

Mutual Exclusion

Companion slides for
The Art of Multiprocessor
Programming
by Maurice Herlihy & Nir Shavit

Mutual Exclusion



- Today we will try to formalize our understanding of mutual exclusion
- We will also use the opportunity to show you how to argue about and prove various properties in an asynchronous concurrent setting

Mutual Exclusion



- Formal problem definitions
- Solutions for 2 threads
- Solutions for n threads
- Fair solutions
- Inherent costs

Warning

- 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

Why is Concurrent Programming so Hard?

- Try preparing a seven-course banquet
 - By yourself
 - With one friend
 - With twenty-seven friends ...
- Before we can talk about programs
 - Need a language
 - Describing time and concurrency

Time

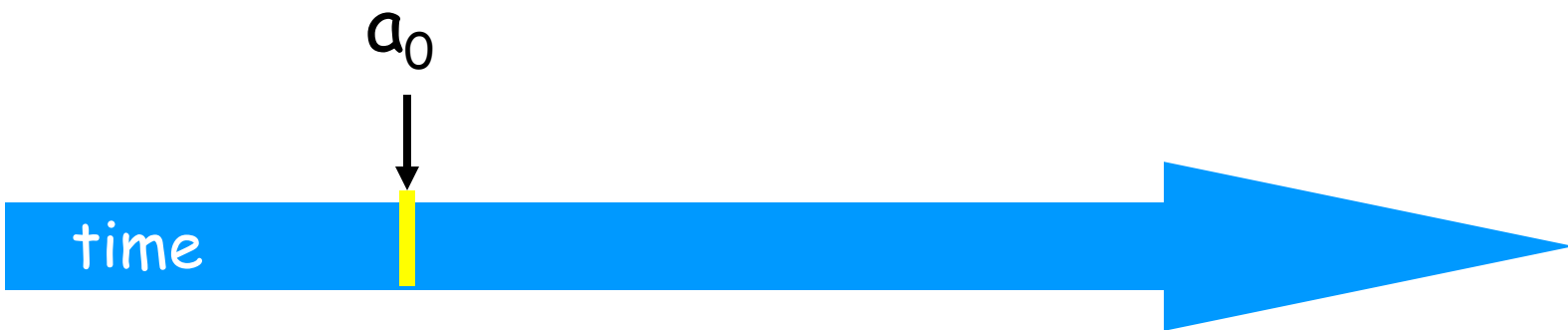
- "Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external." (I. Newton, 1689)
- "Time is, like, Nature's way of making sure that everything doesn't happen all at once." (Anonymous, circa 1968)



time

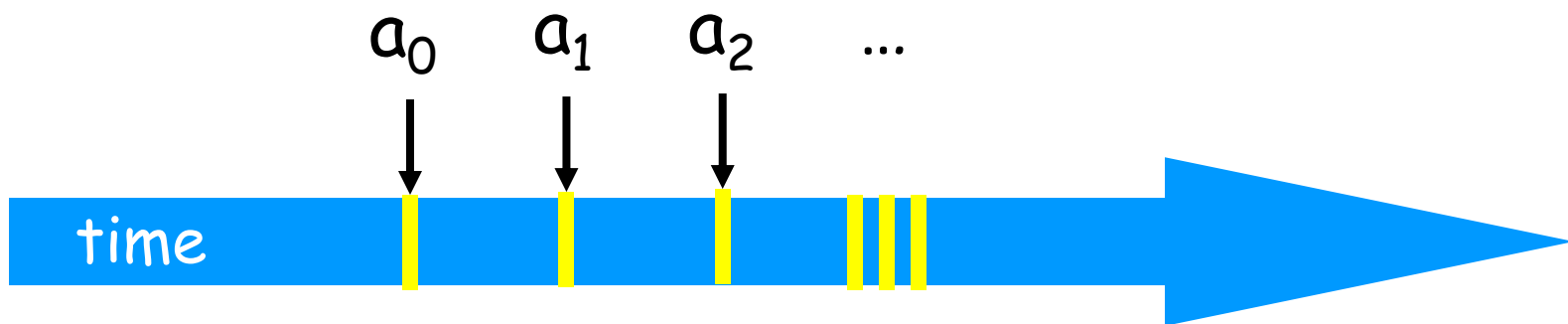
Events

- An *event* a_0 of thread A is
 - Instantaneous
 - No simultaneous events (break ties)



Threads

- A *thread* A is (formally) a sequence a_0, a_1, \dots of events
 - "Trace" model
 - Notation: $a_0 \rightarrow a_1$ indicates order

