



DD245 I
Parallel and Distributed Computing

FDD3008
Distributed Algorithms

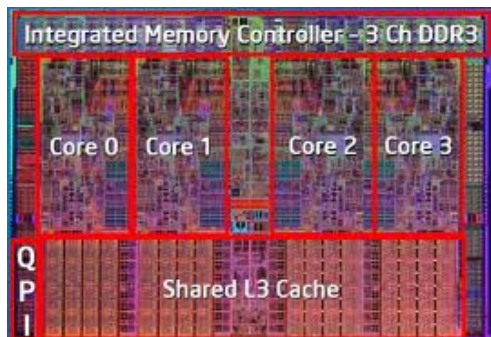
Lecture I

Mads Dam
Autumn/Winter 2011

DD245 I Overview

Lectures

- 12 double lectures
- Twice a week until beg. December
- Focus on theory and principles
- But look at some systems too
- Also some programming issues



Exercises

- Exercises given for each lecture
- Discussion at end of each lecture
- 3 homework sets to be handed in individually
- Homework rules and grading – see the course page
- We may use peer review
- Deadlines apply

DD245 I Overview

Paper presentations

- Chance to present research papers
- See “presentation sessions” in schedule on course web
- Optional, but counts towards final grade
- Select paper you want to present
- Some advice on course web
- Let Mads know by mail at the latest 11 Nov !



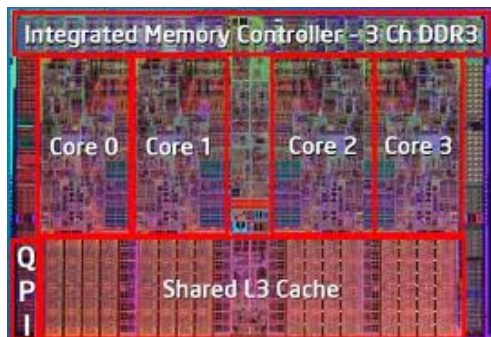
Other matters:

- Course committee/kursnämnd
- Any volunteers?
- KTH social – any interest?
- Anything else?

FDD 3008 Overview

Introductory postgraduate level

- Some additional requirements
- Paper presentation obligatory
- Final report 8-10 pages
- Preferably: Application to PhD project
- Topic selection quite free

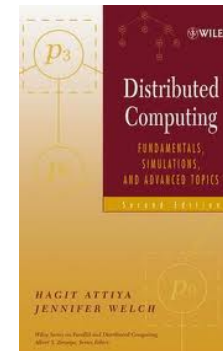
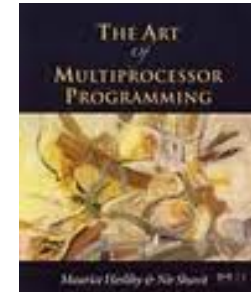


Generally:

- Herlihy's book at bit too informal for pg level
- More precision and formality expected in handins, reports, presentations
- Anyway the material gets more advanced as we move along

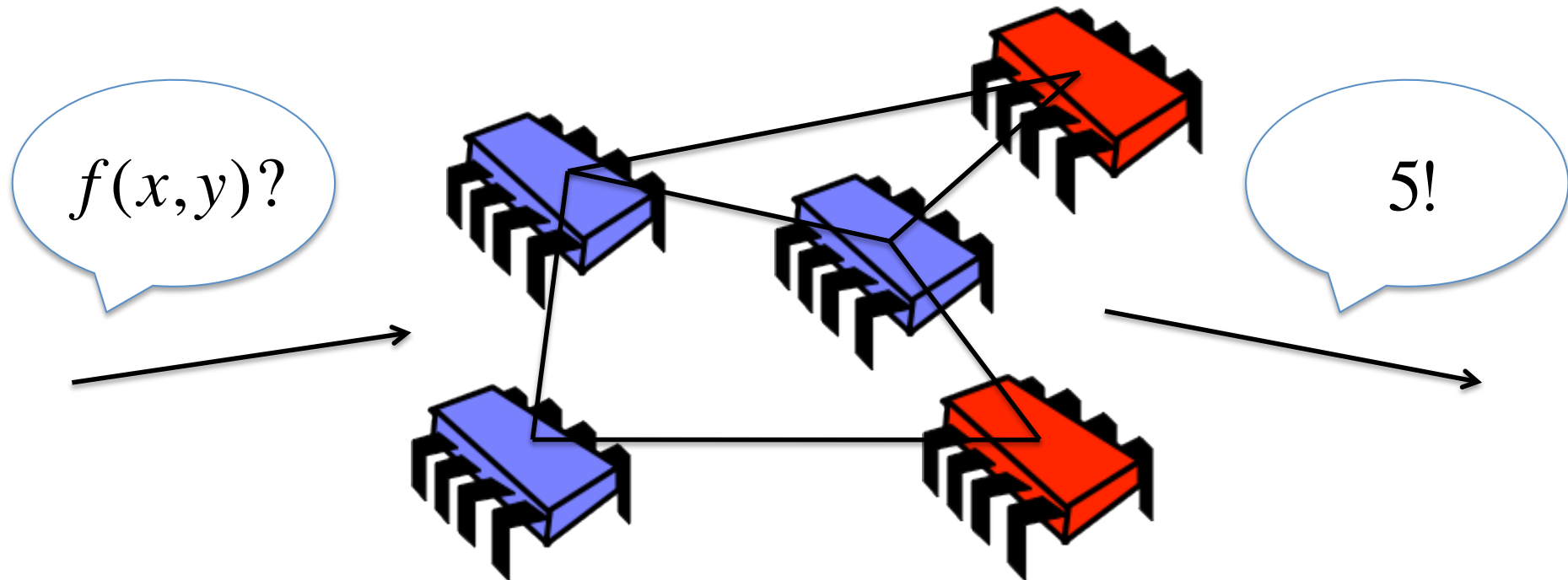
Course Material

- Main textbook: Herlihy and Shavit
- Focus on shared memory model
- Used a lot in first lectures
- Later more attention to message-passing and distribution
- Other textbooks used later on
- Other material (papers, course notes) to be announced on course page
- Slides: Much material due to Maurice Herlihy, Brown, and Roger Wattenhofer, ETH, Zurich, thanks!



What Is Parallel and Distributed Computing About?

How to distribute a computing task between several processors/
threads/nodes/cores?



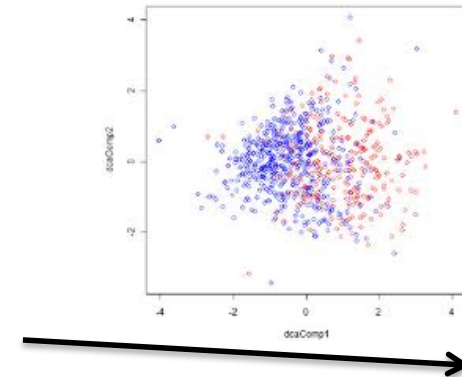
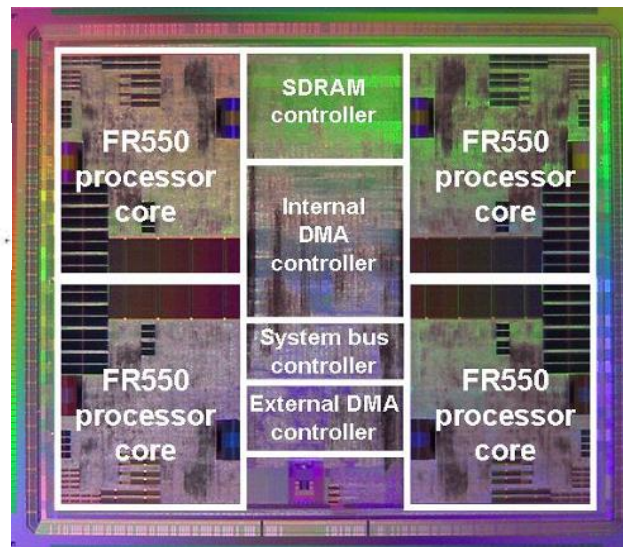
When? How? Performance? Models? Failures? Security? Scale? What is possible? What is not possible? Correctness? Fault tolerance?

What Is Parallel and Distributed Computing About?

How to distribute a computing task between several processors/
threads/nodes/cores?

$$\max_{0 \leq x \leq 1} \left(\max\{2x, x + p, 1\} - \frac{x^2}{2} \right)$$

subject to $x \leq \min \left\{ \frac{(2p - 1) \max\{2x, x + p, 1\}}{px + (1 - p)(1 - x)}, 1 \right\}$.



When? How? Performance? Models? Security? Scale? What is possible?
What is not possible? Correctness? Fault tolerance?

What Is Parallel and Distributed Computing About?

How to distribute a computing task between several processors/
threads/nodes/cores?



When? How? Performance? Models? Security? Scale? What is possible?
What is not possible? Correctness? Fault tolerance?

What Is Parallel and Distributed Computing About?

How to distribute a computing task between several processors/
threads/nodes/cores?



When? How? Performance? Models? Security? Scale? What is possible?
What is not possible? Correctness? Fault tolerance?

What Is Parallel and Distributed Computing About?

How to distribute a computing task between several processors/
threads/nodes/cores?

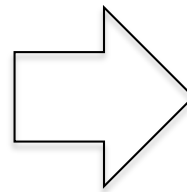
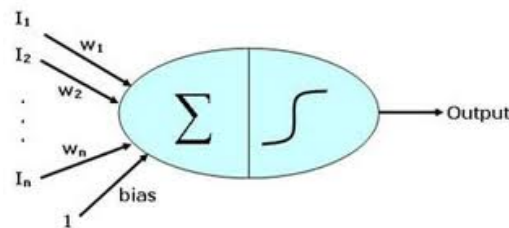


When? How? Performance? Models? Security? Scale? What is possible?
What is not possible? Correctness? Fault tolerance?

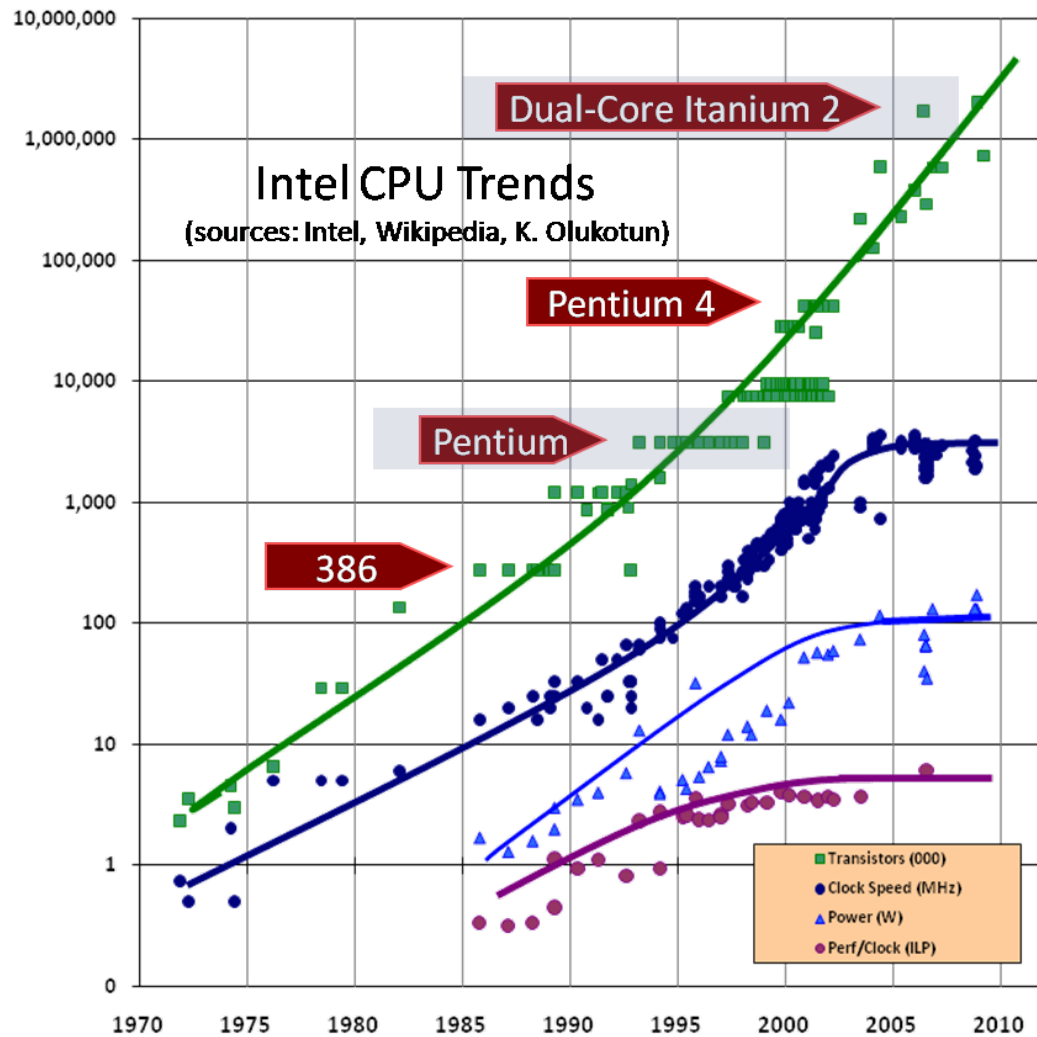
Does Scale Matter?

- It's all about scale!
- Small scale:
 - Small numbers of nodes, fixed interconnects, shared memory, little latency, fixed architectures, rare failures, mostly static
- Large scale:
 - Big numbers, system organization important, heterogenous architectures, message passing, big latency, frequent failures, more dynamic

- Ultimately:




Why Is It Important?



Read: The Free Lunch Is Over, Herb Sutter

Why Is It Important?

- High performance
 - High reliability
 - High availability
- 
- Concurrency is indispensable

But parallelization and distribution is tricky!

