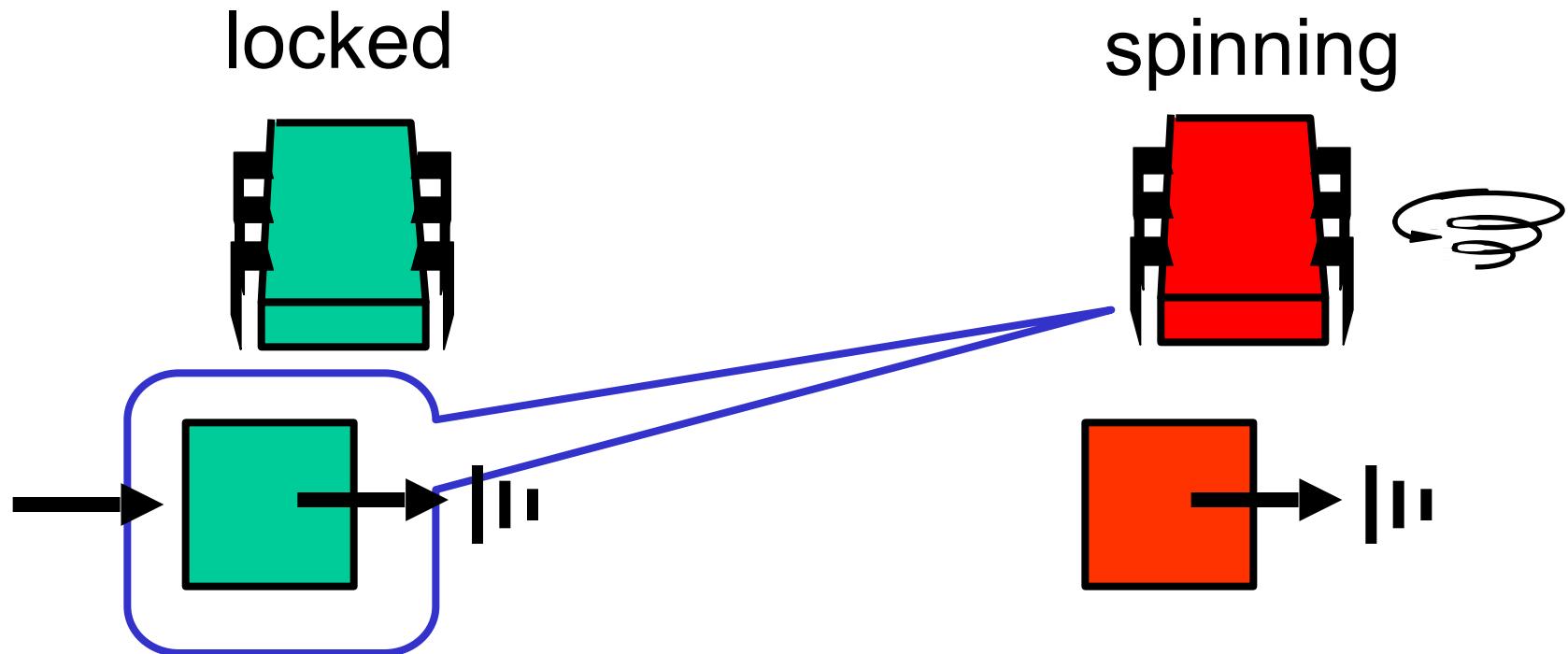
locked



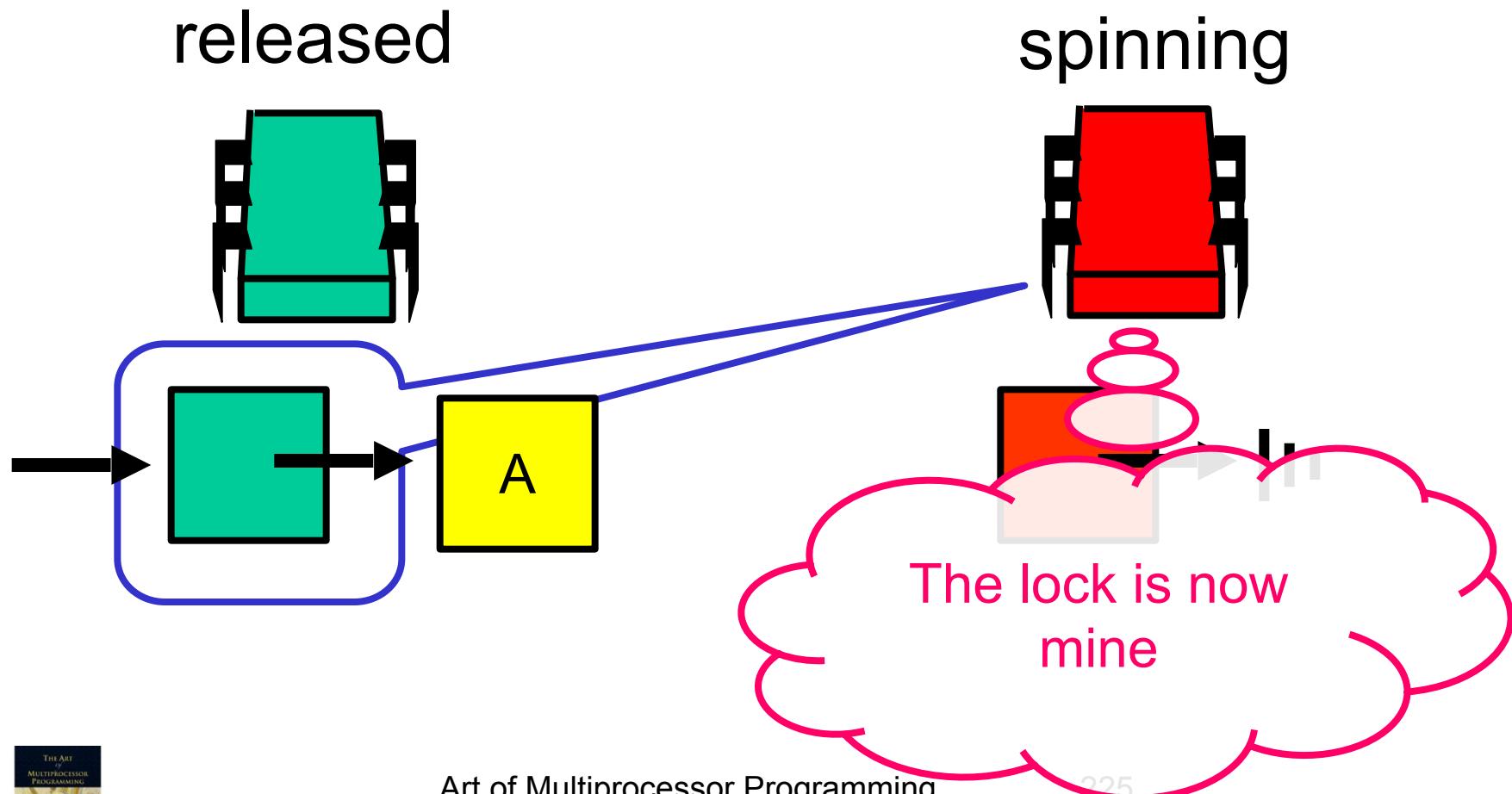
spinning



# Spin on Earlier Node

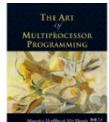


# Spin on Earlier Node



# Time-out Lock

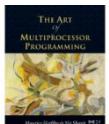
```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```



# Time-out Lock

```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

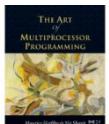
AVAILABLE node signifies  
free lock



# Time-out Lock

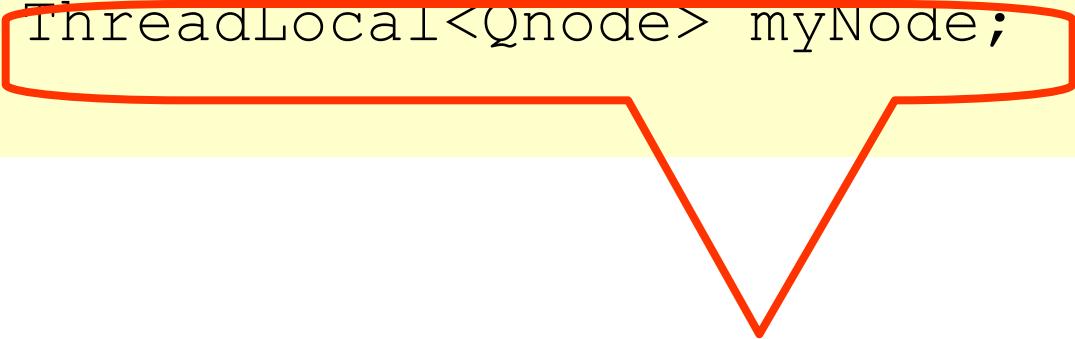
```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

Tail of the queue

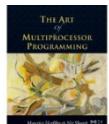


# Time-out Lock

```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

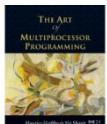


Remember my node ...



# Time-out Lock

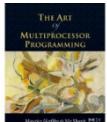
```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred = tail.getAndSet(qnode);  
    if (myPred== null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
    ...
```



# Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

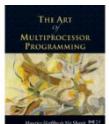
Create & initialize node



# Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

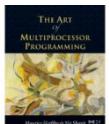
Swap with tail



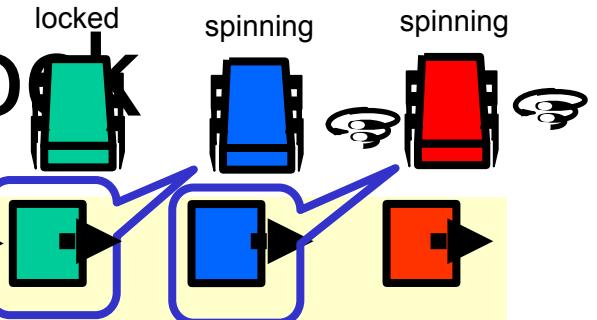
# Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
    ...
```

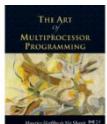
If predecessor absent or released, we are done



# Time-out Lock

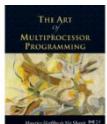


```
...
long start = now();
while (now() - start < timeout) {
    Qnode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }
}
...
...
```



# Time-out Lock

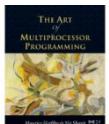
```
...  
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...  
Keep trying for a while ...
```



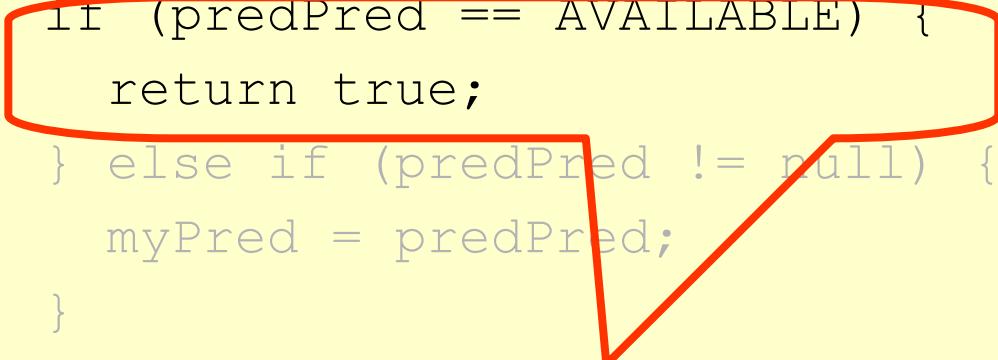
# Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...  
...
```

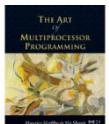
Spin on predecessor's prev field



# Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...  

```

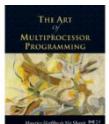
Predecessor released lock



# Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...  
}
```

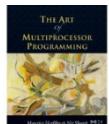
Predecessor aborted,  
advance one



# Time-out Lock

```
...
if (!tail.compareAndSet(qnode, myPred))
    qnode.prev = myPred;
return false;
}
}
```

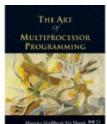
What do I do when I time out?



# Time-out Lock

```
...
if (!tail.compareAndSet(qnode, myPred))
    qnode.prev = myPred;
    return false;
}
}
```

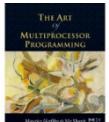
Do I have a successor?  
If CAS fails, I do.  
Tell it about myPred



# Time-out Lock

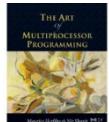
```
...
if (!tail.compareAndSet(qnode, myPred) )
    qnode.prev = myPred;
    return false;
}
}
```

If CAS succeeds: no successor,  
simply return false



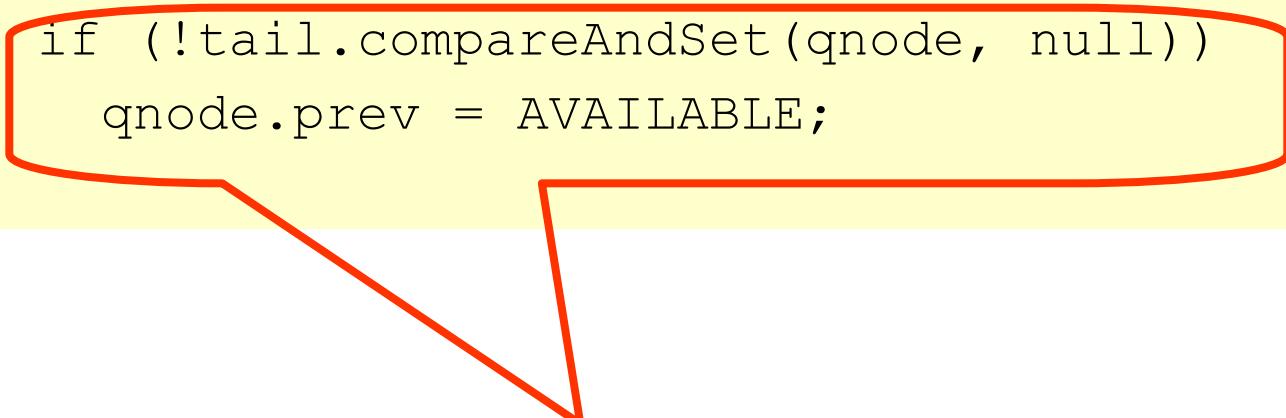
# Time-Out Unlock

```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode, null))  
        qnode.prev = AVAILABLE;  
}
```

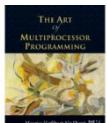


# Time-out Unlock

```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode, null))  
        qnode.prev = AVAILABLE;  
}
```



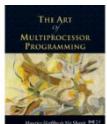
If CAS failed:  
successor exists,  
notify it can enter



# Timing-out Lock

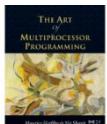
```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode, null))  
        qnode.prev = AVAILABLE;  
}
```

CAS successful: set tail to null, no clean up since no successor waiting



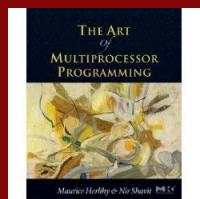
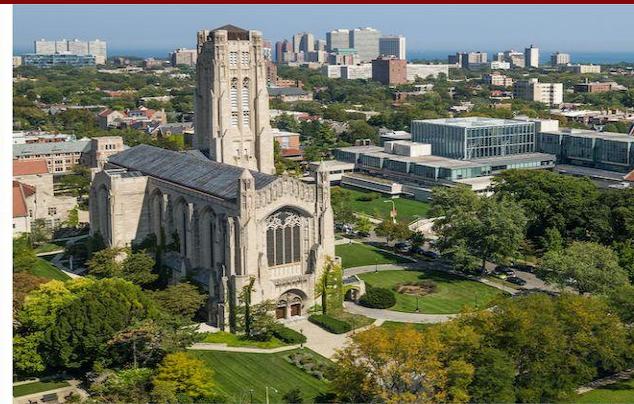
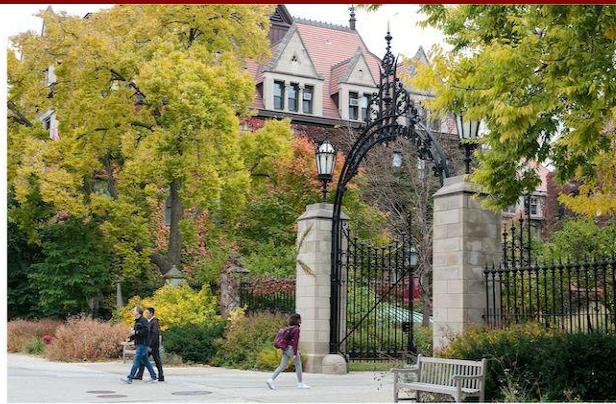
# One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS, ToLock...
- Each better than others in some way
- There is no one solution
- Lock we pick really depends on:
  - the application
  - the hardware
  - which properties are important



# MPCS 52060 - Parallel Programming

## M4: Concurrent Objects (Part 1)



Original slides from “The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit” With modifications by Lamont Samuels

# More Low-Level Synchronization Primitives

# Motivation for Semaphores

- What if we wanted to control access to shared resource?
  - For example, A system can only handle a certain number of users concurrently signed on. After the maximum number of logged in users is reached, then others must wait until others logout.
- A **semaphore** is a synchronization primitive used to control access to a shared resource by multiple threads.
  - It has a capacity ( $c$ ), which allows for having at most ( $c$ ) threads in a critical section. Unlike with locks, where only one thread can be a critical section at a time.
  - You can also think of them as a way to control how many resources are available of a particular entity by allowing resources to be acquired and released safely concurrently.

# Semaphore Pseudo-Implementation

- The capacity variable of a semaphore is an integer value that cannot be directly accessed.
  - Go does not have a semaphore construct so the below examples are pseudocode similar to the implementations of Semaphores in other languages:
- **Creation:** Must initialize it to some capacity integer value

```
var sema *Semaphore  
// NewSemaphore allocates and  
// returns a *Semaphore with its  
// internal capacity initialized  
sema = NewSemaphore(0)
```

- **Behaviors:** It has two main operations (methods in our case) that modify the integer capacity value

```
//Decrement semaphore  
sema.Down()  
// Increment semaphore  
sema.Up()
```

# Semaphore Pseudo-Implementation

```
func (s *Semaphore) Down() {  
    //Wait until value of semaphore s is greater than 0  
    //Decrement the value of semaphore s by one  
}  
  
func (s *Semaphore) Up() {  
    //Increment the value of semaphore s by 1  
    //If there are 1 or more threads waiting, wake one up  
}
```

# Semaphores & Mutual Exclusion

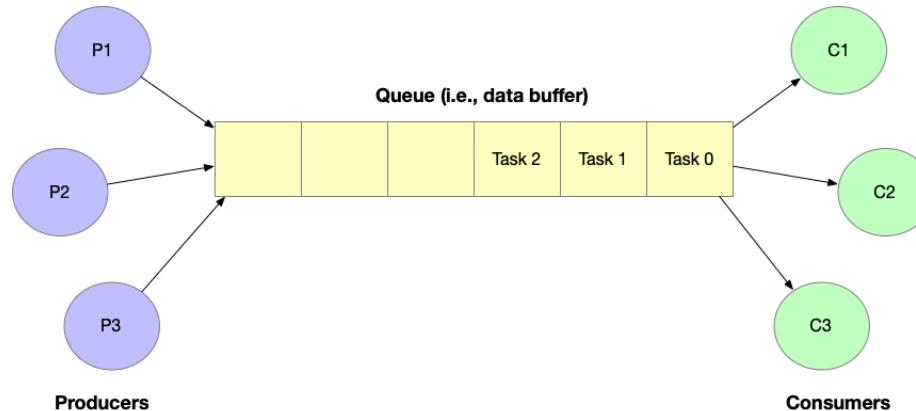
- There are two well known types of semaphores
  - **Binary semaphore** - acts like a mutex by setting its capacity initial value to 1.

```
var mutex_sema *Semaphore  
mutex_sema = NewSemaphore(1)  
mutex_sema.Down()  
//critical section  
mutex_sema.Up()
```

- **Counting semaphore** - initialize the semaphore to be equal to the number of available resources.

# Real-World Example: Producer and Consumer Problem

- Also called the bounded-buffer problem.
- One or more threads generate tasks (**producers**) and one or more threads receive and process them (**consumers**).
- Producers and consumers communicate using a queue of maximum size N and must adhere to the following conditions:
  - Consumers must wait for a producer to produce a task if the queue is empty.
  - Producer must wait for the consumer to consume a task if the queue is full.



# Real-World Example: Producer and Consumer Problem

```
//Producer
```

```
for {
```

```
    //Generate Task
```

```
    sema_mutex.Down()
```

```
    sema_emptyCount.Down()
```

```
    //Put task in Queue
```

```
    sema_fullCount.Up()
```

```
    sema_mutex.Up()
```

```
}
```

```
//Consumer
```

```
for {
```

```
    sema_mutex.Down()
```

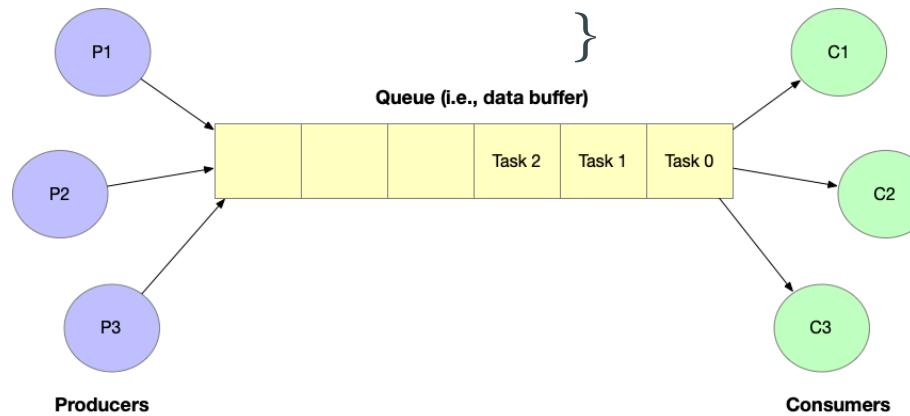
```
    sema_fullCount.Down()
```

```
    //Remove task from Queue
```

```
    sema_emptyCount.Up()
```

```
    sema_mutex.Up()
```

```
    //Process Task
```



# Condition Variables

- A data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to “wake up.”
- A condition variable is always associated with a mutex.

```
// lock mutex
if (condition has occurred) {
    // signal thread(s);
} else {
    // 1. Wait until another thread signals to wake up by unlocking
    // the mutex and block (e.g., sleep, or spin etc.);
    // 2. After the signal happens then the thread wakes, requires the lock
    // and checks to make sure the condition is still true.
}
// unlock mutex
```

# Condition Variables in Go

- `sync.Cond` represents conditional variables in Go:
- Creation: `NewCond(I Locker) *Cond`
- Operations on condition variables:
  - `func (c *Cond) Wait()`: suspends the calling thread and releases the monitor lock. When it resumes, reacquire the lock. Called when condition is not true •
  - `func (c *Cond) Signal()`: resumes one thread waiting in wait() if any. Called when condition becomes true and wants to wake up one waiting thread.
  - `func (c *Cond) Broadcast()`: resumes all threads waiting in wait(). Called when condition becomes true and wants to wake up all waiting threads.

# Demo: Condition Variables

# Concurrent Data Structures

# Concurrent Data Structures

- We assume
  - shared-memory multiprocessors environment
  - concurrently execute multiple threads which communicate and synchronize through data structures in shared memory

# Concurrent Data Structures

- Far more difficult to design than sequential ones
  - Correctness
    - Primary source of difficulty is concurrency
    - The steps of different threads can be interleaved arbitrarily
  - Scalability (performance)
- We will look at
  - Concurrent Linked List/Queue/Stack

# Main performance issue of lock based system

- Sequential bottleneck
  - At any point in time, at most one lock-protected operation is doing useful work.
- Memory contention
  - Overhead in traffic as a result of multiple threads concurrently attempting to access the same memory location.
- Blocking
  - If thread that currently holds the lock is delayed, then all other threads attempting to access are also delayed.
  - Implementation of locks is known as a blocking algorithm
  - Consider non-blocking (lock-free) algorithm

# Nonblocking algorithms

- implemented by a hardware operation
  - atomically combines a load and a store
  - Ex) compare-and-swap(CAS)
- lock-free
  - if there is guaranteed system-wide progress;
  - while a given thread might be blocked by other threads, all CPUs can continue doing other useful work without stalls.
- wait-free
  - if there is also guaranteed per-thread progress.
  - in addition to all CPUs continuing to do useful work, no computation can ever be blocked by another computation.

# Linked List

- Illustrate these patterns ...
- Using a list-based Set
  - Common application
  - Building block for other apps

# Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - add(x) put x in set
  - remove(x) take x out of set
  - contains(x) tests if x in set

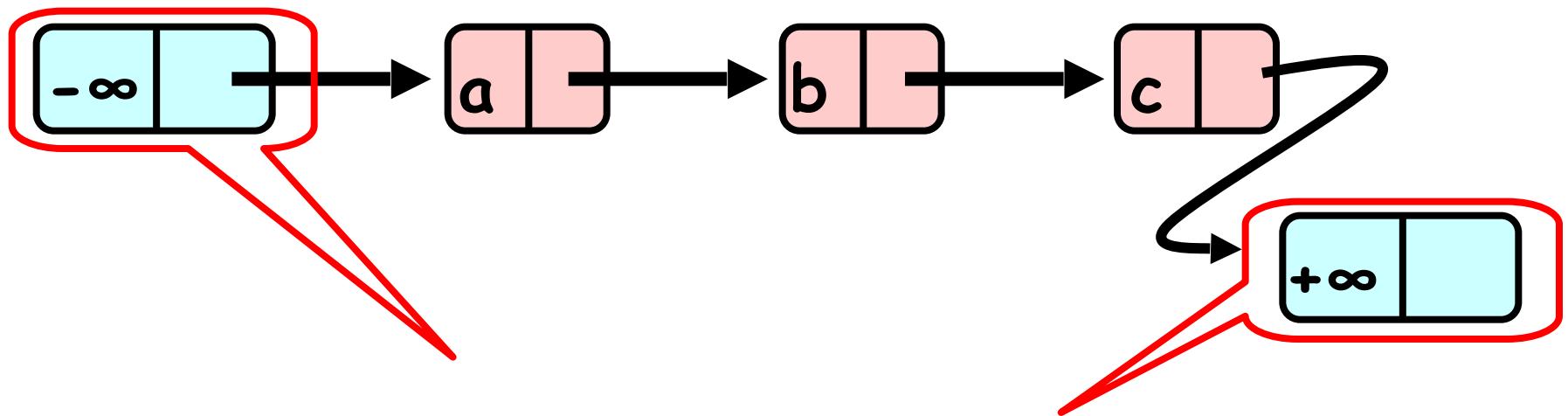
# List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
    public boolean remove(T x);  
    public boolean contains(T x);  
}
```

# List Node

```
public class Node {  
    public T item;      // item of interest  
    public int key;    // usually hash code  
    public Node next; // reference to next node  
}
```

# The List-Based Set



Sorted with Sentinel nodes  
(min & max possible keys)

# Sequential List Based Set

Add()

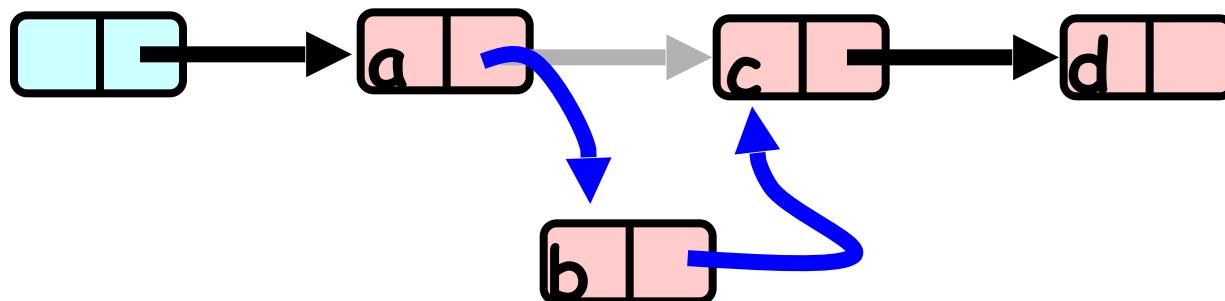


Remove()

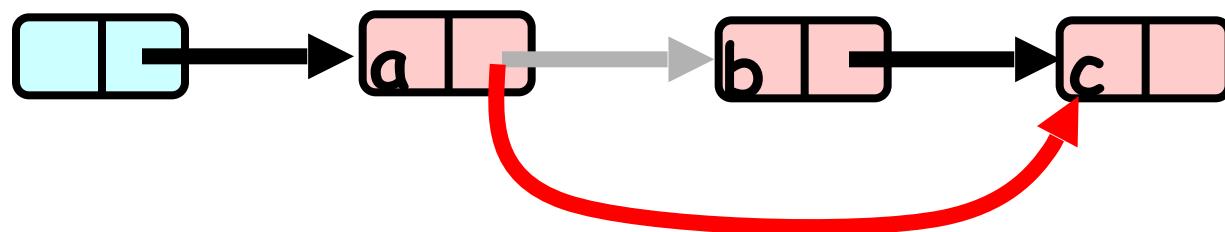


# Sequential List Based Set

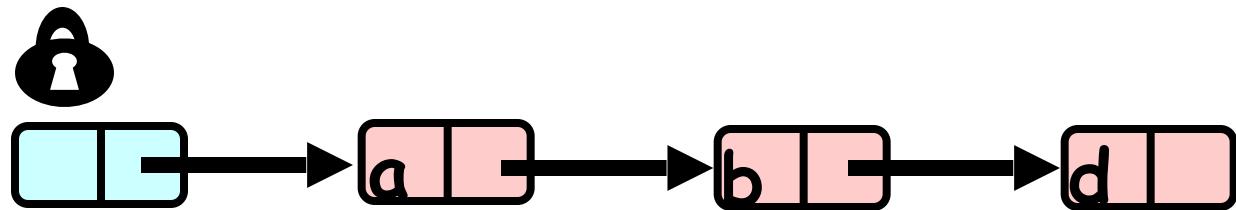
Add()



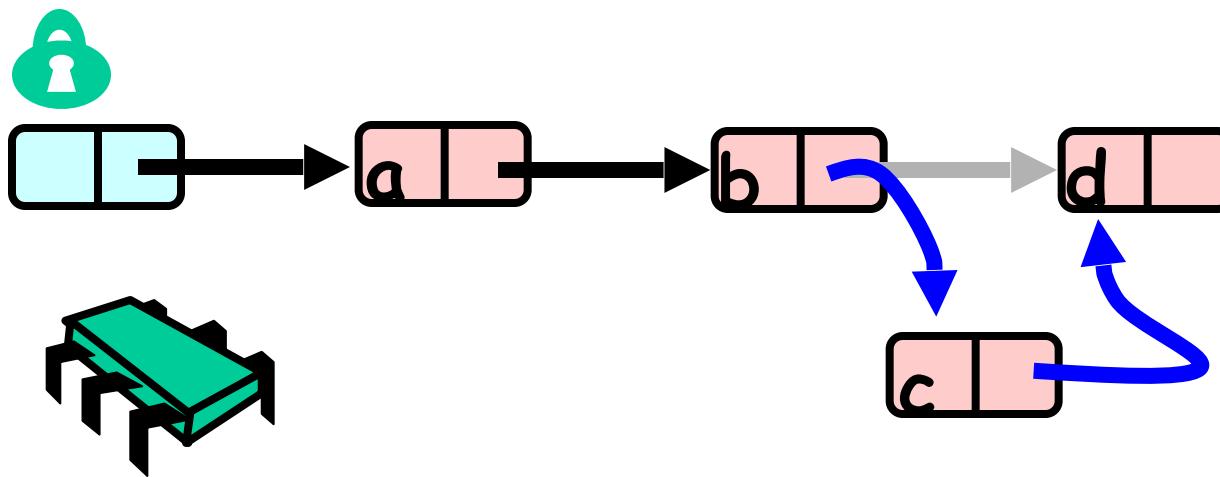
Remove()



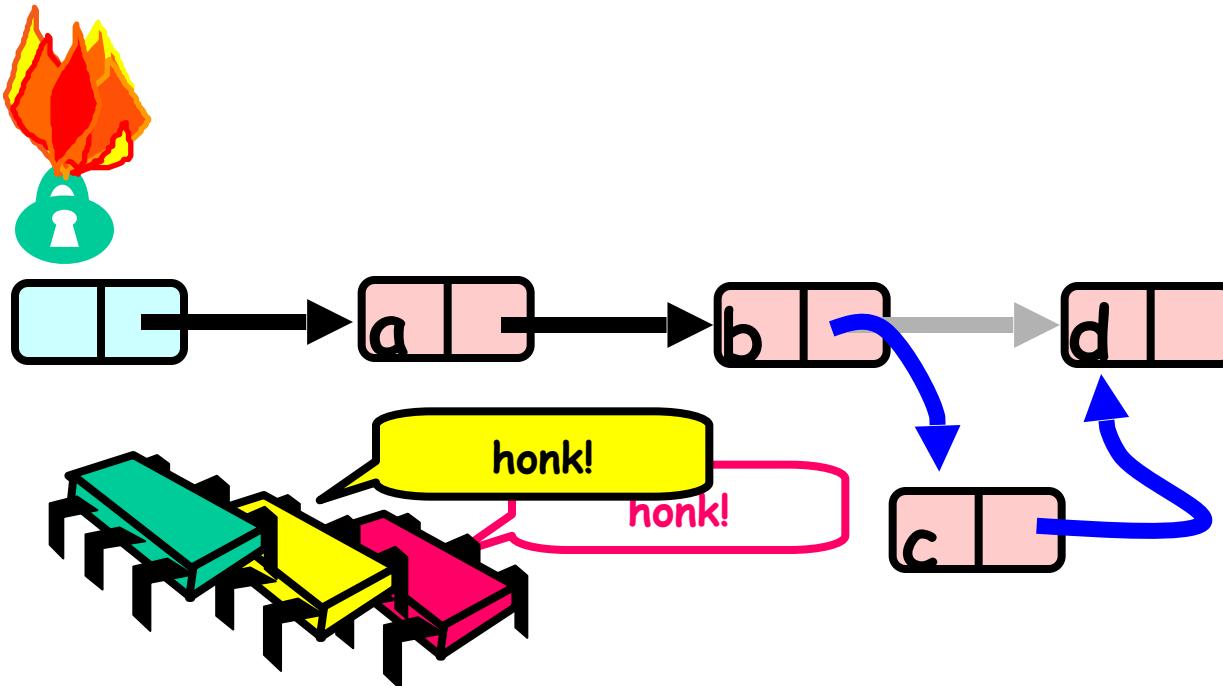
# Course Grained Locking



# Course Grained Locking



# Course Grained Locking



Simple but **hotspot + bottleneck**

# Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Well, some apps inherently parallel ...

# Coarse-Grained Synchronization (Linked List)

```
public class CoarseList<T> {  
    private Node head;  
    private Node tail;  
    private Lock lock = new ReentrantLock();  
  
    public CoarseList() {  
        // Add sentinels to start and end  
        head = new Node(Integer.MIN_VALUE);  
        tail = new Node(Integer.MAX_VALUE);  
        head.next = this.tail;  
    }  
}
```

```
public boolean add(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key) {  
            return false;  
        } else {  
            Node node = new Node(item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

```
public boolean remove(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = this.head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key)  
            pred.next = curr.next;  
        return true;  
    } else {  
        return false;  
    }  
} finally {  
    lock.unlock();  
}
```

```
public boolean contains(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        return (key == curr.key);  
    } finally {  
        lock.unlock();  
    }  
}
```

# Coarse-Grained Locking

- Easy, same as synchronized methods
  - "One lock to rule them all ..."
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention

# Performance Improvement

- For highly-concurrent objects
- Goal:
  - Concurrent access
  - More threads, more throughput

# First: Fine-Grained Synchronization

- Instead of using a single lock ..
- Split object into
  - Independently-synchronized components
- Methods conflict when they access
  - The same component ...
  - At the same time

## Second: Optimistic Synchronization

- Search without locking ...
- If you find it, lock and check ...
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking
  - Mistakes are expensive

## Third: Lazy Synchronization

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done

## Fourth: Lock-Free Synchronization

- Don't use locks at all
  - Use compareAndSet() & relatives ...
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

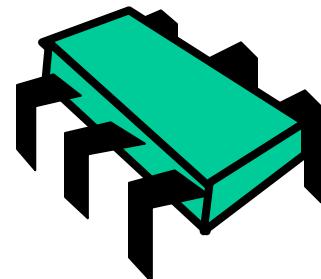
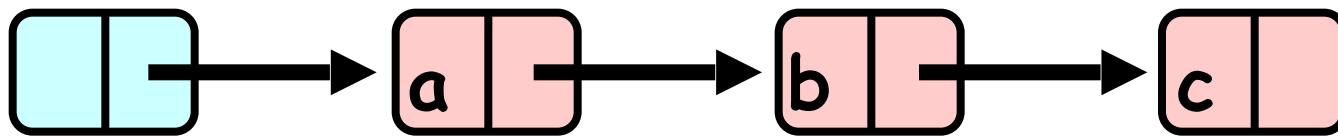
# Fine-grained Locking