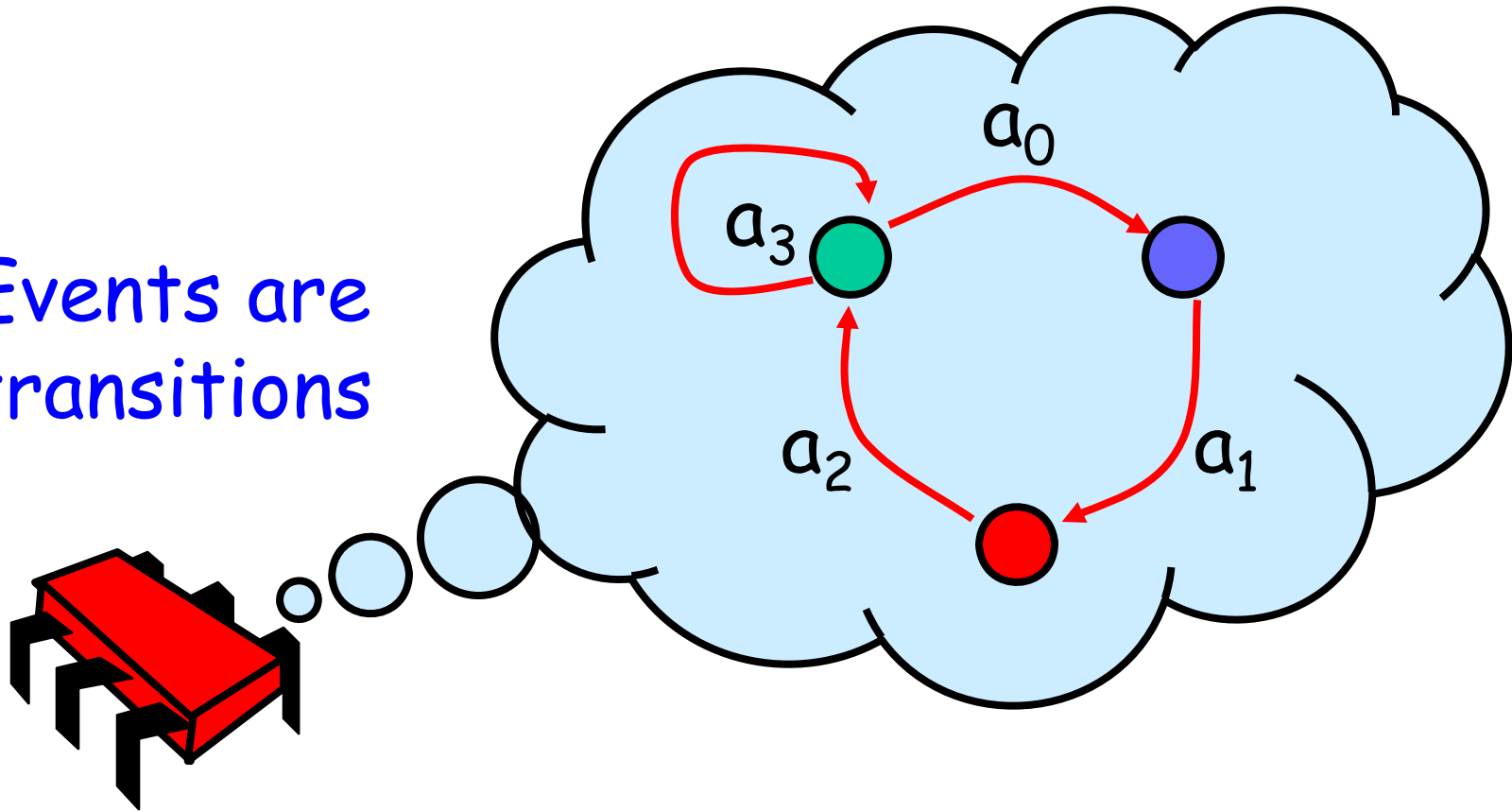


Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things ...

Threads are State Machines

Events are transitions



States

- Thread State
 - Program counter
 - Local variables
- System state
 - Object fields (shared variables)
 - Union of thread states

Concurrency

- Thread A



Concurrency

- Thread A

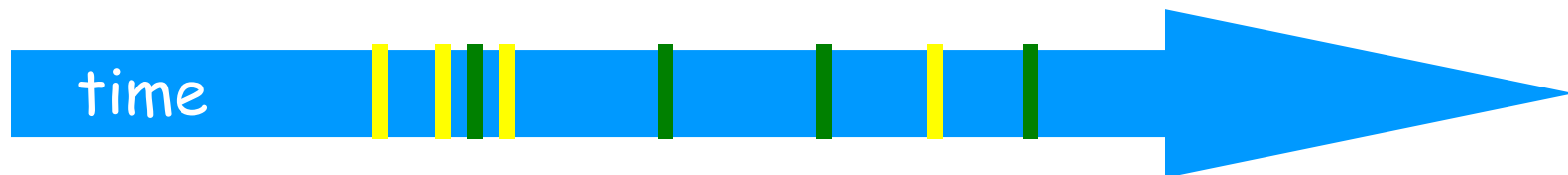


- Thread B



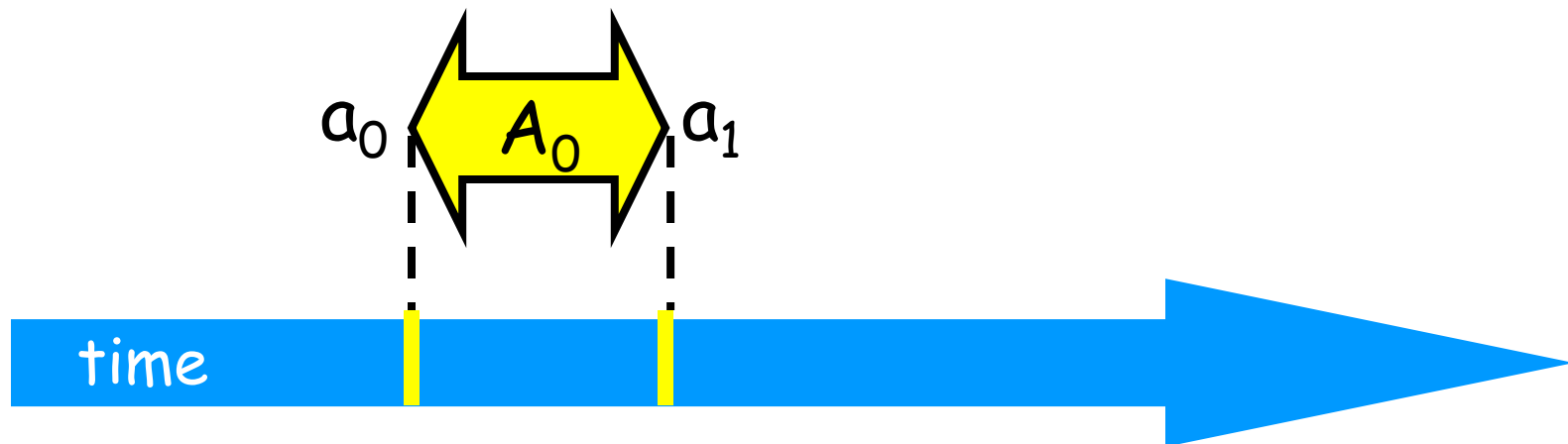
Interleavings

- Events of two or more threads
 - Interleaved
 - Not necessarily independent (why?)