So let's get started

Course Overview

- First small scale
 - Mutual exclusion
 - Linearizability, sequential consistency, the Java memory model
 - Consensus, impossibility results, consensus numbers, registers
 - This part mainly uses Herlihy's book

Course Overview, II

- Then large scale
 - Fault tolerance
 - Byzantine fault tolerance
 - Consistency models, weak consistency
 - P2P systems, DHT's
 - Mostly slides and handouts
- Then some additional topics among:
 - Leader election, spanning trees, vertex colouring, and/or
 - Spin locks, concurrent data structures

Events

Events a, b :

A : $\text{read}(x = \text{true})$ – the event of thread A reading variable x and getting the result true

B : $\text{write}(y = 0)$ – do. of A writing 0 to y

Note: y might have been 0 already – doesn't matter

Events:

- Are instantaneous, $\text{time}(a) = \text{“time at which event } a \text{ occurs”}$
- (Not the same as atomic, not the same as a program statement)
- Asynchrony: No two events occur at the same time



Events, II

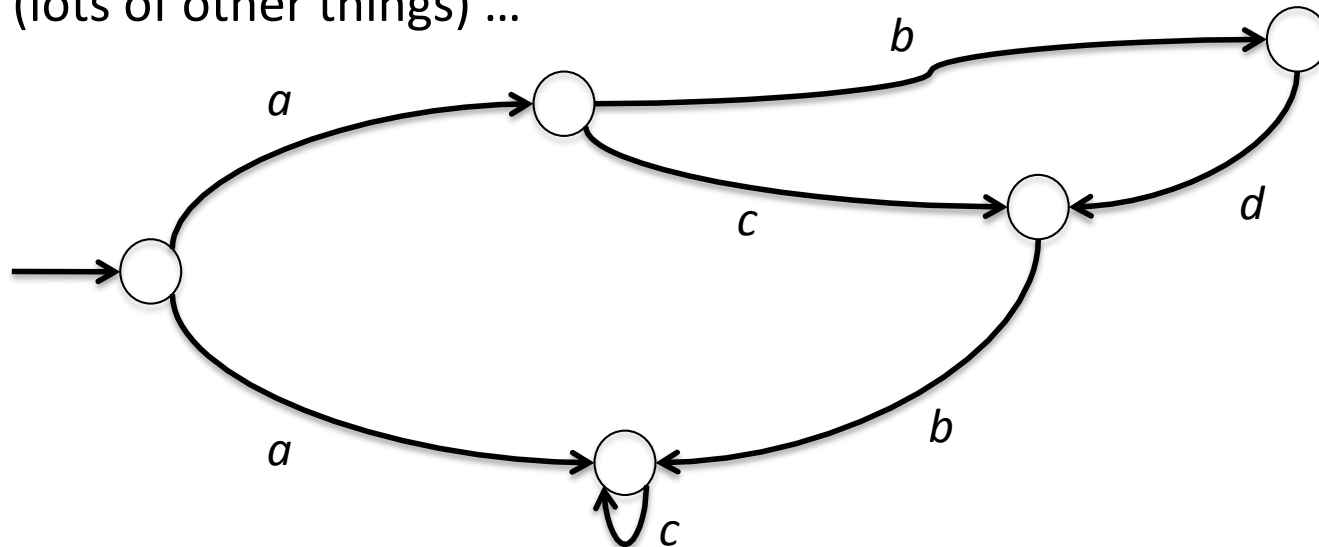
Examples:

- Reading a variable
- Writing a variable
- Outputting a value
- Inputting a value
- Receiving a signal
- Calling a method
- Returning from a method
- Returning normally from a method
- Returning exceptionally from a method
- ... (etc) ... depending on which abstractions we assume

Thread, Process

A thread / process is a *state machine*

- aka transition system
- aka transition graph
- aka ... (lots of other things) ...



- Need to distinguish between *events* and *event occurrences*
- Events are not necessarily “observable”

States

Model dependent

Threads:

- Object states – assignments of values to fields
- Thread states – do. to local variables
- Program counters
- (Maybe more, caches, load state, network state, etc)

Processor state:

- Registers
- Memory
- Caches, pipelines
- Configuration

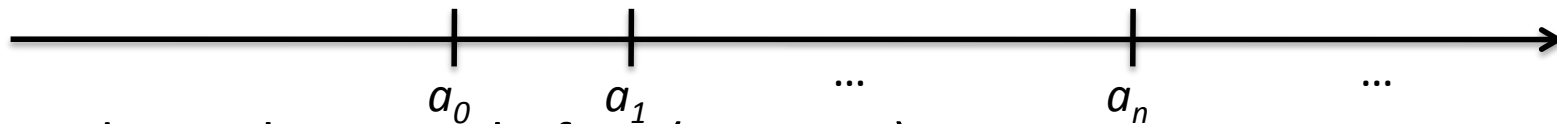
Network state

- “Wire” state
- Buffer states
- Configuration state
 - IP numbers
 - Lots and lots and lots

Traces

A thread produces a *trace* = sequence of events

$$tr = a_0 a_1 \dots a_n \dots$$



Precedence, happens-before (Lamport):

$a \rightarrow b$: Same as $time(a) < time(b)$

The precedence relation is a *total order*

- Irreflexive: $\text{not}(a \rightarrow a)$
- Antisymmetric: If $a \rightarrow b$ then $\text{not}(b \rightarrow a)$
- Transitive: If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
- Total: For all $a \neq b$, either $a \rightarrow b$ or $b \rightarrow a$

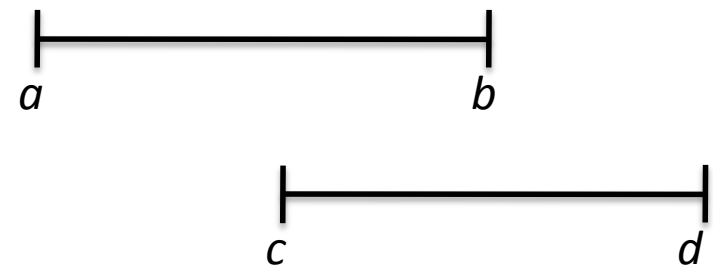
Intervals

(a,b) = the interval between a and b
= the set of time point from $time(a)$ to $time(b)$
(endpoints included)

$(a,b) \rightarrow (c,d)$
= (a,b) precedes (c,d)
= $b \rightarrow c$
= (a,b) ends before (c,d) starts



$(a,b), (c,d)$ concurrent, if not $(a,b) \rightarrow (c,d)$



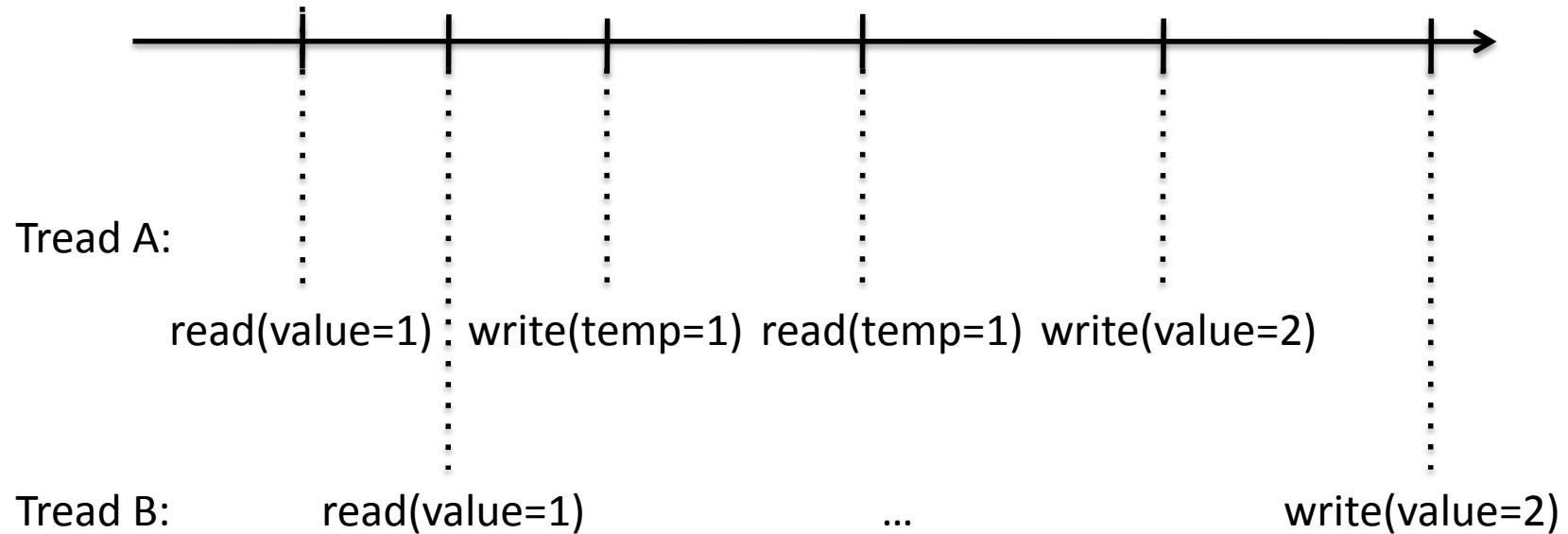
Critical Sections

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

Say: Assign globally unique sequence numbers

What if two threads call `getAndIncrement` in parallel?

Critical Sections, II



```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

Locks

```
public interface Lock {  
    public void lock();  
    public void unlock();  
}
```

`Lock.lock()`: Entering critical section

`Lock.unlock()`: Exiting

Using Locks

```
public class Counter {  
    private long value;  
    private Lock lock;  
    public long getAndIncrement() {  
        lock.lock();  
        try {  
            int temp = value;  
            value = value + 1;  
        } finally {  
            lock.unlock();  
        }  
        return temp;  
    }  
}
```

Properties

Mutual exclusion:

- CS_A^i : The i :th interval during which thread A is in its critical section
- For any A, B, i, j , $CS_A^i \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^i$

Absence of deadlock:

- If some thread calls `lock()` and never returns
- (i.e. it has acquired the lock)
- then other threads acquire and release the lock infinitely often

No starvation:

- Each call to `lock()` eventually returns

2 Threads, First Attempt

```
class LockOne implements Lock {
private boolean[] flag = new boolean[2];

public void lock() {
    flag[thisThread] = true;
    while (flag[otherThread]) {} }

public void unlock() {
    flag[thisThread] = false }
}
```

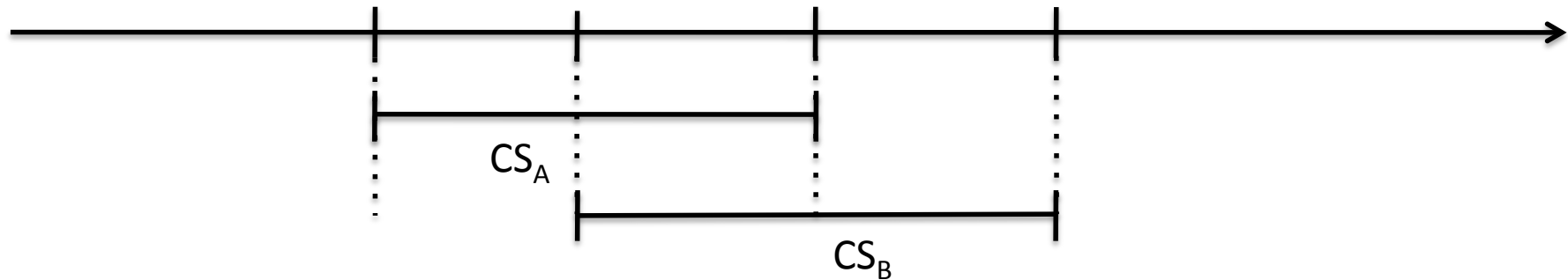
Notes:

- LockOne deadlocks
- flag array should be declared volatile, why?

2 Threads, First Attempt

Lemma: Mutual exclusion holds

Proof: Suppose

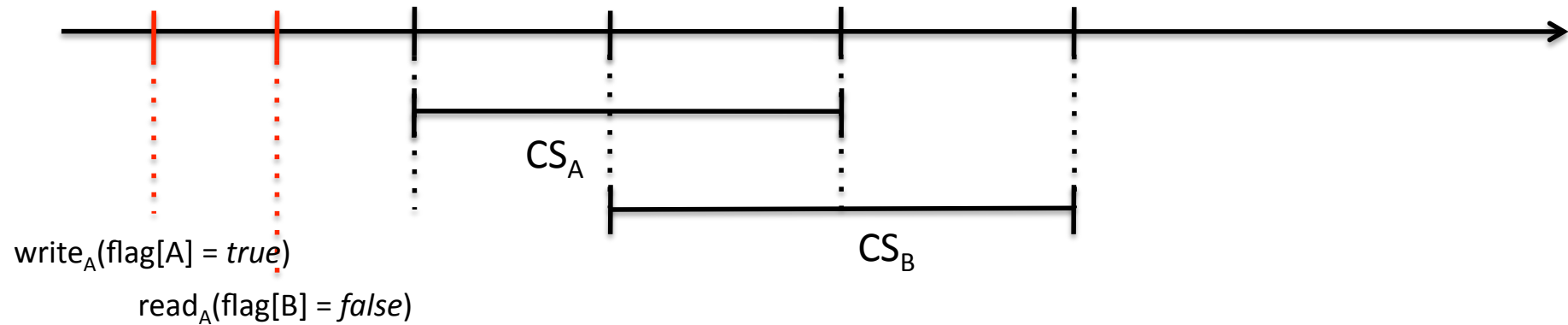


```
class LockOne implements Lock {
private boolean[] flag = new boolean[2];

public void lock() {
    flag[thisThread] = true;
    while (flag[otherThread]) {} }

public void unlock() {
    flag[thisThread] = false }
```

2 Threads, First Attempt

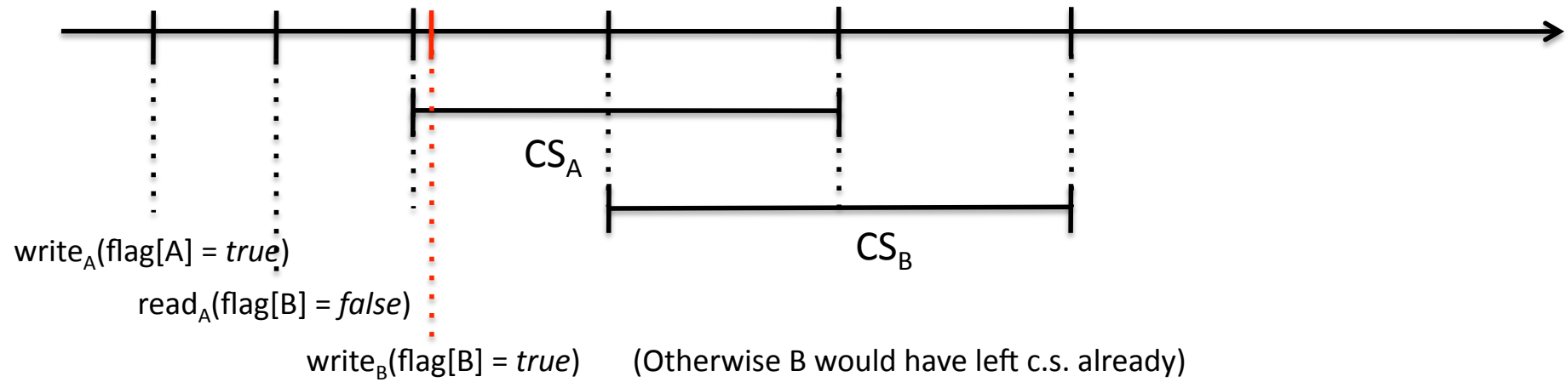