- Requires **careful thought**
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

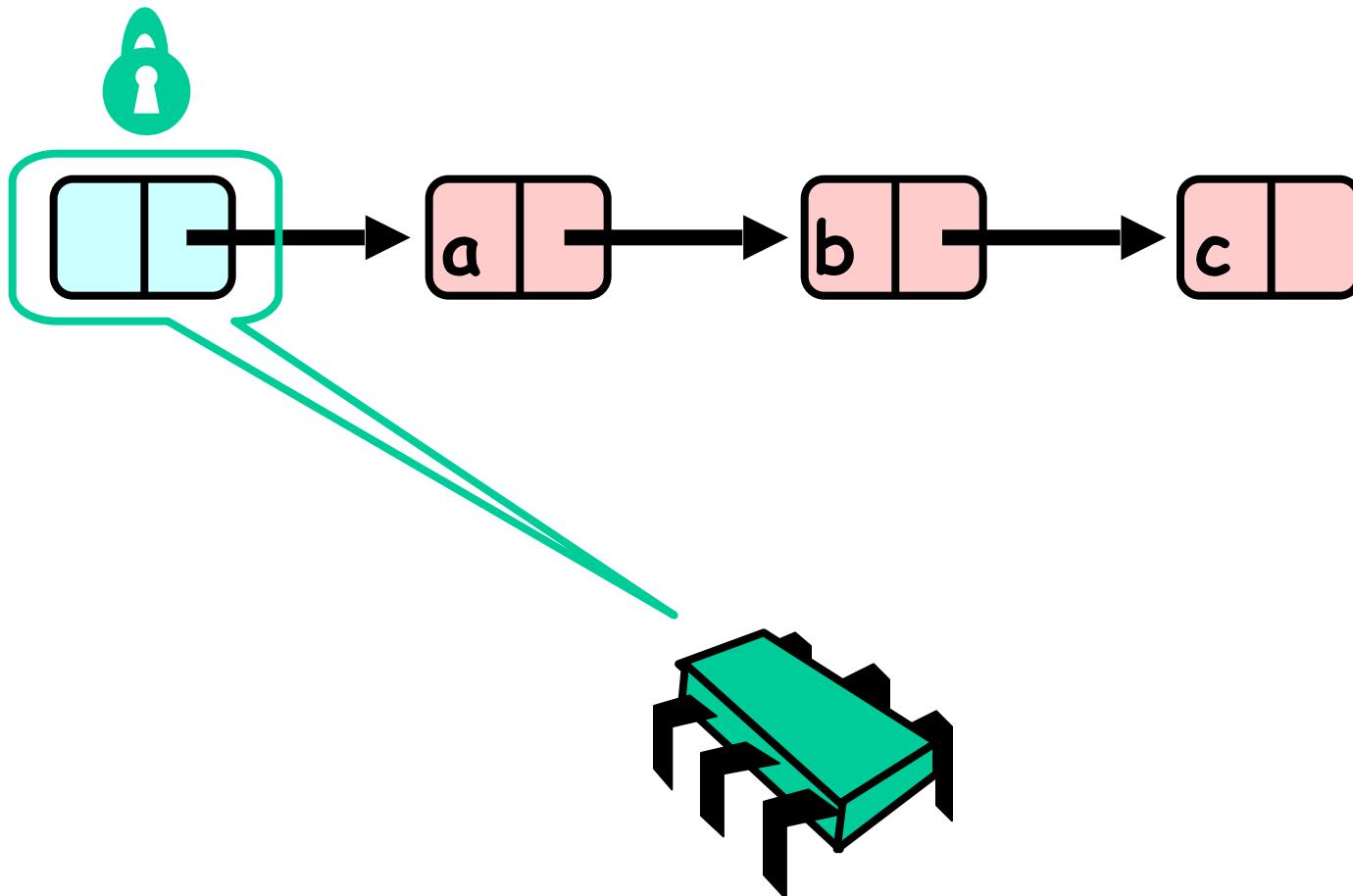
# Fine-grained Locking

- Use multiple locks of small granularity to protect different parts of the data structure
- Goal
  - To allow concurrent operations to proceed in parallel when they do not access the same parts of the data structure

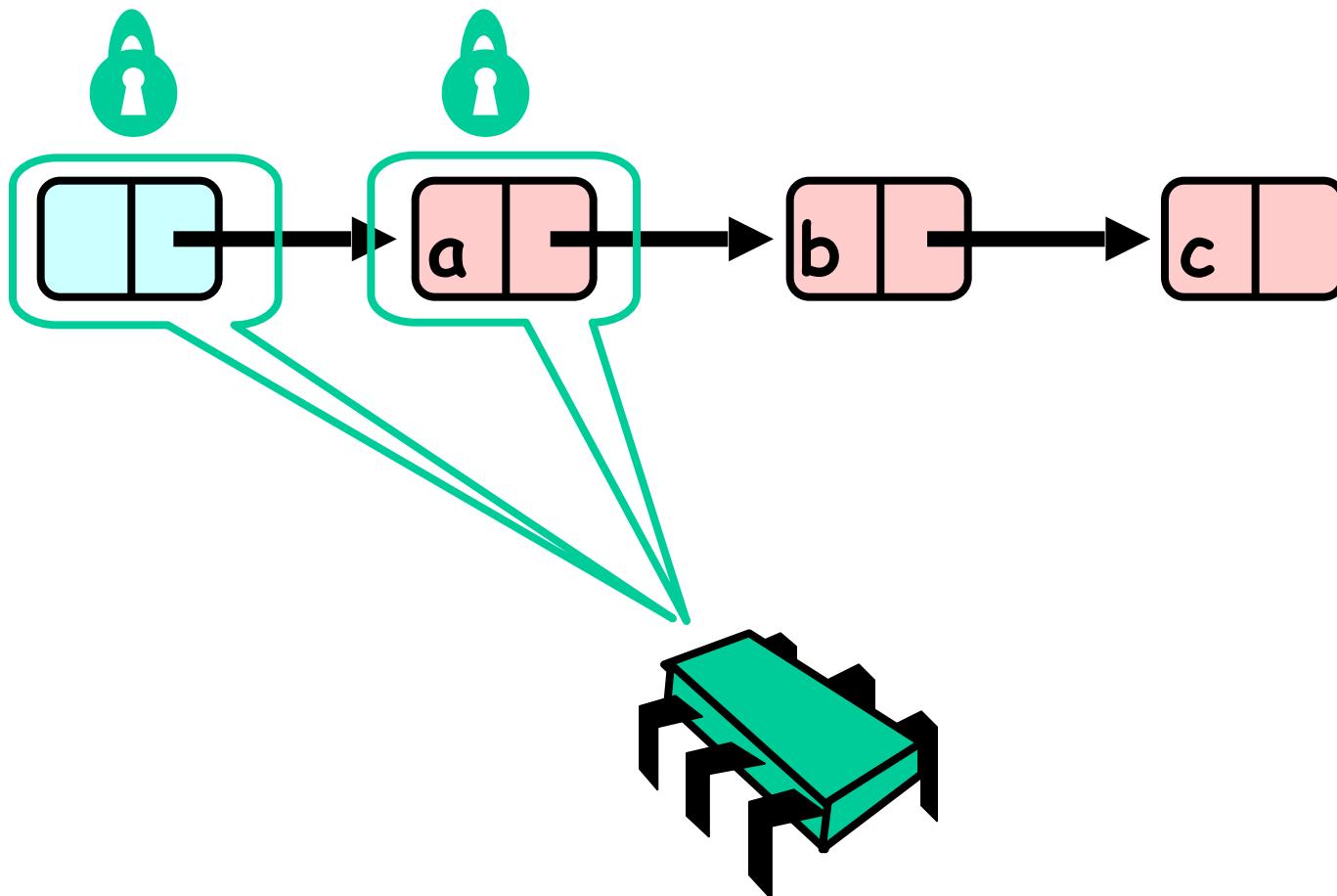
# Hand-over-Hand locking



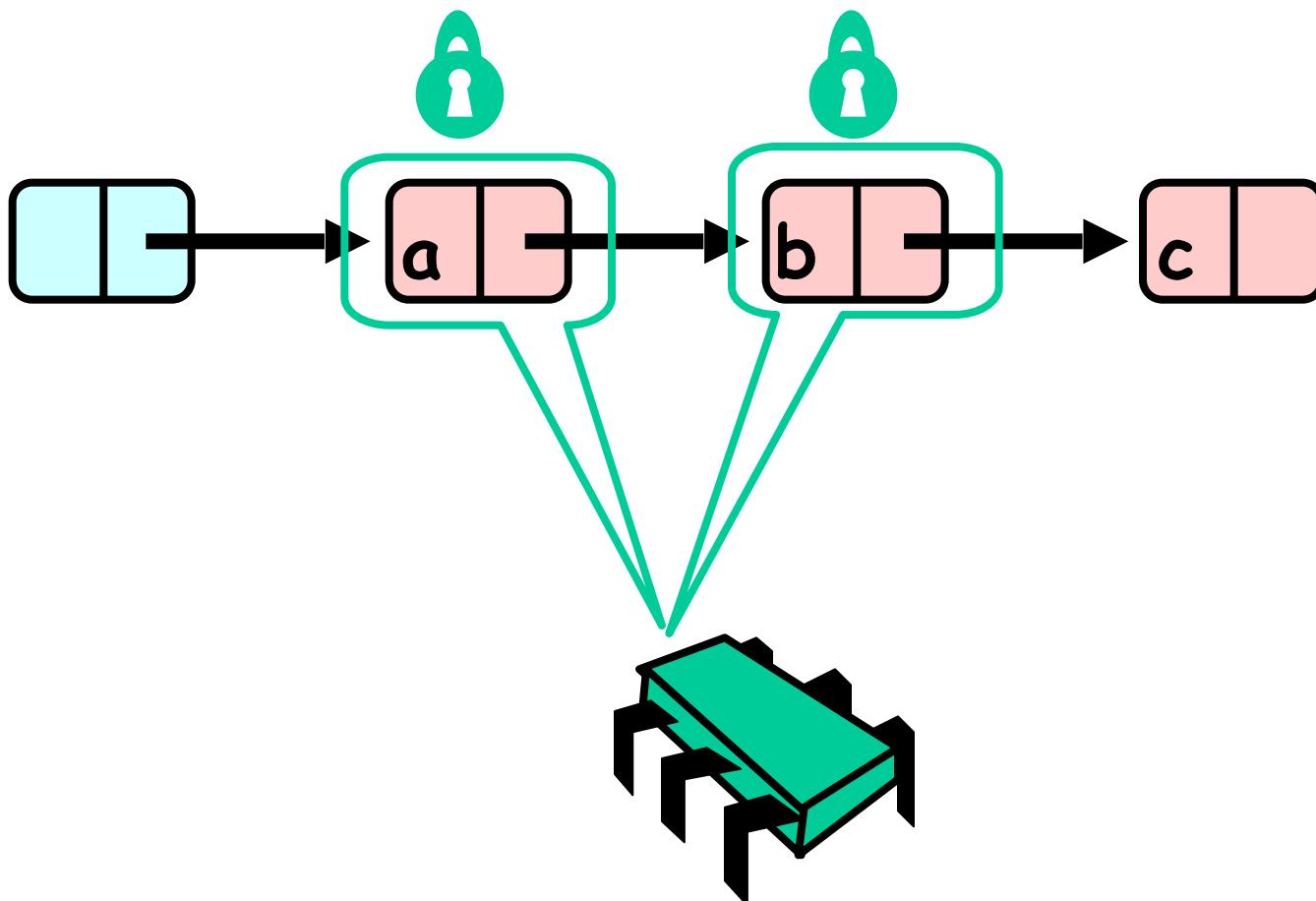
# Hand-over-Hand locking



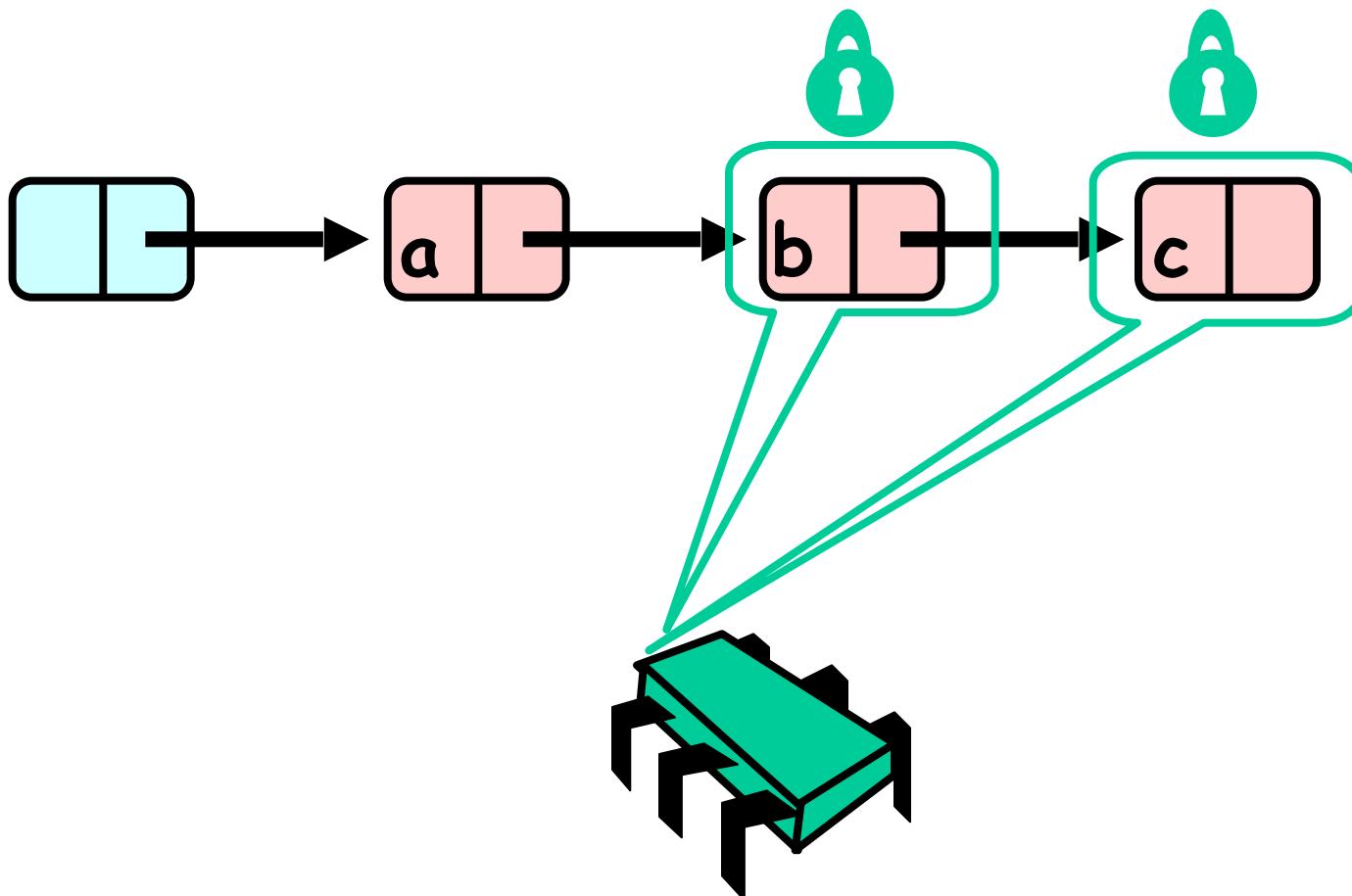
# Hand-over-Hand locking



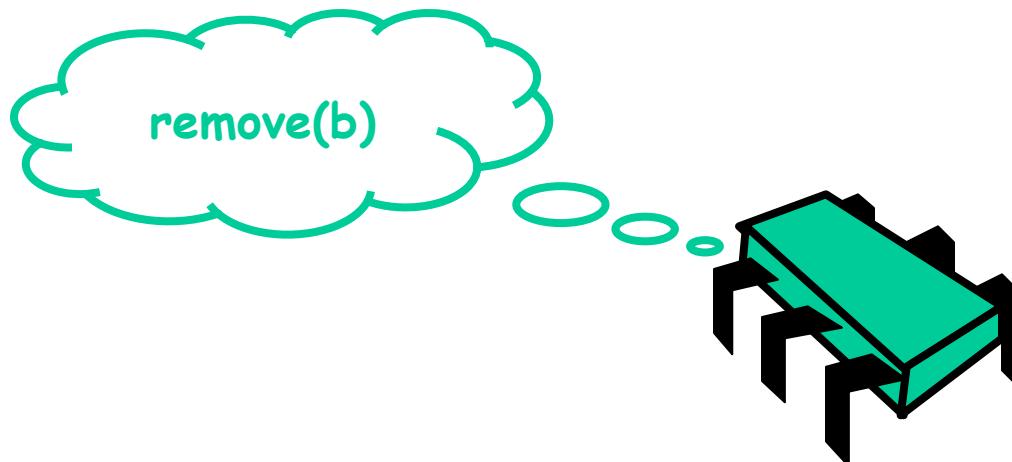
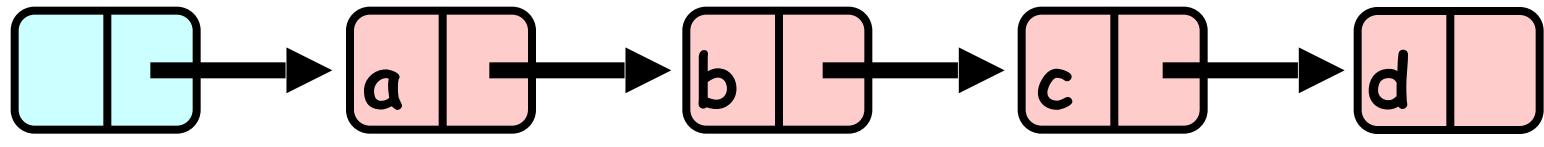
# Hand-over-Hand locking