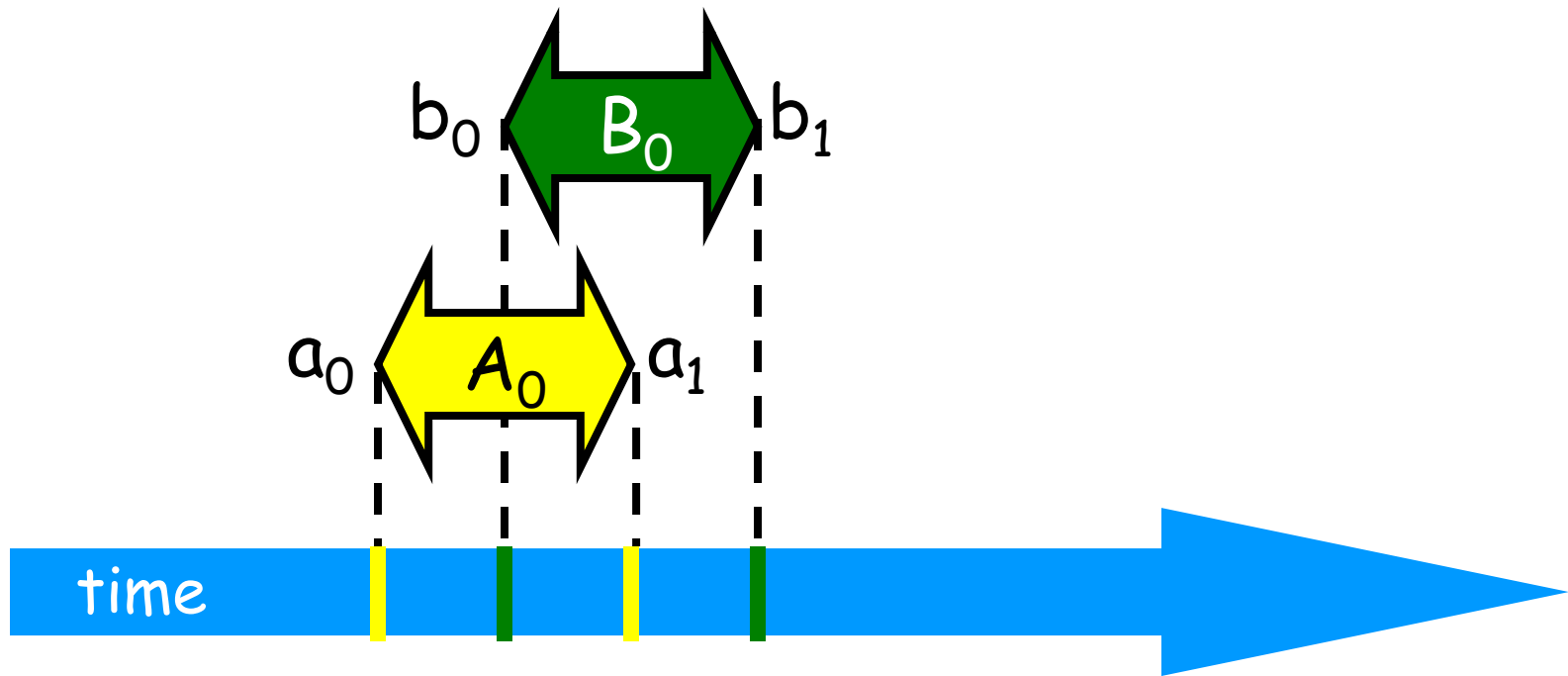


Intervals

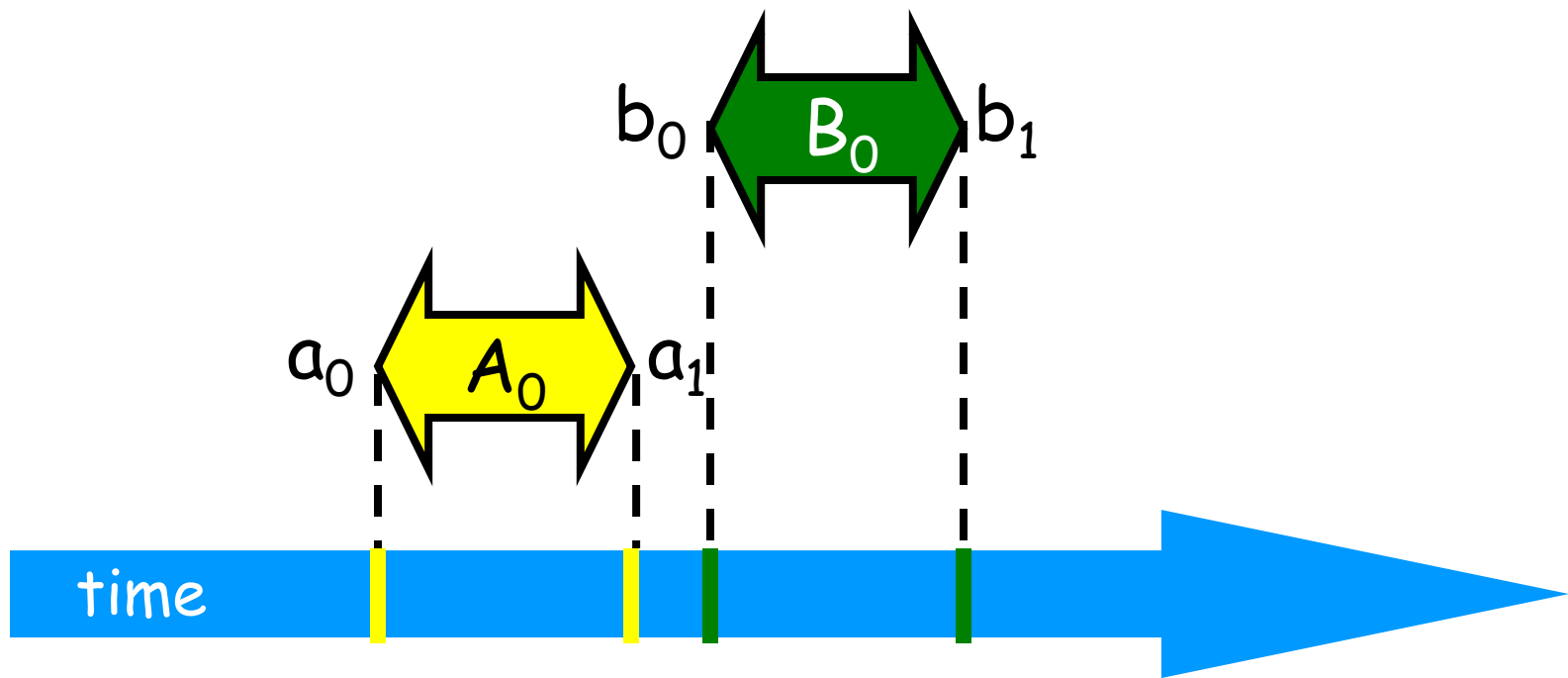
- An *interval* $A_0 = (a_0, a_1)$ is
 - Time between events a_0 and a_1