```
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];

    public void lock() {
        flag[thisThread] = true;
        while (flag[otherThread]) {} }

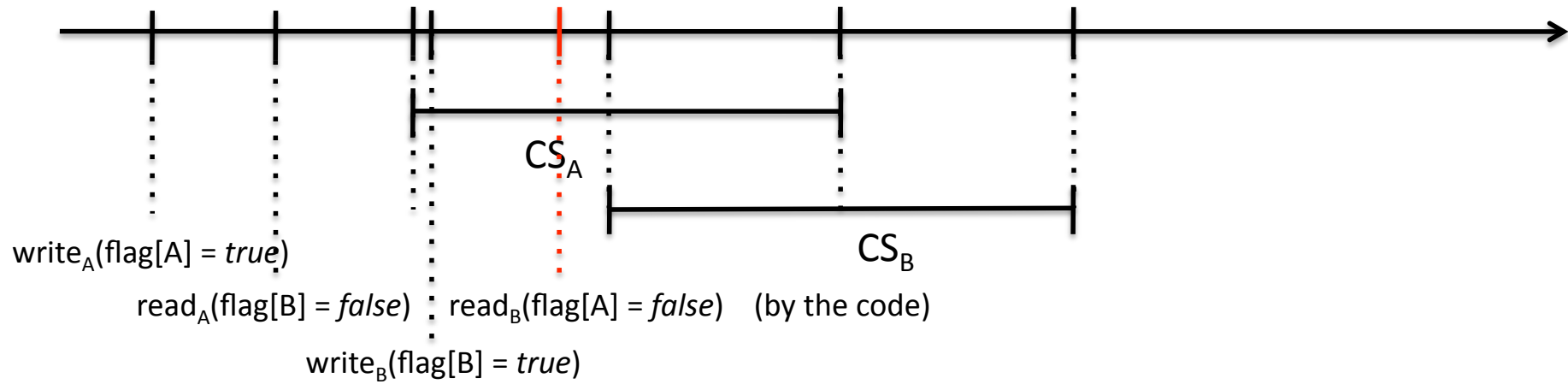
    public void unlock() {
        flag[thisThread] = false }
}
```

2 Threads, First Attempt



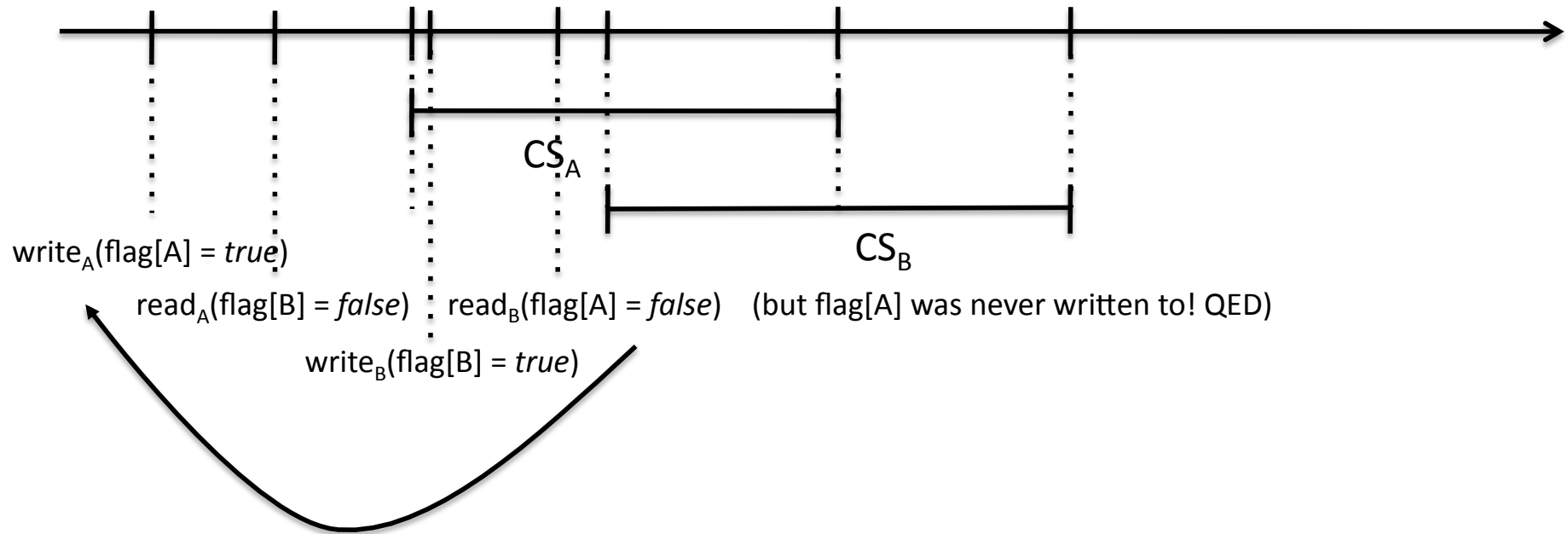
```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
  
    public void lock() {  
        flag[thisThread] = true;  
        while (flag[otherThread]) {} }  
  
    public void unlock() {  
        flag[thisThread] = false }  
}
```

2 Threads, First Attempt



```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
  
    public void lock() {  
        flag[thisThread] = true;  
        while (flag[otherThread]) {} }  
  
    public void unlock() {  
        flag[thisThread] = false }  
}
```

2 Threads, First Attempt



```
class LockOne implements Lock {
private boolean[] flag = new boolean[2];

public void lock() {
    flag[thisThread] = true;
    while (flag[otherThread]) {} }

public void unlock() {
    flag[thisThread] = false }
```


2 Threads, Second Attempt

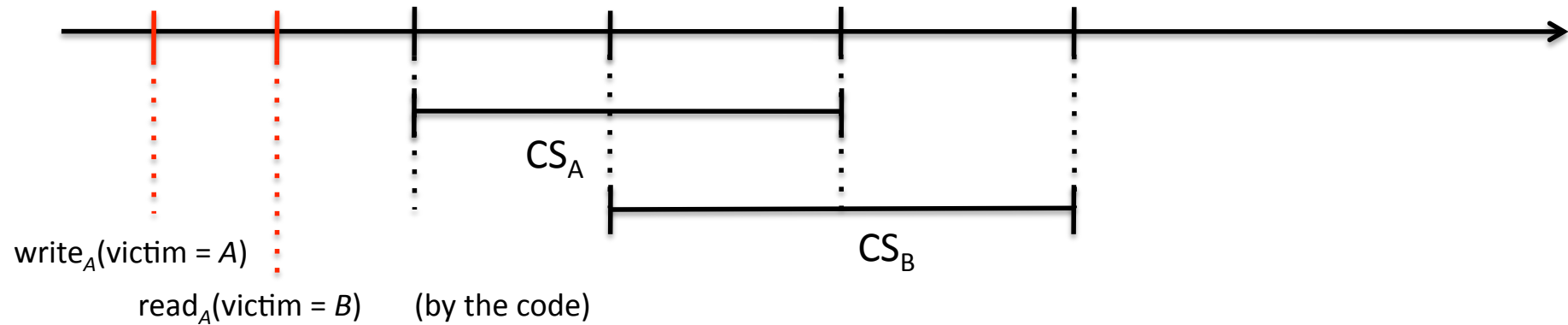
```
public class LockTwo implements Lock {
    private int victim;
    public void lock() {
        victim = thisThread;
        while (victim == thisThread) {};
    }

    public void unlock() {}
}
```

Notes:

- victim = A : A defers to B
- LockTwo also deadlocks

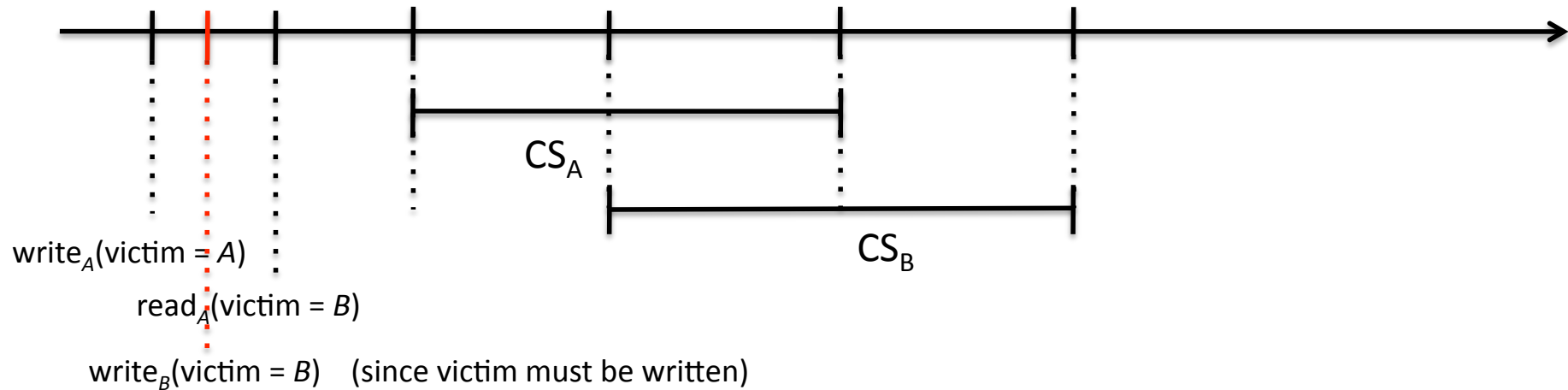
LockTwo, Mutual Exclusion



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

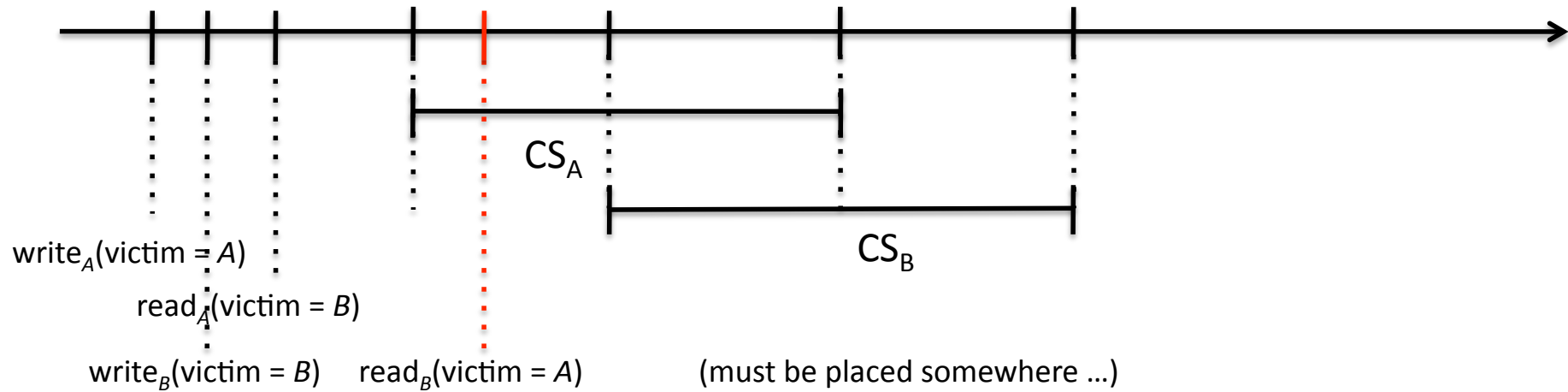
    public void unlock() {}
}
```

LockTwo, Mutual Exclusion



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

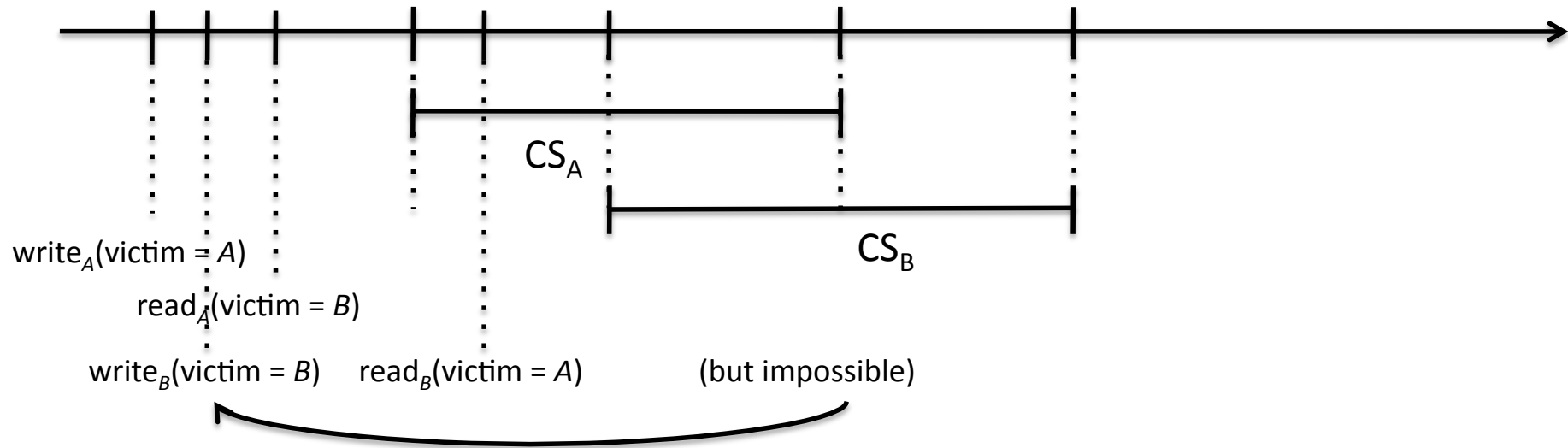
LockTwo, Mutual Exclusion



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

    public void unlock() {}
}
```

LockTwo, Mutual Exclusion



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

    public void unlock() {}
}
```