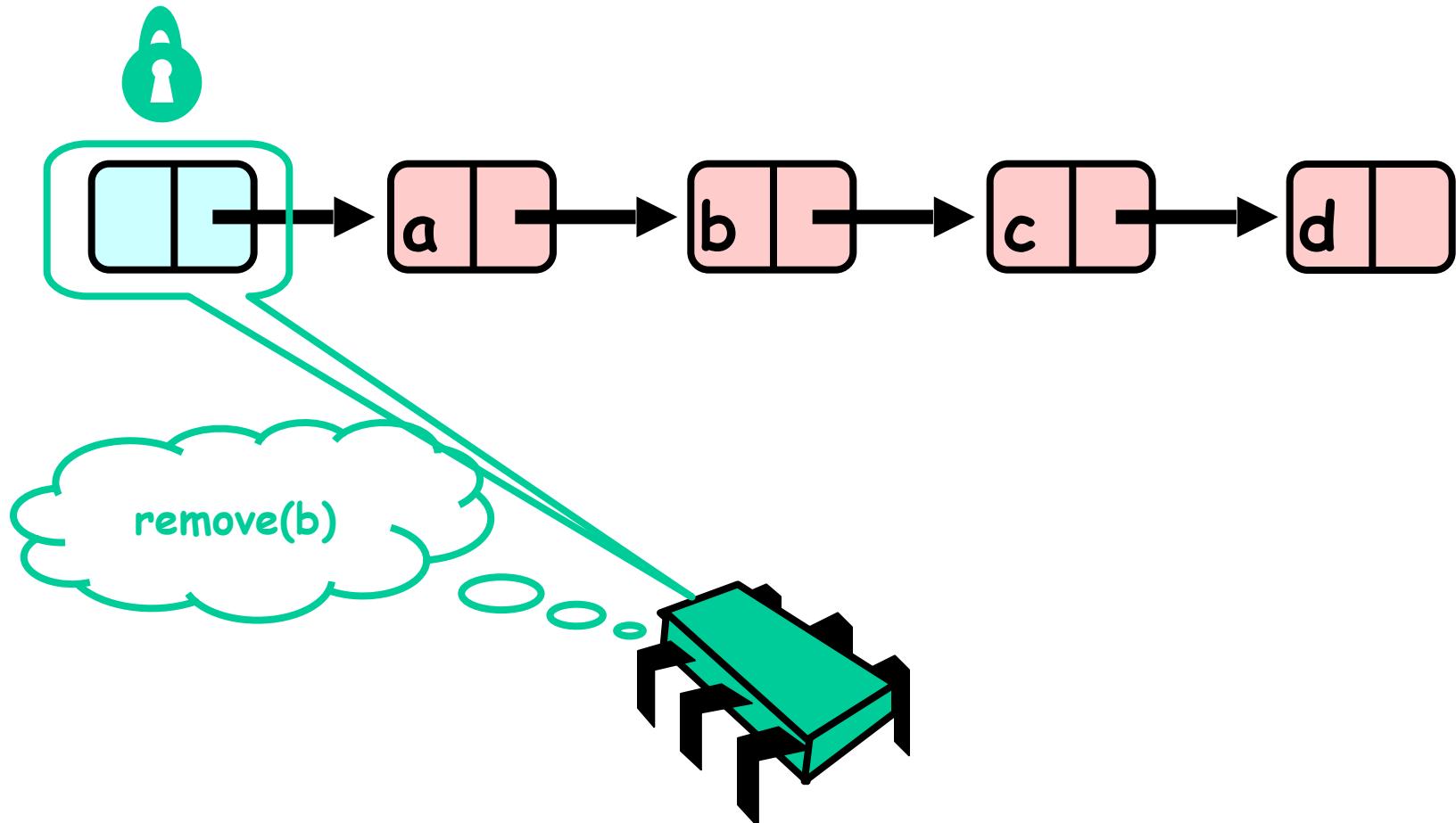
# Hand-over-Hand locking



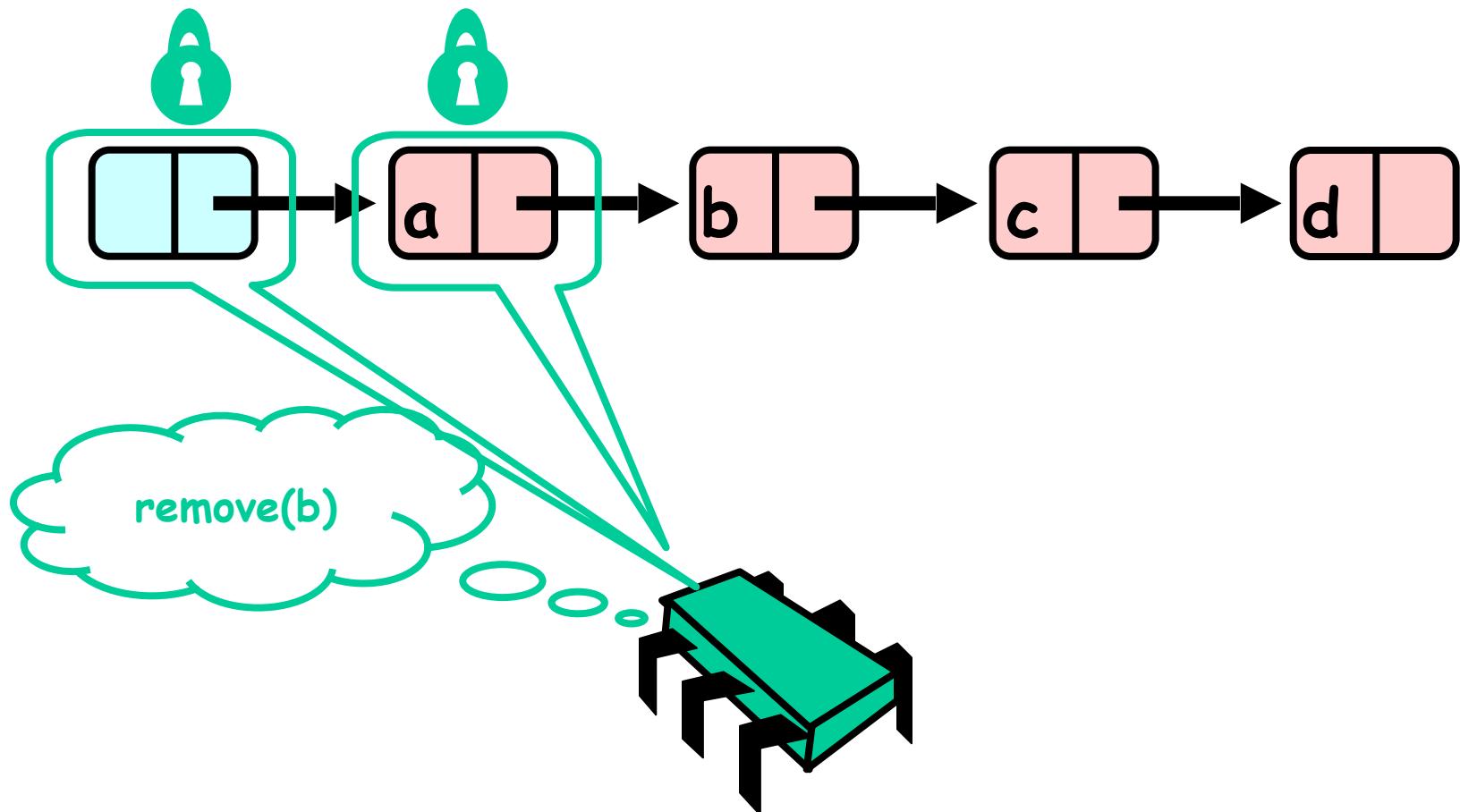
# Removing a Node



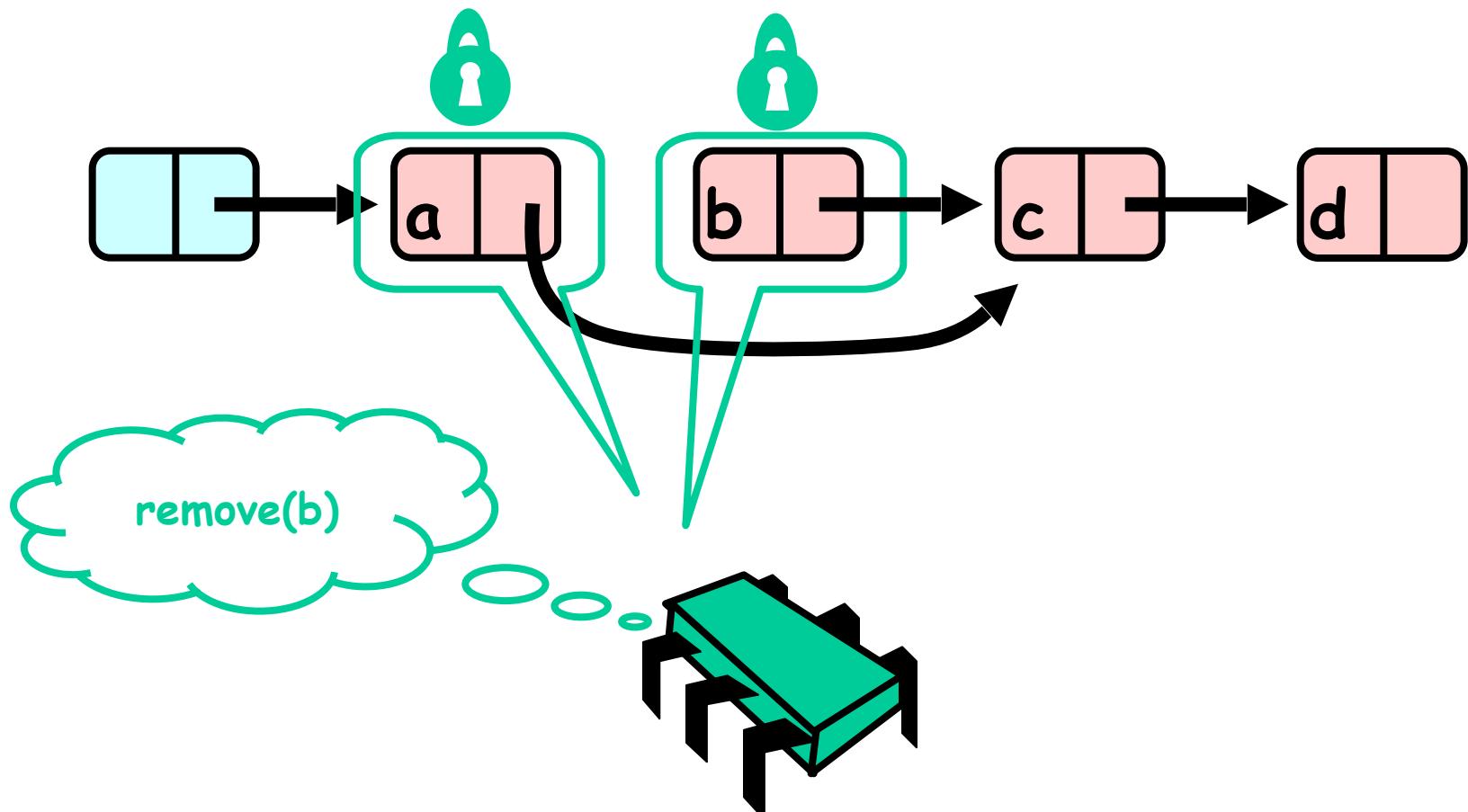
# Removing a Node



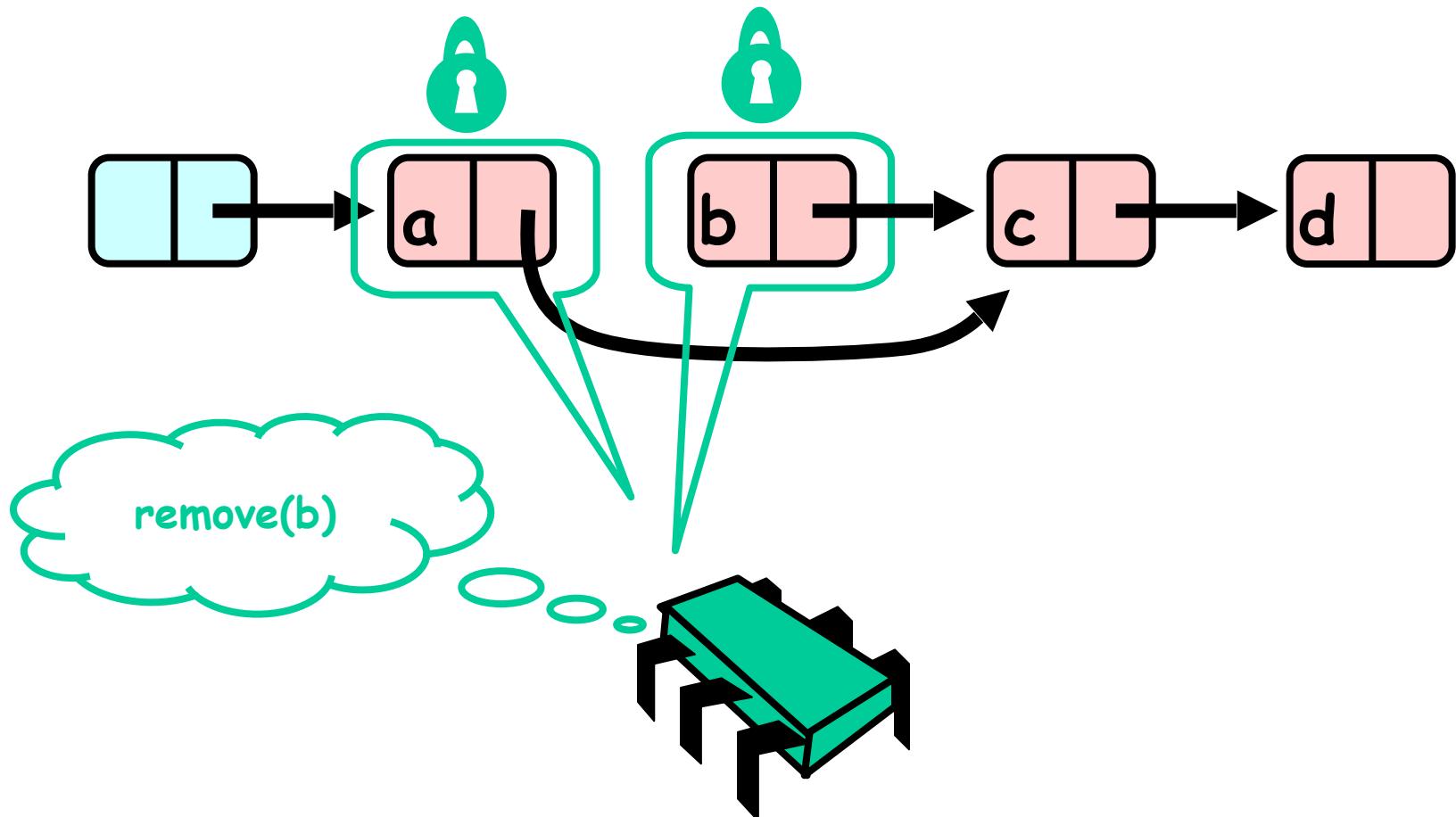
# Removing a Node



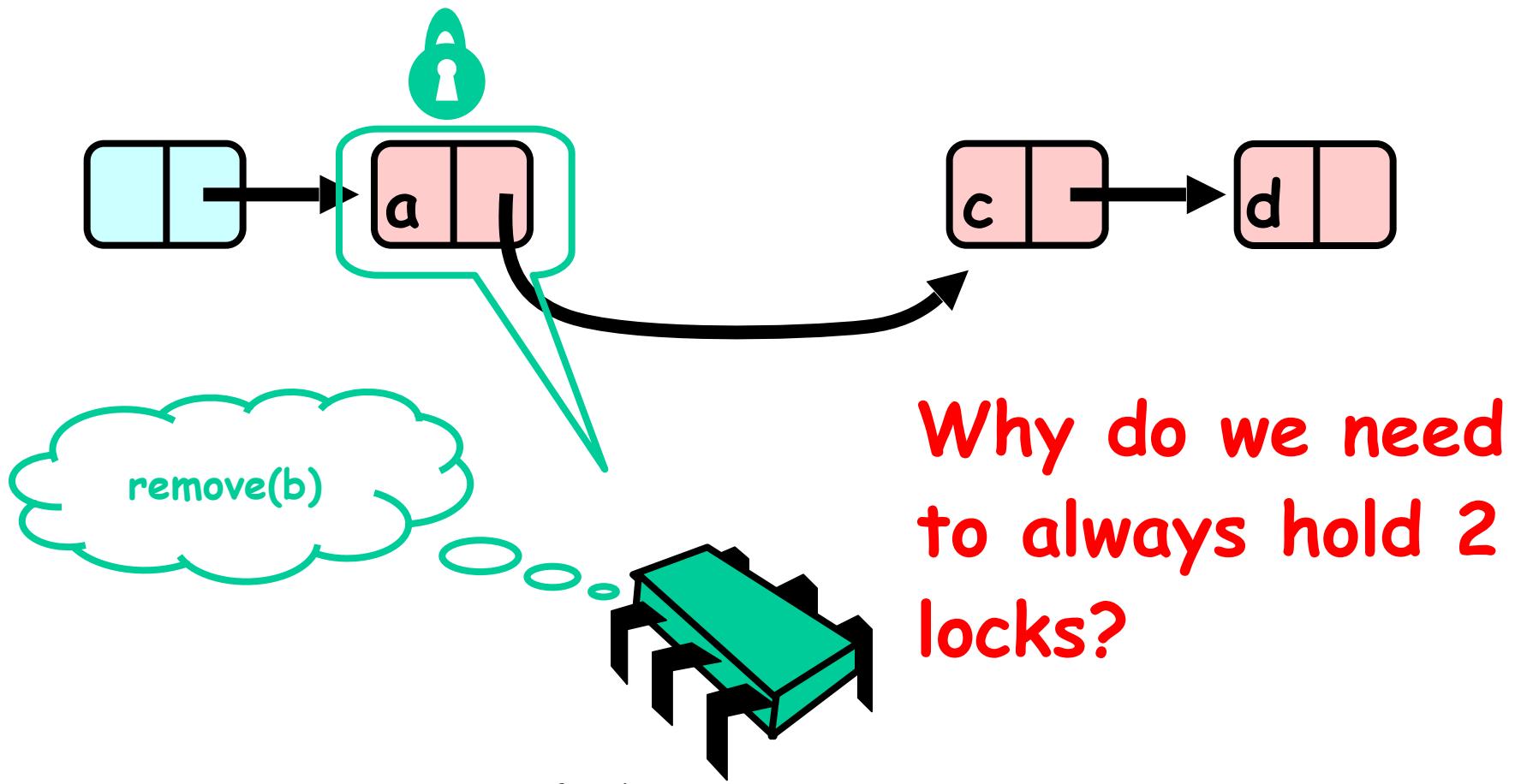
# Removing a Node



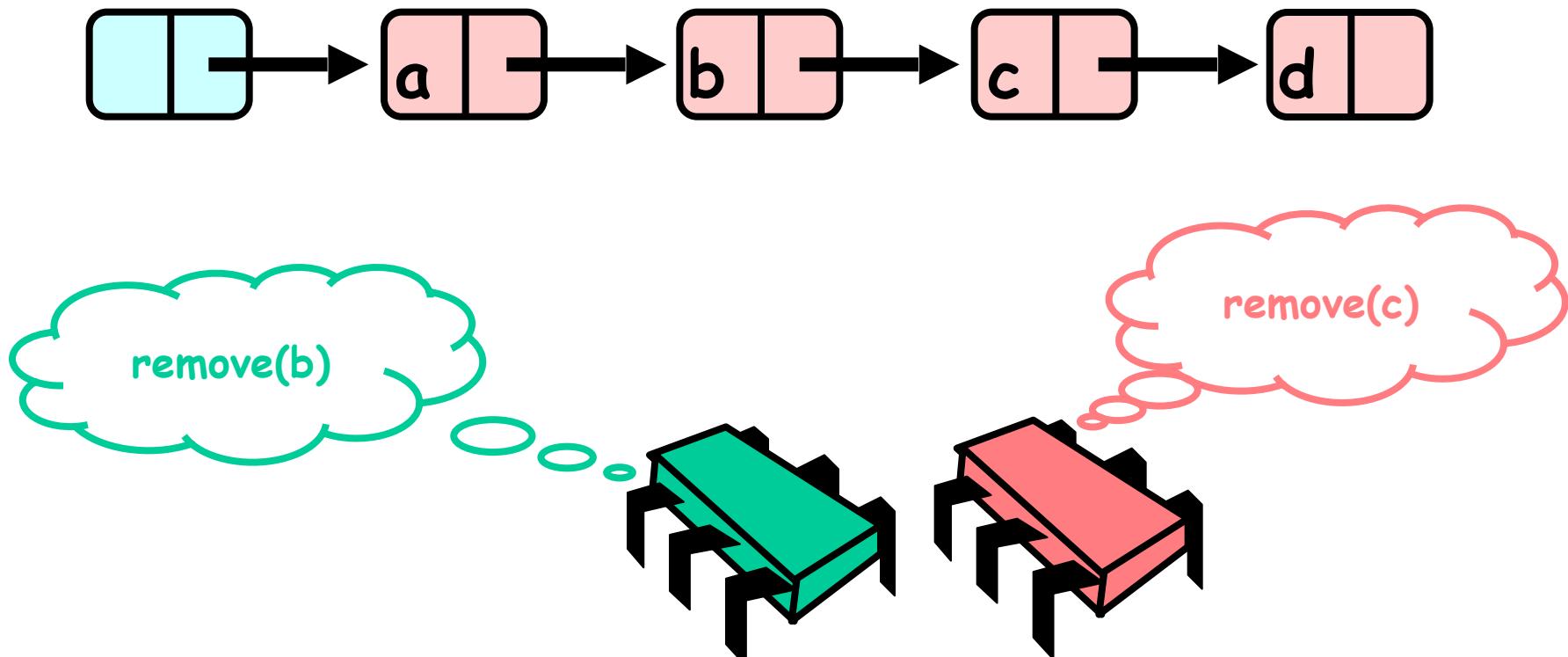
# Removing a Node



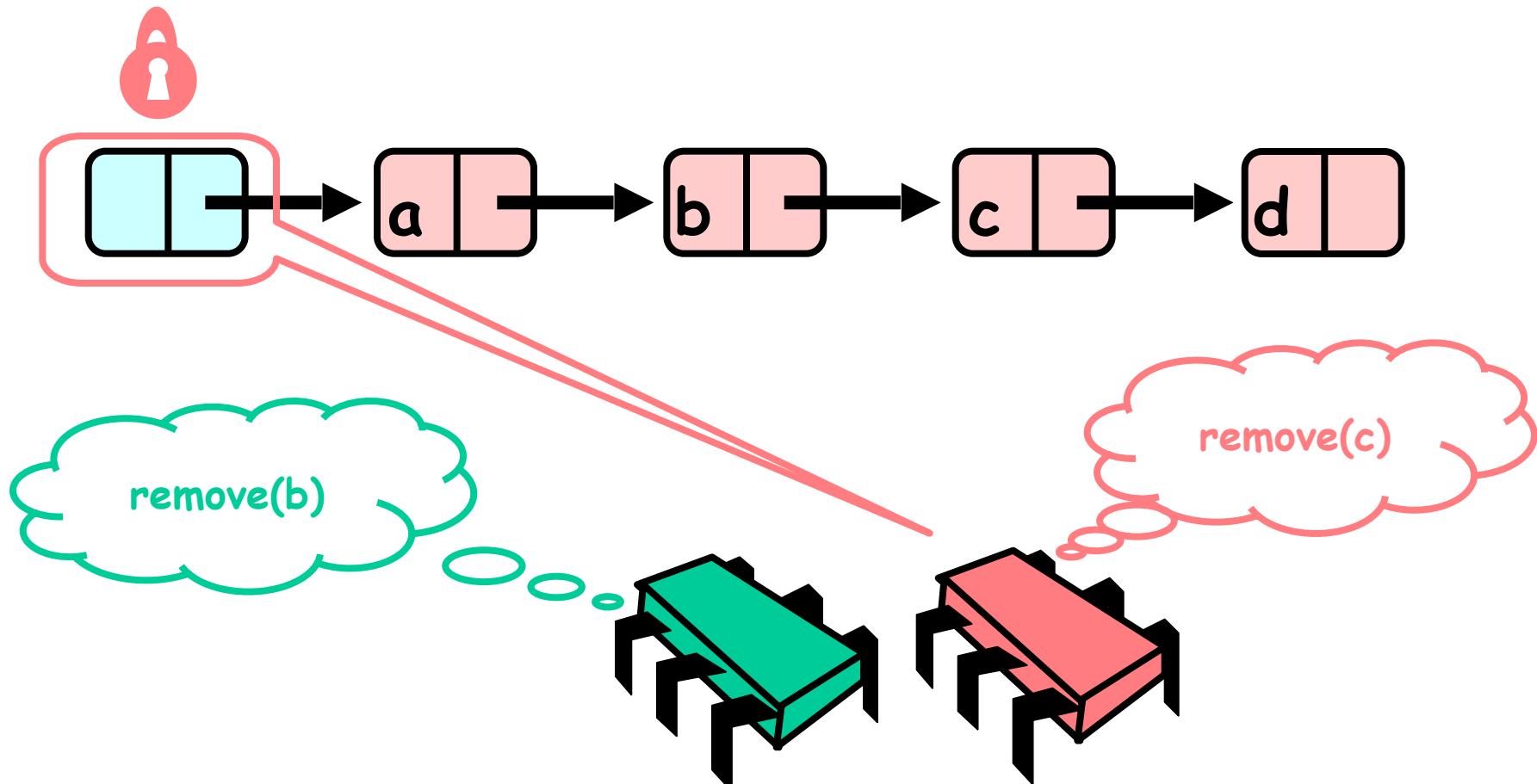
# Removing a Node



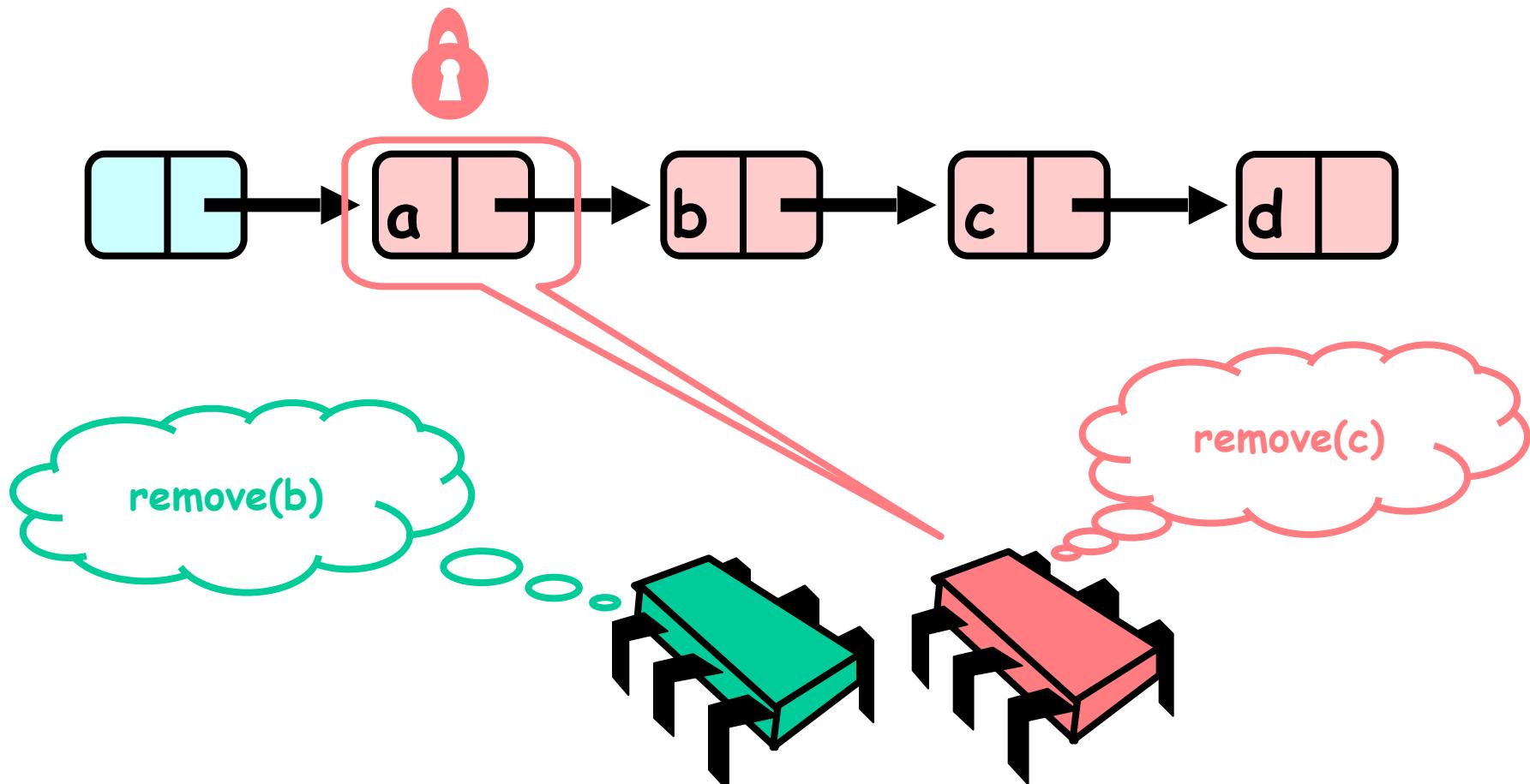
# Concurrent Removes



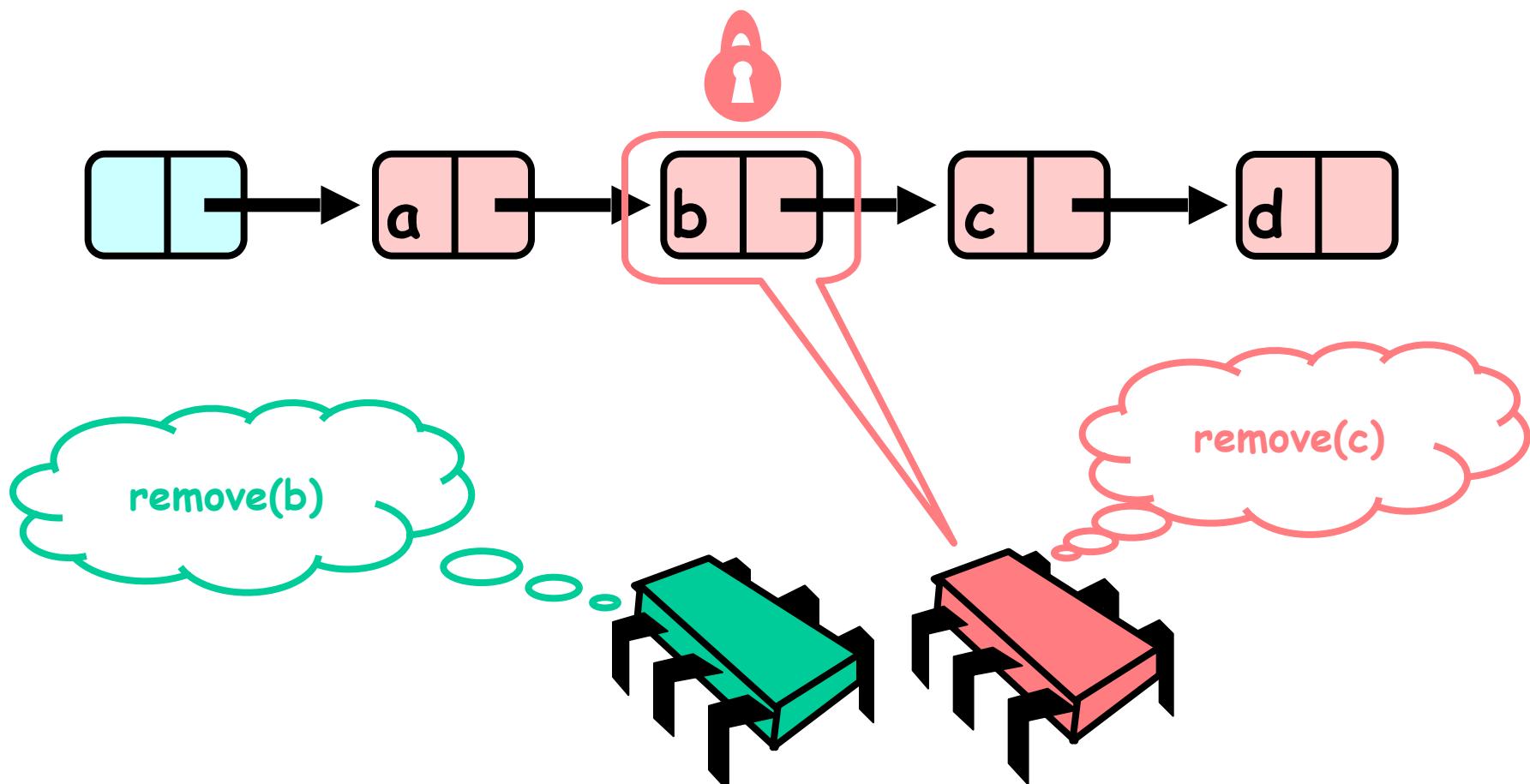
# Concurrent Removes



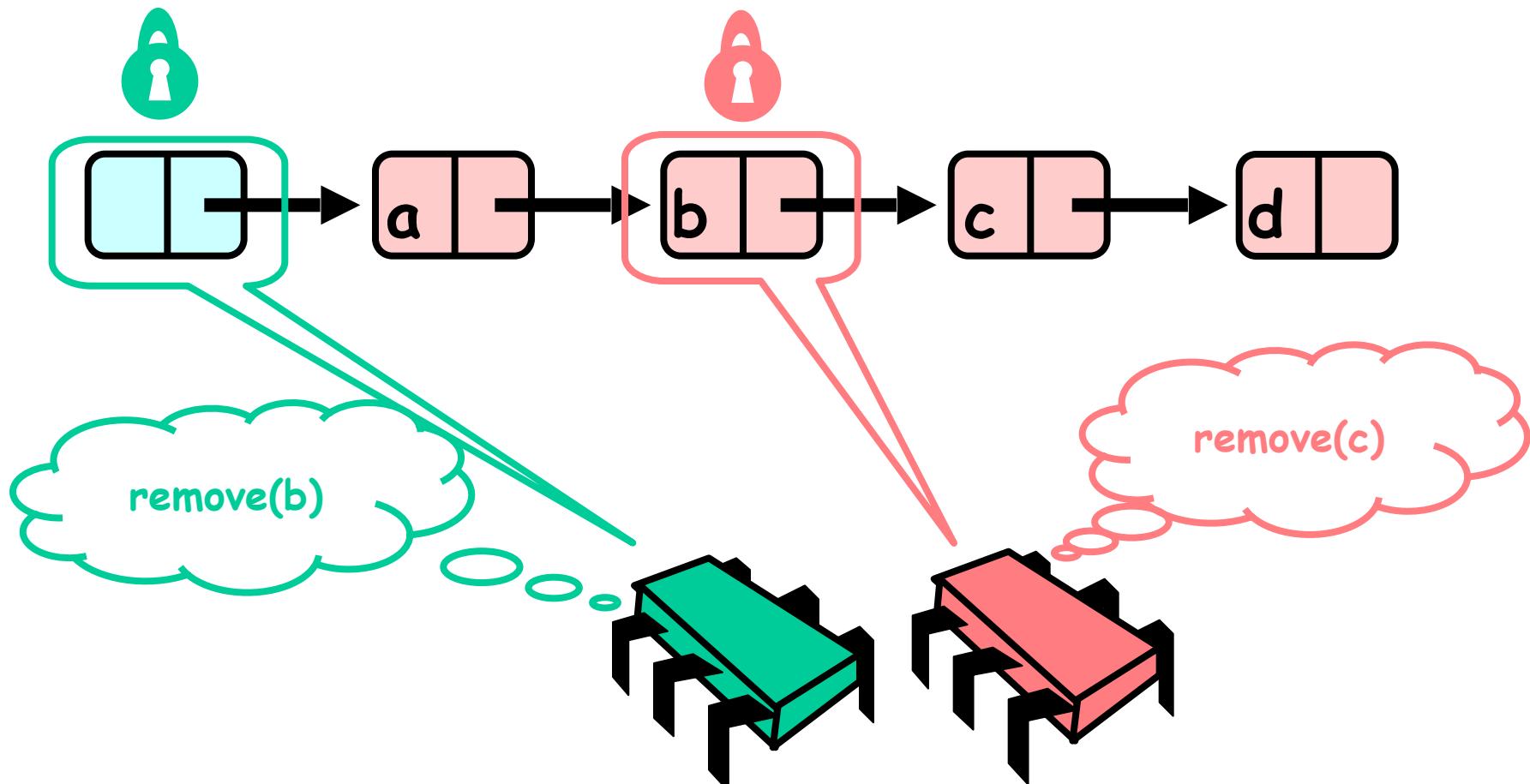
# Concurrent Removes



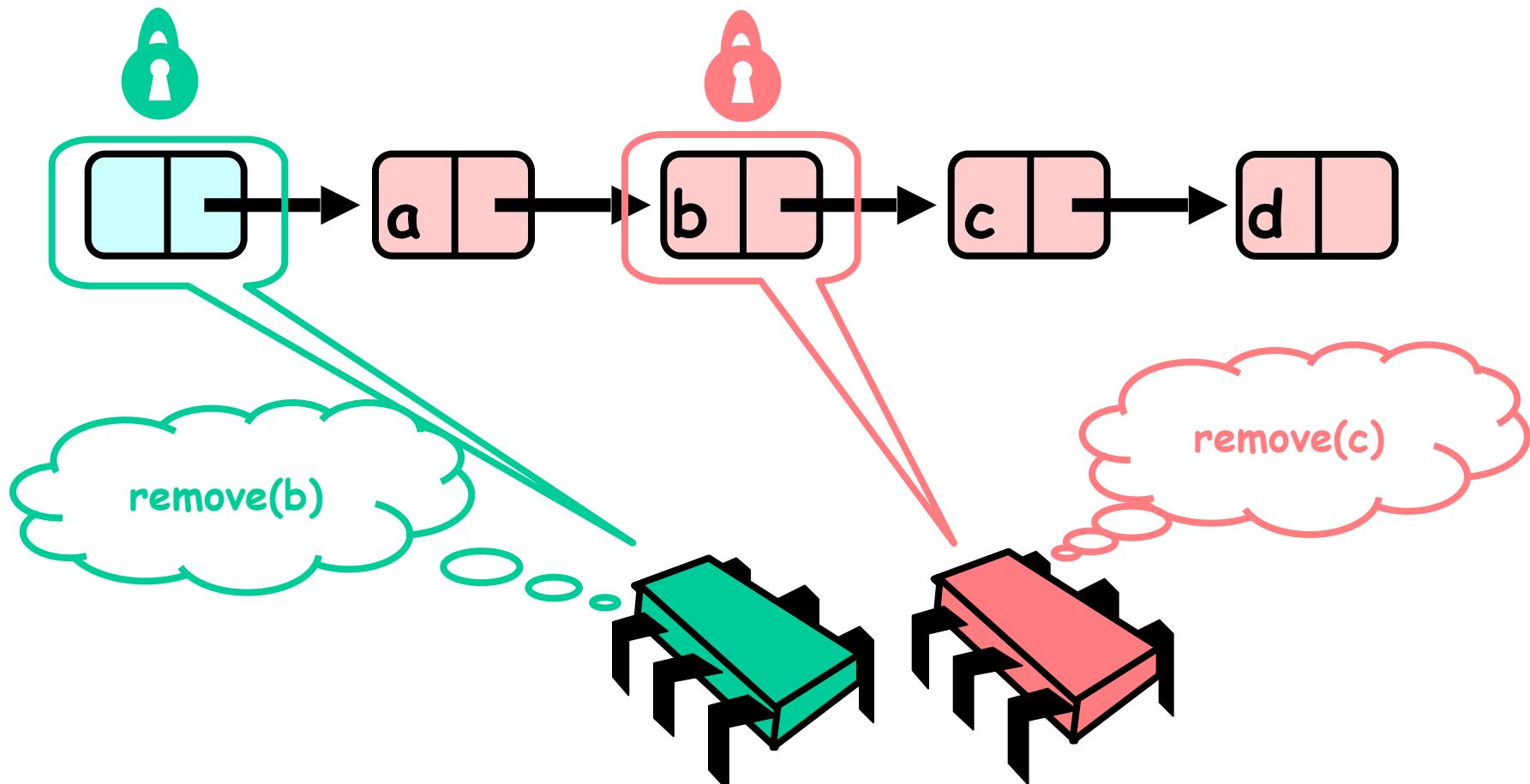
# Concurrent Removes



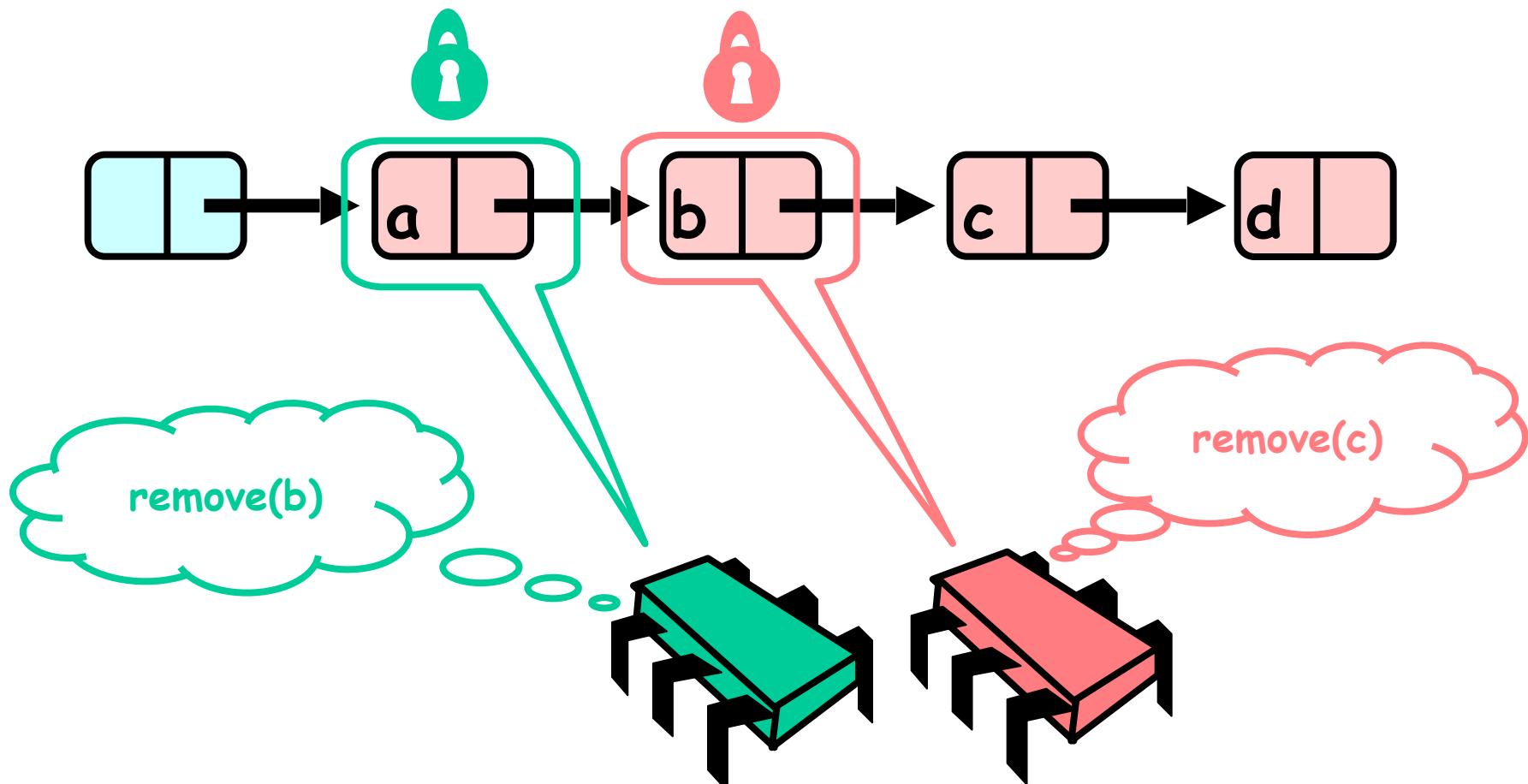
# Concurrent Removes



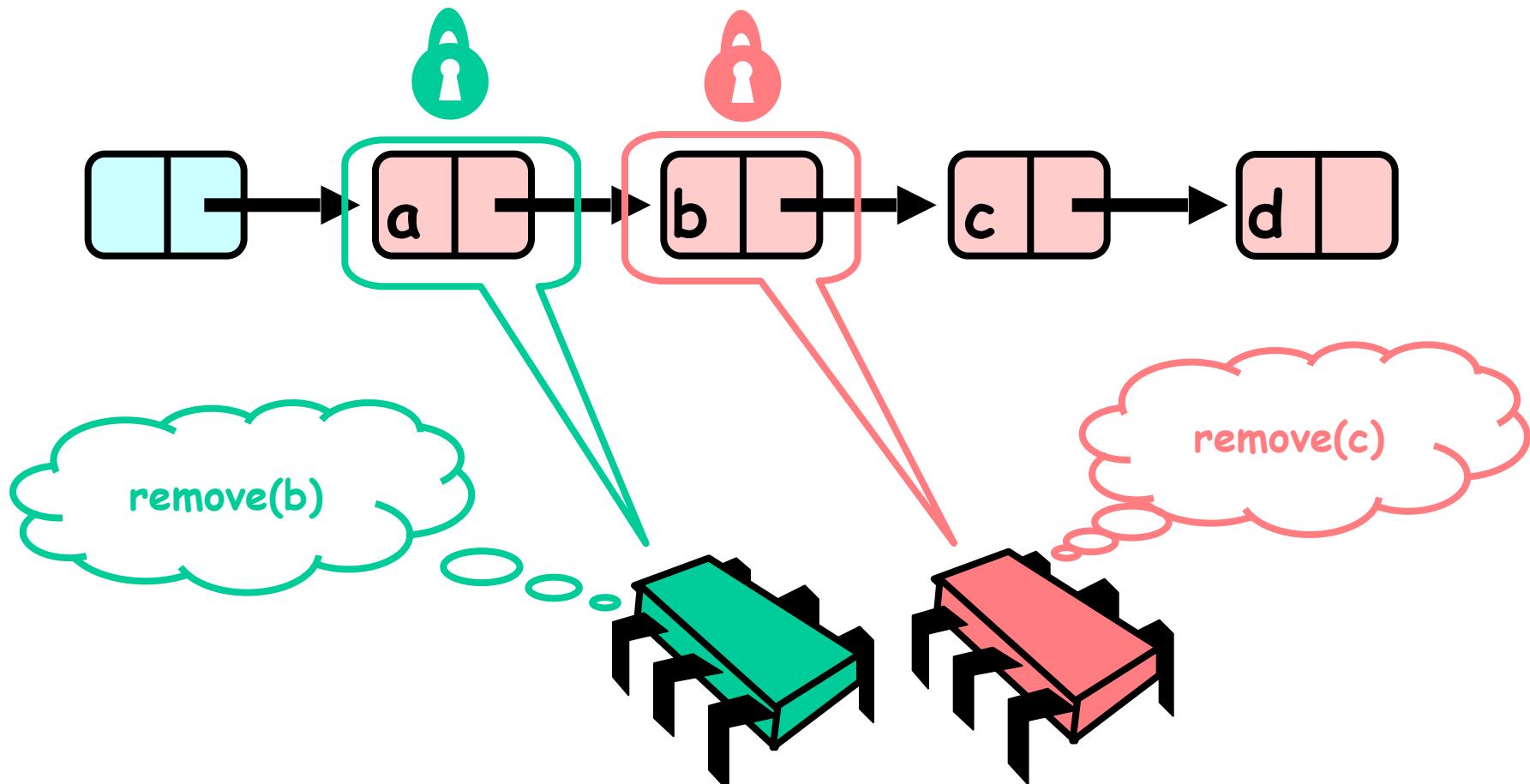
# Concurrent Removes



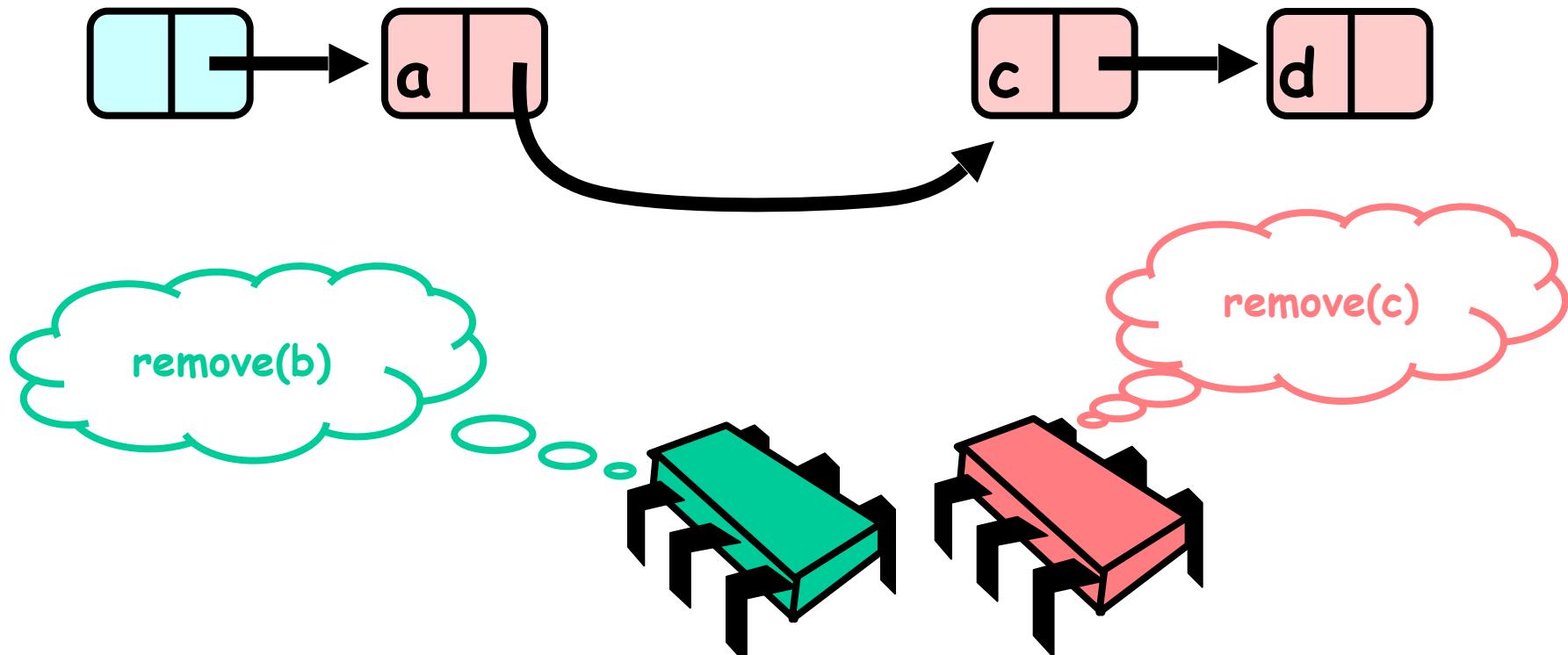
# Concurrent Removes



# Concurrent Removes

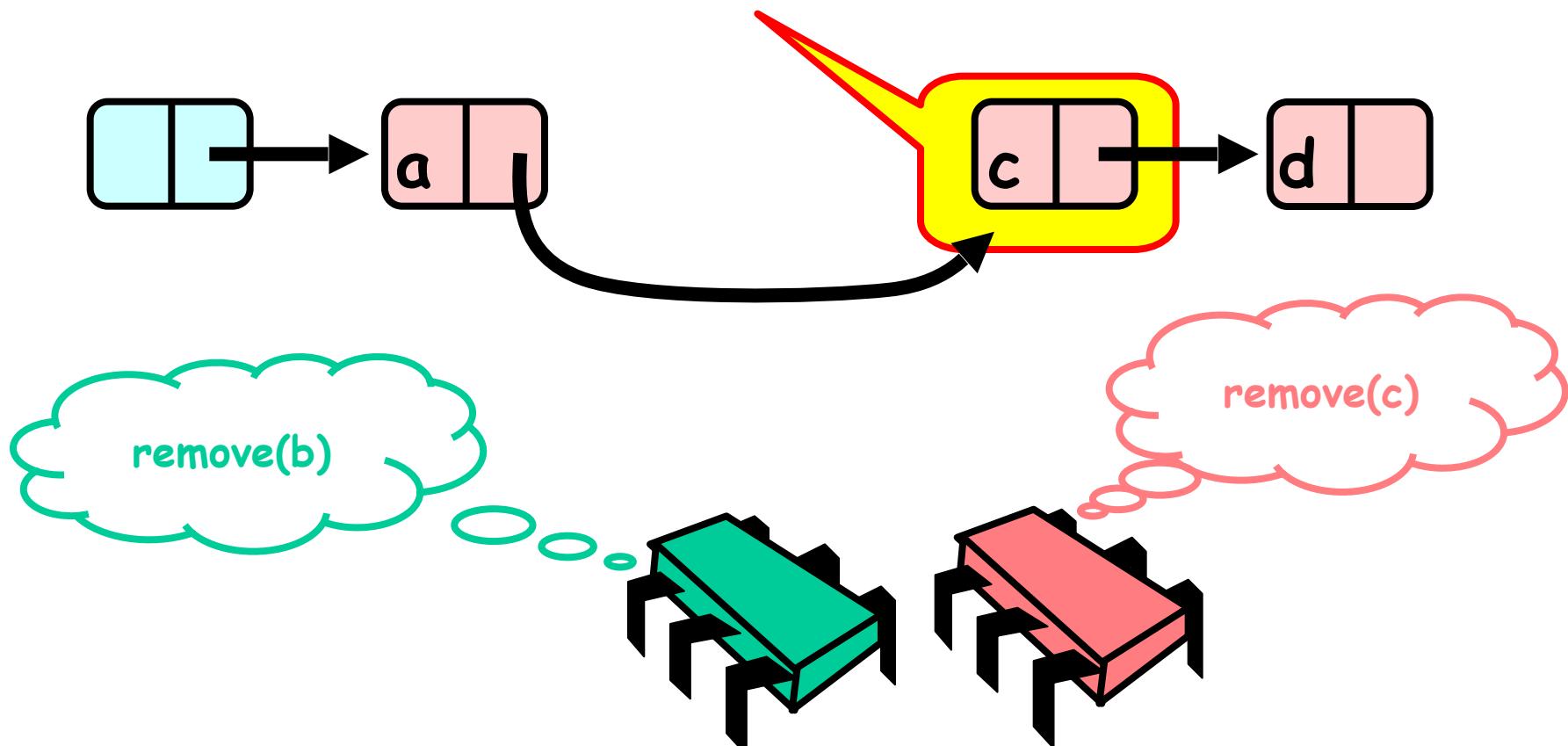


# Uh, Oh



# Uh, Oh

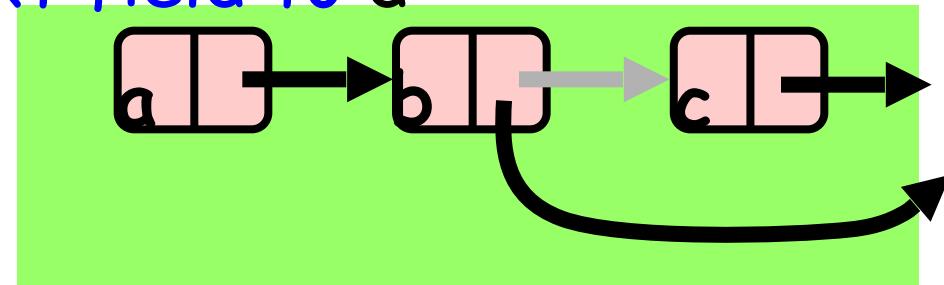
Bad news, C not removed



# Problem

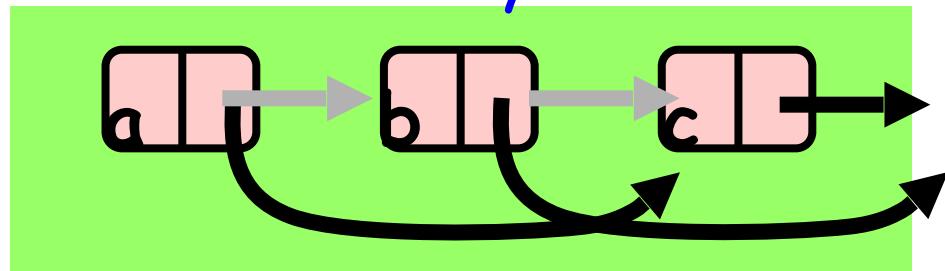
- To delete node c

- Swing node b's next field to d



- Problem is,

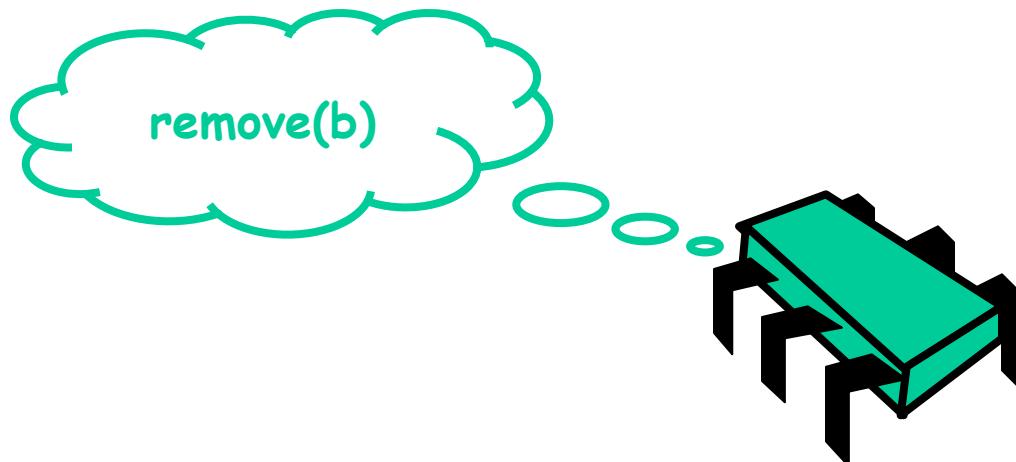
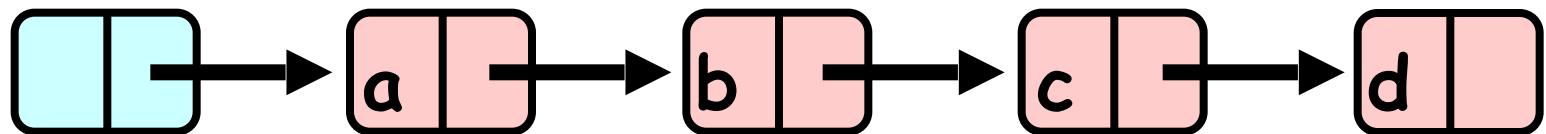
- Someone deleting b concurrently could direct a pointer to c