Intervals may Overlap

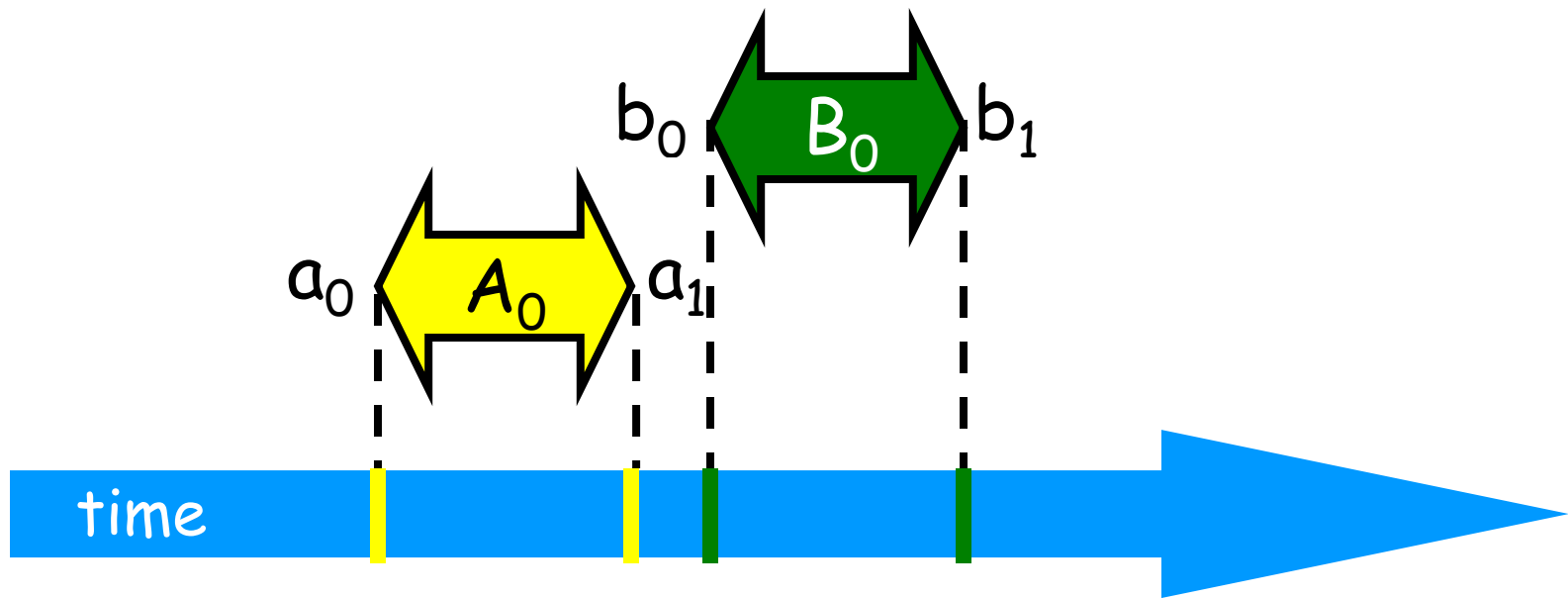


Intervals may be Disjoint

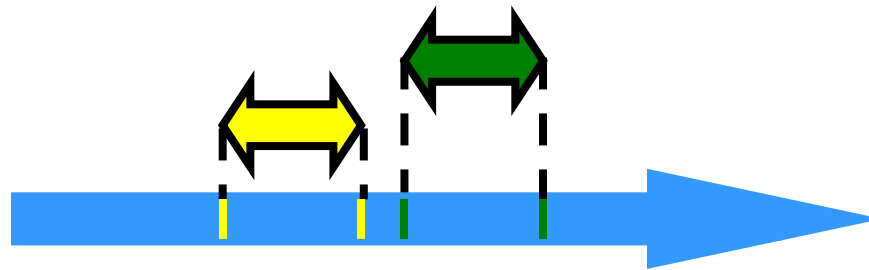


Precedence

Interval A_0 precedes interval B_0

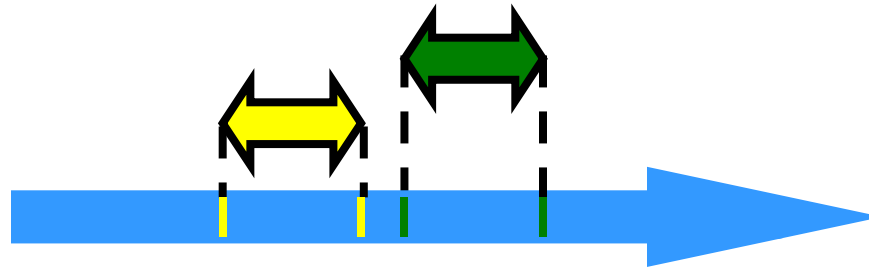


Precedence



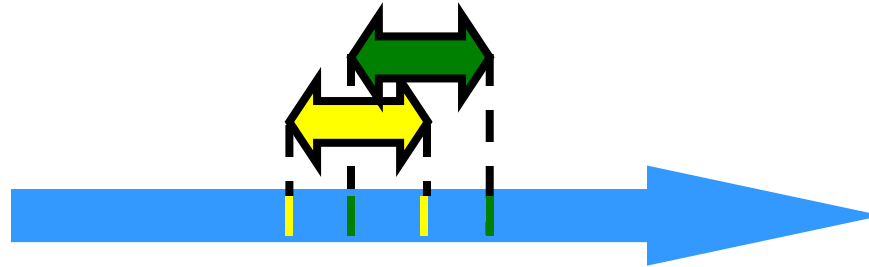
- Notation: $A_0 \rightarrow B_0$
- Formally,
 - End event of A_0 before start event of B_0
 - Also called "happens before" or "precedes"