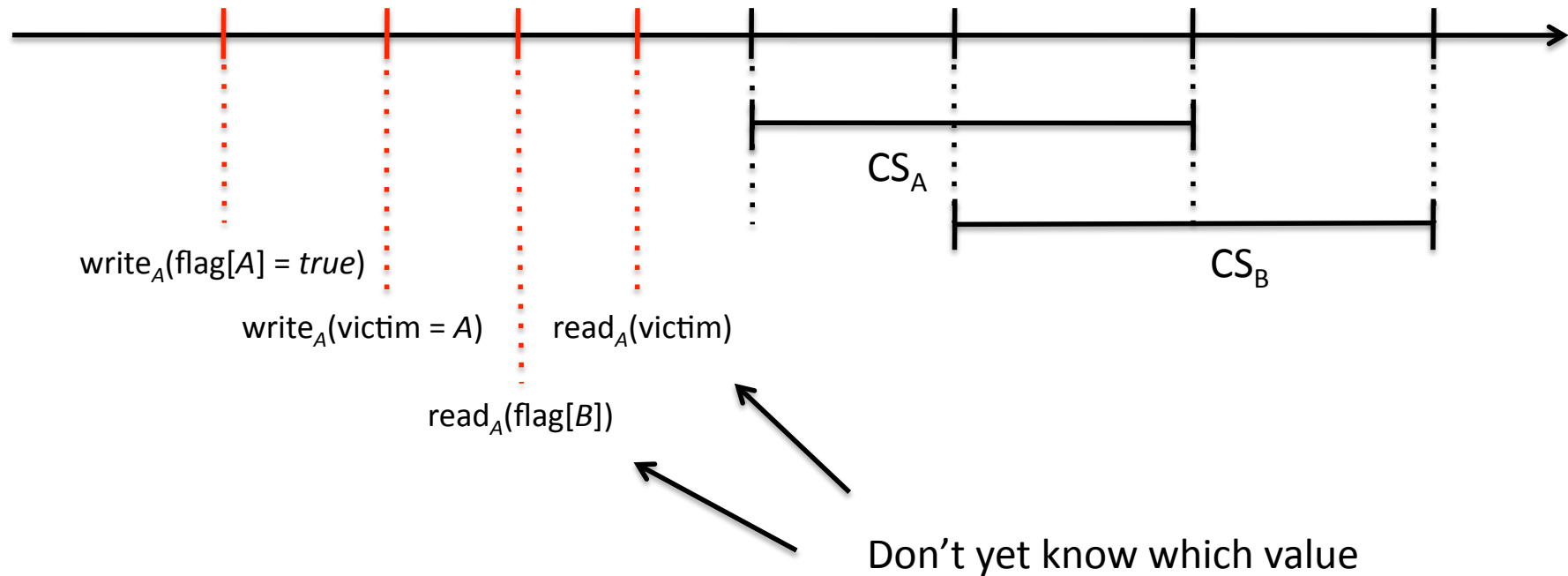
Peterson's Algorithm

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

Notes:

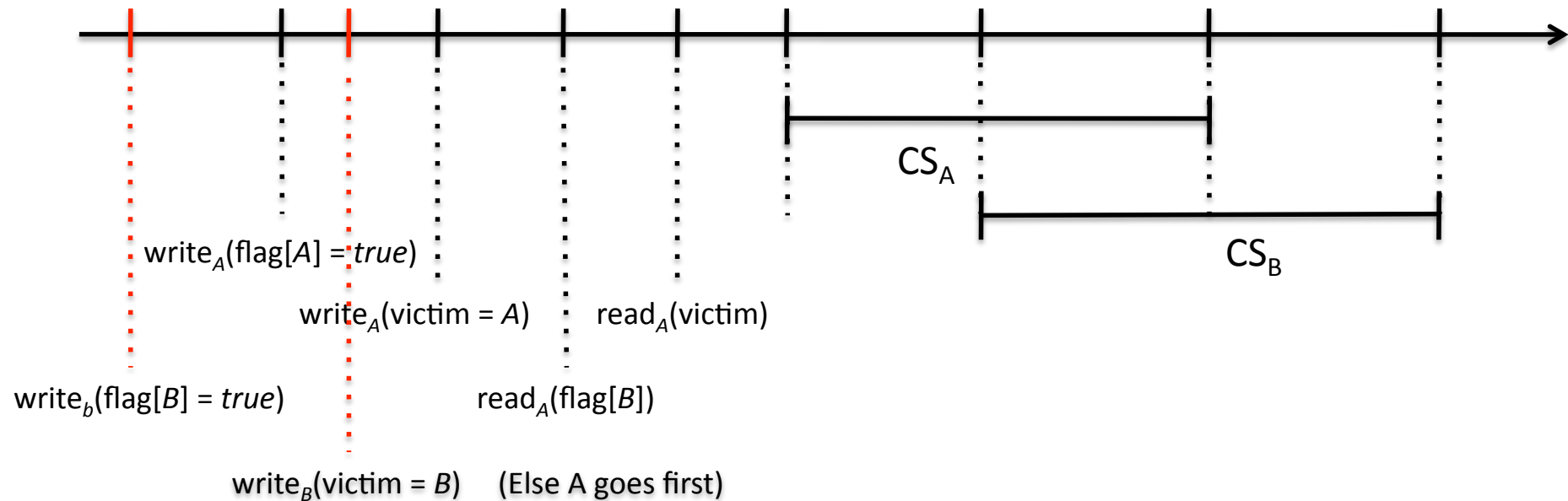
- Flag your interest and step back if other thread also wants a go

Peterson's Algorithm, Mutual Exclusion



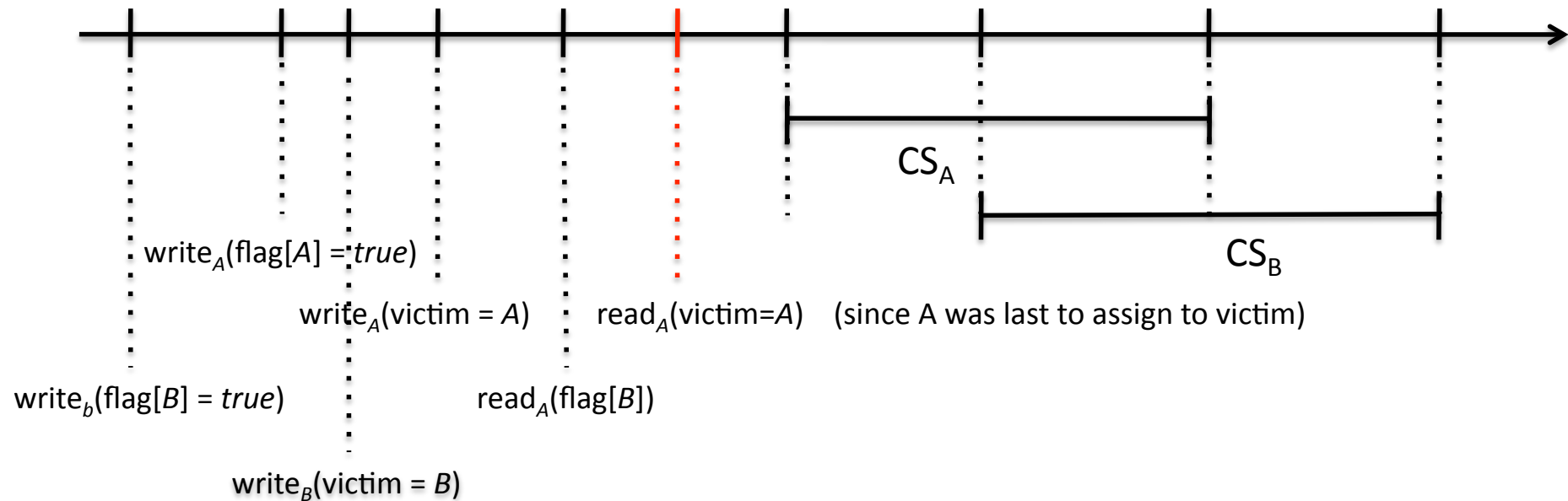
```
public void lock() {
    flag[thisThread] = true;
    victim = thisThread;
    while (flag[otherThread] && victim == thisThread) {};
}
public void unlock() {
    flag[thisThread] = false;
}
```

Peterson's Algorithm, Mutual Exclusion



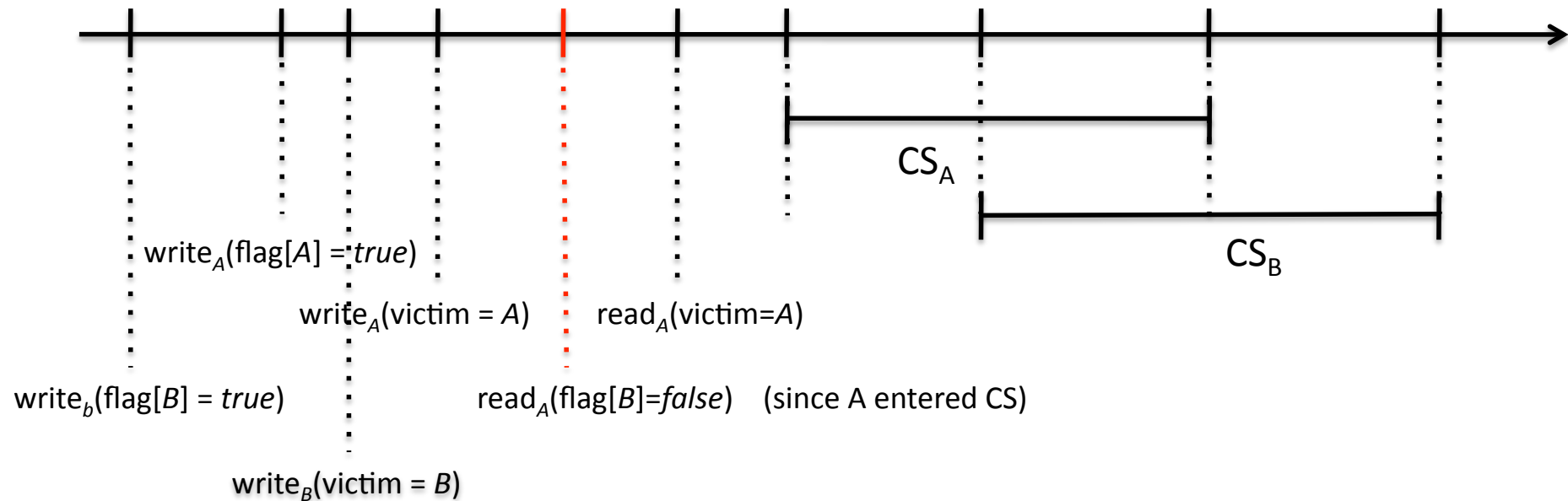
```
public void lock() {  
    flag[thisThread] = true;  
    victim = thisThread;  
    while (flag[otherThread] && victim == thisThread) {};  
}  
public void unlock() {  
    flag[thisThread] = false;  
}
```


Peterson's Algorithm, Mutual Exclusion



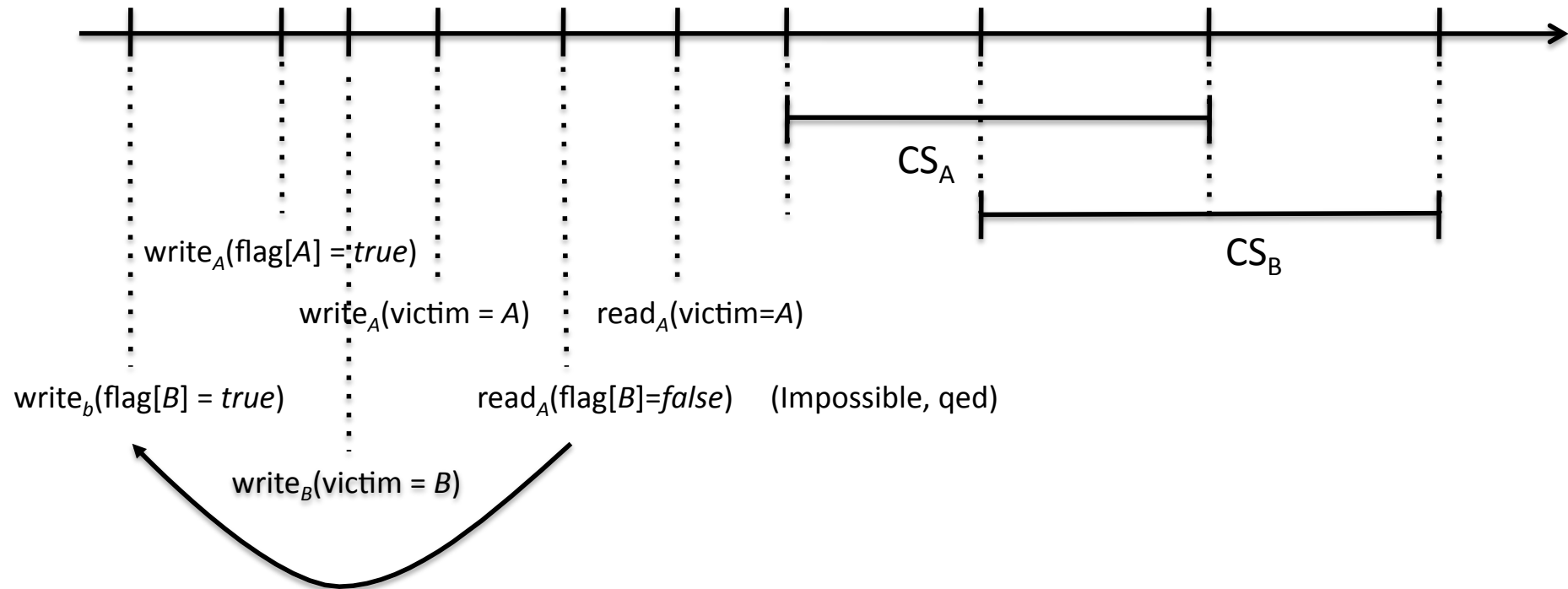
```
public void lock() {
    flag[thisThread] = true;
    victim = thisThread;
    while (flag[otherThread] && victim == thisThread) {};
}
public void unlock() {
    flag[thisThread] = false;
}
```

Peterson's Algorithm, Mutual Exclusion



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

Peterson's Algorithm, Mutual Exclusion



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

Peterson's Algorithm, No Starvation

Suppose thread A spins from time t onwards

Then $\text{victim} = A$ and $\text{flag}[B] = \text{true}$ infinitely often (i.o.)

Since $\text{flag}[B] = \text{true}$ i.o.:

Either B enters and exits CS_B i.o.