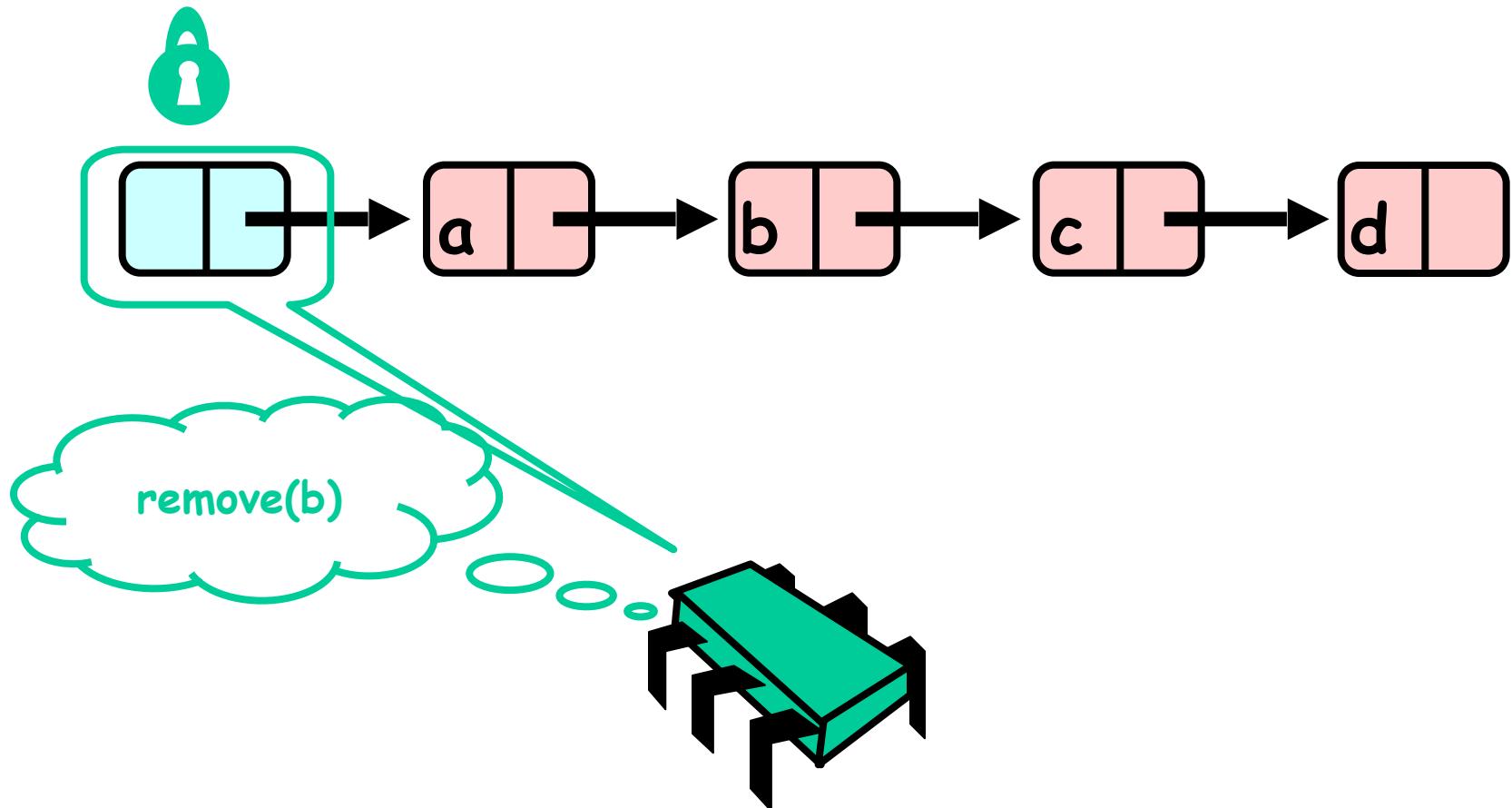
# Insight

- If a node is locked
  - No one can delete node's successor
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works

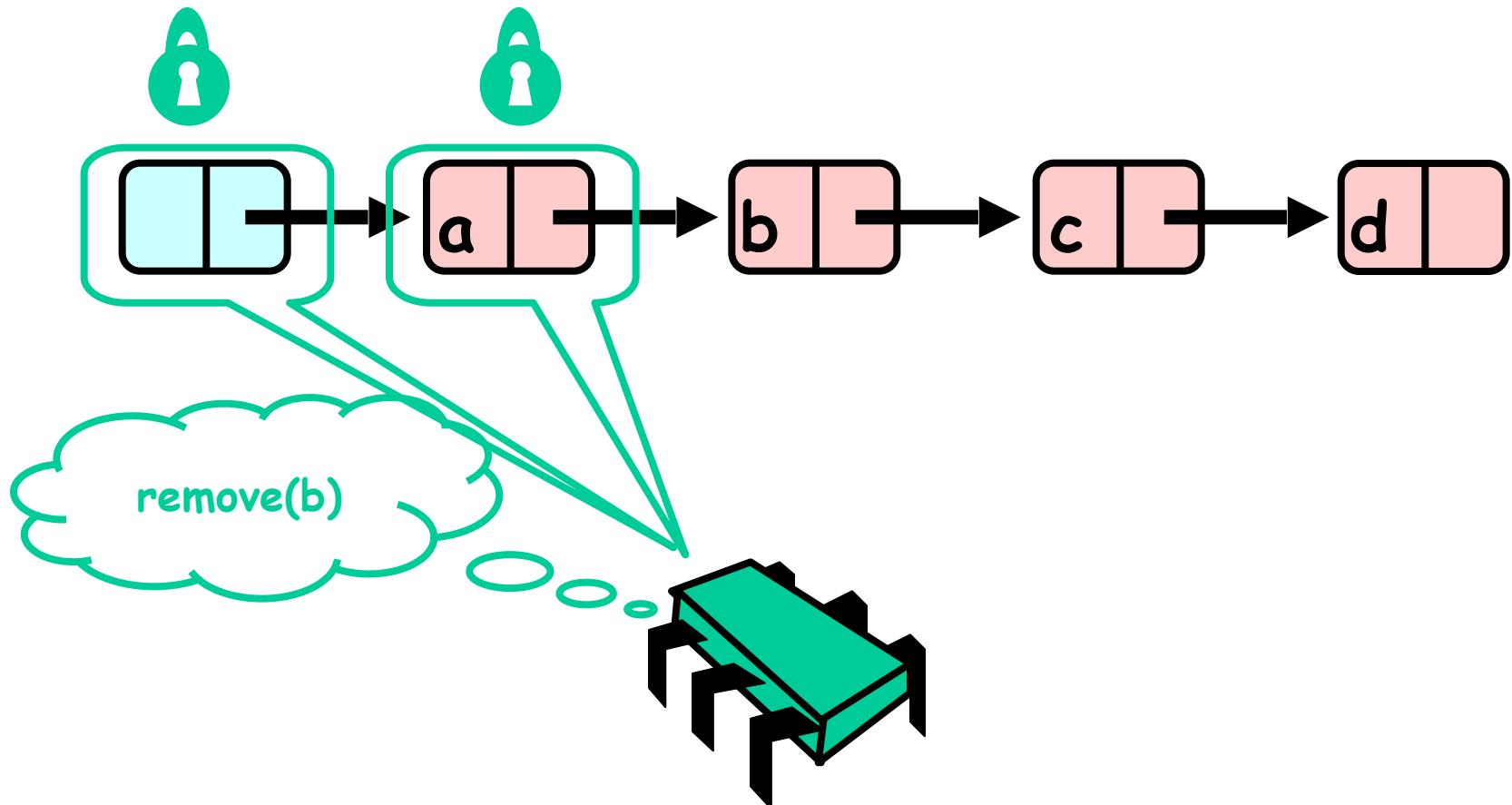
# Hand-Over-Hand Again



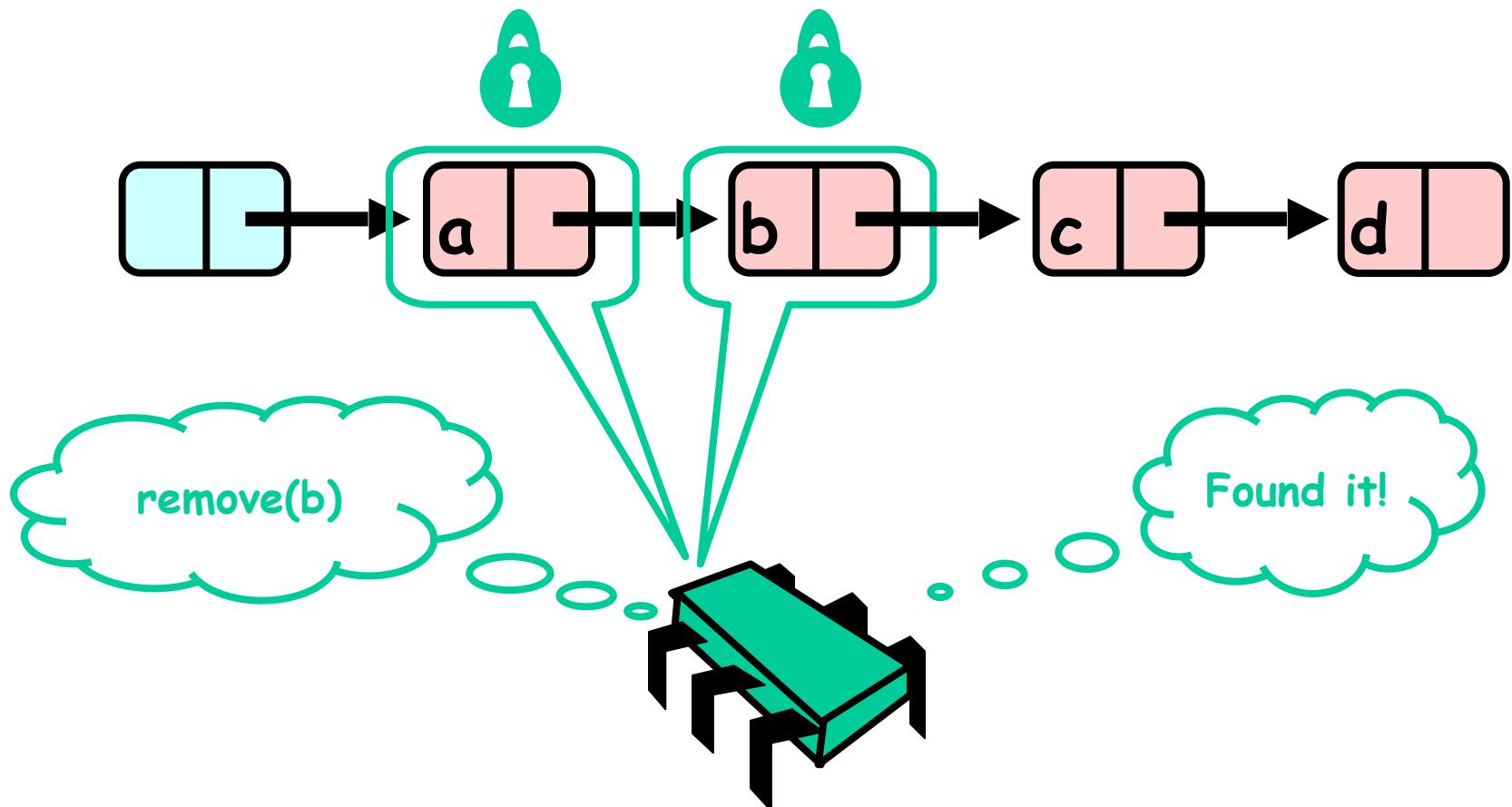
# Hand-Over-Hand Again



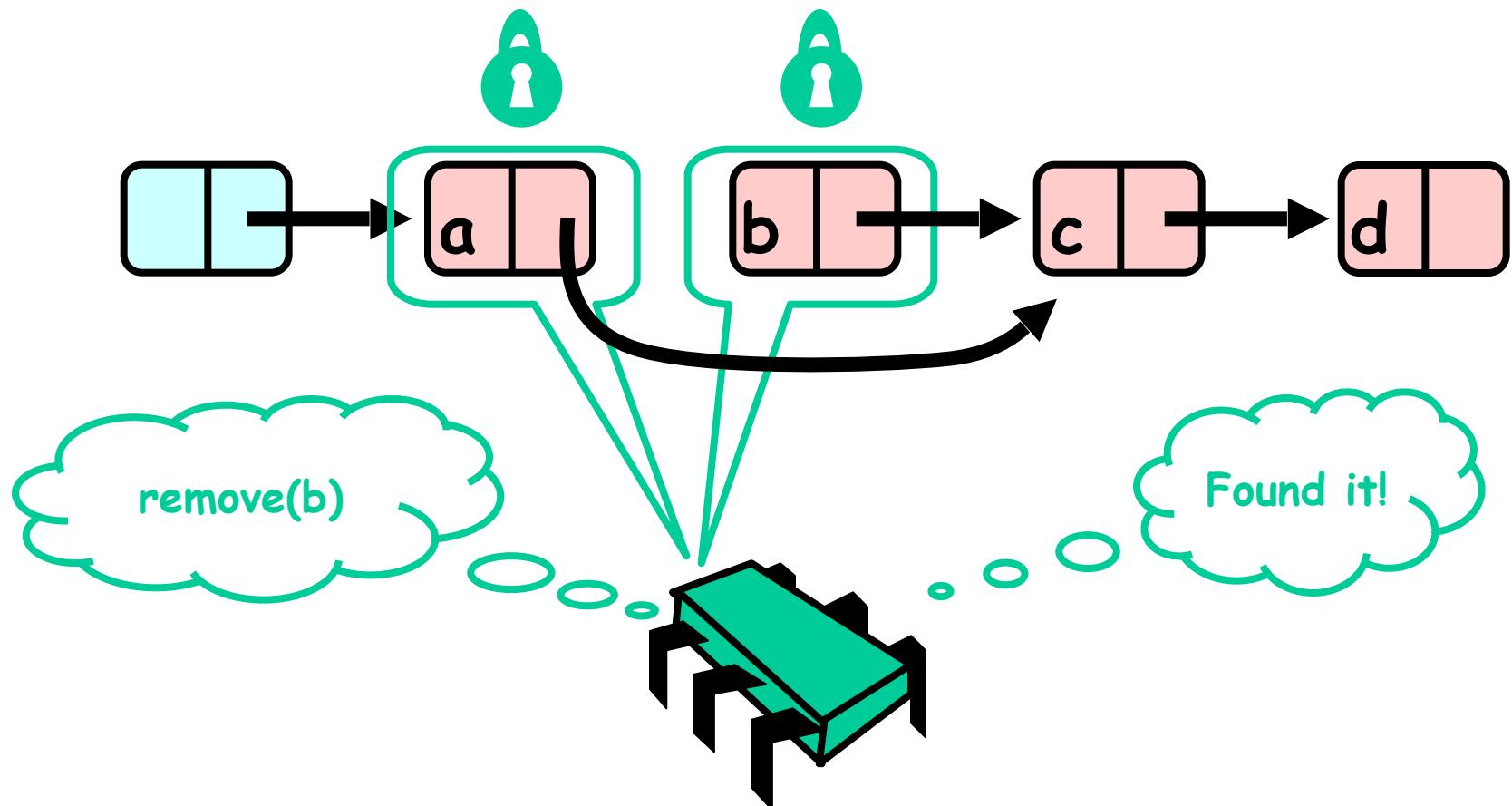
# Hand-Over-Hand Again



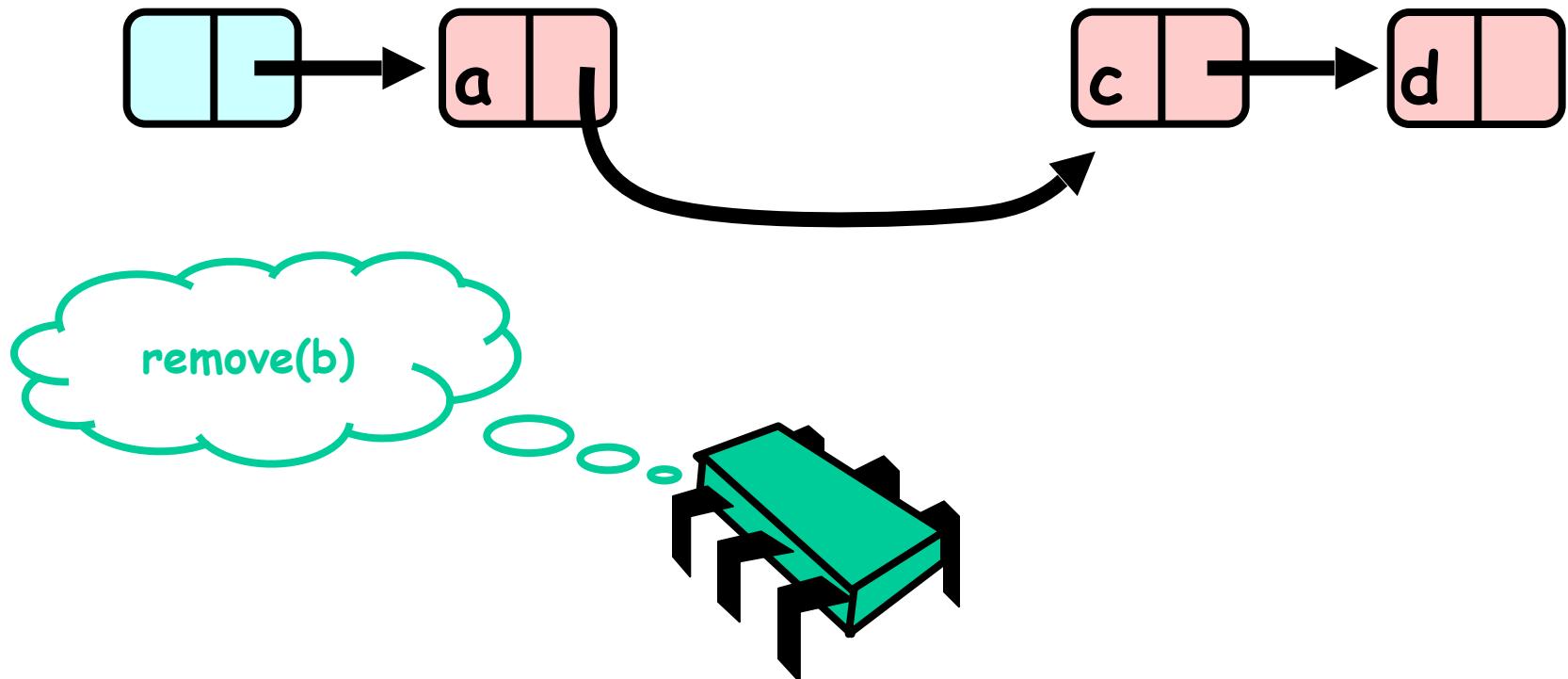
# Hand-Over-Hand Again



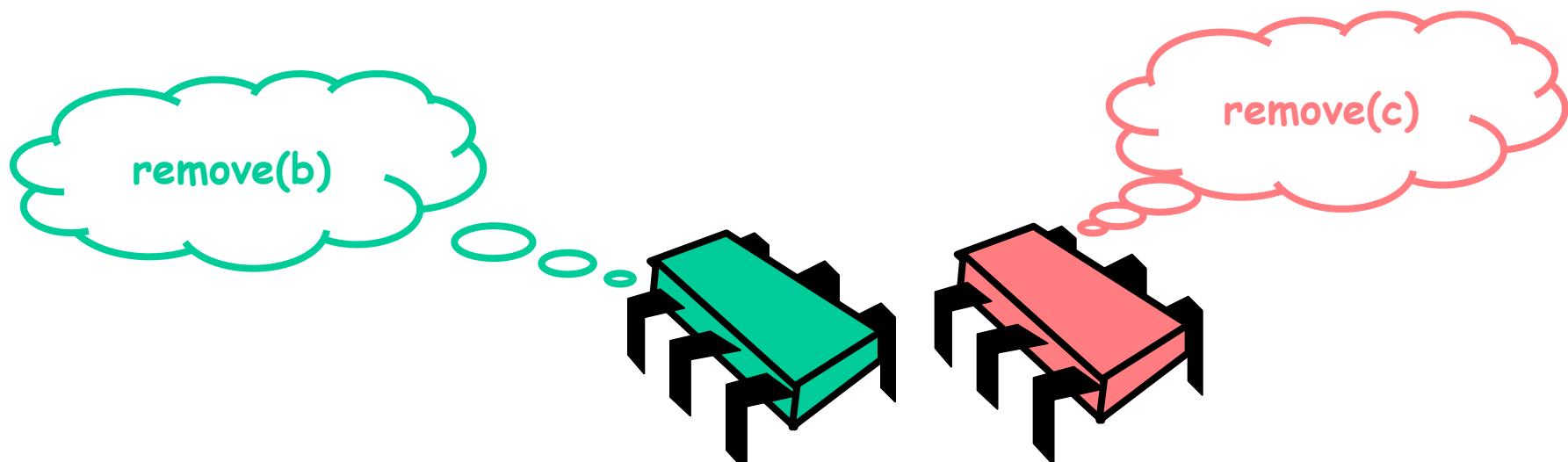
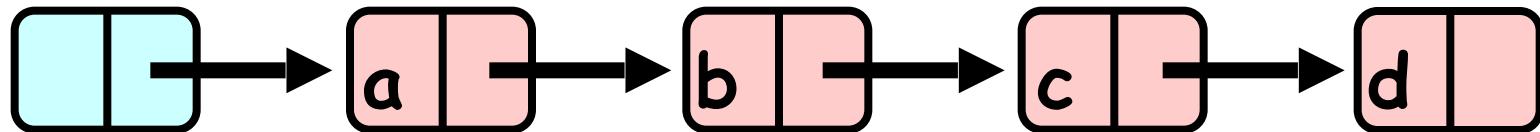
# Hand-Over-Hand Again