Precedence Ordering



- Remark: $A_0 \rightarrow B_0$ is just like saying
 - 1066 AD \rightarrow 1492 AD,
 - Middle Ages \rightarrow Renaissance,
- Oh wait,
 - what about this week *vs* this month?

Precedence Ordering



- Never true that $A \rightarrow A$
- If $A \rightarrow B$ then not true that $B \rightarrow A$
- If $A \rightarrow B$ & $B \rightarrow C$ then $A \rightarrow C$
- Funny thing: $A \rightarrow B$ & $B \rightarrow A$ might both be false!

Partial Orders

(you may know this already)

- Irreflexive:
 - Never true that $A \rightarrow A$
- Antisymmetric:
 - If $A \rightarrow B$ then not true that $B \rightarrow A$
- Transitive:
 - If $A \rightarrow B$ & $B \rightarrow C$ then $A \rightarrow C$

Total Orders

(you may know this already)

- Also
 - Irreflexive
 - Antisymmetric
 - Transitive
- Except that for every distinct A, B ,
 - Either $A \rightarrow B$ or $B \rightarrow A$

Repeated Events

```
while (multiple) {  
    a0; a1;  
}
```

k -th occurrence
of event a_0

a_0^k

k -th occurrence of
interval $A_0 = (a_0, a_1)$

A_0^k

Implementing a Counter

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

Make these steps
indivisible using
locks

Locks (Mutual Exclusion)

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

Locks (Mutual Exclusion)

```
public interface Lock {
```

```
    public void lock();
```

acquire lock

```
    public void unlock();
```

```
}
```

Locks (Mutual Exclusion)

```
public interface Lock {
```

```
    public void lock();
```

acquire lock

```
    public void unlock();
```

release lock

```
}
```

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;  
    }  
}
```

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;  
    }  
}
```

lock.lock(); acquire Lock

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;  
    }  
}
```

Release lock
(no matter what)

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;  
    }  
}
```

Critical
section







Mutual Exclusion

- Let CS_i^k  be thread i 's k -th critical section execution



Mutual Exclusion

- Let CS_i^k  be thread i 's k -th critical section execution
- And CS_j^m  be thread j 's m -th critical section execution

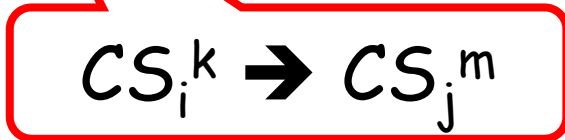
Mutual Exclusion

- Let CS_i^k  be thread i 's k -th critical section execution
- And CS_j^m  be j 's m -th execution
- Then either
 -   or  

Mutual Exclusion

- Let CS_i^k  be thread i 's k -th critical section execution
- And CS_j^m  be j 's m -th execution
- Then either

-   or  



Mutual Exclusion

- Let CS_i^k \leftrightarrow be thread i 's k -th critical section execution
- And CS_j^m \leftrightarrow be j 's m -th execution
- Then either

- \leftrightarrow \leftrightarrow or \leftrightarrow \leftrightarrow

$CS_i^k \rightarrow CS_j^m$

$CS_i^m \rightarrow CS_i^k$

Deadlock-Free



- If some thread calls **lock()**
 - And never returns
 - Then other threads must complete **lock()** and **unlock()** calls infinitely often
- System as a whole makes progress
 - Even if individuals starve

Starvation-Free



- If some thread calls `lock()`
 - It will eventually return
- Individual threads make progress

Two-Thread vs n -Thread Solutions

- Two-thread solutions first
 - Illustrate most basic ideas
 - Fits on one slide
- Then n -Thread solutions

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 volatile boolean[] flag =  
        new boolean[2];  
  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

LockOne

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

Set my flag

LockOne

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

Set my flag

**Wait for other
flag to go false**

LockOne Satisfies Mutual Exclusion

- Assume CS_A^j overlaps CS_B^k
- Consider each thread's last (j-th and k-th) read and write in the lock() method before entering
- Derive a contradiction

From the Code

- $\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow$
 $\text{read}_A(\text{flag}[B]==\text{false}) \rightarrow CS_A$
- $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow$
 $\text{read}_B(\text{flag}[A]==\text{false}) \rightarrow CS_B$

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

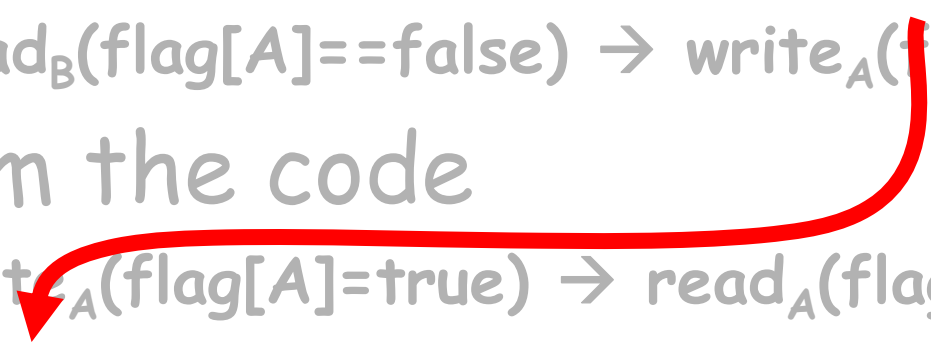
From the Assumption

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$
- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[B] = \text{true})$