But then $\text{victim} = B$ almost always

("almost always" = always, from some time t' onwards)

Else B spins in the $\text{lock}()$ loop

But then $\text{victim} = B$ almost always, a contradiction

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

Lamport's 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;  
        }  
    }  
}
```

Used to
declare
interest,
as
earlier

Smallest non-zero label
gets to go

Lamport's Bakery Algorithm

I want the cs

Doorway

```
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)) {};  
    }  
}
```

Pick a new ticket, later than any already in use

Wait until the new ticket is minimal
Uses lexicographic order to break symmetry

Lamport's Bakery Algorithm

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

The Bakery Algorithm, Correctness

Lemma: The Bakery algorithm is deadlock free

Proof: Eventually a waiting thread will hold the least (*label, thread*) pair. QED

Lemma: The Bakery algorithm is first-come-first-served

Proof: First through the doorway gets the smallest “ticket”. QED

Lemma: The Bakery algorithm satisfies mutual exclusion

Proof: Check the textbook.

Timestamps

Bakery algorithm:

- Timestamp: (*label*, *thread*) pair
- Timestamp domain unbounded for correctness

Two operations needed:

- scan: Sample the label array
- label: Produce a new label, strict upper bound of scanned labels

Possible to do this using only bounded data?

Bounded Timestamps

Possible to build bounded timestamp system which

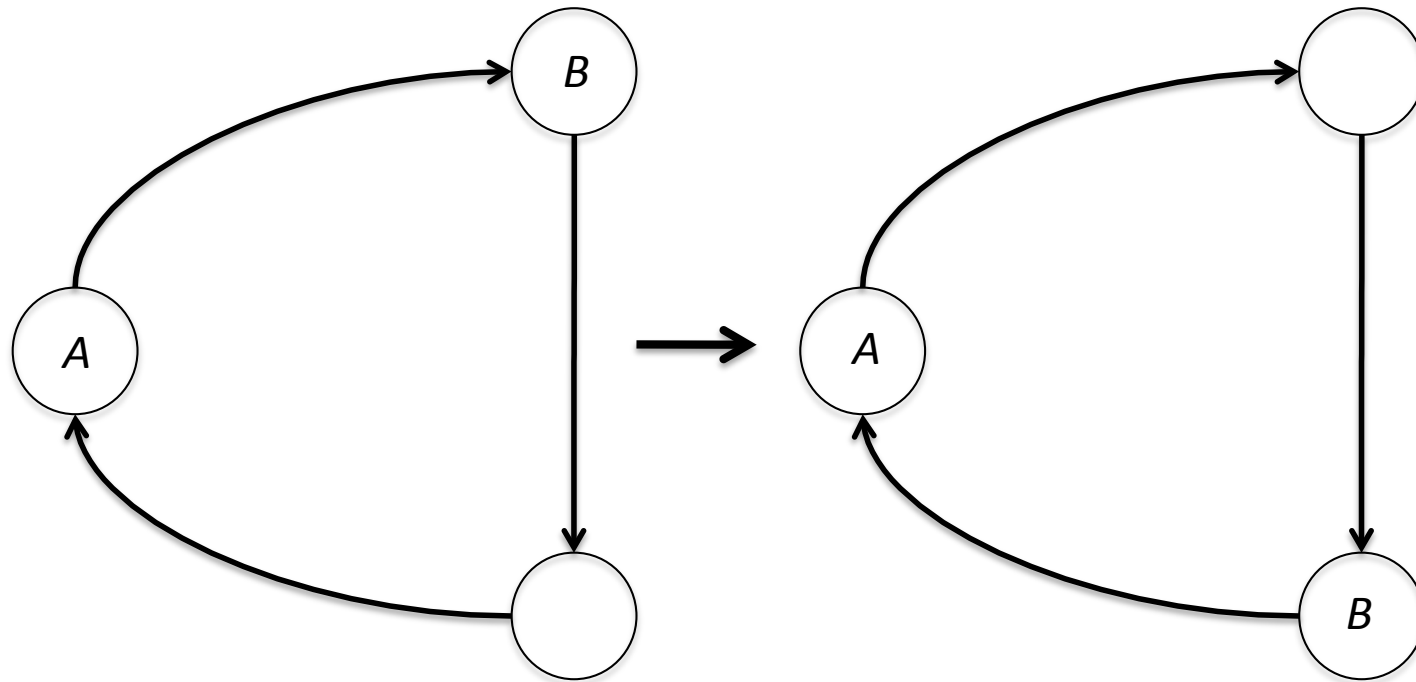
- Is wait-free – i.e. no locks
- For multiple threads

(cf. Dolev-Shavit-SIAM-97)

Here only sequential solution

- Not wait-free
- So useless for the Bakery algorithm
- Still of interest

Two Thread Bounded Precedence Graph

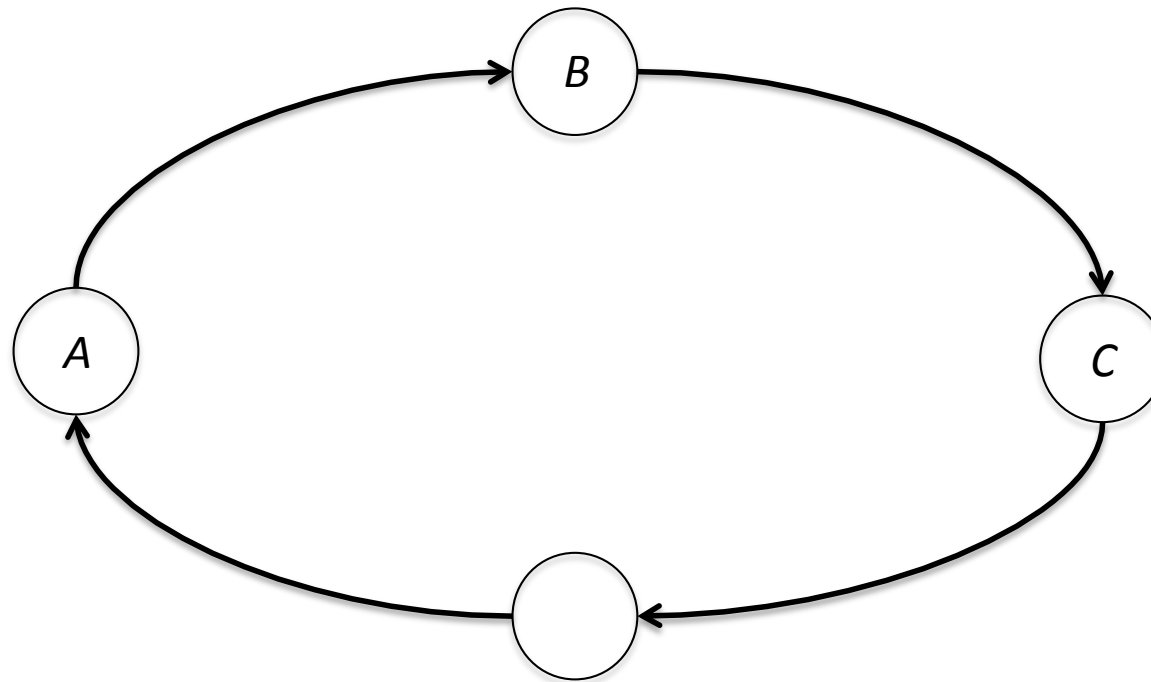