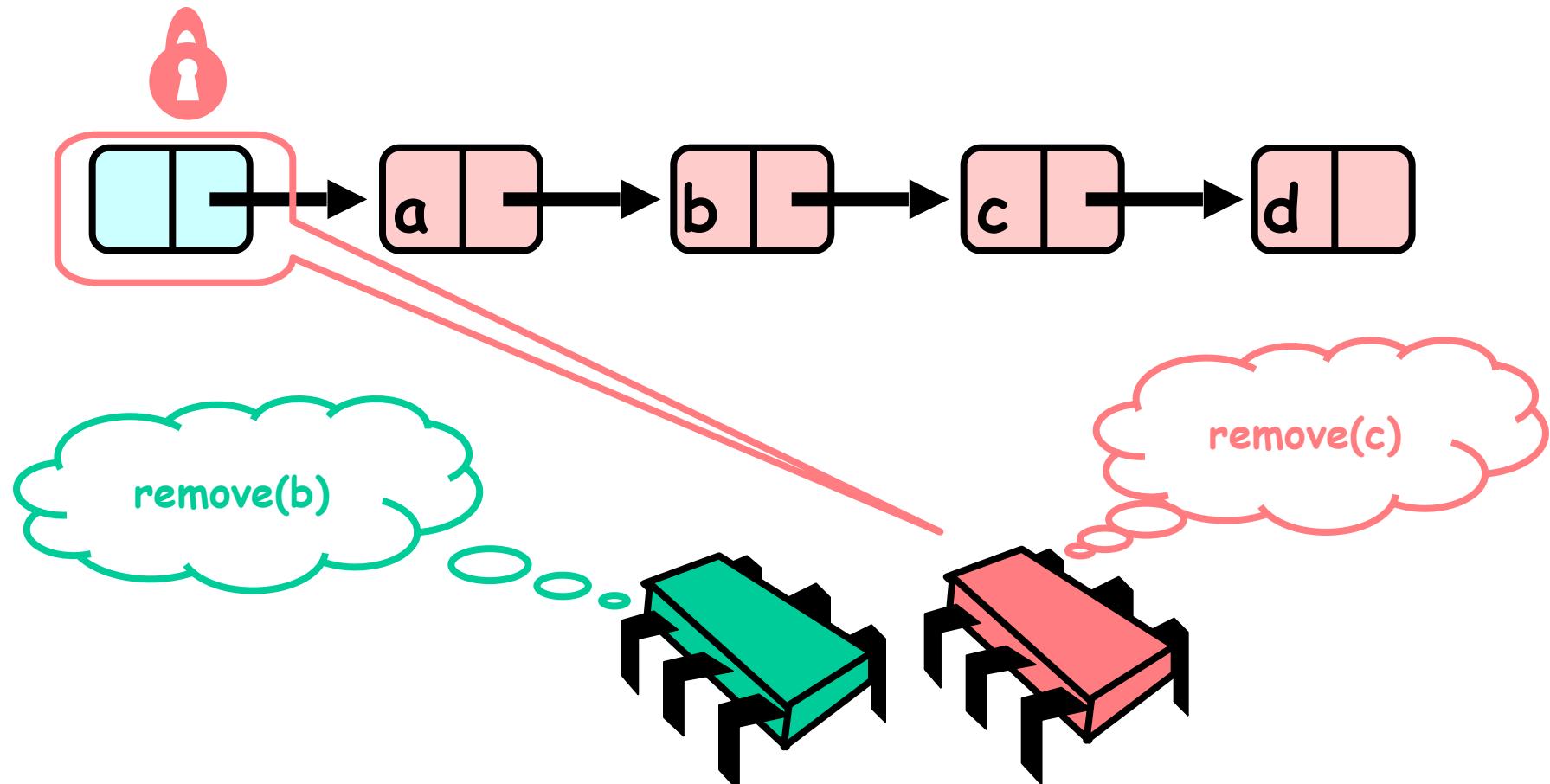
# Hand-Over-Hand Again



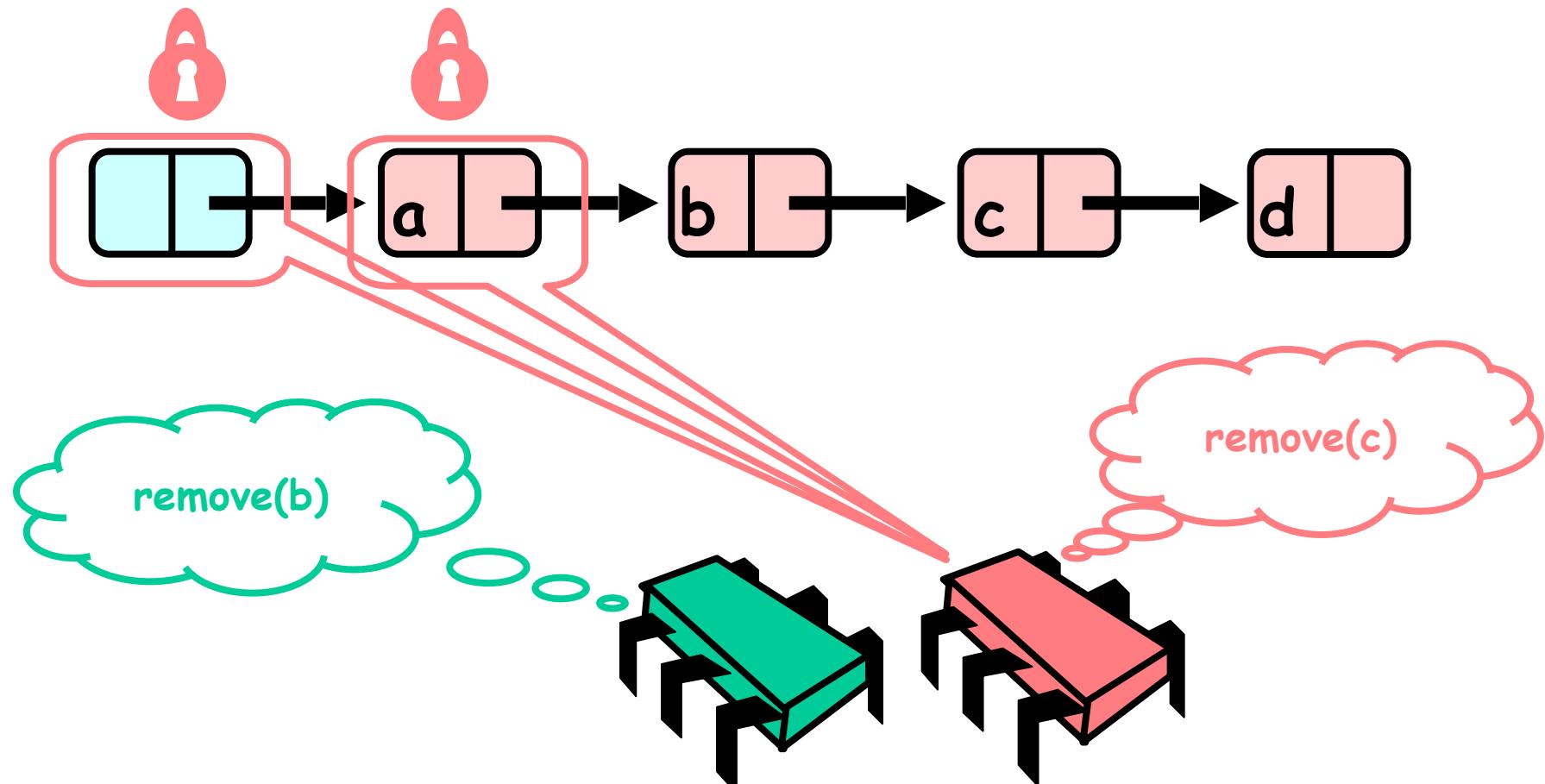
# Removing a Node



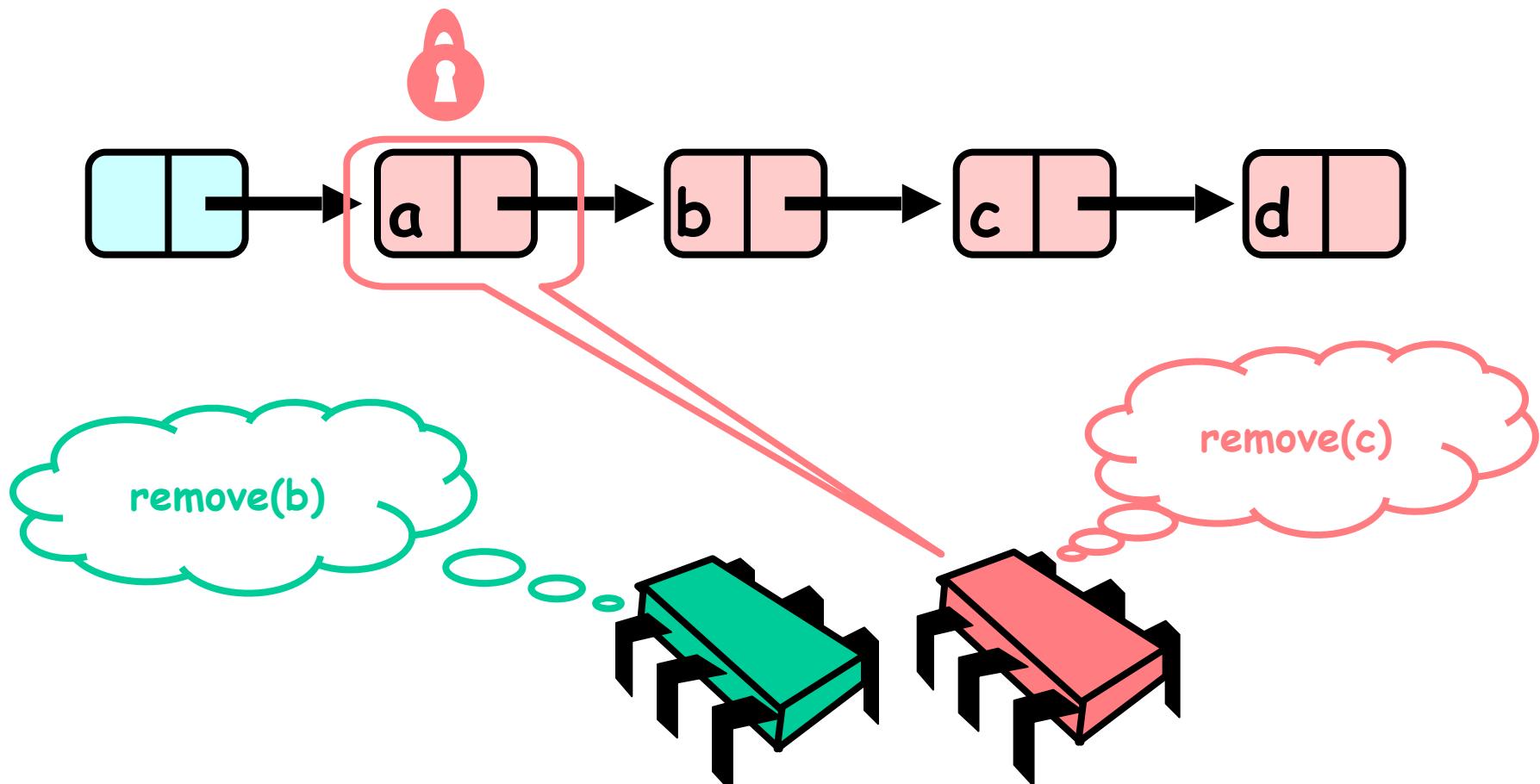
# Removing a Node



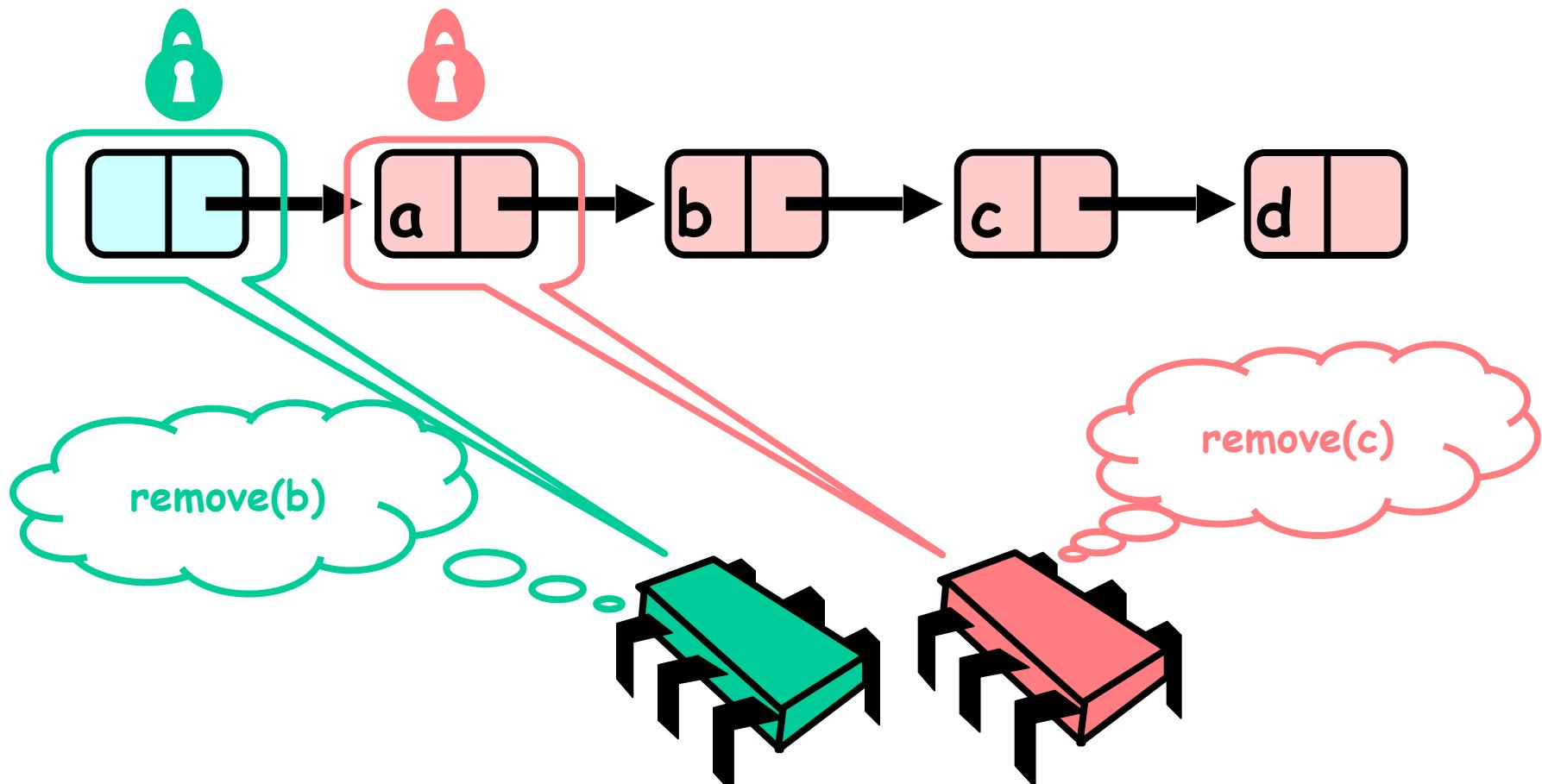
# Removing a Node



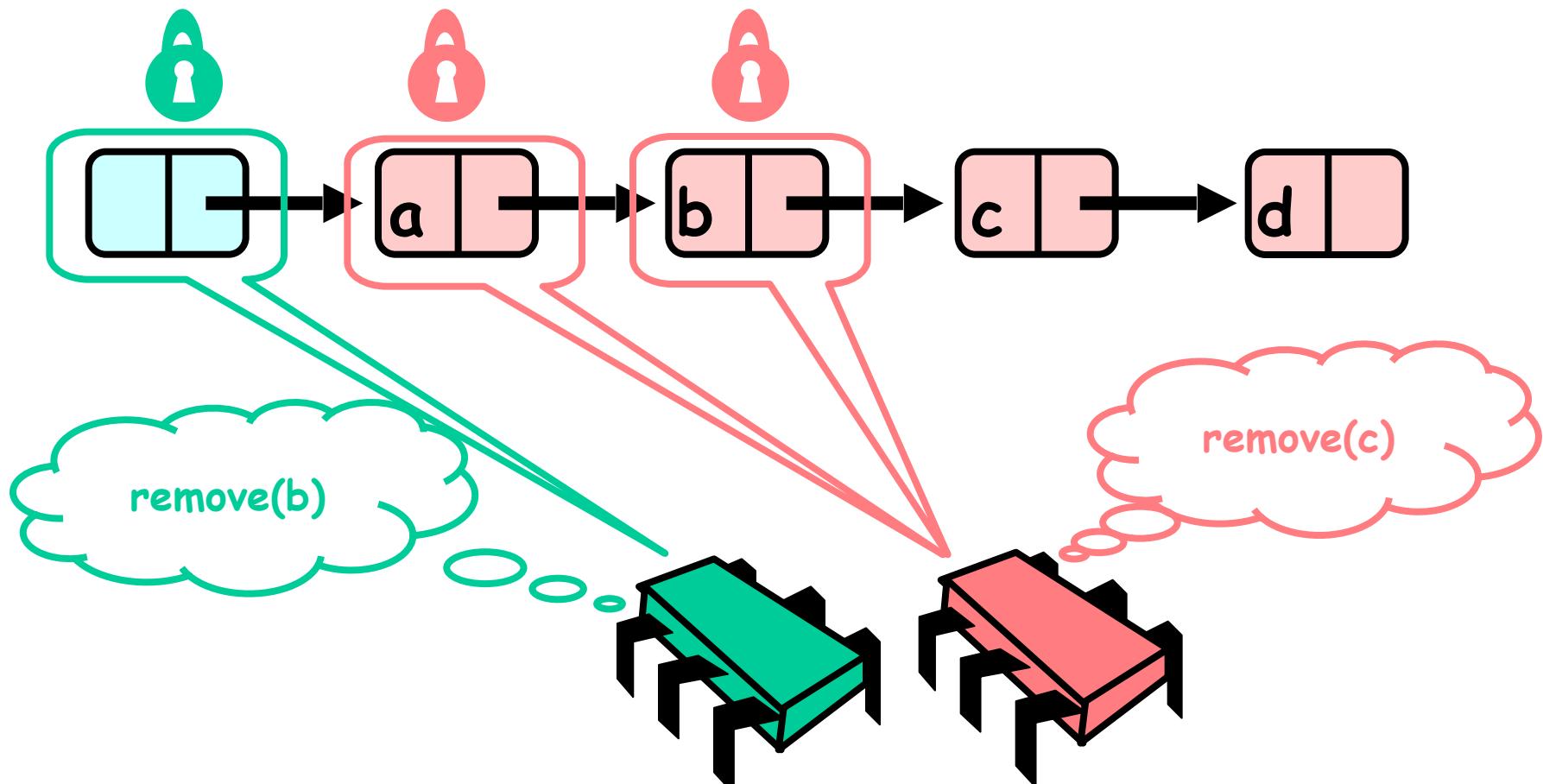
# Removing a Node



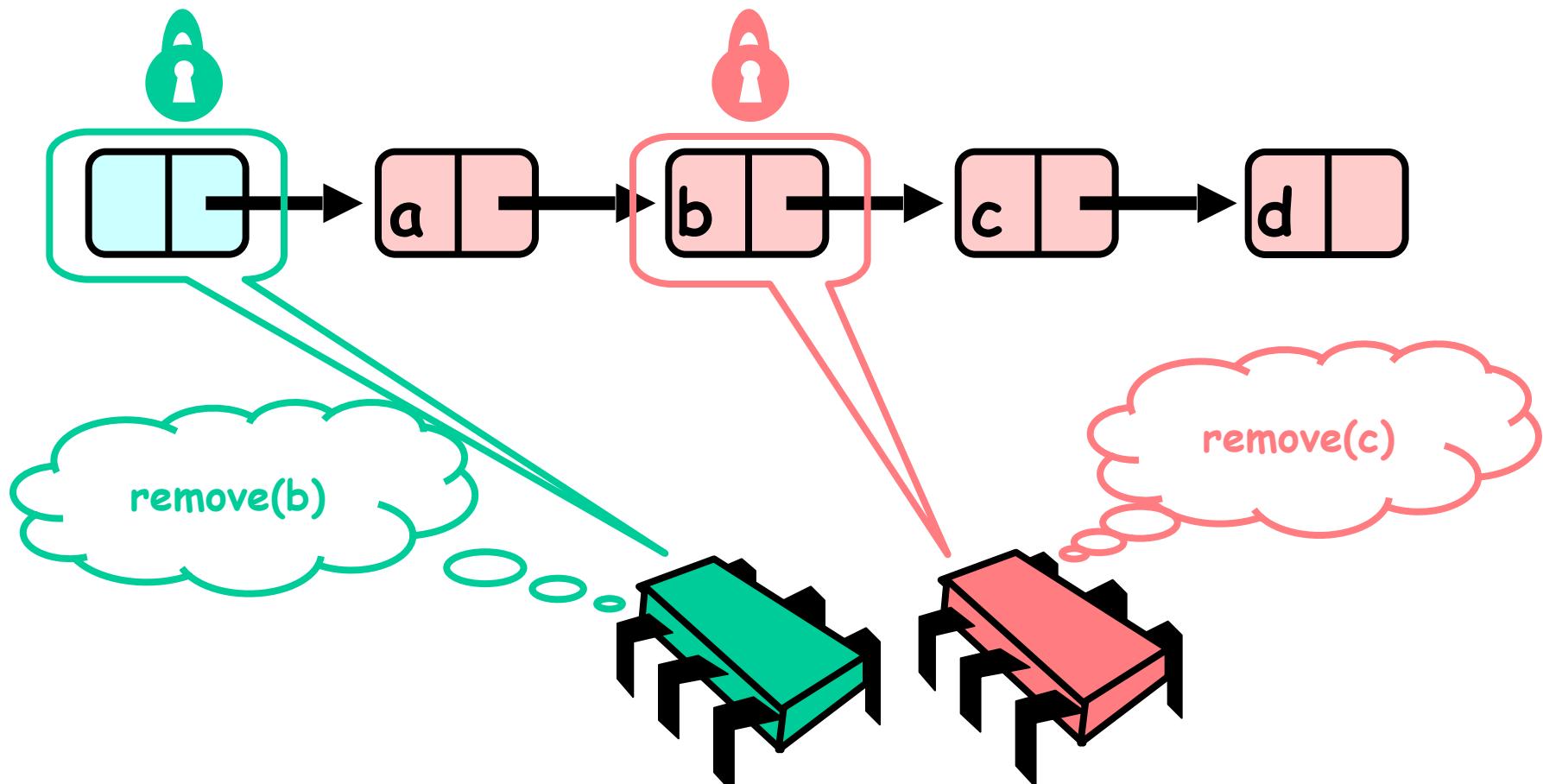
# Removing a Node



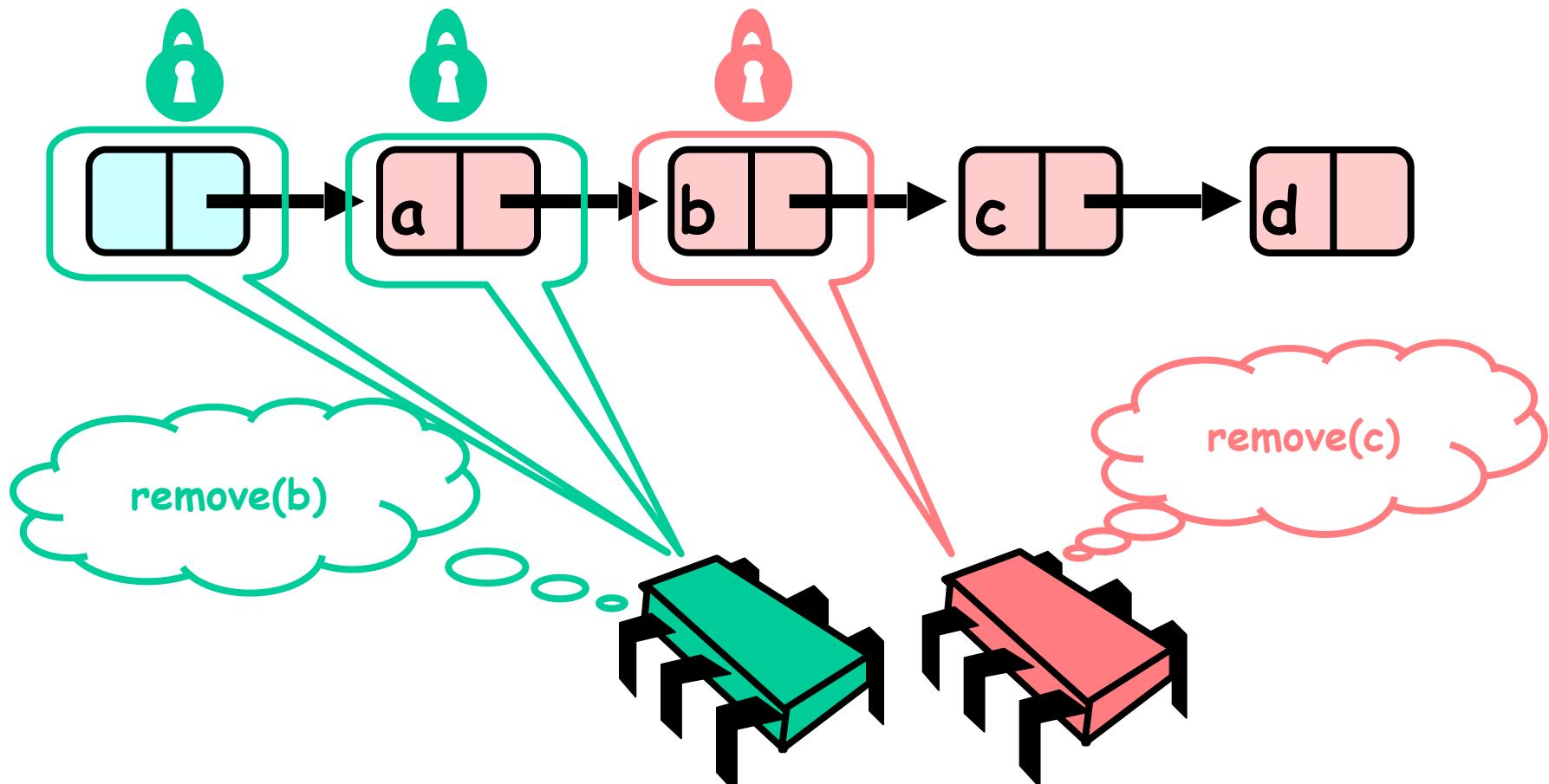
# Removing a Node



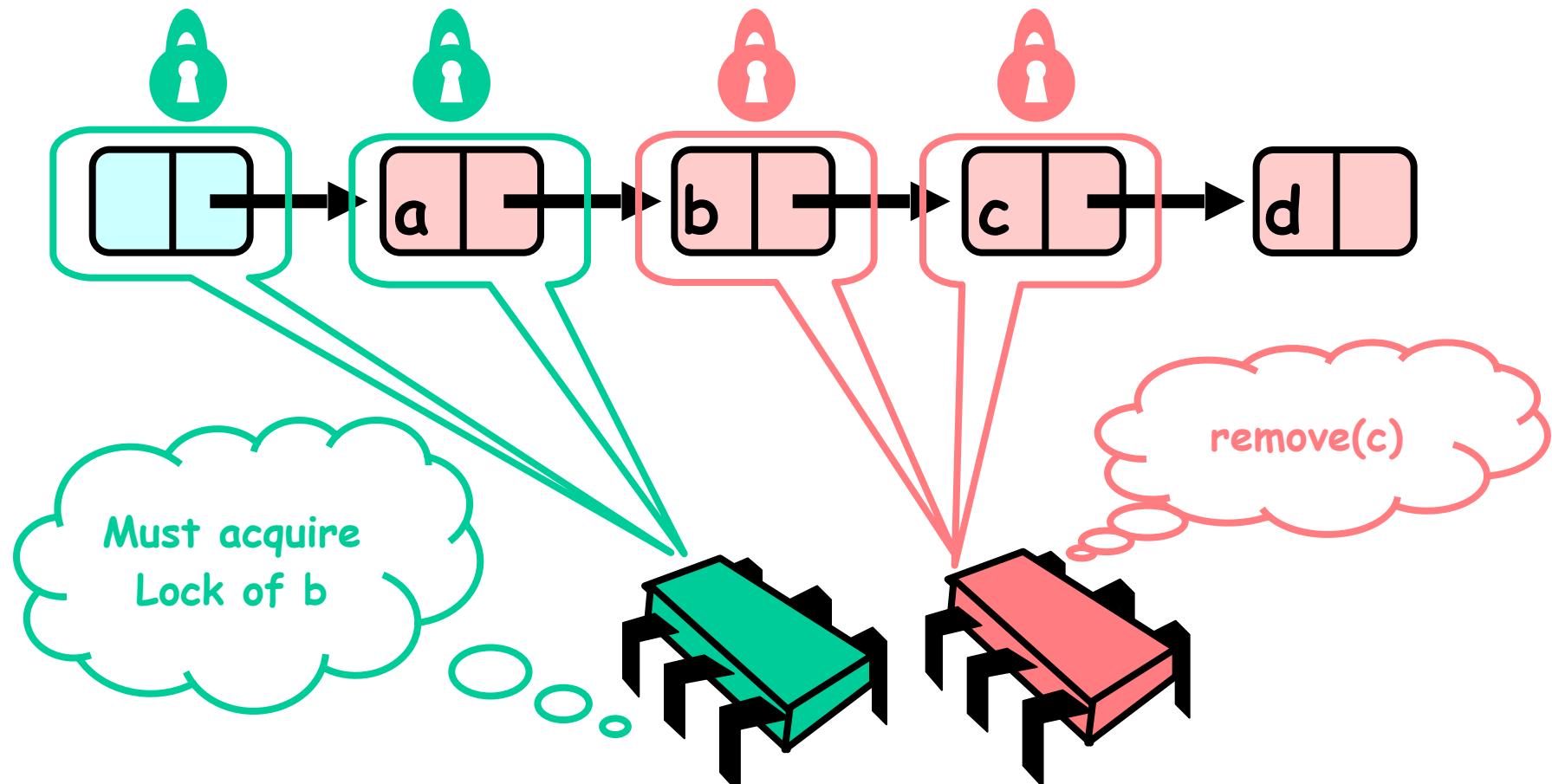
# Removing a Node



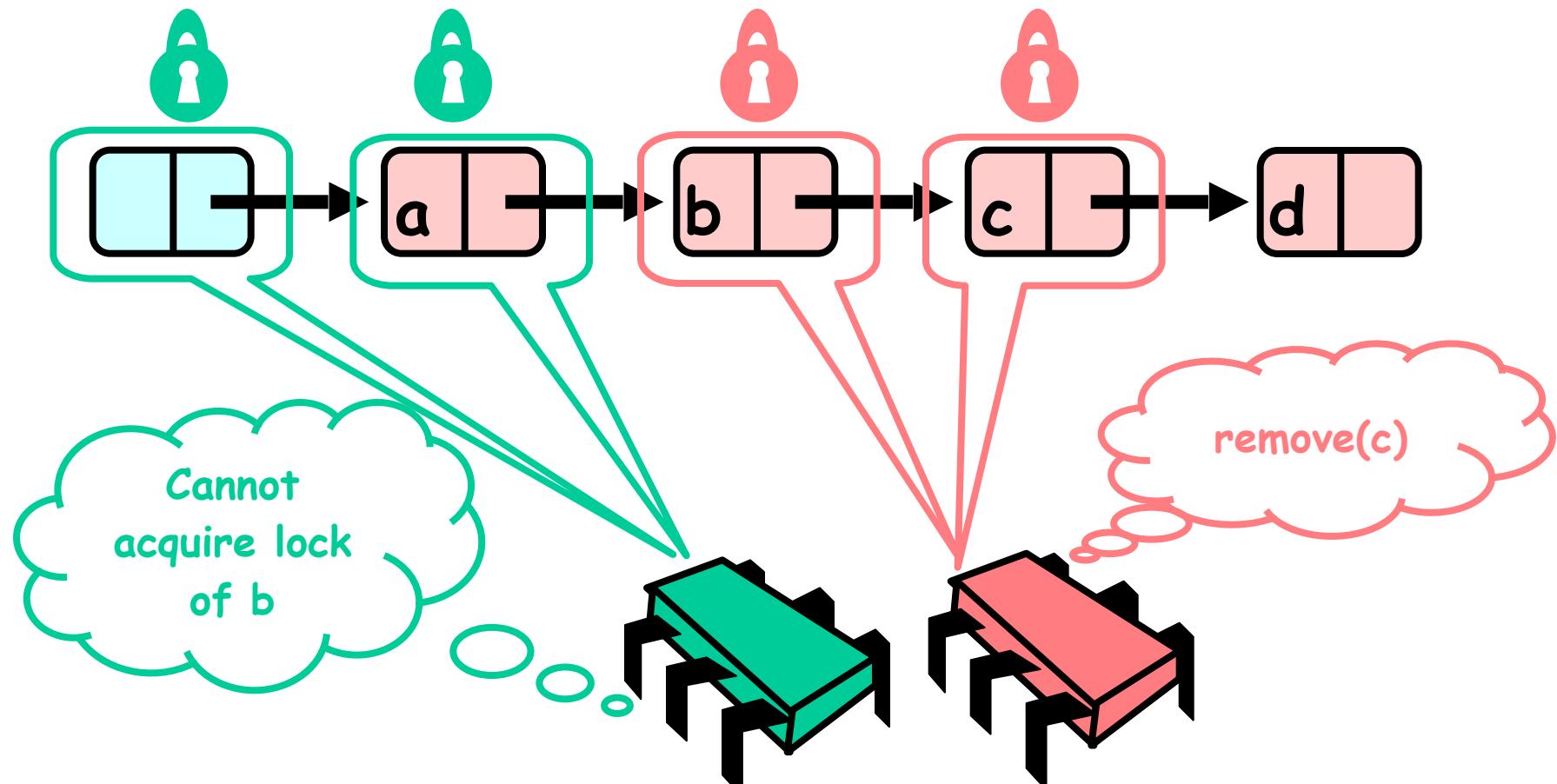
# Removing a Node



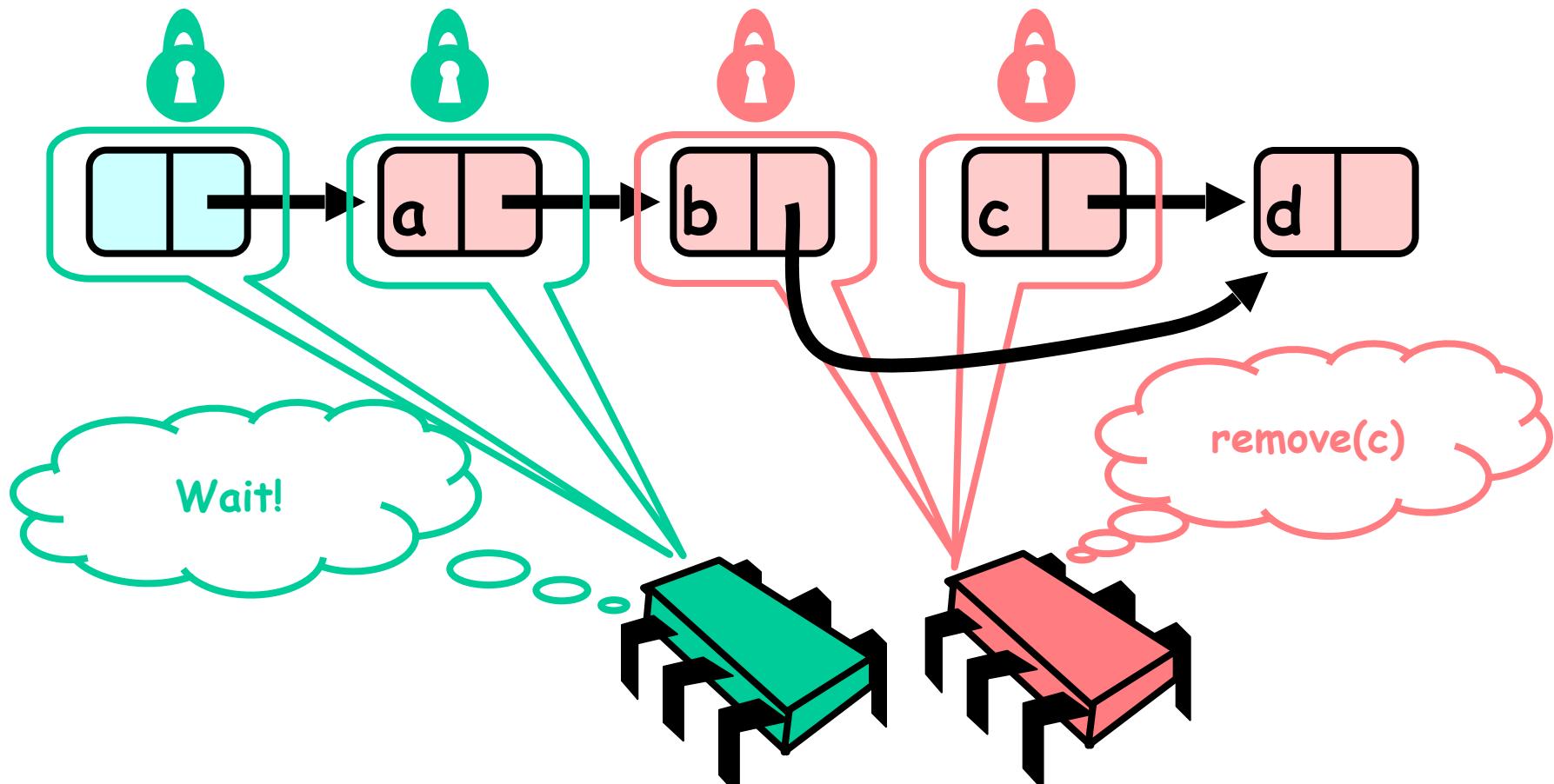
# Removing a Node



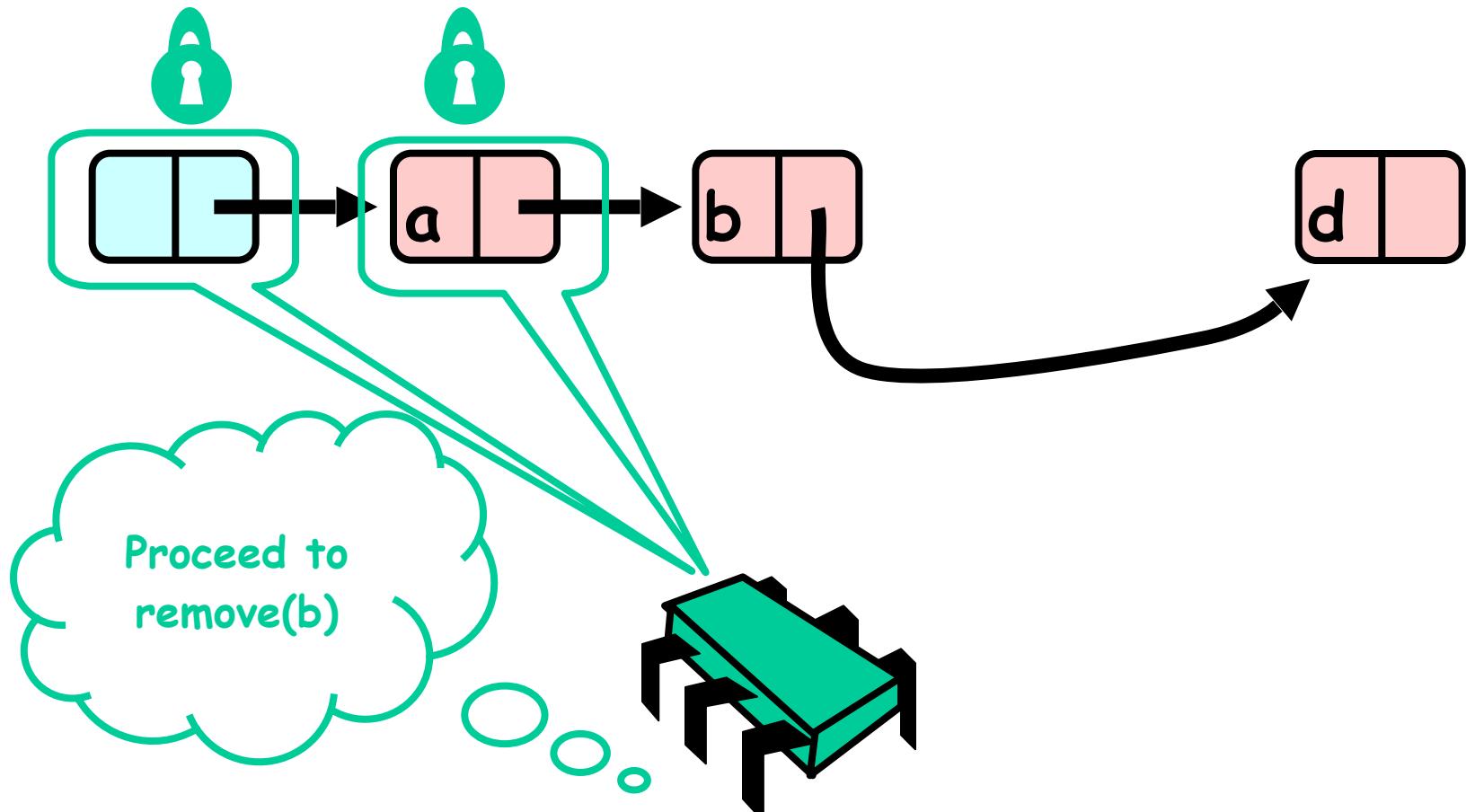
# Removing a Node



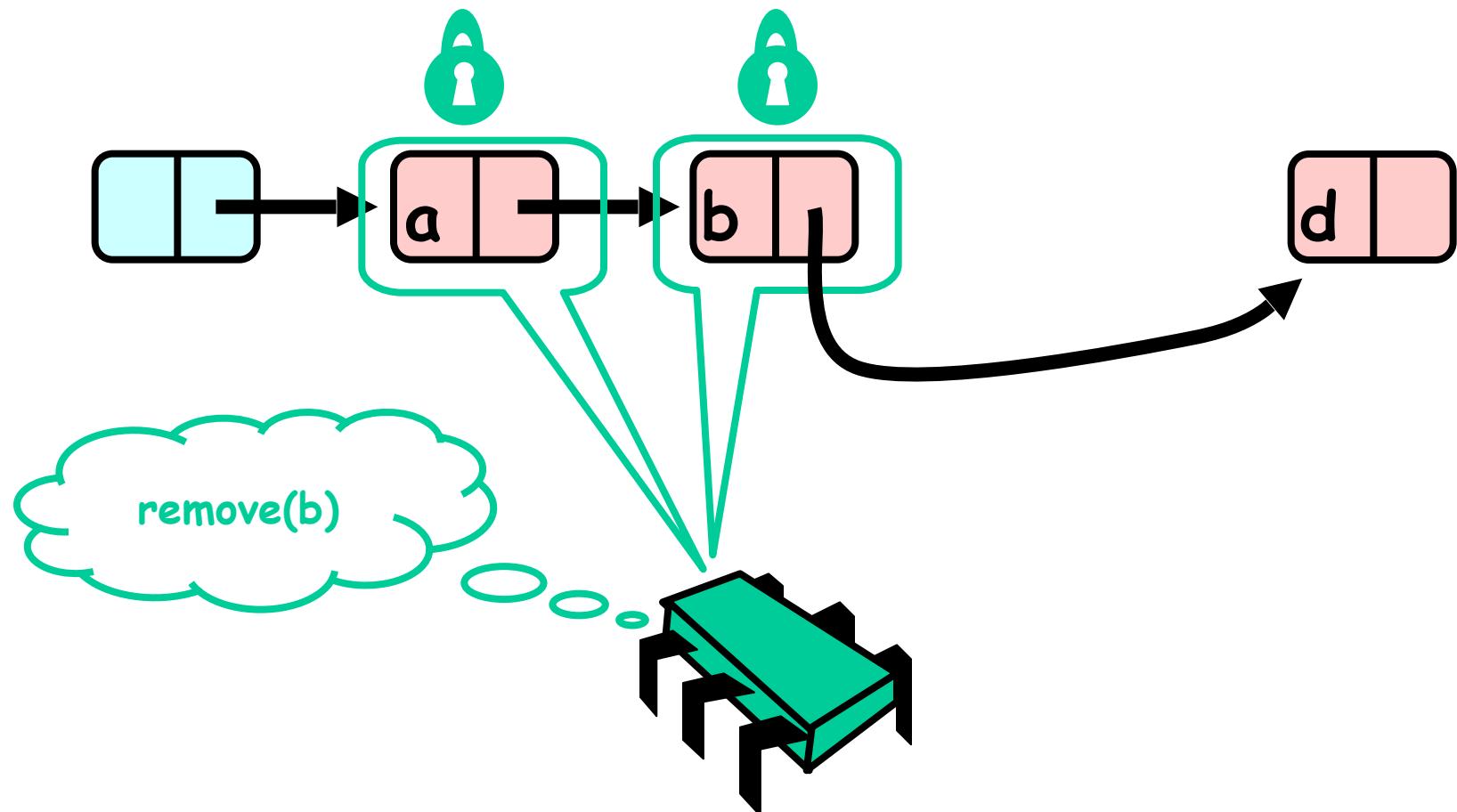
# Removing a Node



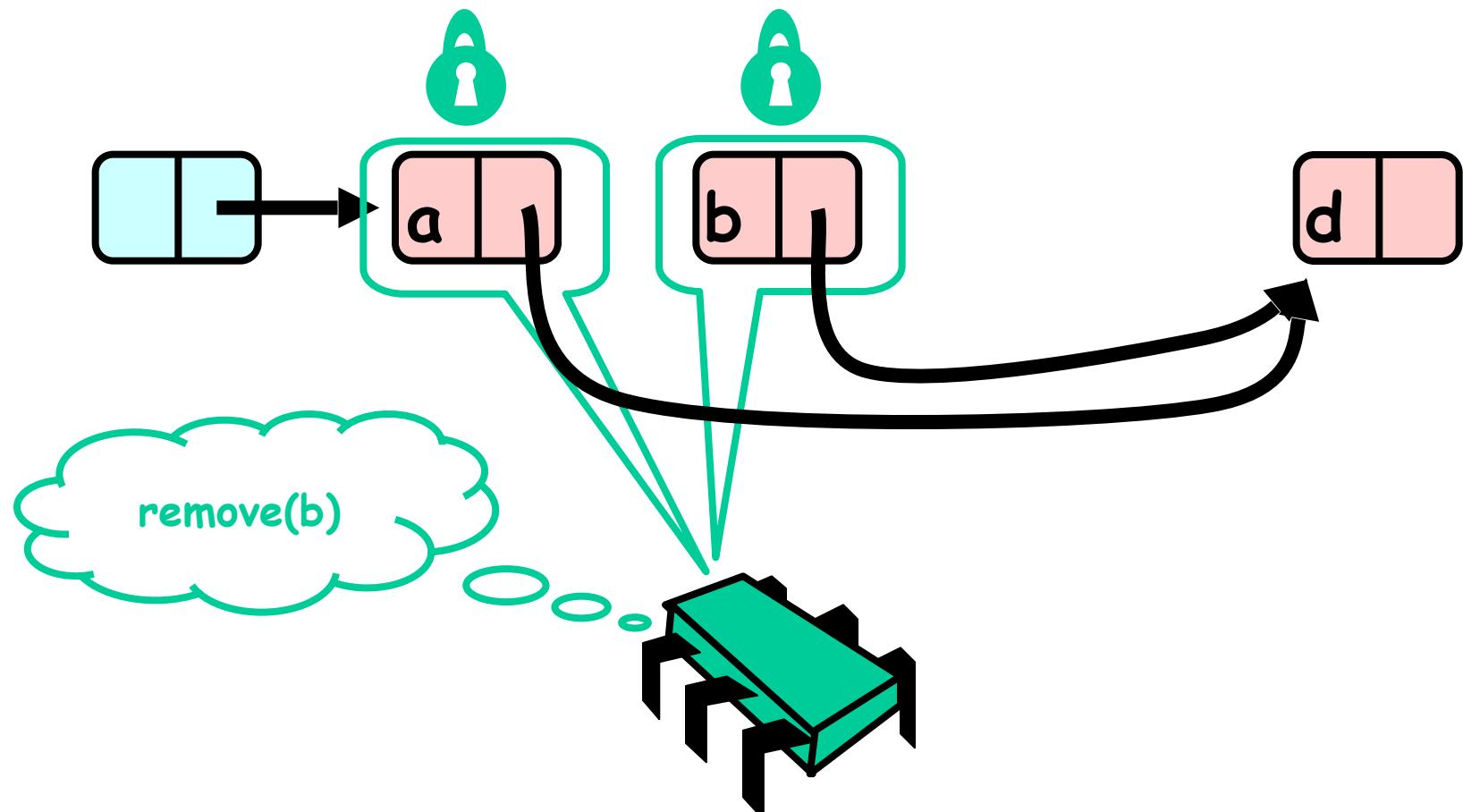
# Removing a Node



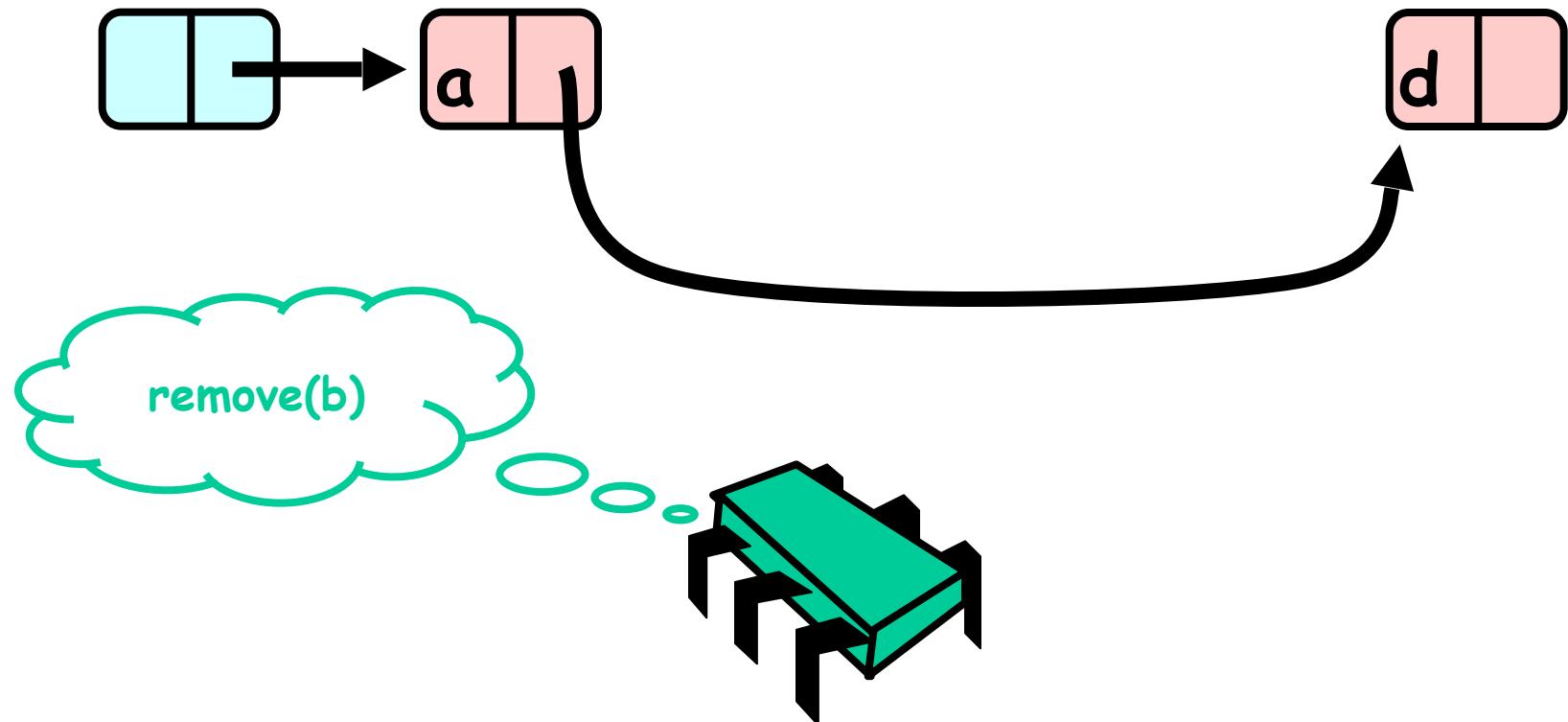
# Removing a Node



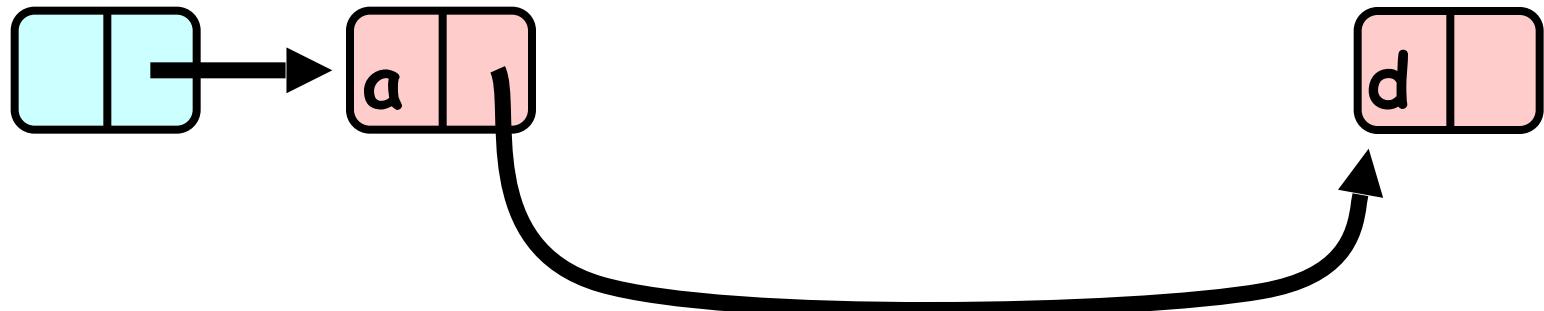
# Removing a Node



# Removing a Node



# Removing a Node



```
public boolean add(T item) {  
    int key = item.hashCode();  
    head.lock();  
    Node pred = head;  
    try {  
        Node curr = pred.next;  
        curr.lock();  
        try {  
            while (curr.key < key) {  
                pred.unlock();  
                pred = curr;  
                curr = curr.next;  
                curr.lock();  
            }  
            if (curr.key == key) return false;  
            Node newNode = new Node(item);  
            newNode.next = curr;  
            pred.next = newNode;  
            return true;  
        } finally {  
            curr.unlock();  
        }  
    } finally {  
        pred.unlock();  
    }  
}
```

## Fine-Grained Synchronization: hand-over-hand locking Linked List

```
public boolean remove(T item) {  
    Node pred = null, curr = null;  
    int key = item.hashCode();  
    head.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        curr.lock();  