Combining

- Assumptions:
 - $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$
 - $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$
- From the code
 - $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$
 - $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

- Assumptions:
 - $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$
 - $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$
 - From the code
 - $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$
 - $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$
- 

Combining

- Assumptions:

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

- Assumptions:

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

- Assumptions:

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

- Assumptions:

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

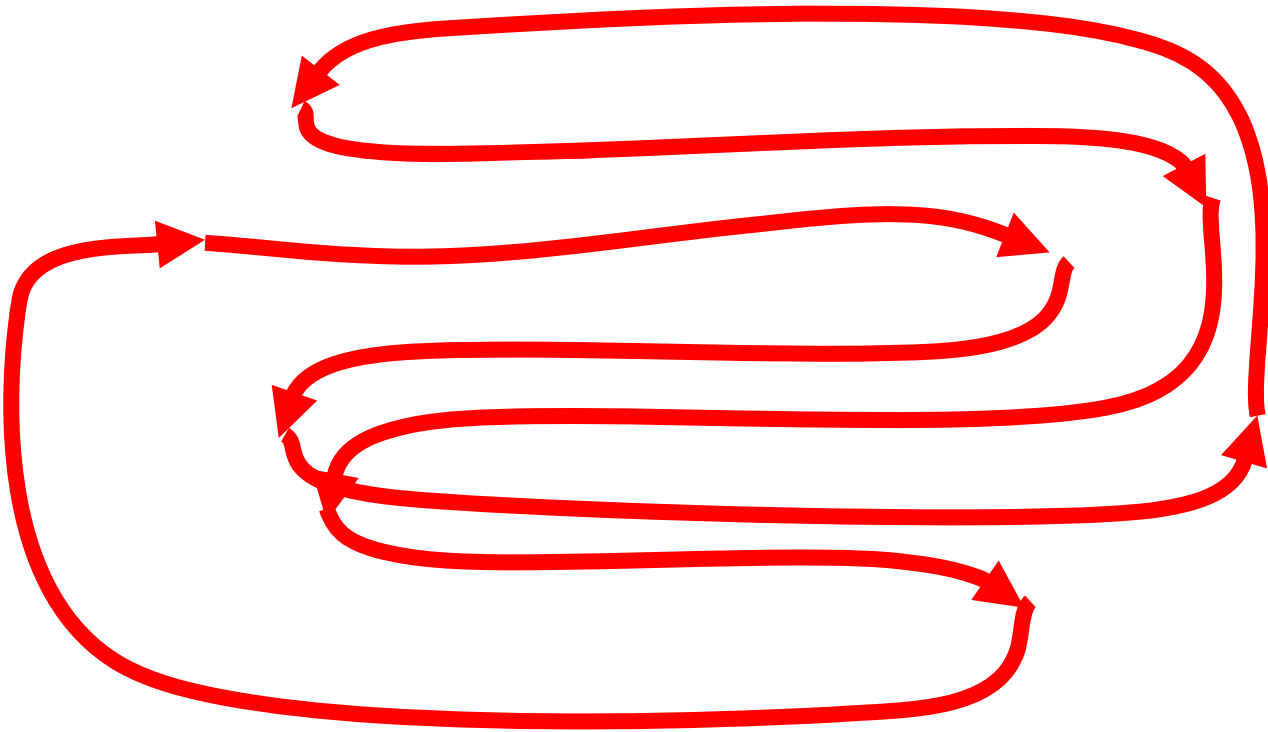
- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Cycle!



LockTwo

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

LockTwo

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

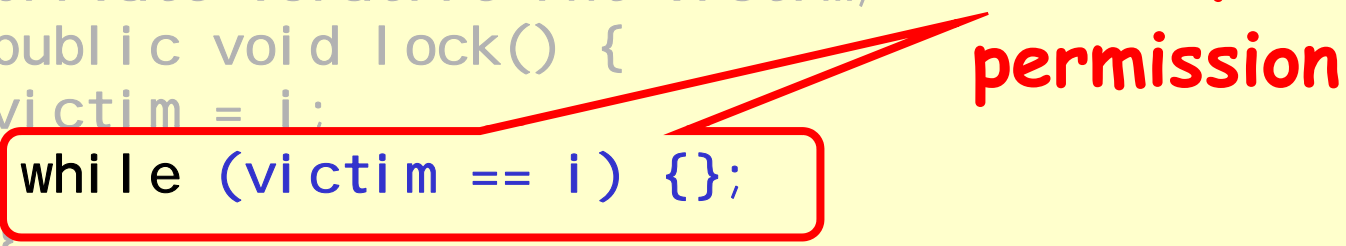
    public void unlock() {}
}
```

Let other go first

LockTwo

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

Wait for permission



LockTwo

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

Nothing to do



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

LockTwo Claims

- Satisfies mutual exclusion
 - If thread i in CS
 - 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() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};  
}  
public void unlock() {  
    flag[i] = false;  
}
```