$A \rightarrow B$: A has precedence over B

$B \rightarrow A$: B has precedence over A

Note that precedence here is not transitive!

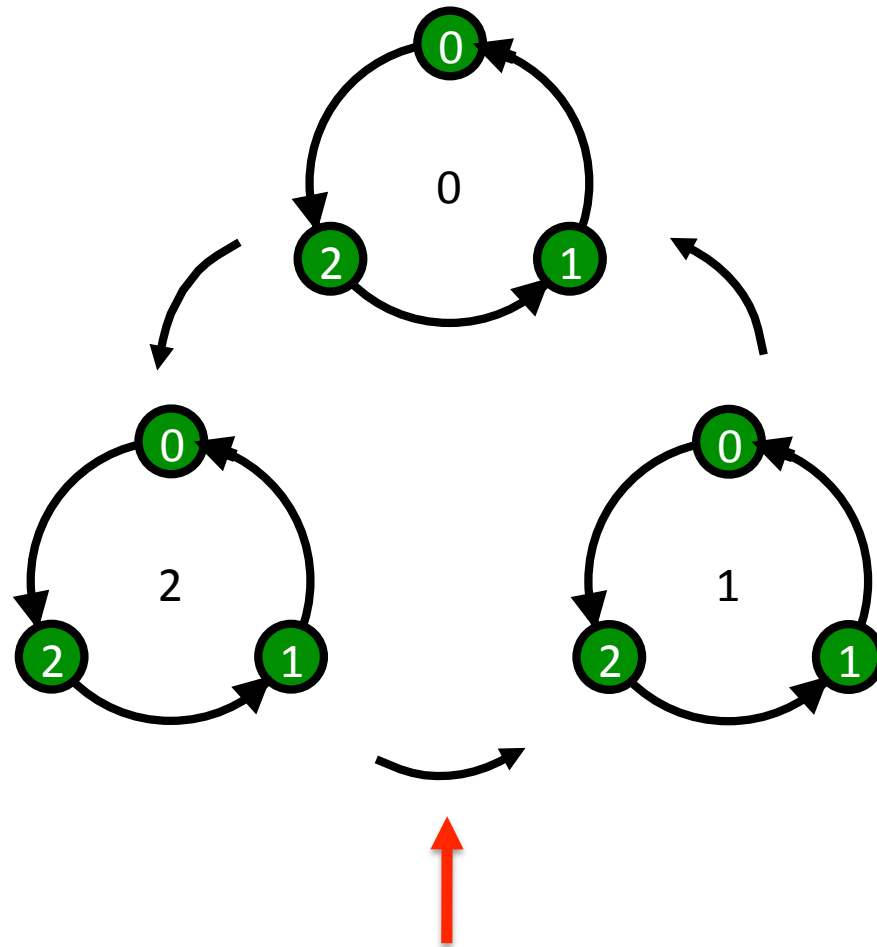
Three Thread Bounded Precedence Graph?



But \rightarrow is not transitive!

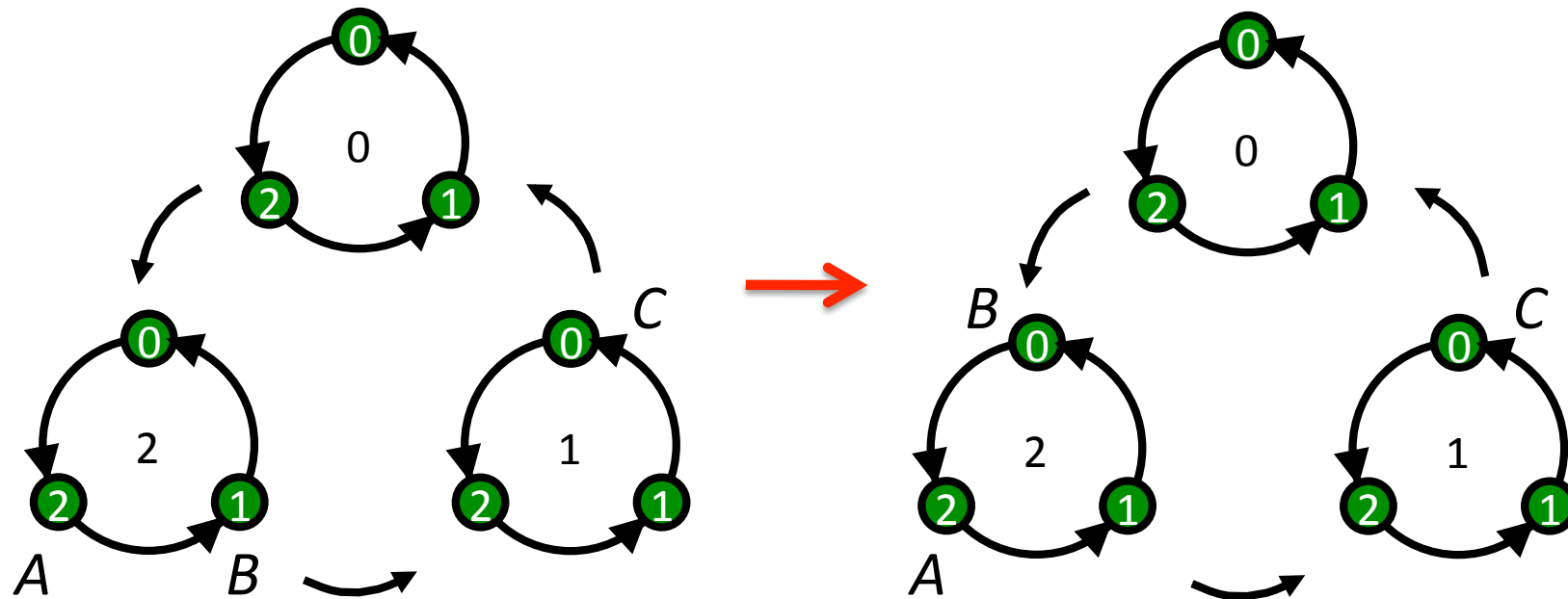
Eh? Want $A \rightarrow C$ as well? What if B wants precedence?

Three Thread Bounded Precedence Graph



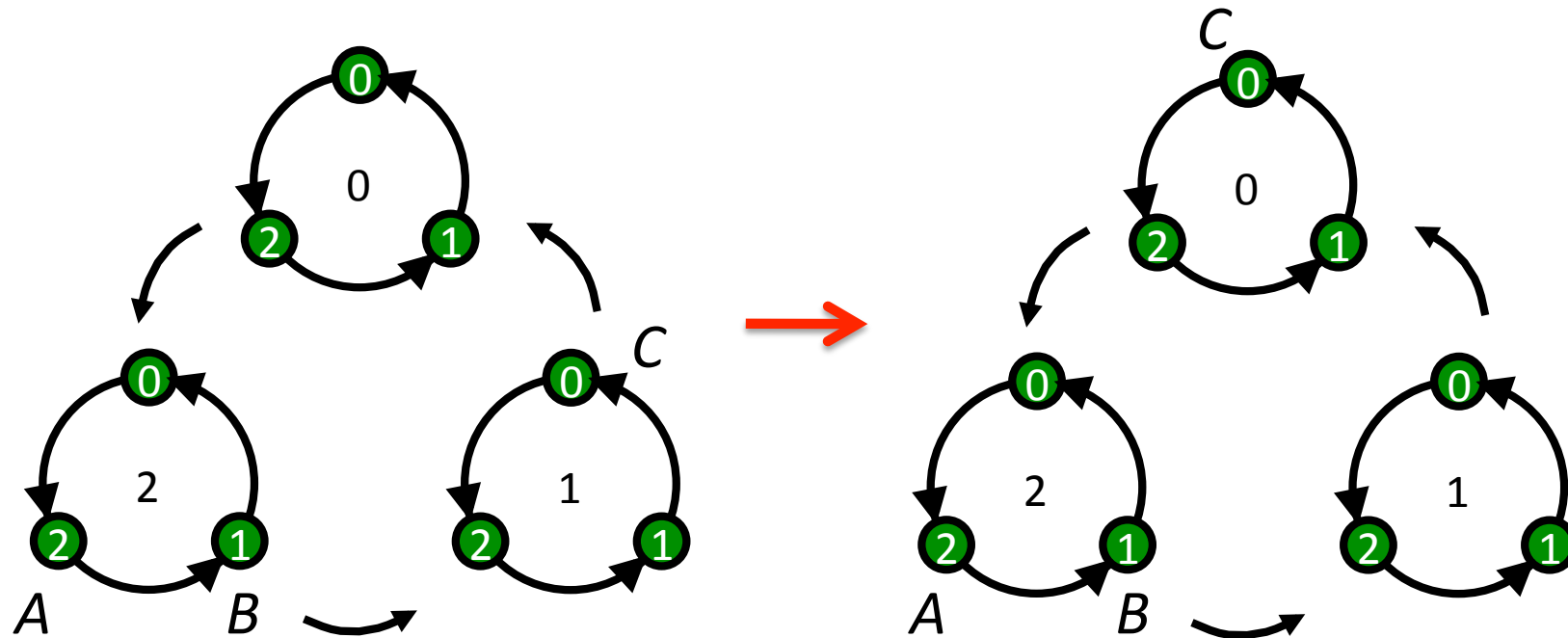
Edge from all vertex in cycle 2 to any vertex in cycle 1 etc

Three Thread Bounded Precedence Graph



B moves

Three Thread Bounded Precedence Graph

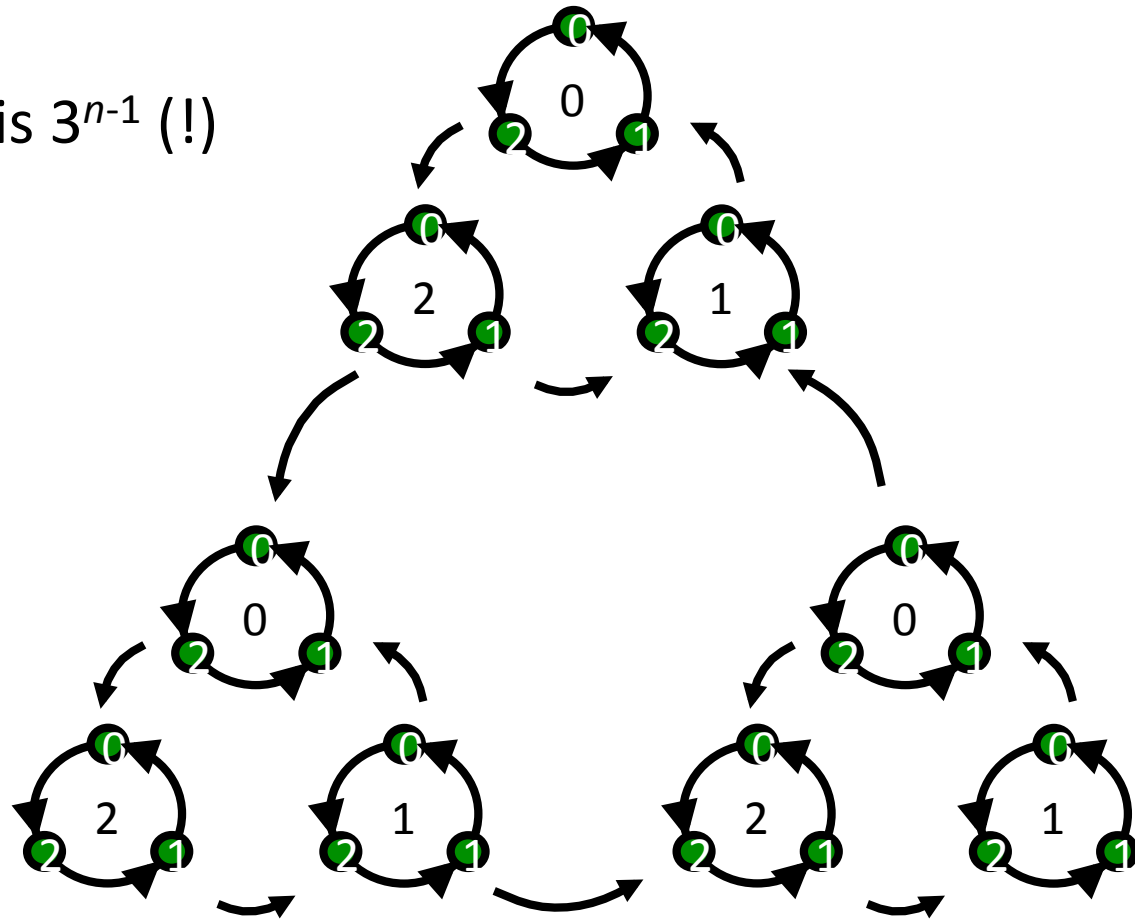


C moves

n Thread Bounded Precedence Graph

And so on ...

n thread graph size is 3^{n-1} (!)



So Is the Bakery Algorithm the Way to Go?

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

Shared Memory

- Shared read/write memory locations are called **registers** (for historical reasons)
- Come in different flavors
 - Multi-Reader-Single-Writer (`flag[]`)
 - Multi-Reader-Multi-Writer (`victim[]`)
 - Less interesting for now: SRMW and SRSW

n MRSW Registers Are Needed

Theorem: At least n MRSW (multi-reader/single-writer) registers are needed to solve deadlock-free mutual exclusion.

- n registers like `flag[]`...

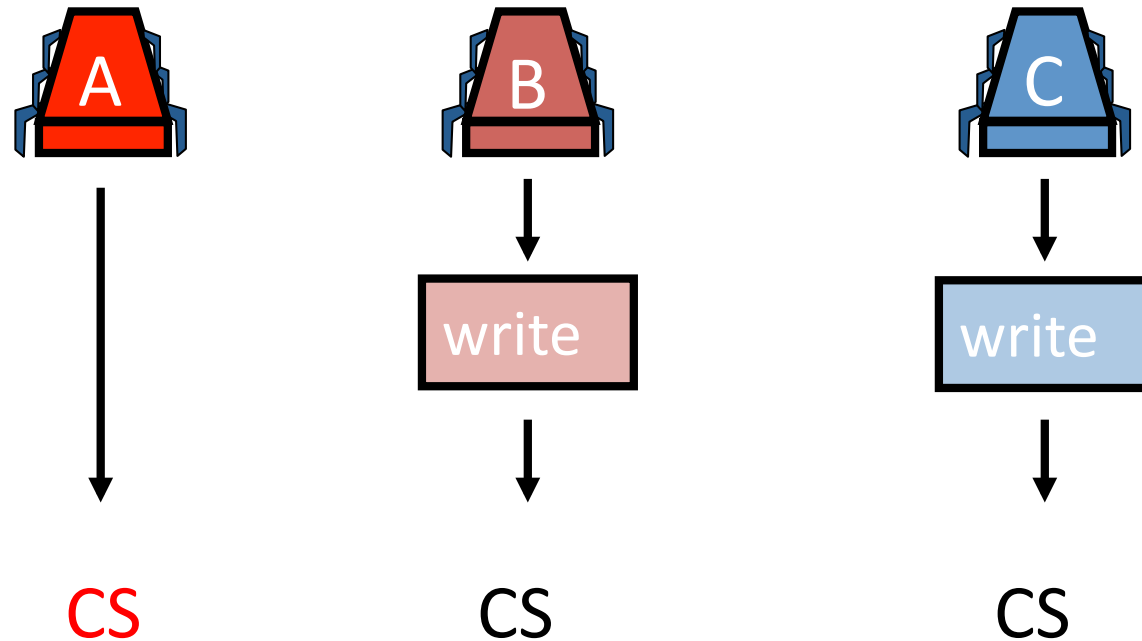
This is a *lower bound* result – generally tough

Proof technique

- Assume that an algorithm exists
- (for deadlock free mutual exclusion using $< n$ MRSW registers)
- Show a bad execution that violates properties must exist

Proof: n MRSW Registers Are Needed

Each thread must write to some register



... can't tell whether A is in critical section

Upper Bound

Bakery algorithm uses $2n$ MRSW registers

- So the bound is (pretty) tight

But what if we use MRMW registers?