        try {  
            while (curr.key < key) {  
                pred.unlock();  
                pred = curr;  
                curr = curr.next;  
                curr.lock();  
            }  
            if (curr.key == key) {  
                pred.next = curr.next;  
                return true;  
            }  
            return false;  
        } finally {  
            curr.unlock();  
        }  
    } finally {  
        pred.unlock();  
    }  
}
```

```
public boolean contains(T item) {  
    Node last = null, pred = null, curr  
    = null;  
    int key = item.hashCode();  
    head.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        curr.lock();  
        try {  
            while (curr.key < key) {  
                pred.unlock();  
                pred = curr;  
                curr = curr.next;  
                curr.lock();  
            }  
            return (curr.key == key);  
        } finally {  
            curr.unlock();  
        }  
    } finally {  
        pred.unlock();  
    }  
}
```

# Adding Nodes

- To add node  $e$ 
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted

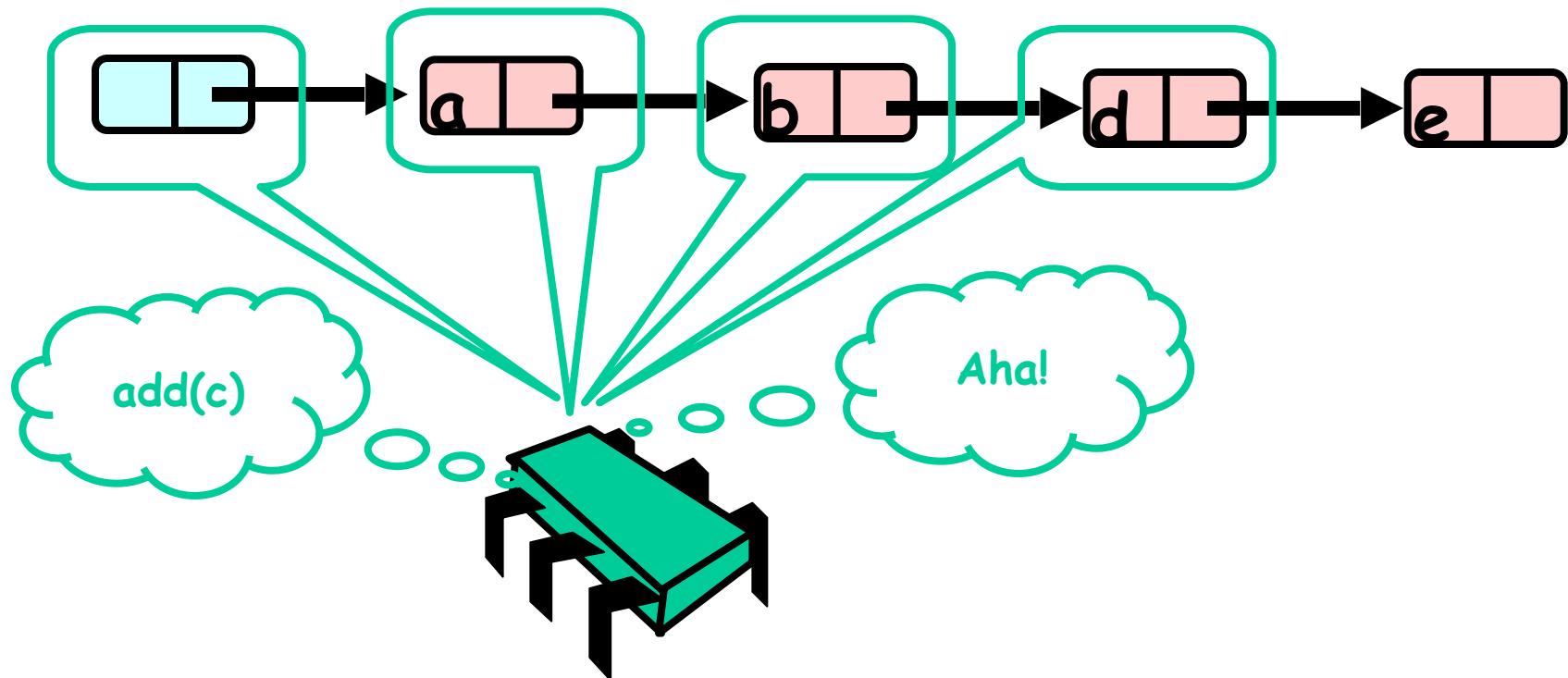
# Drawbacks

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

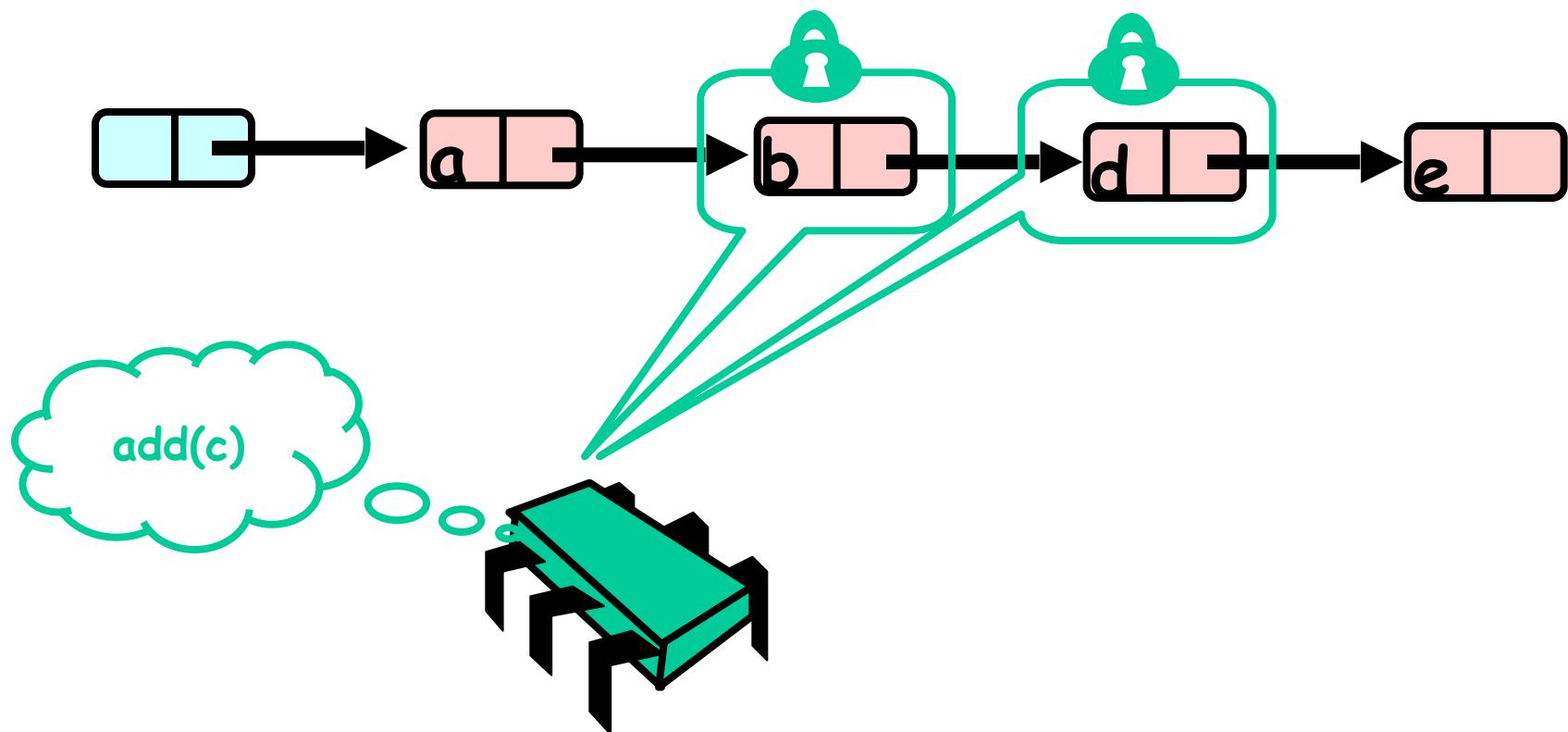
# Optimistic Synchronization

- Find nodes without locking
- Lock nodes
- Check that everything is OK

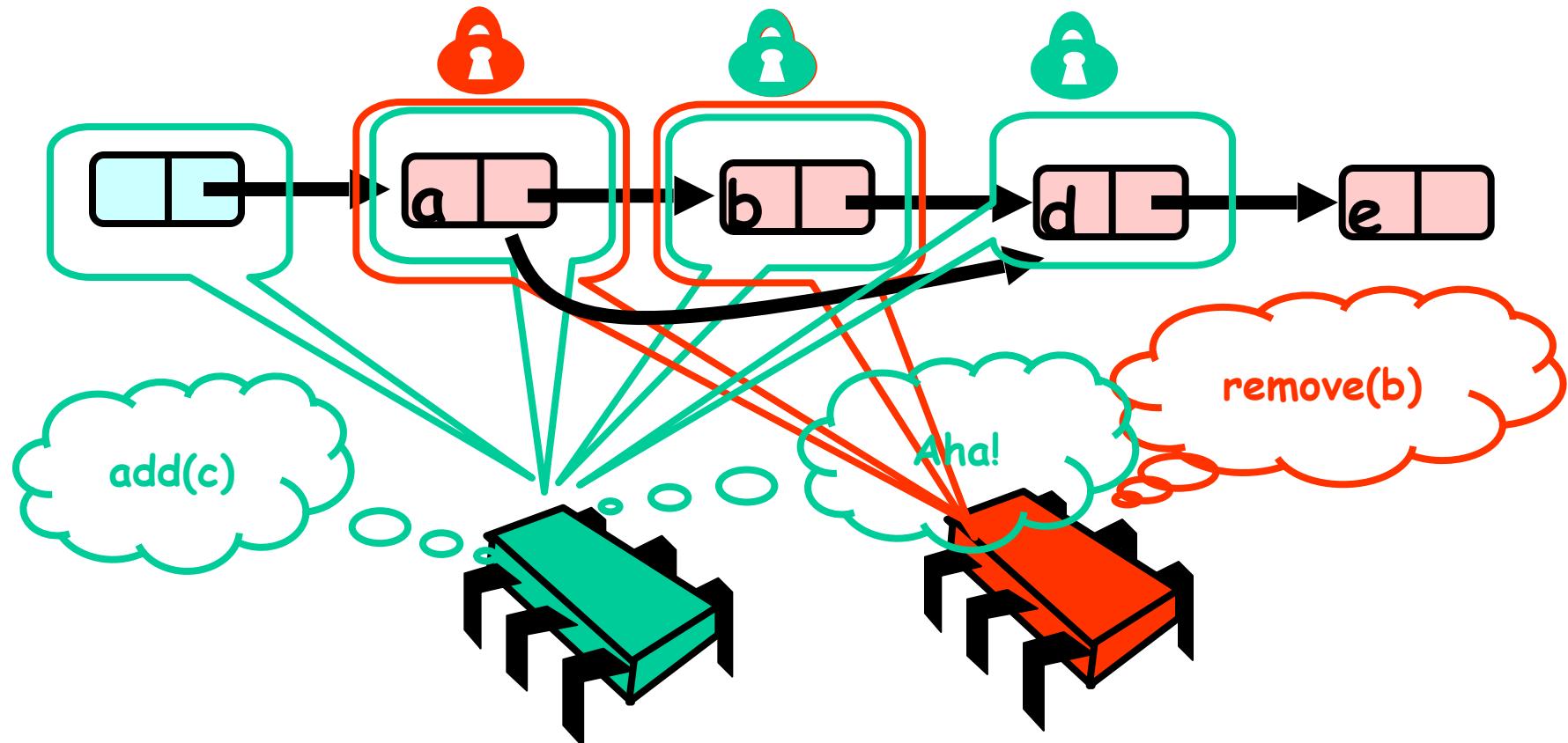
# Optimistic: Traverse without Locking



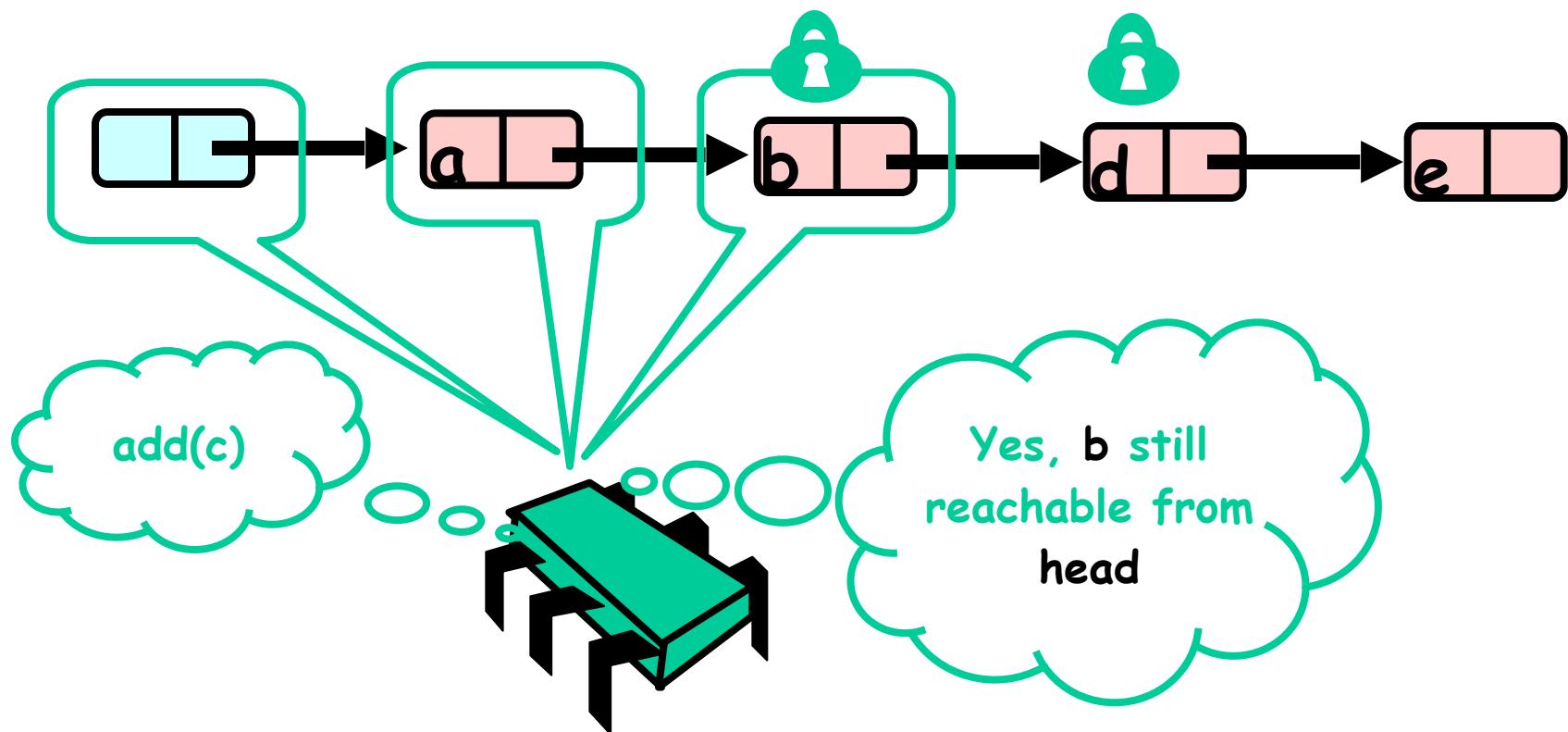
# Optimistic: Lock and Load



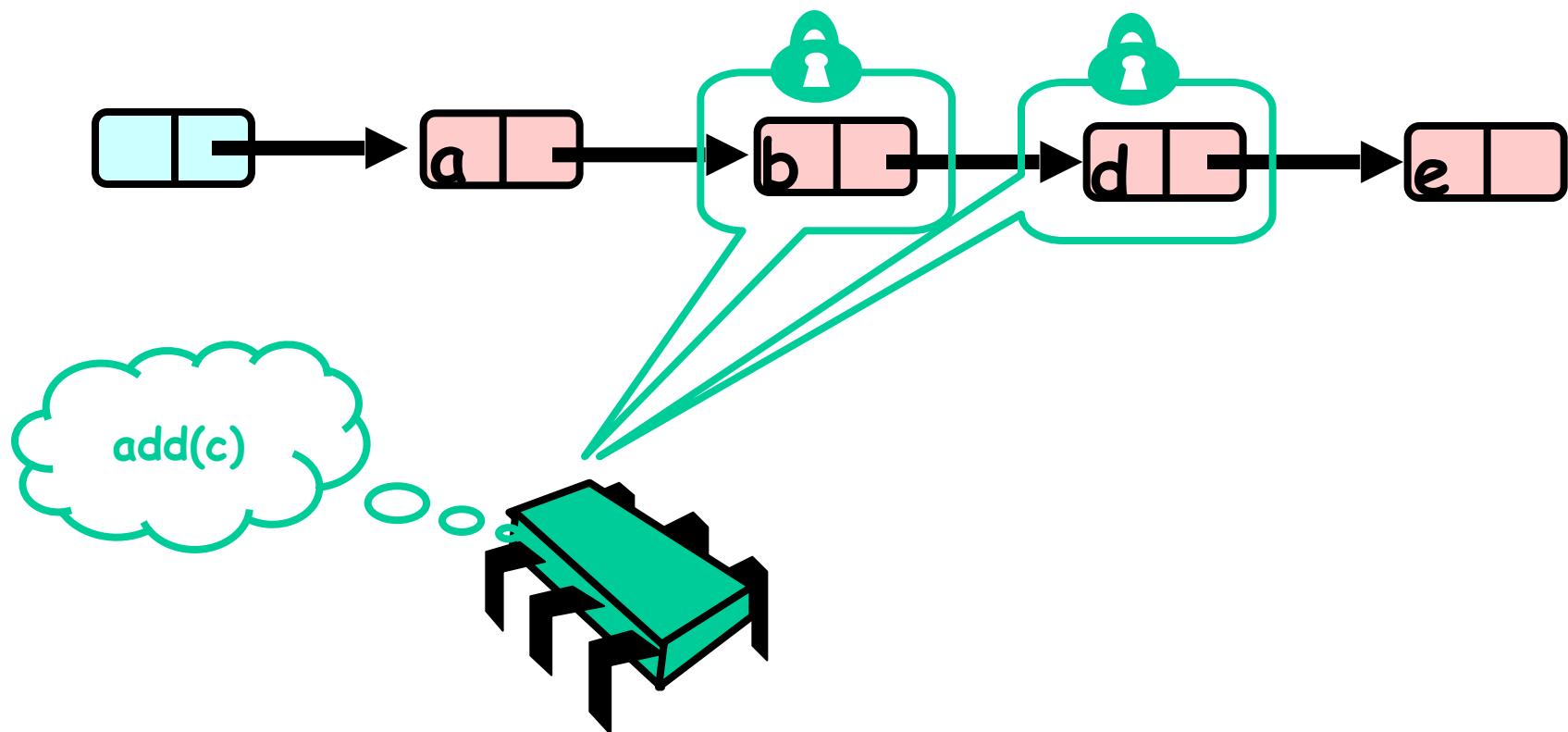
# What could go wrong?



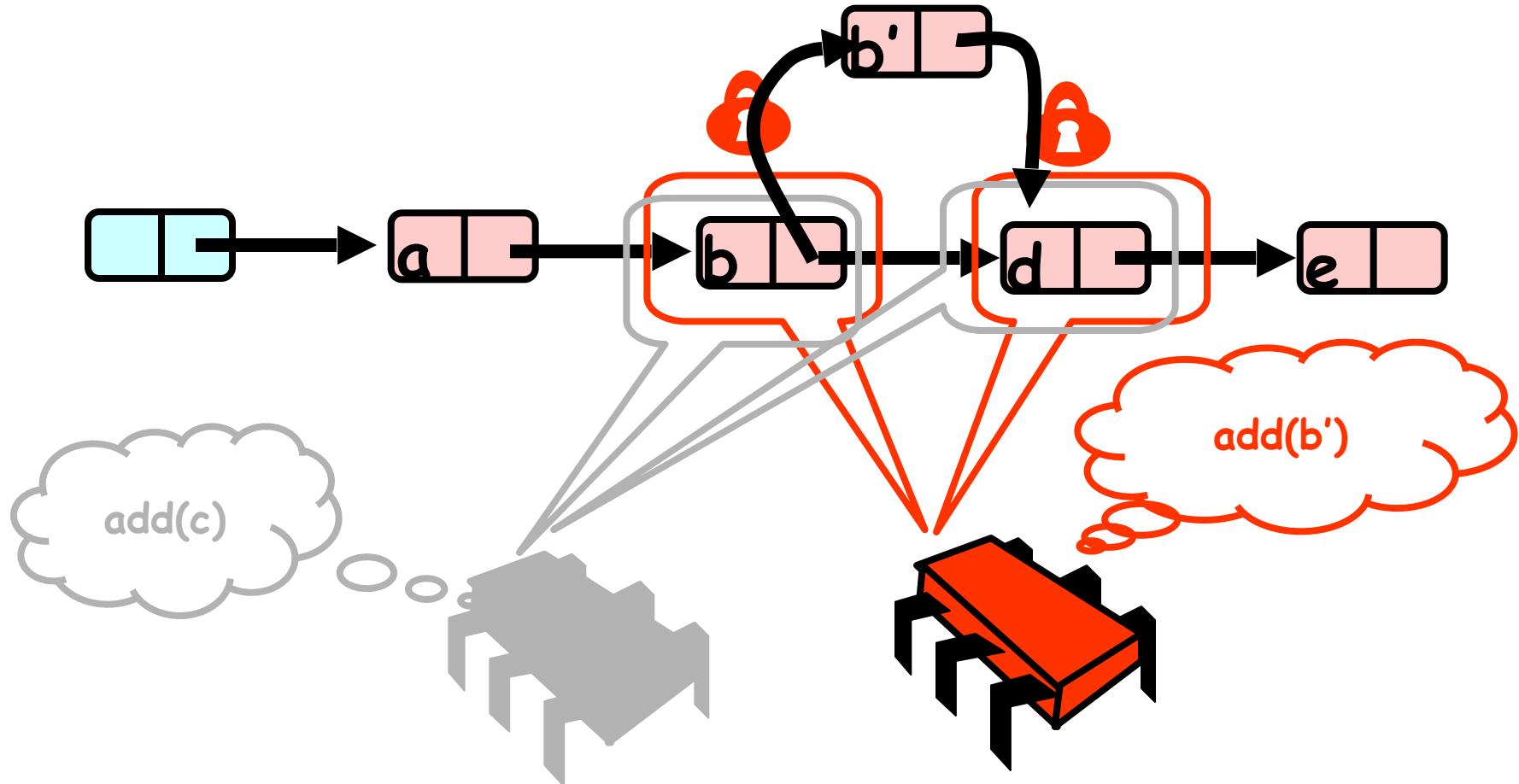
# Validate - Part 1 (while holding locks)



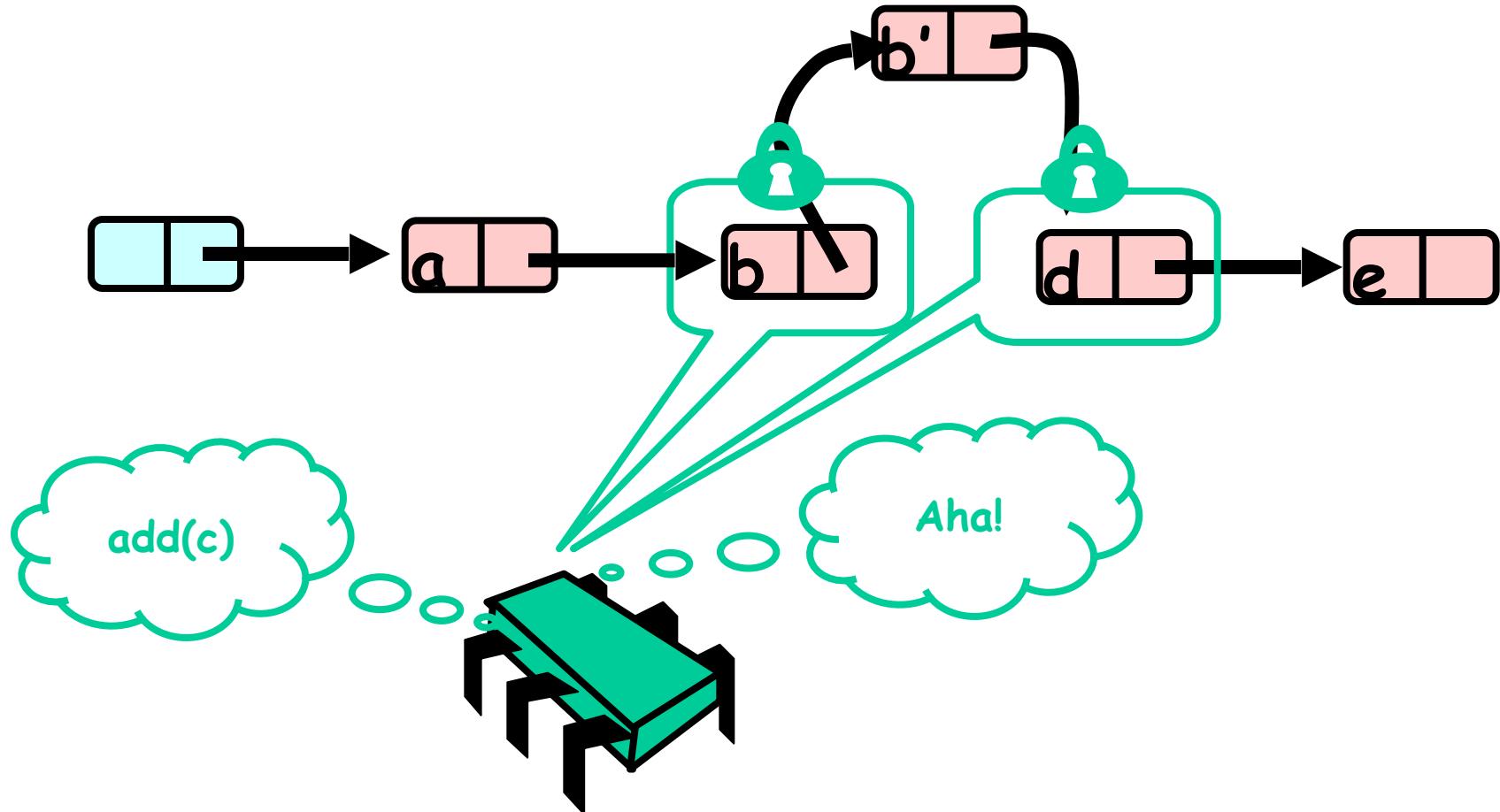
# What Else Can Go Wrong?



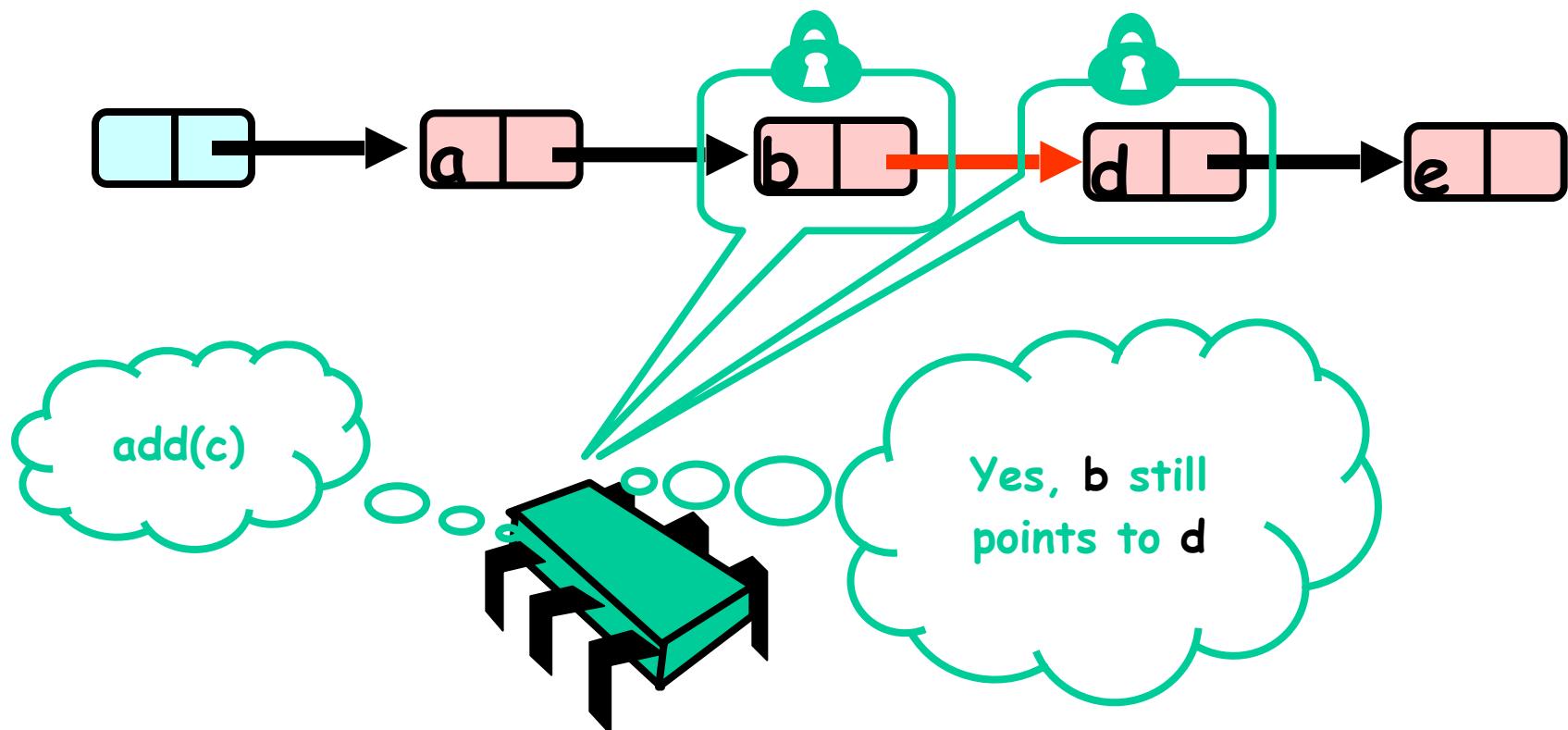
# What Else Can Go Wrong?