Peterson's Algorithm

Announce I'm
interested

```
public void Lock() {  
    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

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

Mutual Exclusion

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

- If thread **0** in critical section,
 - flag[0]=true,
 - victim = 1
- If thread **1** in critical section,
 - flag[1]=true,
 - victim = 0

Cannot both be true

Deadlock Free

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

- Thread blocked
 - only at while loop
 - only if it is the victim
- One or the other must not be the victim

Starvation Free

- Thread i blocked only if j repeatedly re-enters so that

$flag[j] == true$ and
 $victim == i$

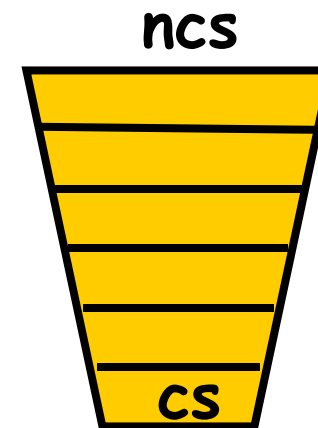
- When j re-enters
 - it sets $victim$ to j .
 - So i gets in

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

The Filter Algorithm for n Threads

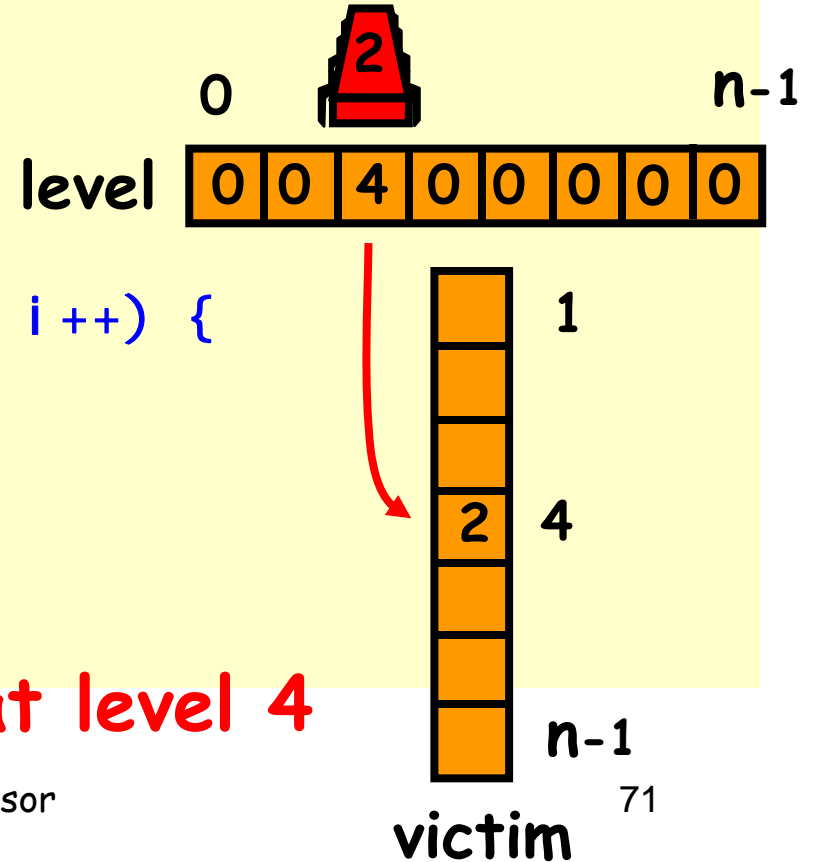
There are $n-1$ "waiting rooms" called levels

- At each level
 - At least one enters level
 - At least one blocked if many try
- Only one thread makes it through



Filter

```
class Filter implements Lock {  
    volatile int[] level; // level[i] for thread i  
    volatile int[] victim; // victim[L] for level L  
  
    public Filter(int n) {  
        level = new int[n];  
        victim = new int[n];  
        for (int i = 1; i < n; i++) {  
            level[i] = 0;  
        }  
        ...  
    }  
}
```



Thread 2 at level 4

Filter

```
class Filter implements Lock {  
    ...  
  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
  
            while (( $\exists k \neq i$  level[k] >= L) &&  
                victim[L] == i );  
  
        }  
    }  
  
    public void unlock() {  
        level[i] = 0;  
    }  
}
```

Filter

```
class Filter implements Lock {
    ...

    public void Lock() {
        for (int L = 1; L < n; L++) {
            level[i] = L;
            victim[L] = i;
            while (( $\exists k \neq i$ ) level[k] >= L) &&
                victim[L] == i);
        }
    }

    public void release(int i) {
        level[i] = 0;
    }
}
```

One level at a time

Filter

```
class Filter implements Lock {
```

```
...
```

```
public void lock() {
```

```
    for (int L = 1; L < n; L++) {
```

```
        level[i] = L;
```

```
        victim[L] = i;
```

```
        while (( $\exists k \neq i$ ) level[k] >= L) &&
```

```
                victim[L] == i);
```

```
    }
```

```
public void release(int i)
```

```
    level[i] = 0;
```

```
}
```

Announce
intention to
enter level L

Filter

```
class Filter implements Lock {
    int level[n];
    int victim[n];
    public void lock() {
        for (int L = 1; L < n; L++) {
            level[i] = L;
            victim[L] = i;
            while (( $\exists k \neq i$ ) level[k] >= L) &&
                victim[L] == i);
        }
    }
    public void release(int i) {
        level[i] = 0;
    }
}
```

Give priority to anyone but me

Filter

Wait as long as someone else is at same or higher level, and I'm designated victim

```
public void lock() {  
    for (int L = 1; L < n; L++) {  
        level[i] = L;  
        victim[L] = i;  
        while (( $\exists k \neq i$ ) level[k] >= L) &&  
            victim[L] == i);  
    }  
}  
public void release(int i) {  
    level[i] = 0;  
}
```

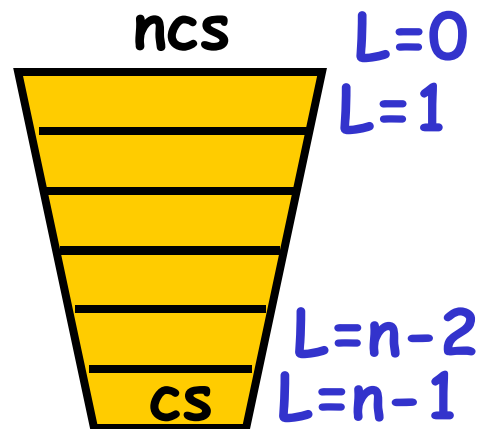
Filter