- Like `victim[]` ?

Bad News Theorem

At least n MRMW multi-reader/**multi-writer** registers are needed to solve deadlock-free mutual exclusion.

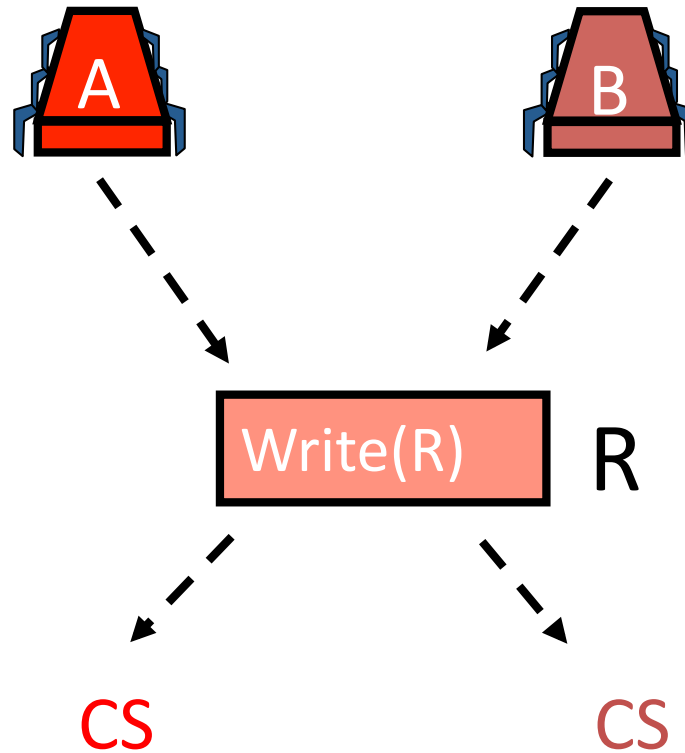
(So multiple writers don't help)

Theorem (First 2-Threads)

Theorem: Deadlock-free mutual exclusion for 2 threads requires at least 2 multi-reader multi-writer registers

Proof: assume one register suffices and derive a contradiction

Two Thread Execution

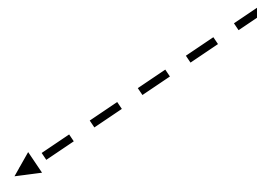
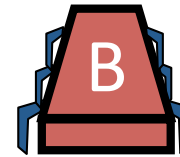


- Threads run, reading and writing R
- Deadlock free so at least one gets in

Covering State for One Register Always Exists

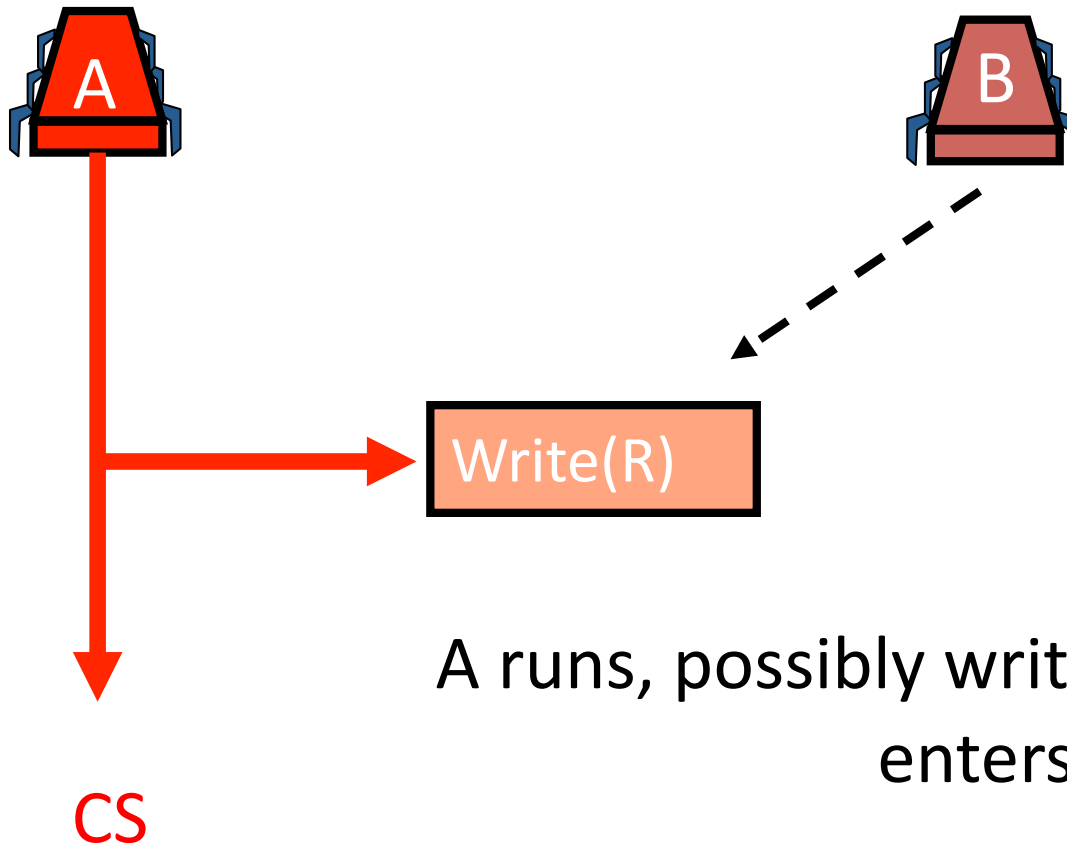
In any protocol
B has to write
to the register
before entering
CS, so stop it
just before

Write(R)

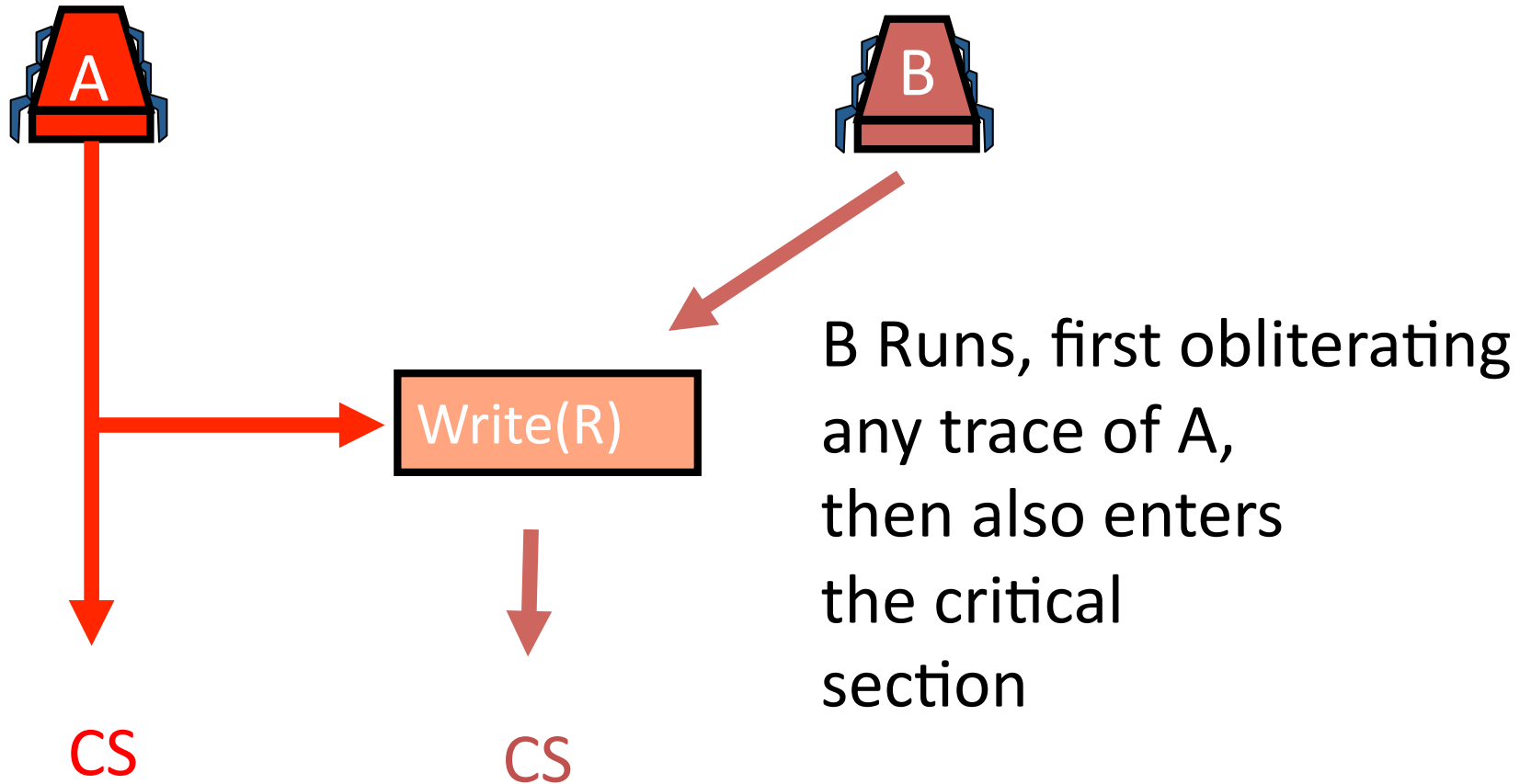


Cover of register R: A thread is about to write to R
and R's value allows other threads to enter CS
Covering state: All shared registers are covered

Proof: Assume Cover of 1



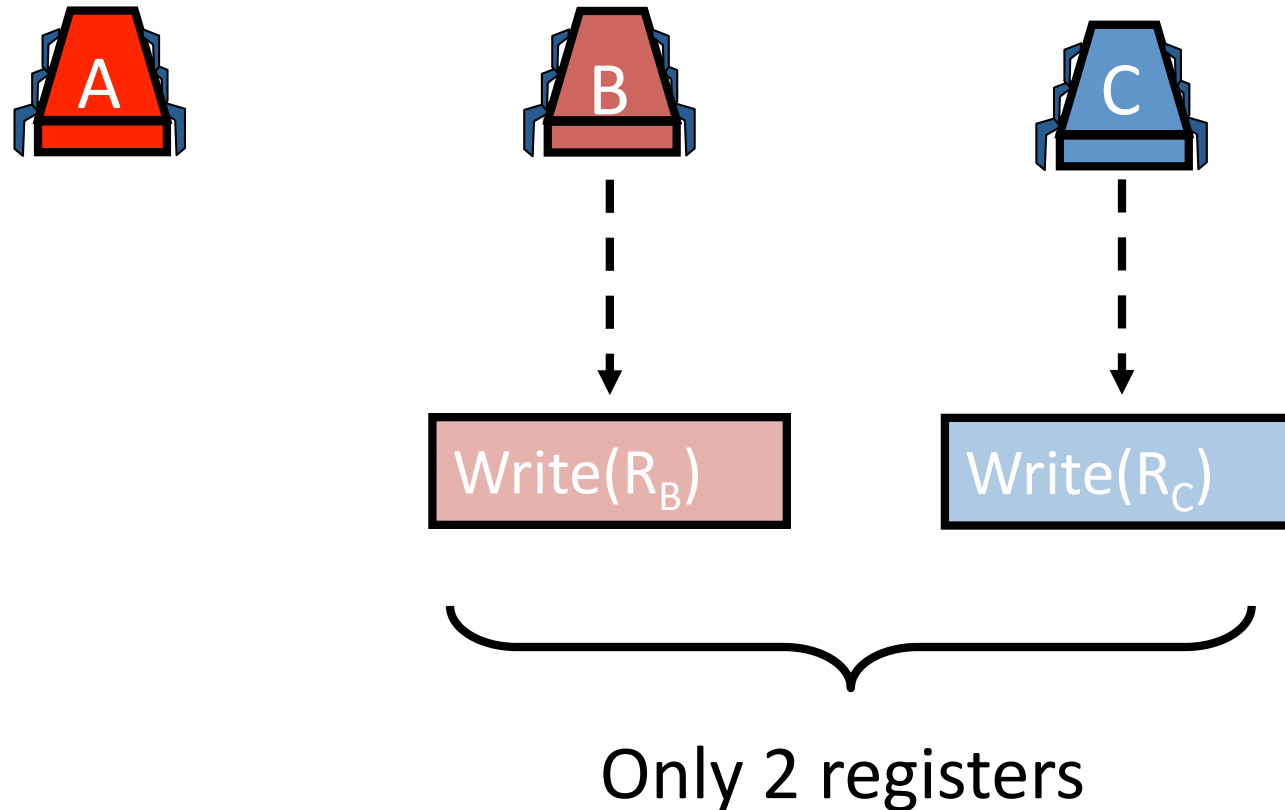
Proof: Assume Cover of 1



Theorem

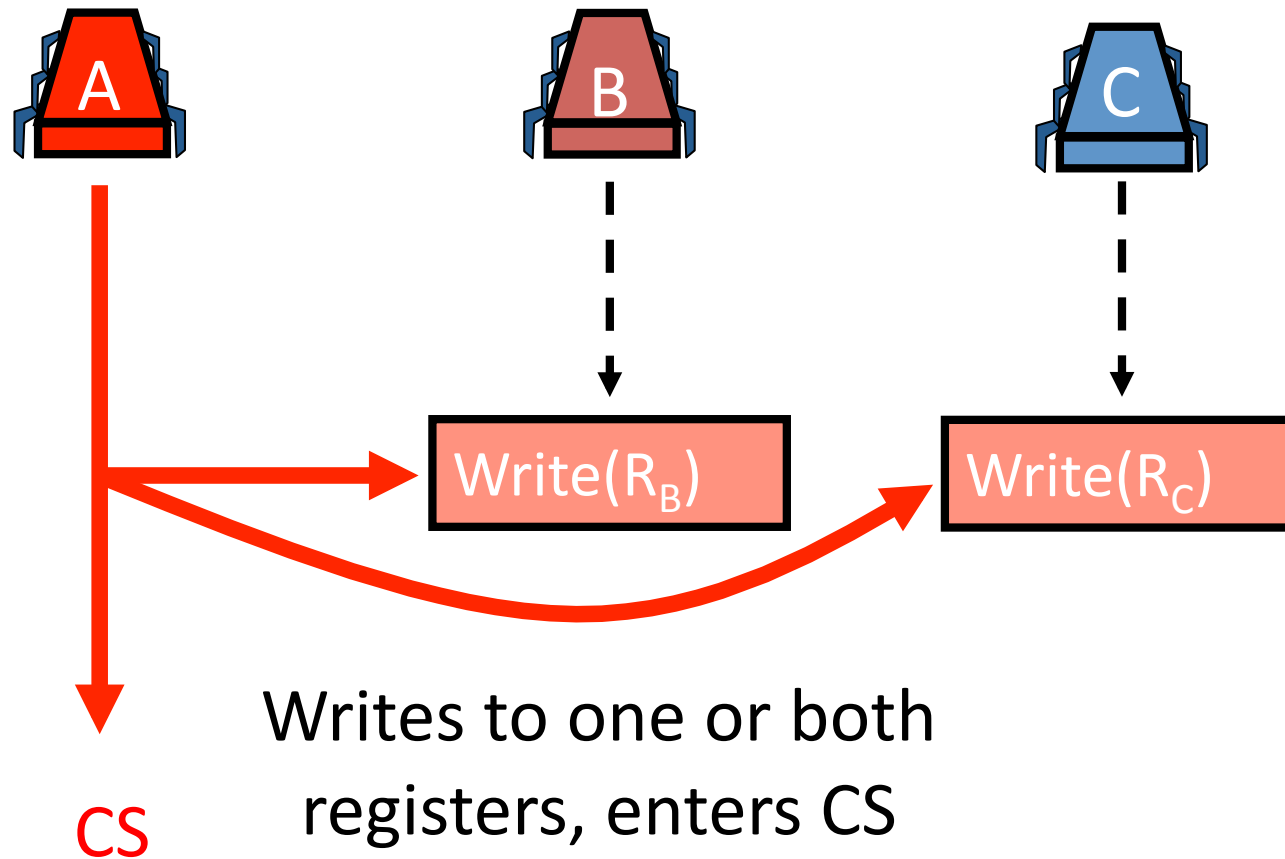
Deadlock-free mutual exclusion for 3 threads requires at least 3 multi-reader multi-writer registers

Proof: Assume Cover of 2

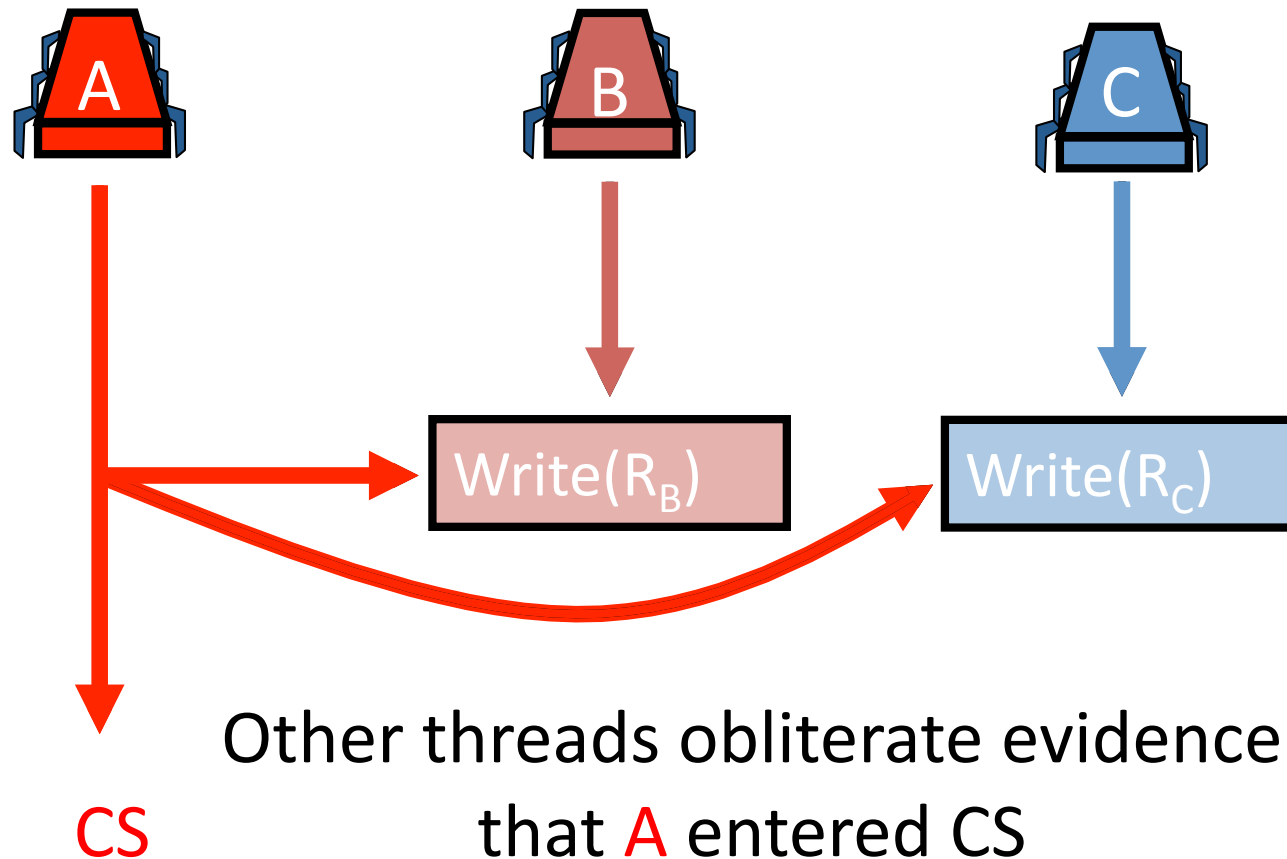


We show later that a cover of 2 can be reached

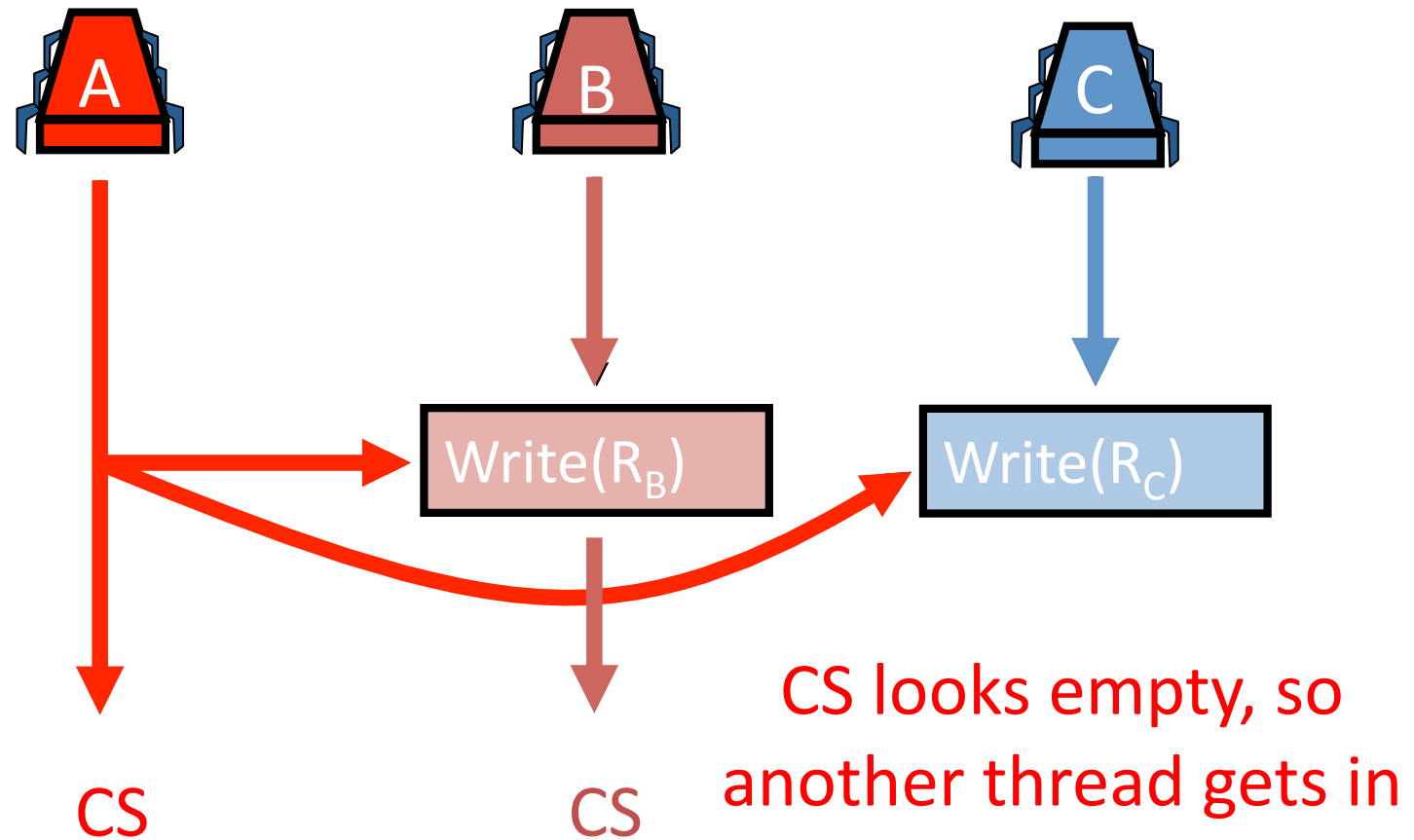
Run A Solo



Obliterate Traces of A



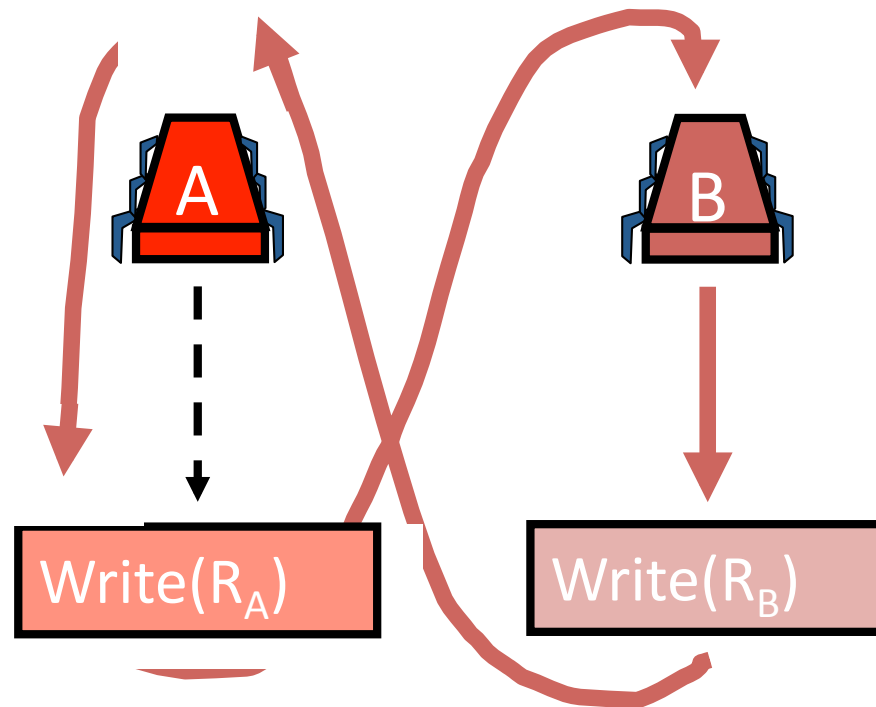
Mutual Exclusion Fails



Covering State Can Be Reached

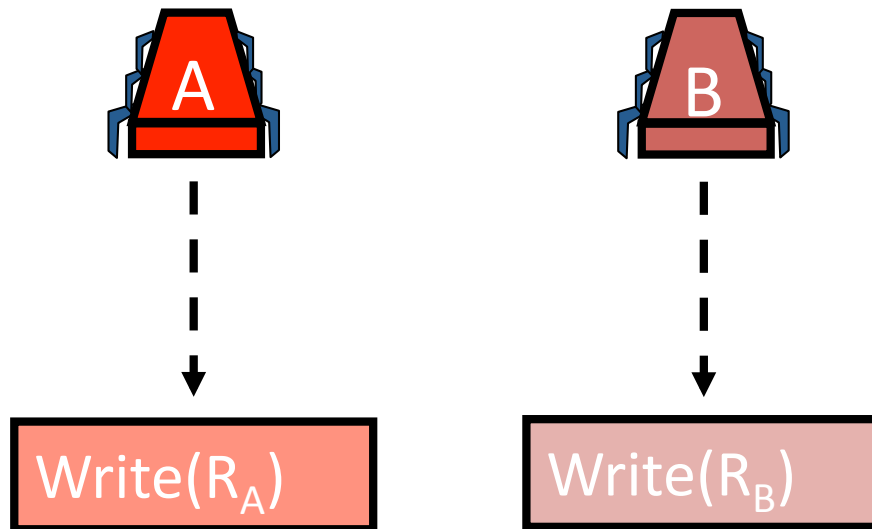
- Proved: a contradiction starting from a covering state for 2 registers
- Claim: a covering state for 2 registers is reachable from any state where CS is empty
- Problem:
 - Registers may have multiple writers
 - So threads may interfere