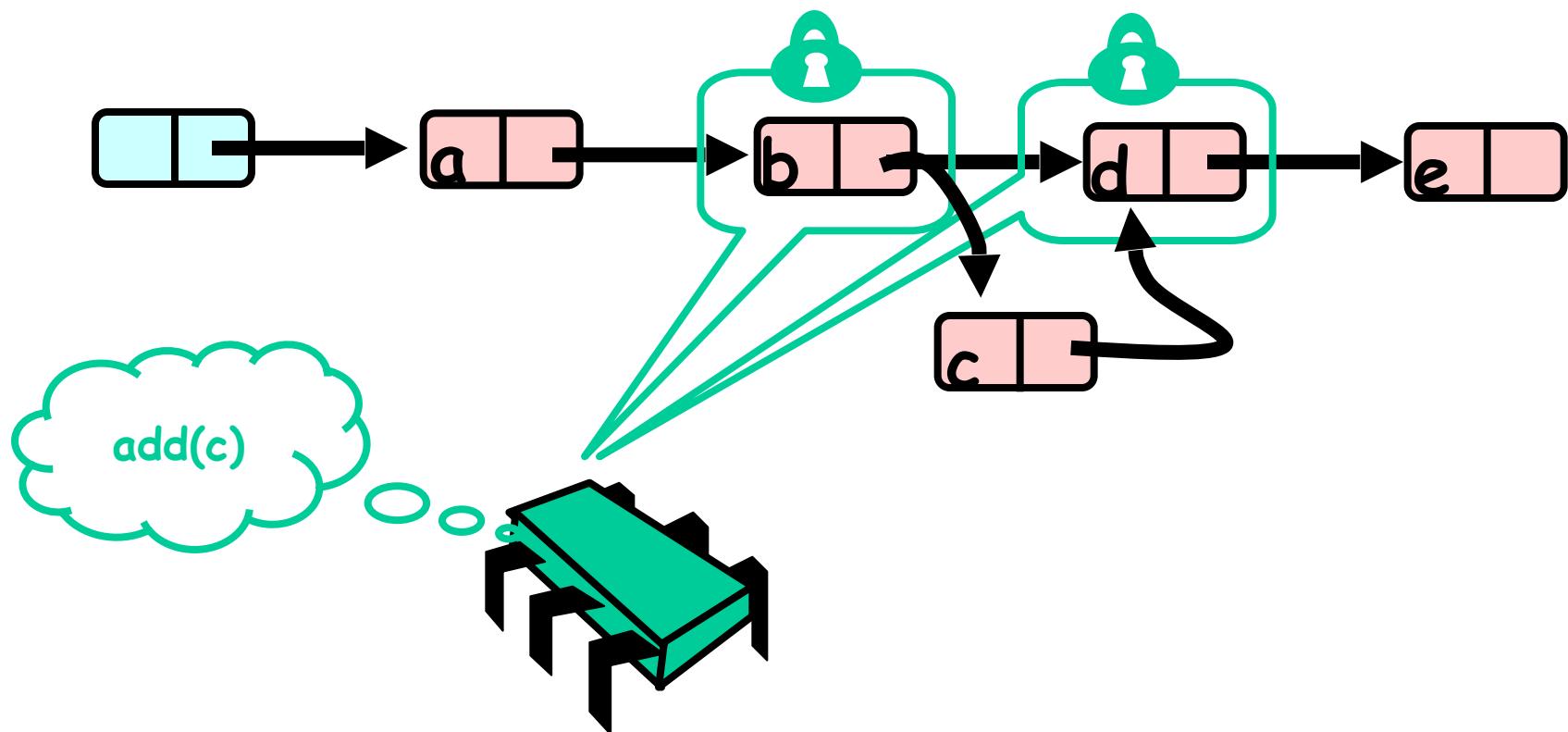
# What Else Can Go Wrong?



# Validate Part 2 (while holding locks)



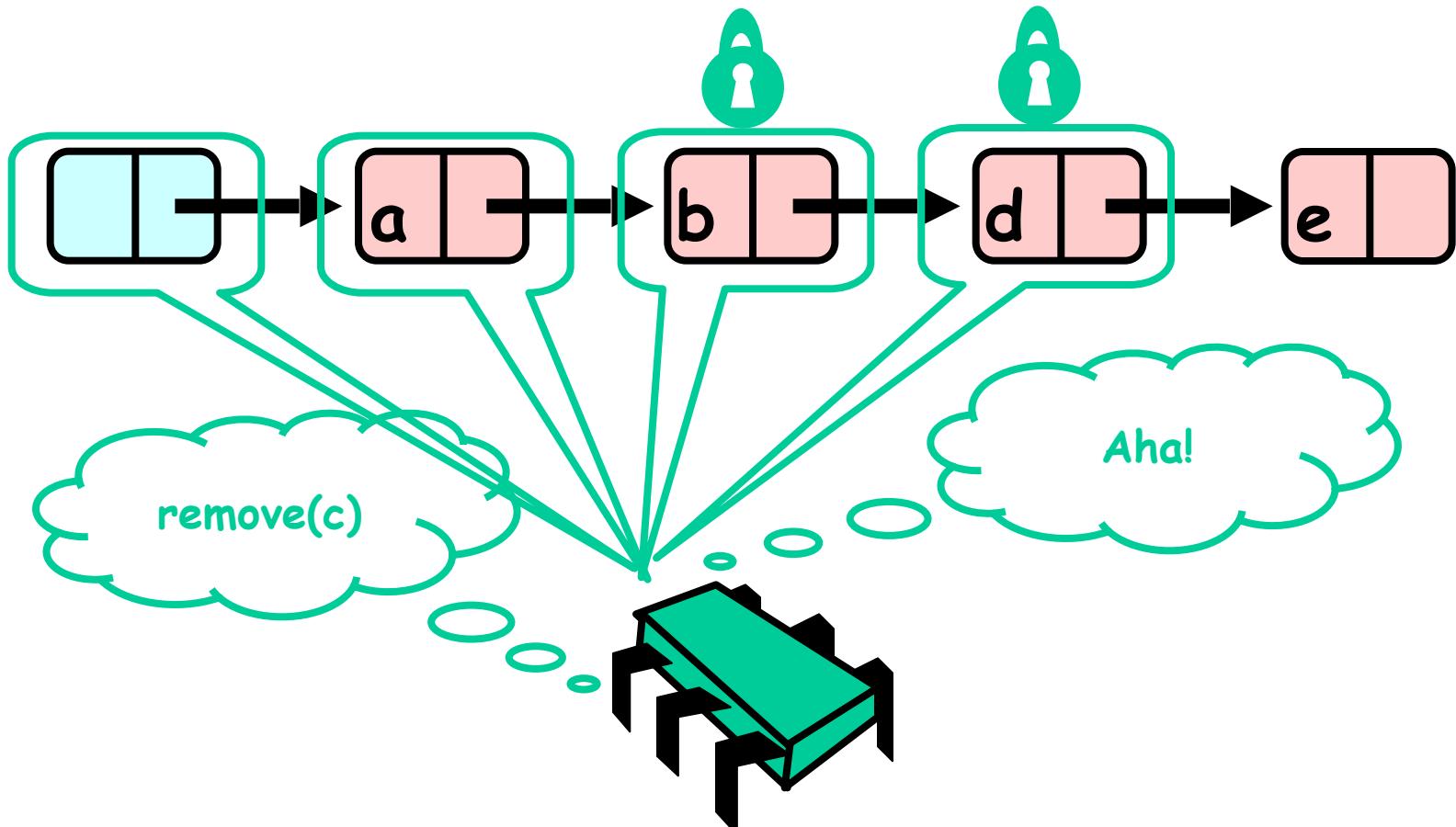
# Optimistic: Critical Point



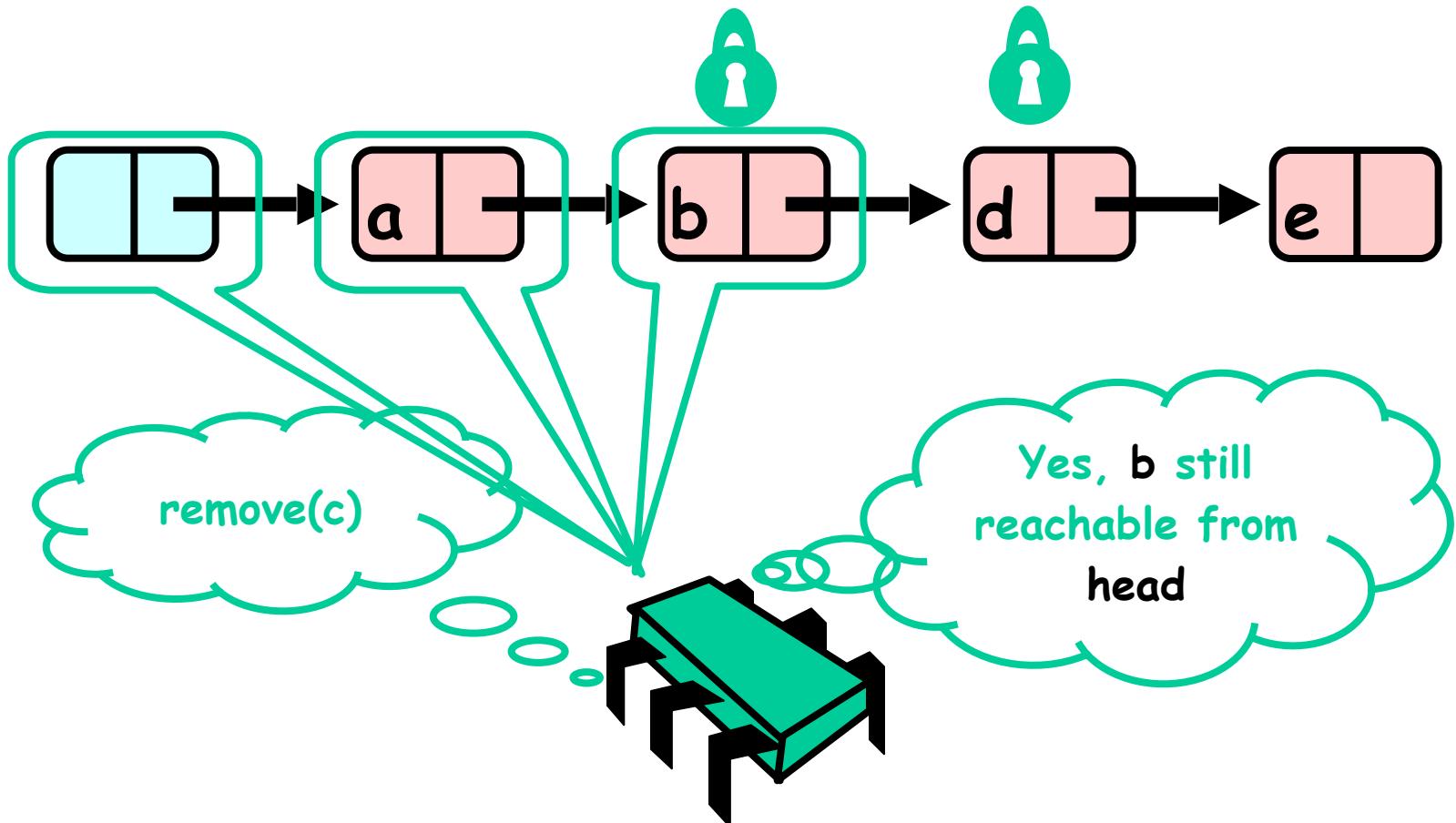
# Correctness

- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return true

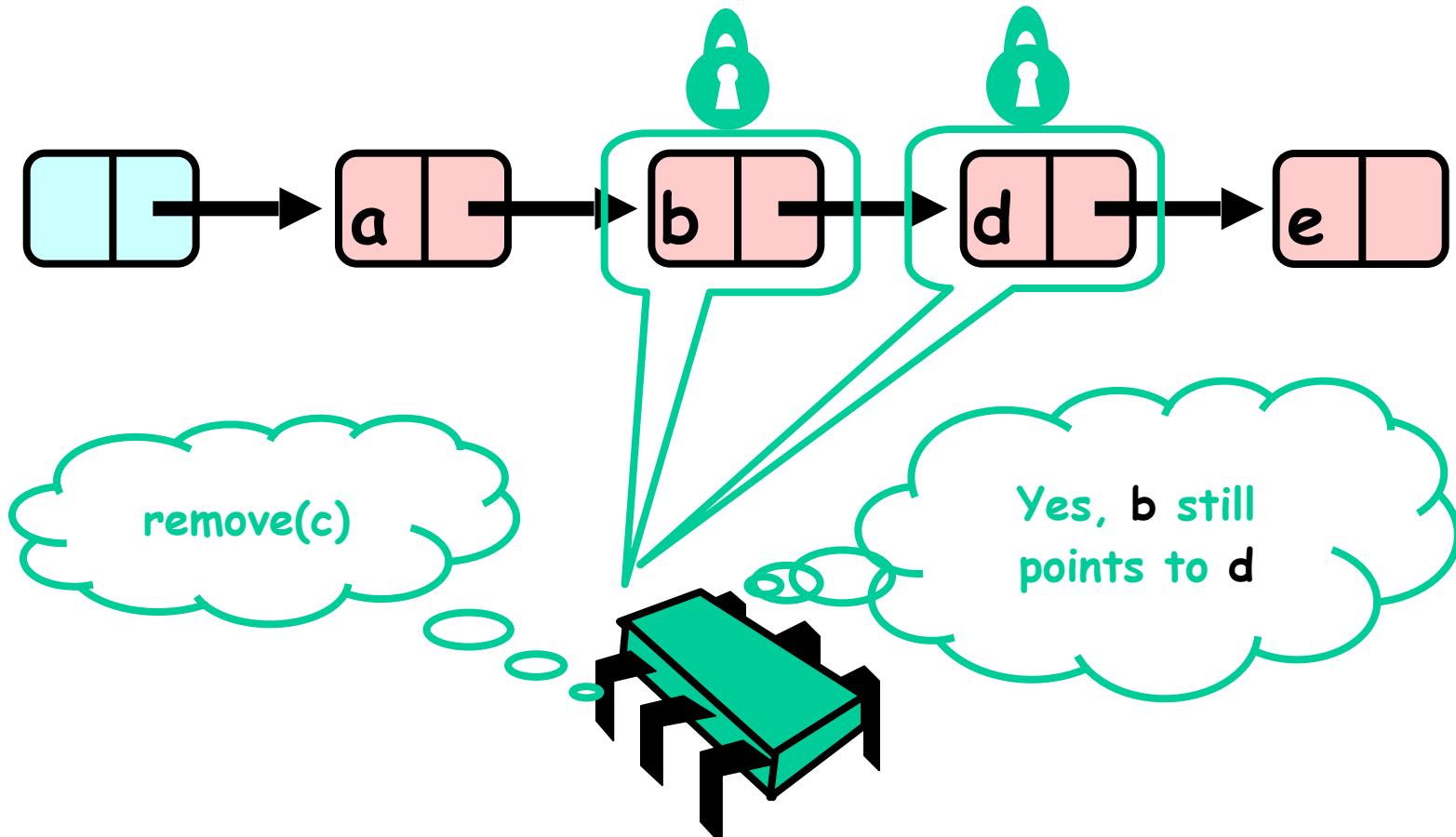
# Unsuccessful Remove



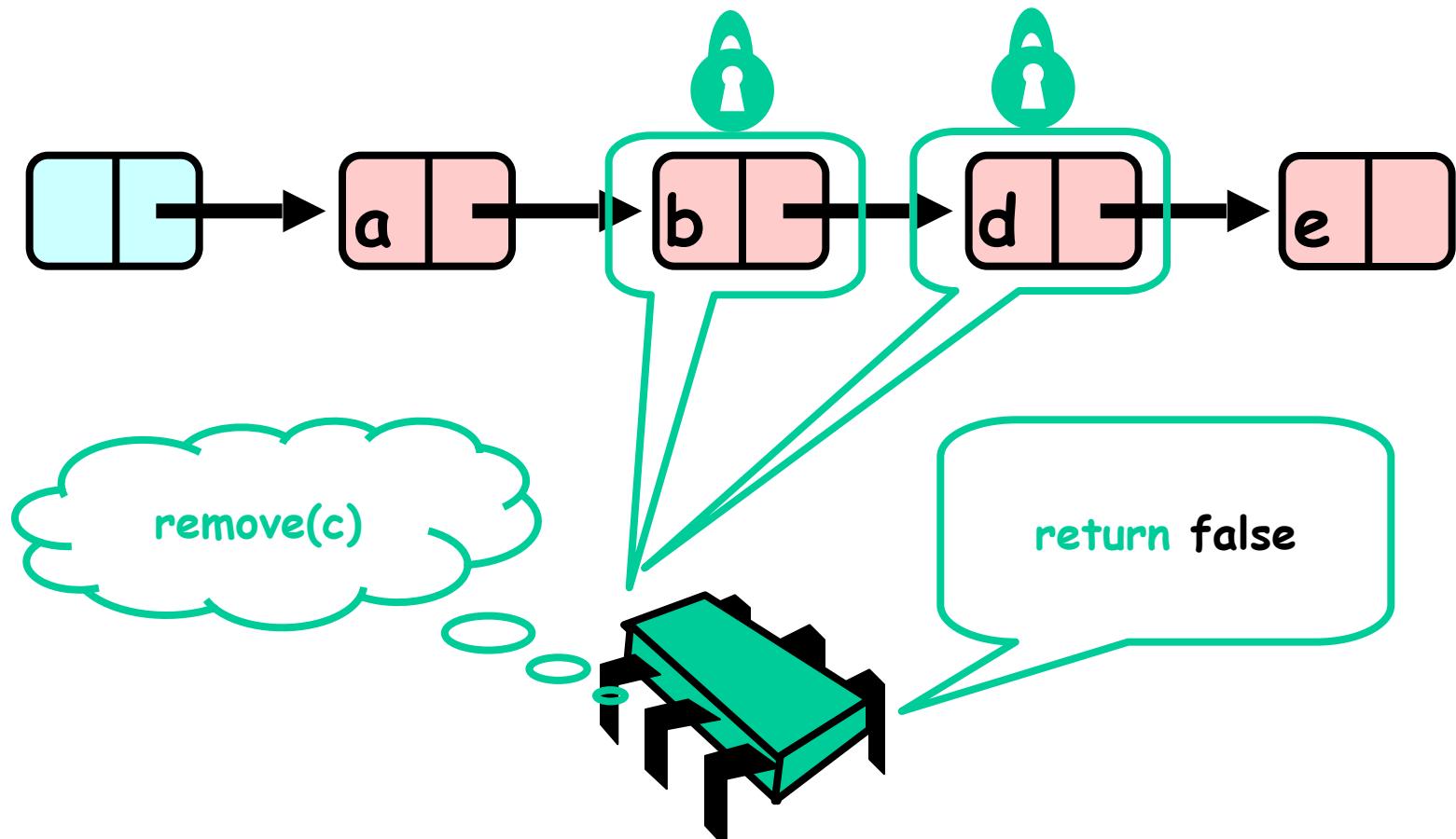
# Validate (1)



# Validate (2)



# OK Computer



# Correctness

- If
  - Nodes b and d both locked
  - Node b still accessible
  - Node d still successor to b
- Then
  - Neither will be deleted
  - No thread can add c after b
  - OK to return false

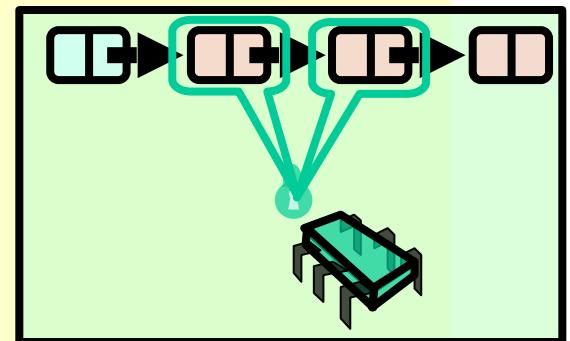
# Validation

```
private boolean  
validate(Node pred,  
        Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

# Validation

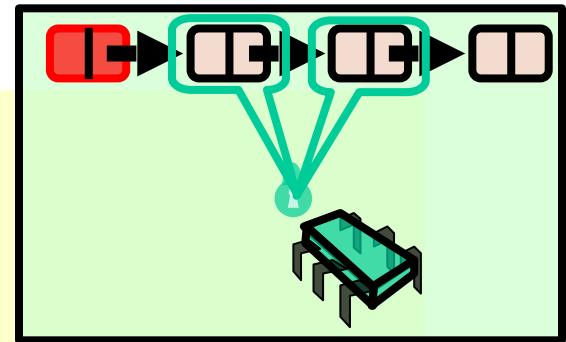
```
private boolean  
validate(Node pred,  
        Node curr) {  
  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

**Predecessor &  
current nodes**



# Validation

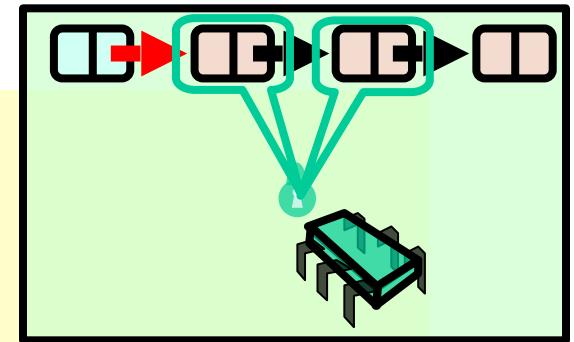
```
private boolean  
validate(Node pred,  
        Node curr) {  
Node node = head;  
while (node.key <= pred.key) {  
    if (node == pred)  
        return pred.next == curr;  
    node = node.next;  
}  
return false;  
}
```



Begin at the  
beginning

# Validation

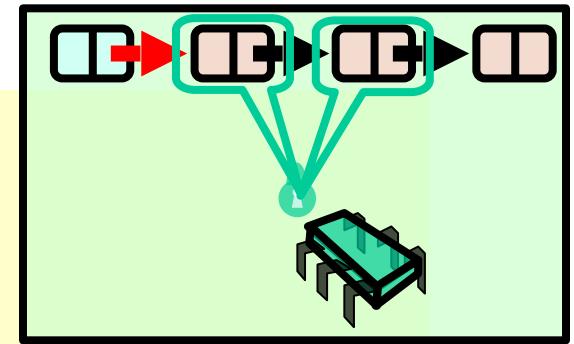
```
private boolean  
validate(Node pred,  
        Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```



Search range of keys

# Validation

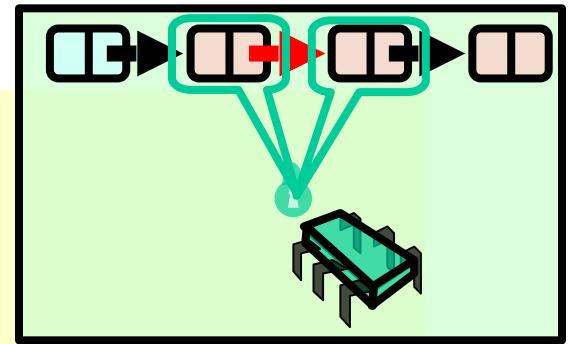
```
private boolean  
validate(Node pred,  
        Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```



Predecessor reachable

# Validation

```
private boolean  
validate(Node pred,  
        Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

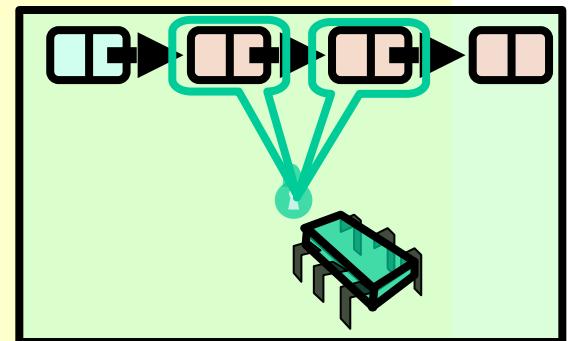


Is current node next?

# Validation

```
private boolean  
validate(Node pred,  
        Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

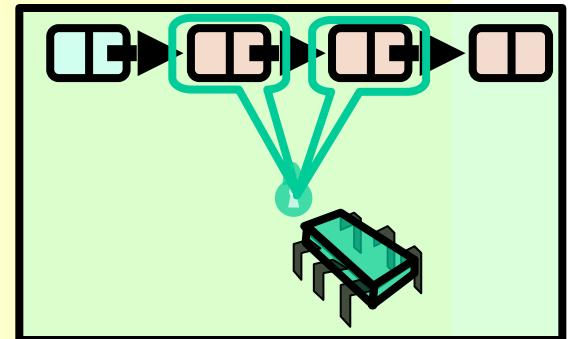
Otherwise move on



# Validation

```
private boolean  
validate(Node pred,  
        Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
return false;  
}
```

**Predecessor not reachable**



```
public boolean add(T item) {  
    int key = item.hashCode();  
    while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key < key) {  
            pred = curr; curr = curr.next;  
        }  
        pred.lock(); curr.lock();  
        try {  
            if (validate(pred, curr)) {  
                if (curr.key == key) {  
                    return false;  
                } else {  
                    Node node = new Entry(item);  
                    entry.next = curr;  
                    pred.next = node;  
                    return true;  
                }  
            }  
        } finally {  
            pred.unlock(); curr.unlock();  
        }  
    }  
}
```

## Optimistic Synchronization

```
public boolean remove(T item) {  
    int key = item.hashCode();  
    while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key < key) {  
            pred = curr; curr = curr.next;  
        }  
        pred.lock(); curr.lock();  
        try {  
            if (validate(pred, curr)) {  
                if (curr.key == key) {  
                    pred.next = curr.next;  
                    return true;  
                } else {  
                    return false;  
                }  
            }  
        } finally {  
            pred.unlock(); curr.unlock();  
        }  
    }  
}
```

```
public boolean contains(T item) {  
    int key = item.hashCode();  
    while (true) {  
        Node pred = this.head;  
        Node curr = pred.next;  
        while (curr.key < key) {  
            pred = curr; curr = curr.next;  
        }  
        try {  
            pred.lock(); curr.lock();  
            if (validate(pred, curr)) {  
                return (curr.key == key);  
            }  
        } finally {  
            pred.unlock(); curr.unlock();  
        }  
    }  
}
```

```
private boolean validate(Node pred, Node  
curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

# Optimistic List

- Limited hot-spots
  - Targets of `add()`, `remove()`, `contains()`
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought ...

# So Far, So Good

- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - `contains()` method acquires locks

# Evaluation

- Optimistic is effective if
  - cost of scanning twice without locks is less than
  - cost of scanning once with locks
- Drawback
  - contains() acquires locks
  - 90% of calls in many apps

# Lazy List

- Like optimistic, except
  - Scan once
  - `contains(x)` never locks ...
- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”

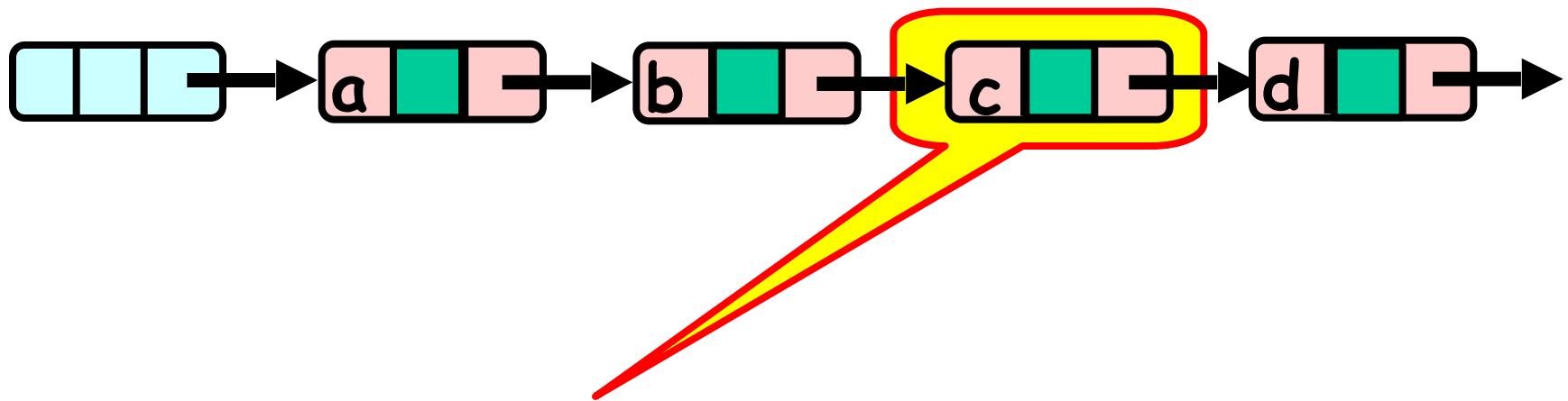
# Lazy List

- remove()
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

# Lazy Removal

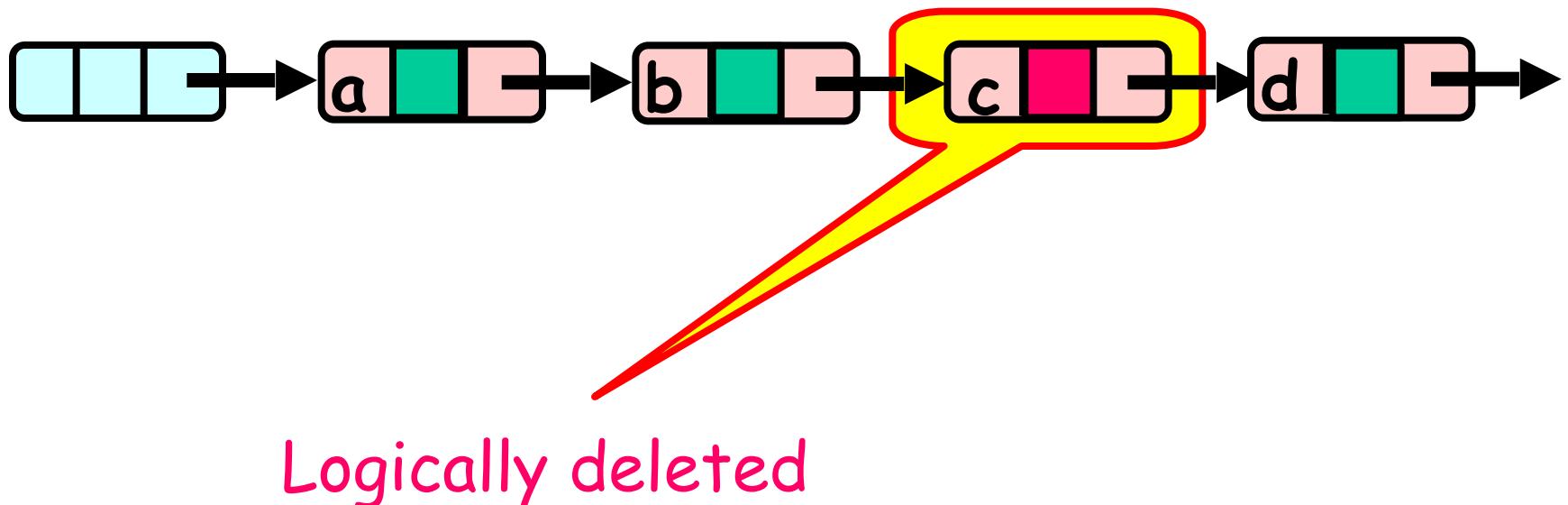


# Lazy Removal

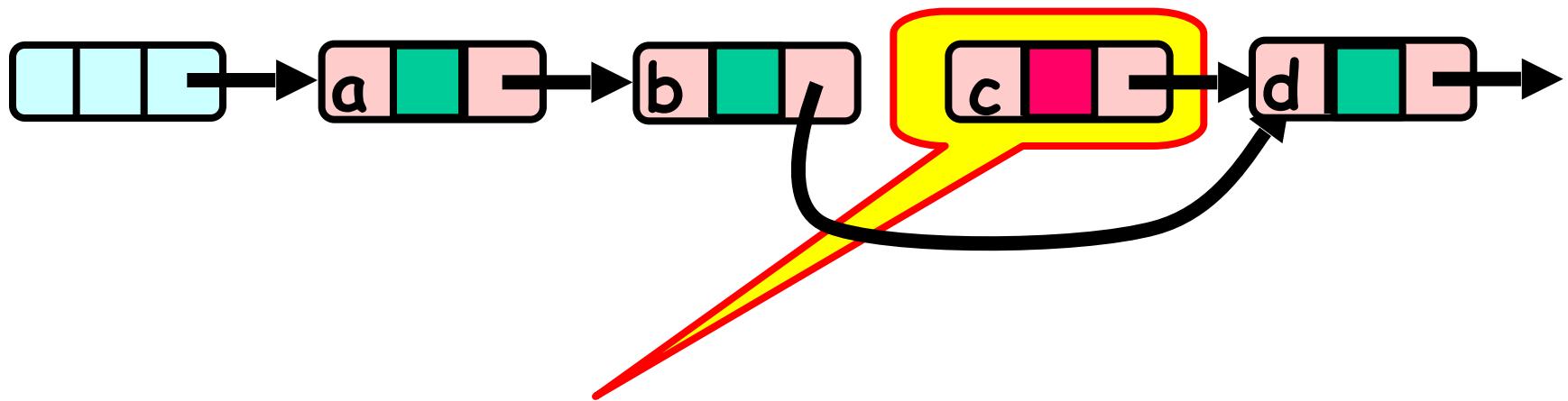


Present in list

# Lazy Removal

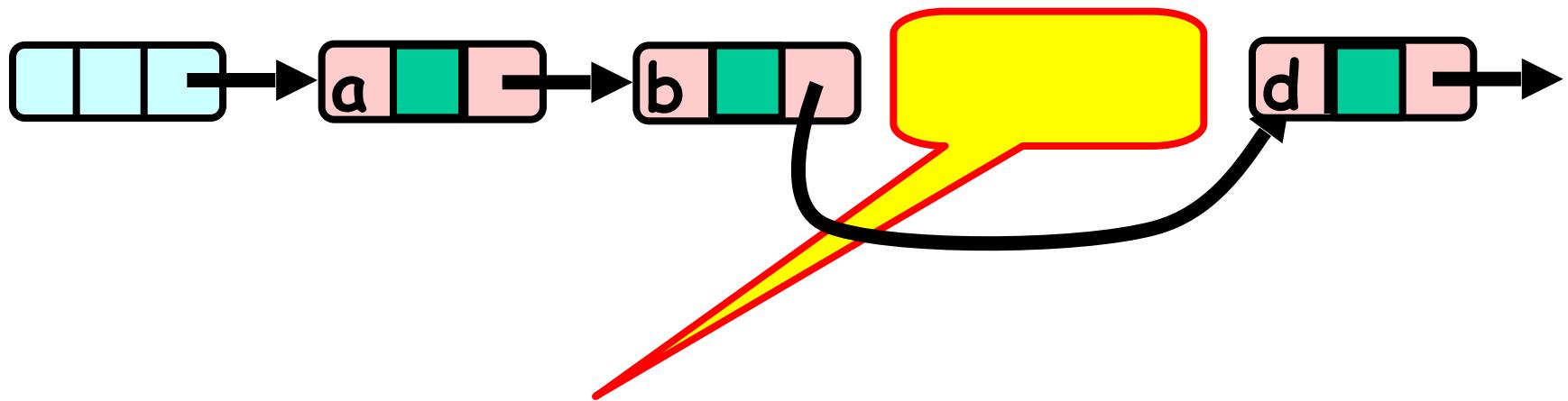


# Lazy Removal



Physically deleted

# Lazy Removal



Physically deleted

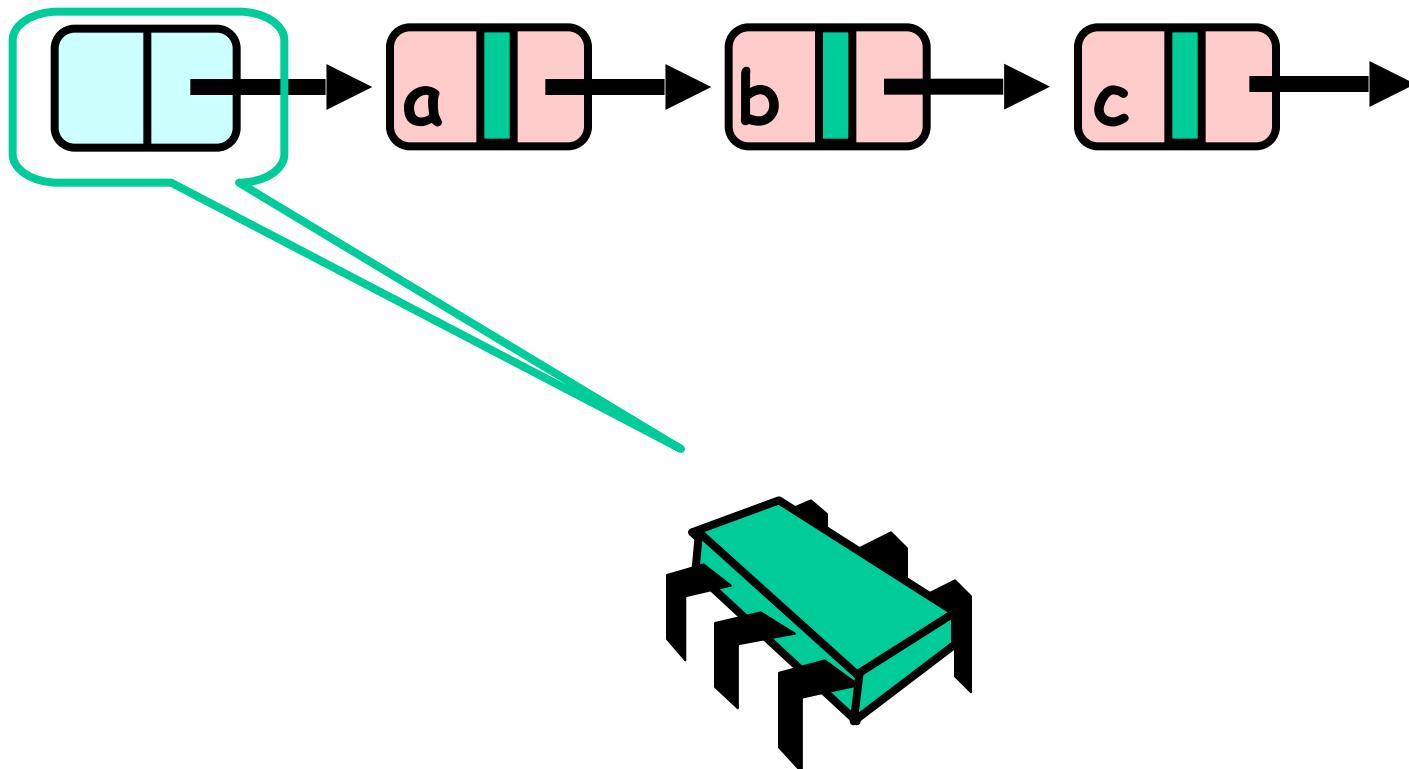
# Lazy List

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls ...
- Must still lock pred and curr nodes.

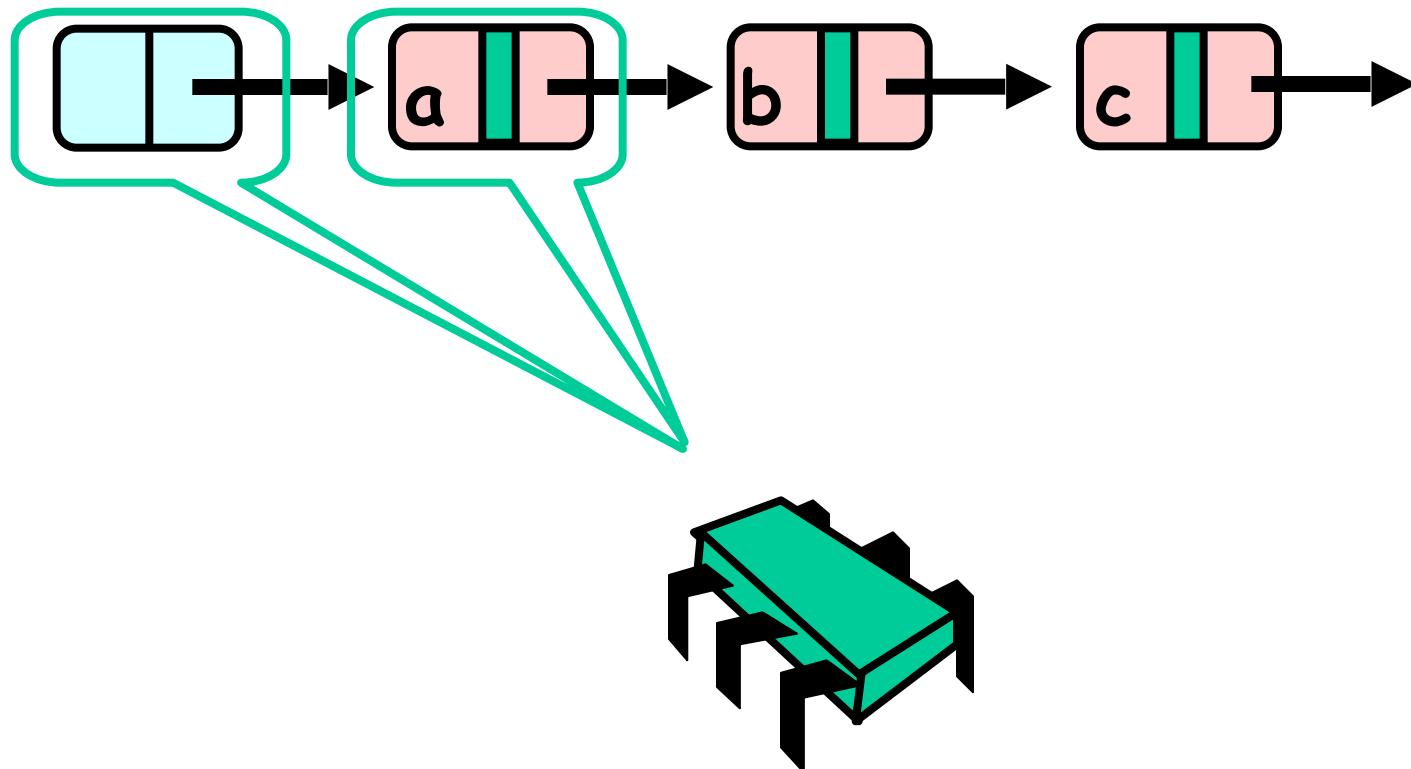
# Validation

- No need to rescan list!
- Check that pred is not marked
- Check that curr is not marked
- Check that pred points to curr

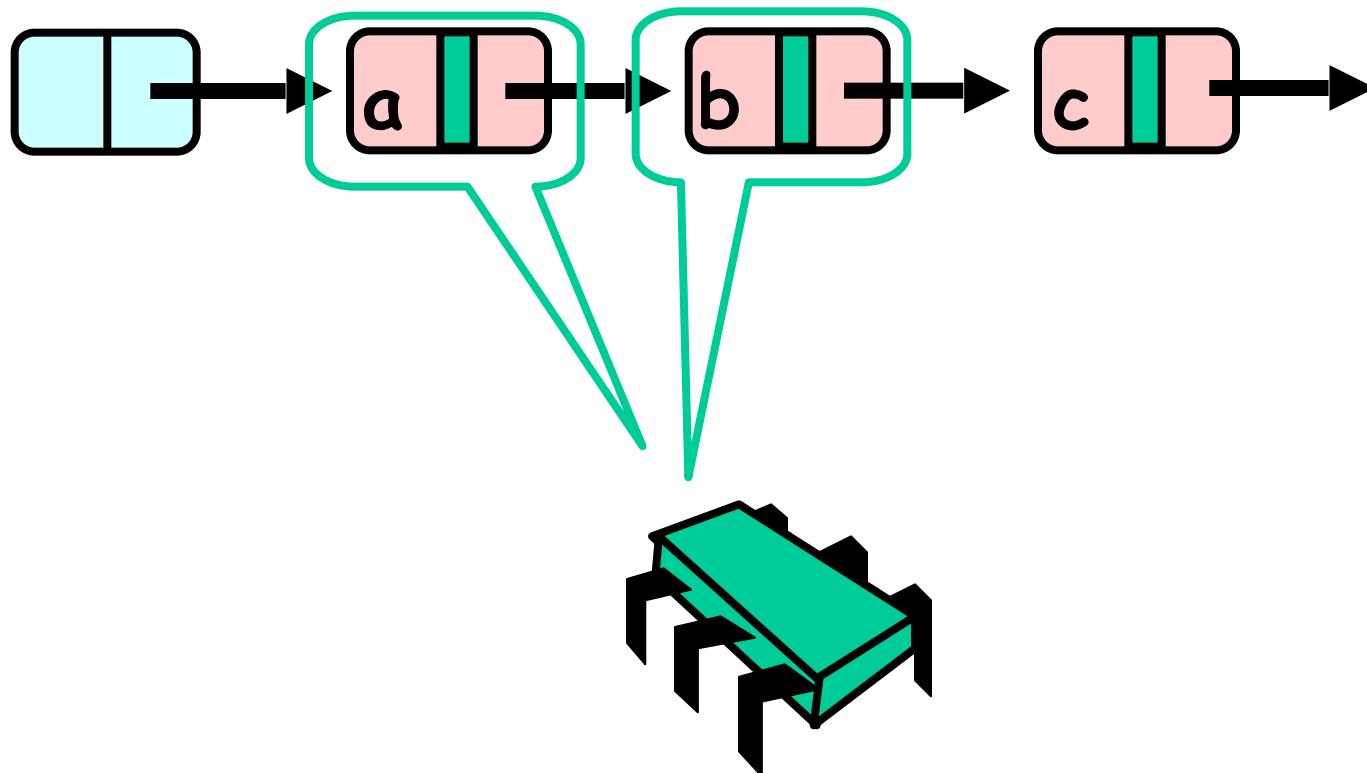
# Business as Usual



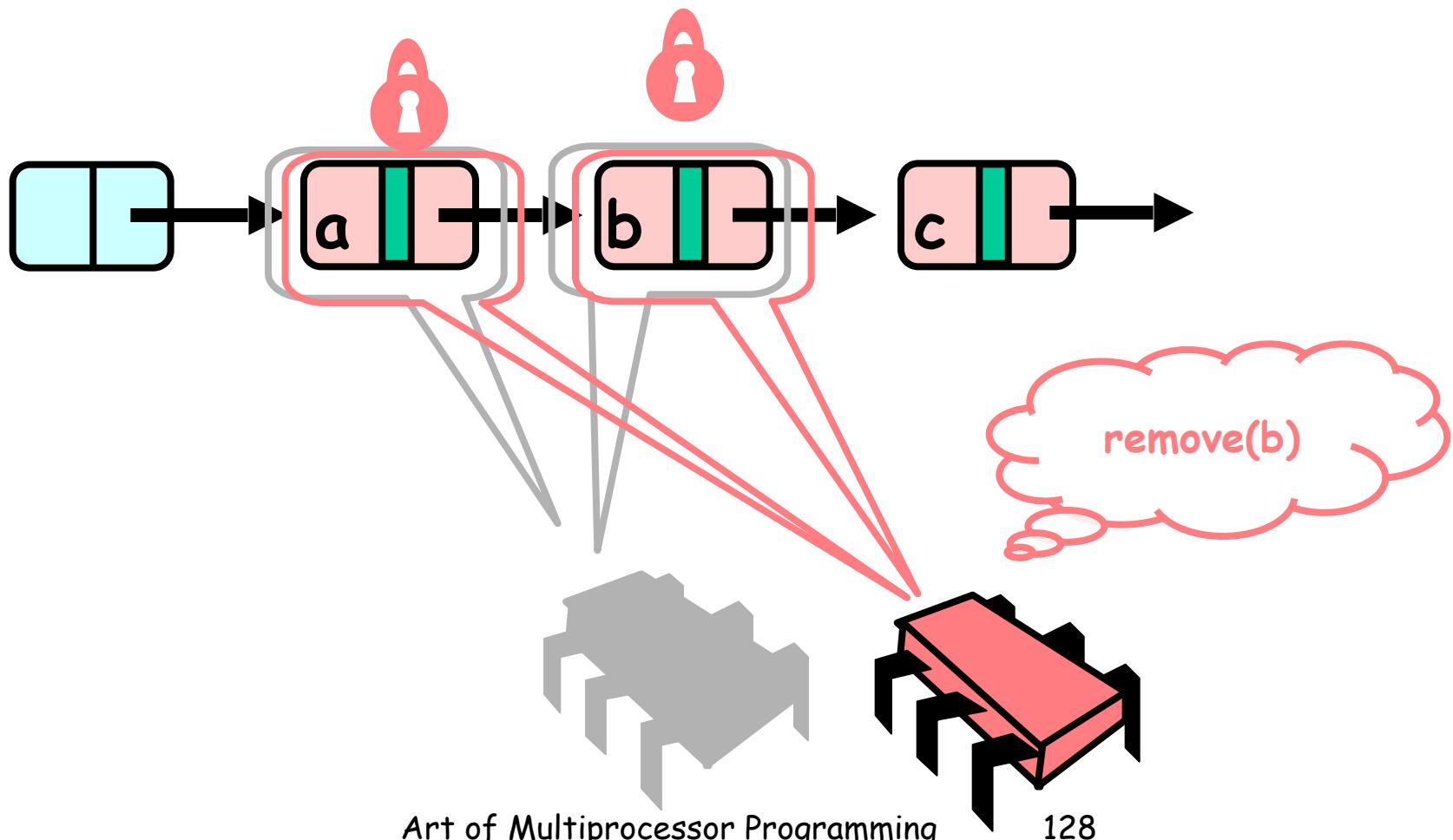
# Business as Usual



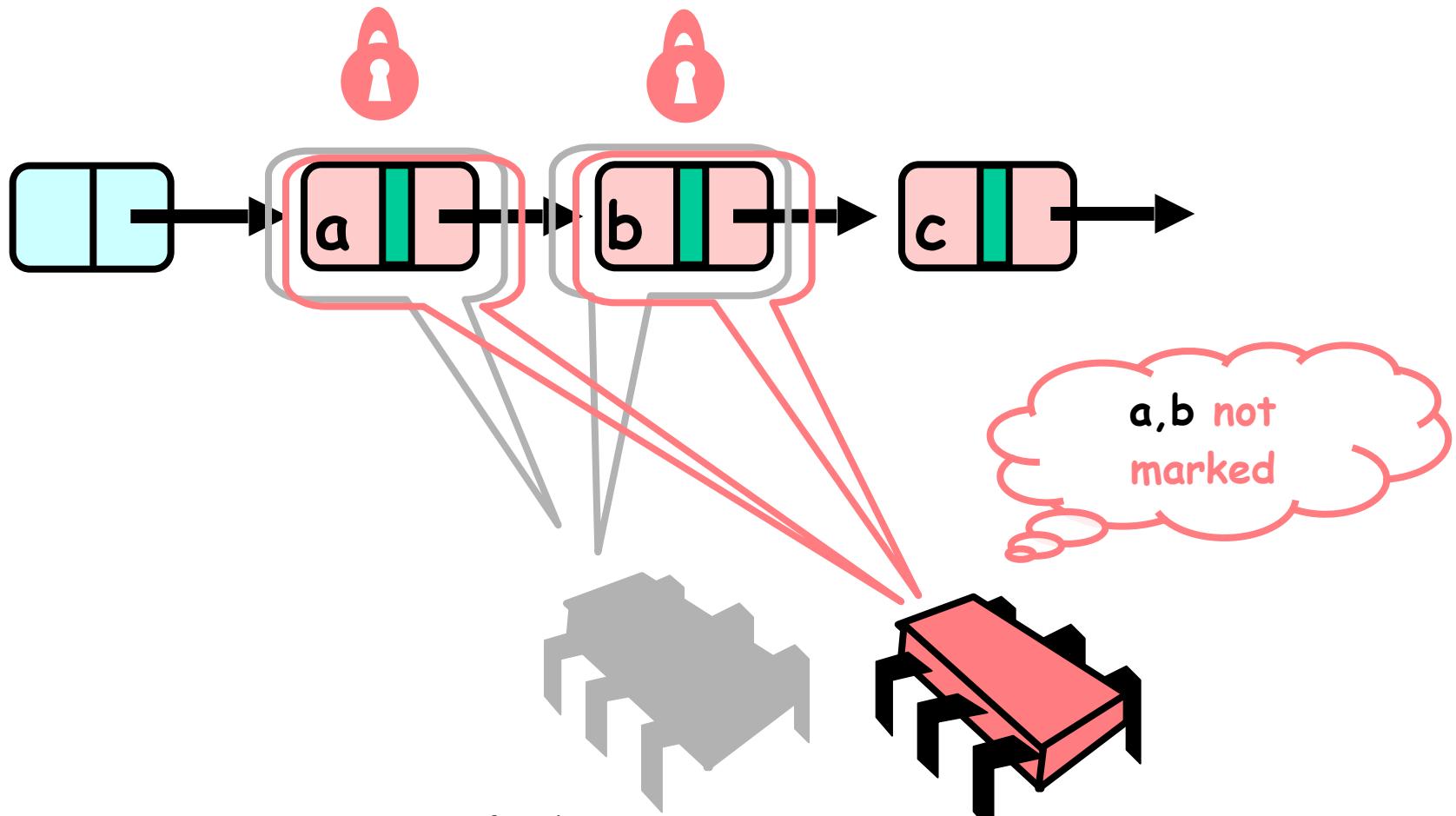
# Business as Usual



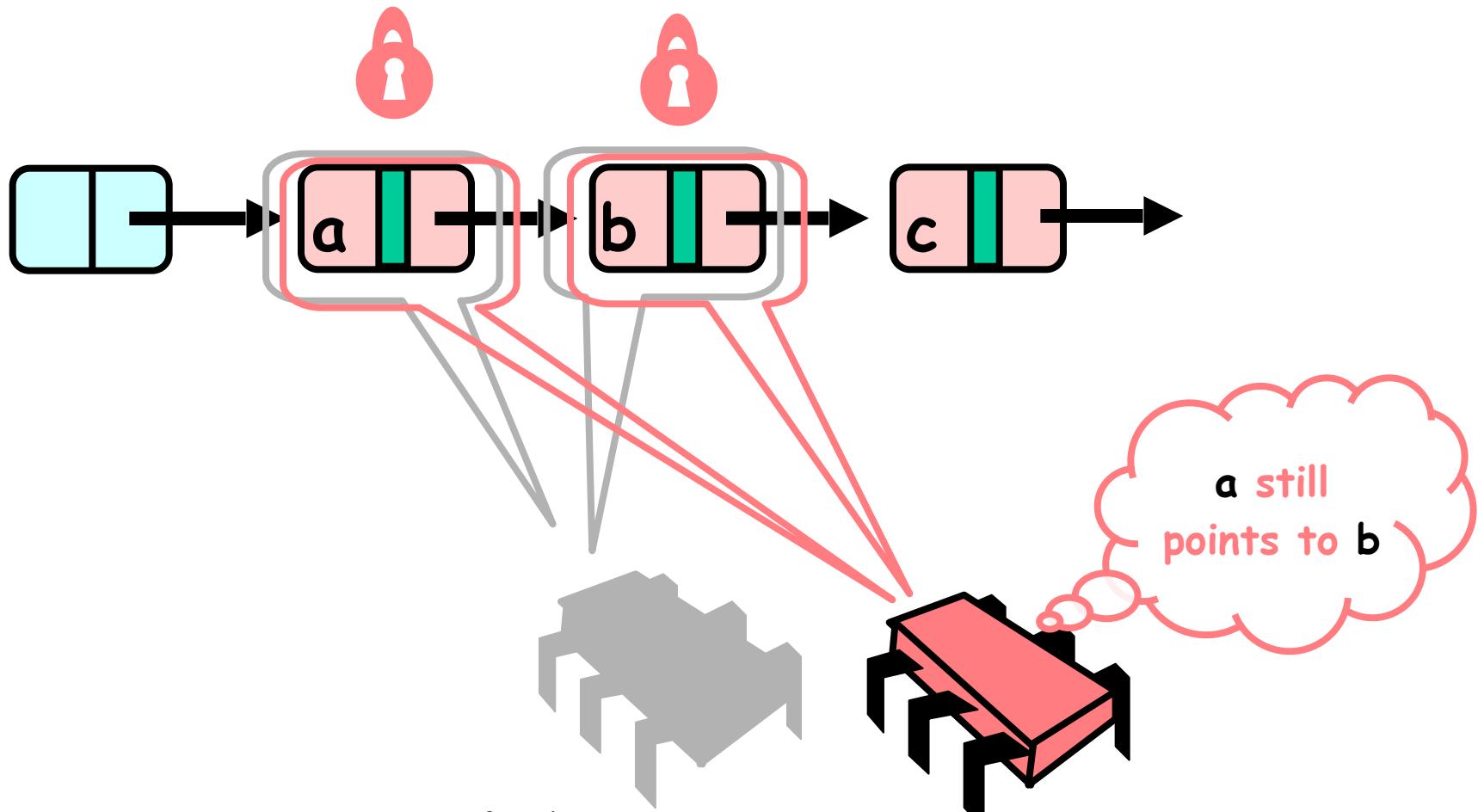
# Business as Usual



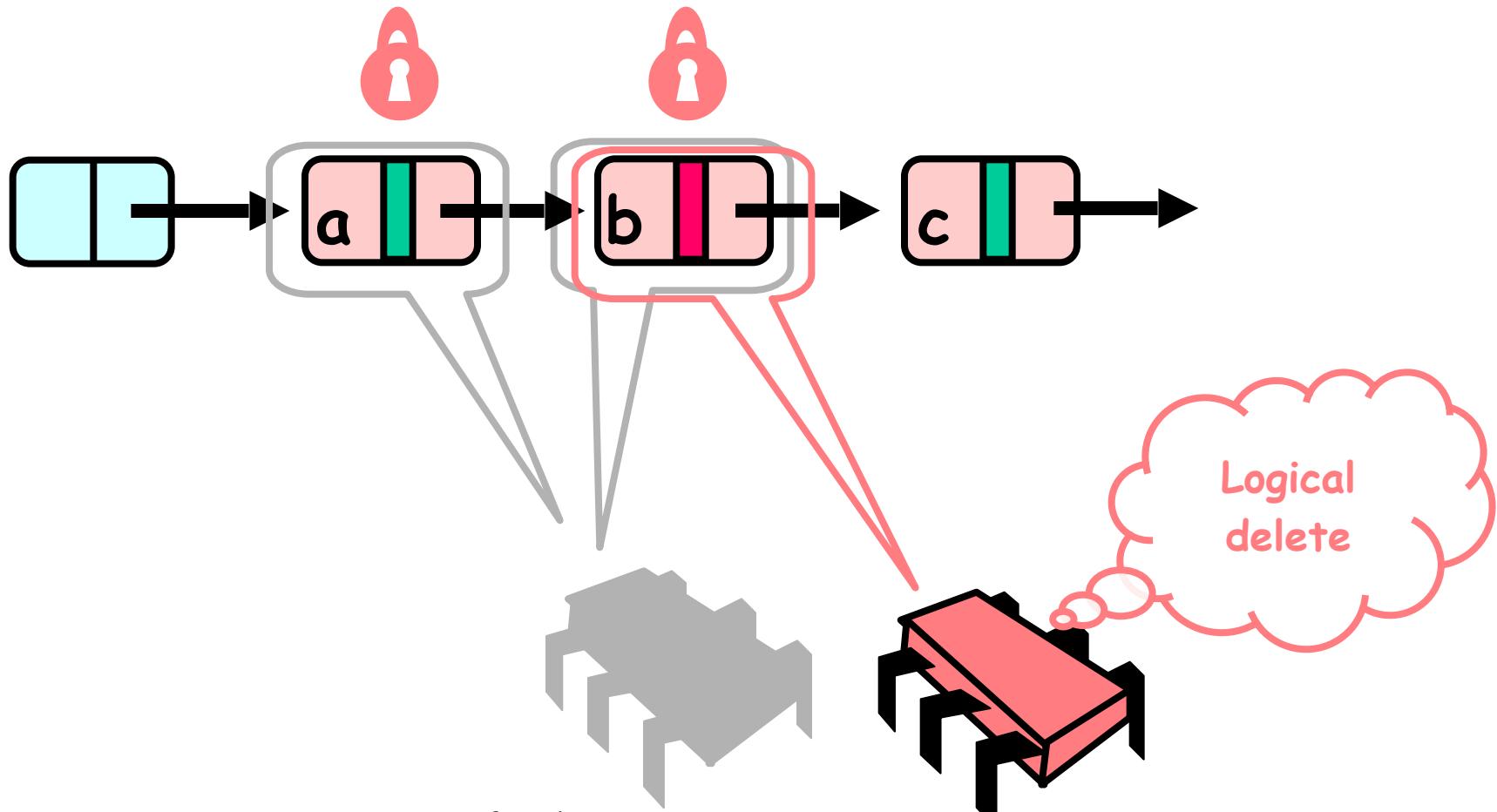
# Business as Usual



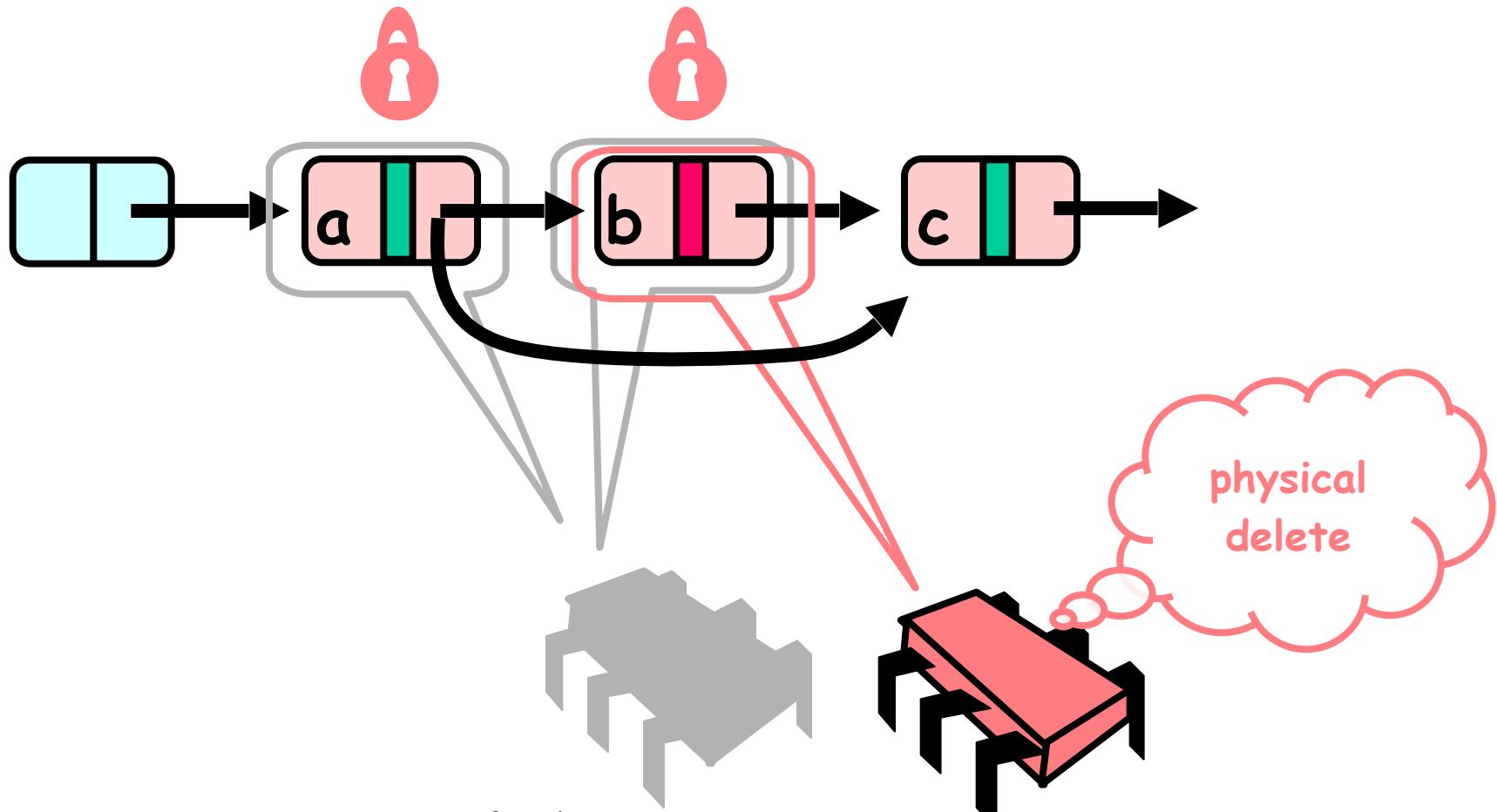
# Business as Usual



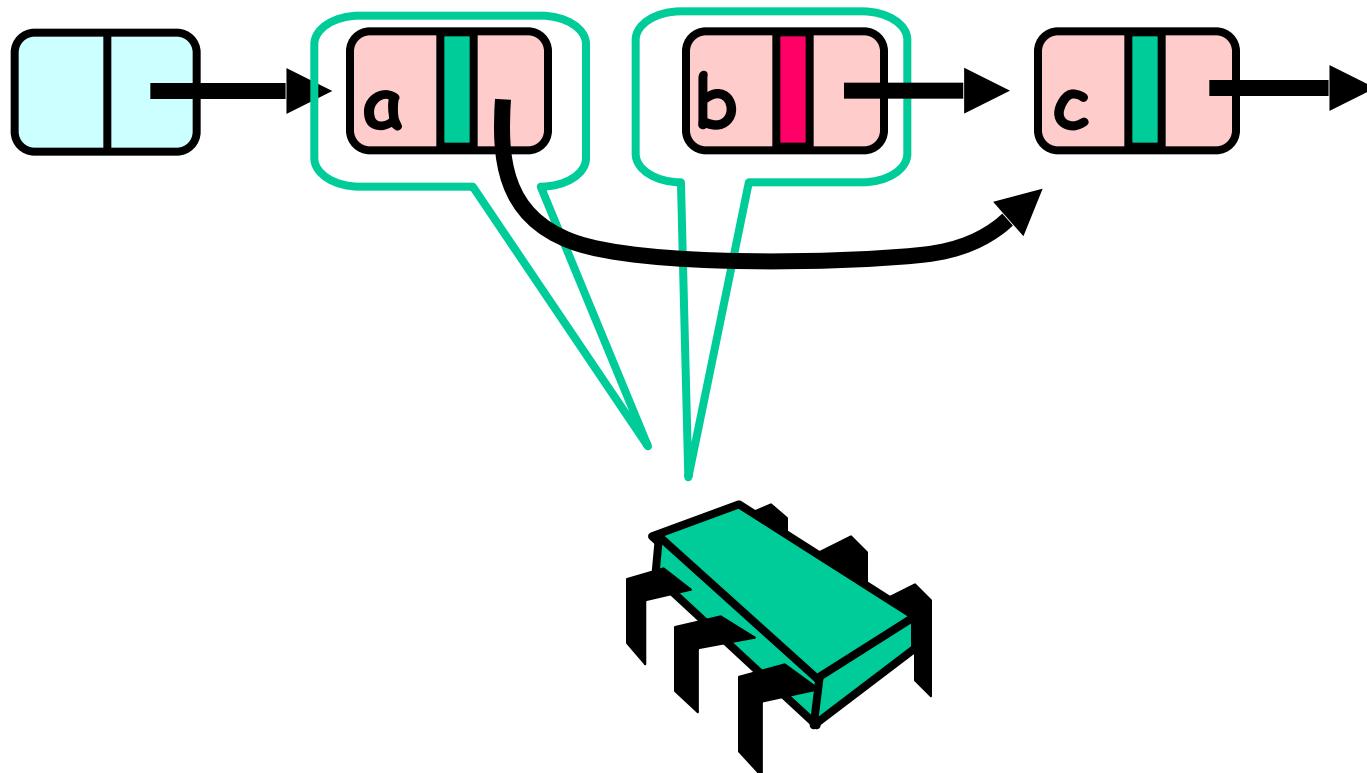
# Business as Usual



# Business as Usual



# Business as Usual



# Invariant

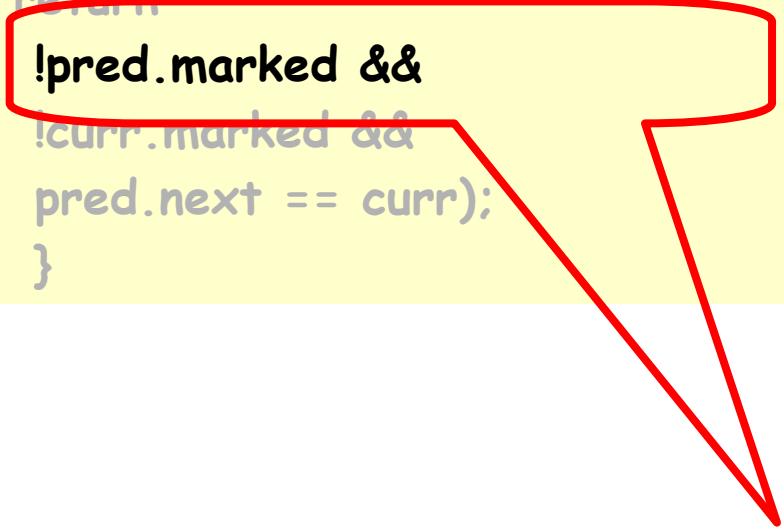
- If not marked then item in the set
- and reachable from head
- and if not yet traversed it is  
reachable from pred

# Validation

```
private boolean  
    validate(Node pred, Node curr) {  
    return  
        !pred.marked &&  
        !curr.marked &&  
        pred.next == curr);  
    }
```

# List Validate Method

```
private boolean  
    validate(Node pred, Node curr) {  
    return  
        !pred.marked &&  
        !curr.marked &&  
        pred.next == curr);  
    }
```



**Predecessor not  
Logically removed**

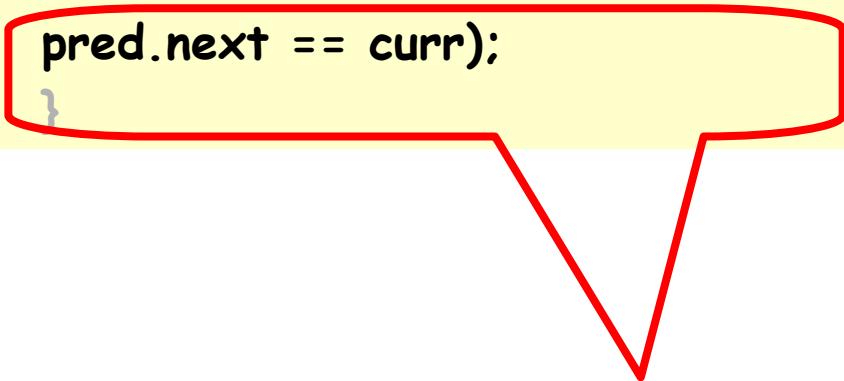
# List Validate Method

```
private boolean  
    validate(Node pred, Node curr) {  
    return  
        !pred.marked &&  
        !curr.marked &&  
        pred.next == curr);  
    }
```

Current not  
Logically removed

# List Validate Method

```
private boolean  
    validate(Node pred, Node curr) {  
    return  
        !pred.marked &&  
        !curr.marked &&  
        pred.next == curr);  
    }
```



**Predecessor still  
Points to current**

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Start at the head

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Search key range

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Traverse without locking  
(nodes may have been removed)

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Present and undeleted?

```
public boolean add(T item) {  
    int key = item.hashCode();  
    while (true) {  
        Node pred = this.head;  
        Node curr = head.next;  
        while (curr.key < key) {  
            pred = curr; curr = curr.next;  
        }  
        pred.lock();  
        try {  
            curr.lock();  
            try {  
                if (validate(pred, curr)) {  
                    if (curr.key == key) {  
                        return false;  
                    } else {  
                        Node Node = new Node(item);  
                        Node.next = curr;  
                        pred.next = Node;  
                        return true;  
                    }  
                }  
            } finally { // always unlock  
                curr.unlock();  
            }  
        } finally { // always unlock  
            pred.unlock();  
        }  
    }  
}
```

## Lazy Synchronization

```
public boolean remove(T item) {  
    int key = item.hashCode();  
    while (true) {  
        Node pred = this.head;  
        Node curr = head.next;  
        while (curr.key < key) {  
            pred = curr; curr = curr.next;  
        }  
        pred.lock();  
        try {  
            curr.lock();  
            try {  
                if (validate(pred, curr)) {  
                    if (curr.key != key) {  
                        return false;  
                    } else {  
                        curr.marked = true;  
                        pred.next = curr.next;  
                        return true;  
                    }  
                }  
            } finally {  
                curr.unlock();  
            }  
        } finally {  
            pred.unlock();  
        }  
    }  
}
```

```
public boolean contains(T item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key)  
        curr = curr.next;  
    return curr.key == key && !curr.marked;  
}
```

```
private boolean validate(Node pred, Node  
curr) {  
    return !pred.marked && !curr.marked &&  
pred.next == curr;  
}
```

# Evaluation

- Good:
  - contains() doesn't lock
  - Good because typically high % contains()
  - Uncontented calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays

# Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin”
    - Cache miss, page fault, descheduled ...
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler....

# Reminder: Lock-Free Data Structures



- No matter what ...
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call, even if others halt at malicious times
  - Implies that implementation can't use locks