 - So how can we be sure that two threads may each cover a register at the same time?

Covering State for Two



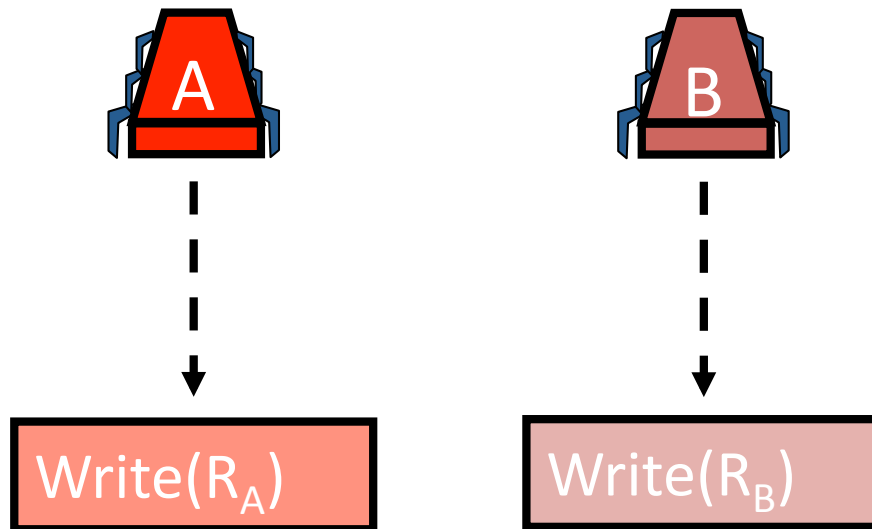
If we run **B** through CS 3 times, **B** must return twice to cover some register, say R_B

Covering State for Two



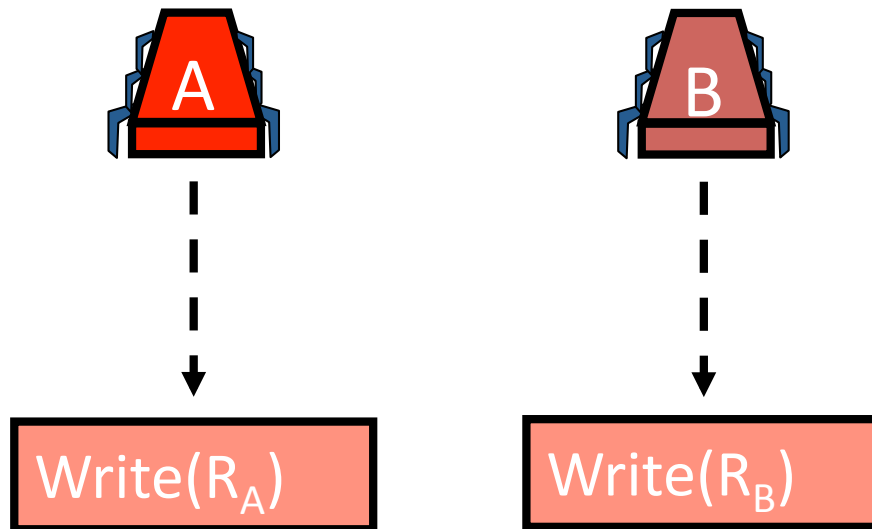
- Start with B covering register R_B for the 1st time
- Run A until about to write to uncovered R_A
 - This must be possible
 - Else let A write R_B only – do that, let A enter CS, let B write R_B
 - This state looks the same to B as one where A did nothing

Covering State for Two



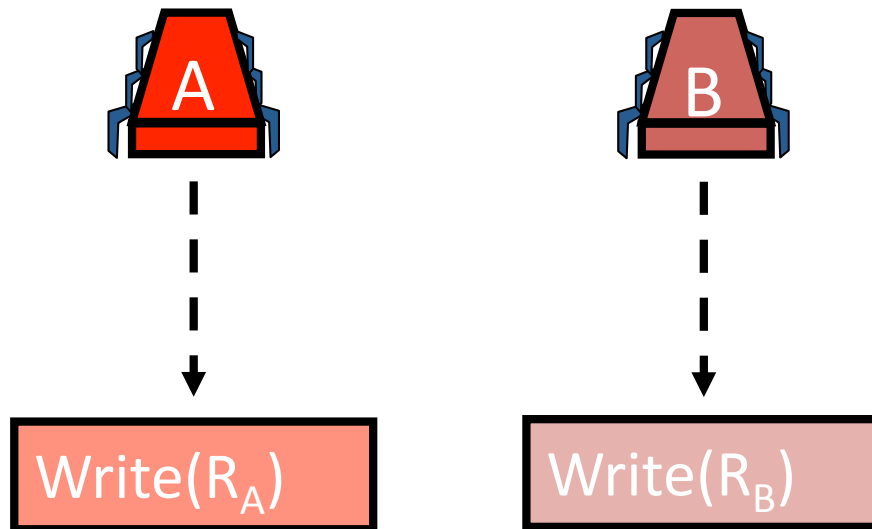
- We were here:
- Start with B covering register R_B for the 1st time
- Run A until about to write to uncovered R_A
- Are we done?

Covering State for Two



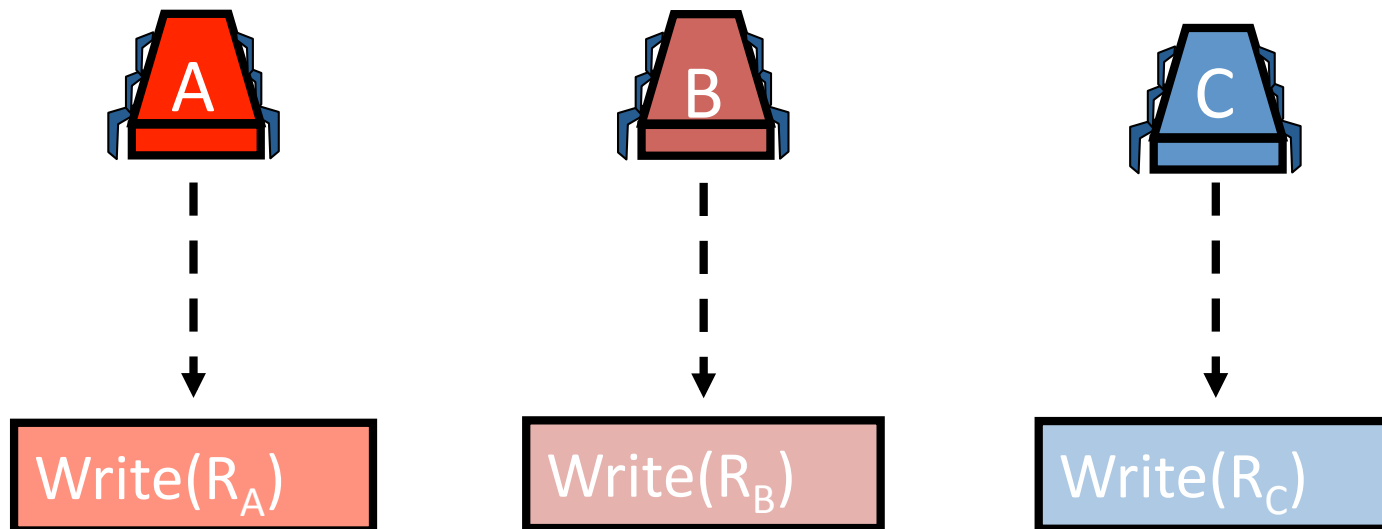
- **NO!** A could have written to R_B
- So CS no longer looks empty
- This could prevent C from entering CS!

Covering State for Two



- Run **B** obliterating traces of **A** in R_B
- Run **B** again until it is about to write to R_B
- Remember **B** wrote R_B then (maybe) R_A then R_B again
- After writing to R_A , **B** leaves CS so CS looks empty to **C**
- Now we are done

Inductively We Can Show



- There is a covering state
 - Where k threads not in CS cover k distinct registers
 - Proof follows when $k = n - 1$

Summary of Lecture

- In the 1960's many **incorrect** solutions to starvation-free mutual exclusion using RW-registers were published...
- Today we know how to solve FIFO n thread mutual exclusion using $2n$ RW-Registers
- This is optimal within factor 2
- But n registers for n thread mutual exclusion is not a scalable solution
- Solution: Hardware supported atomic registers
- Coming up: Which registers, what can they do?