```
class Filter implements Lock {  
    int level[n];  
    int victim[n];  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists k \neq i$ ) level[k] >= L) &&  
                victim[L] == i);  
        }  
    }  
}
```

Thread enters level L when it completes the loop

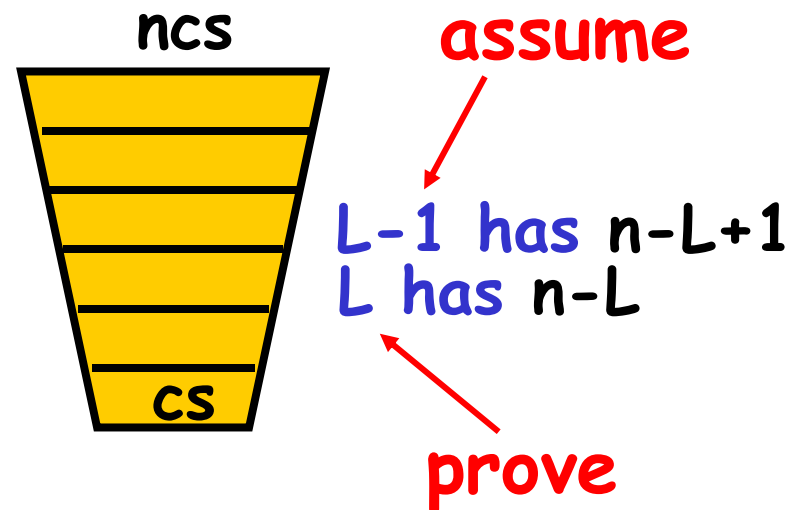
Claim

- Start at level $L=0$
- At most $n-L$ threads enter level L
- Mutual exclusion at level $L=n-1$

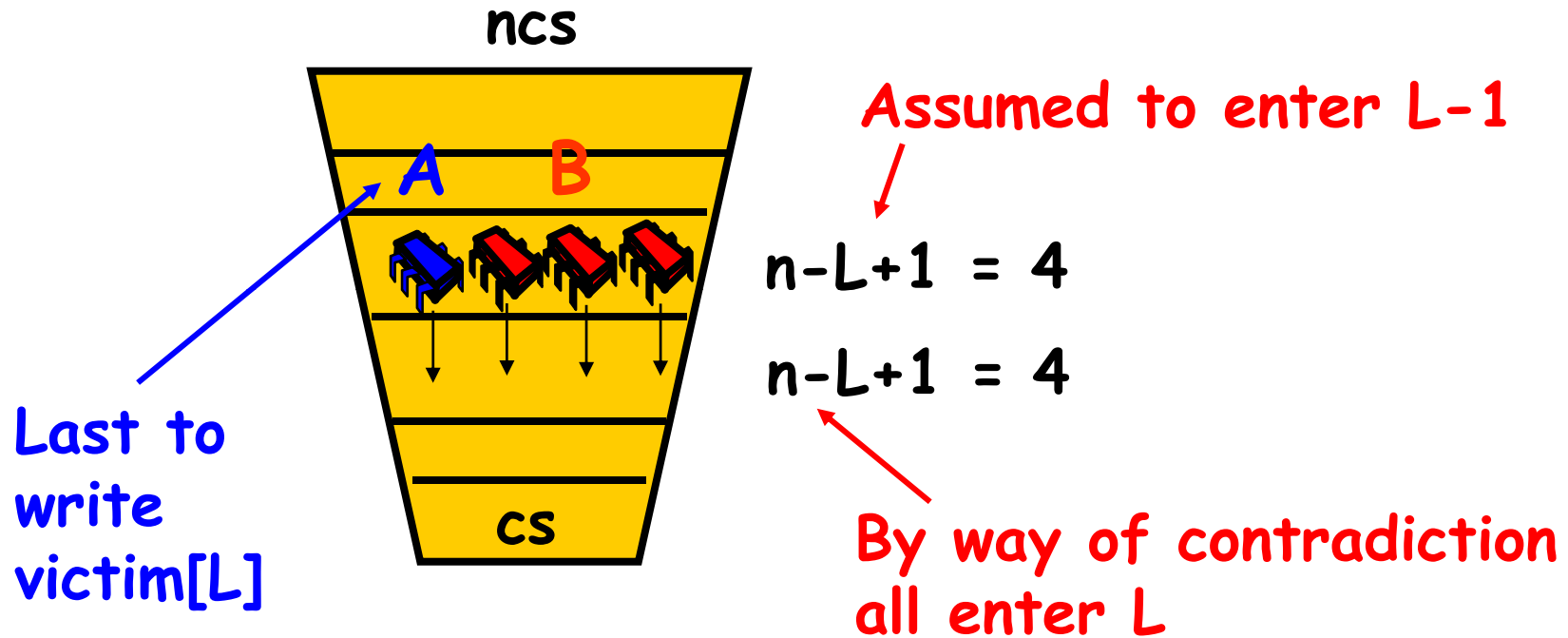


Induction Hypothesis

- No more than $n-L+1$ at level $L-1$
- Induction step: by contradiction
- Assume all at level $L-1$ enter level L
- A last to write `victim[L]`
- B is any other thread at level L



Proof Structure



Show that A must have seen B at level L and since $\text{victim}[L] == A$ could not have entered

From the Code

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B)$

```
public void lock() {  
    for (int L = 1; L < n; L++) {  
        level[i] = L;  
        victim[L] = i;  
        while (( $\exists k \neq i$ ) level[k] >= L  
            && victim[L] == i) {};  
    }  
}
```

From the Code

(2) $\text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B])$

```
public void lock() {  
    for (int L = 1; L < n; L++) {  
        level[i] = L;  
        victim[L] = i;  
        while (( $\exists k \neq i$ ) level[k] >= L  
            && victim[L] == i) {};  
    }  
}
```

By Assumption

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

By assumption, A is the last thread to write **victim[L]**

Combining Observations

- (1) $\text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B)$
- (3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$
- (2) $\text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B])$

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2) $\rightarrow \text{read}_A(\text{level}[B])$

```
public void lock() {
    for (int L = 1; L < n; L++) {
        level[i] = L;
        victim[L] = i;
        while (( $\exists k \neq i$ ) level[k] >= L
            && victim[L] == i) {};
    }
}
```

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2)

$\rightarrow \text{read}_A(\text{level}[B])$

**Thus, A read $\text{level}[B] \geq L$,
A was last to write $\text{victim}[L]$,
so it could not have entered level L!**

No Starvation

- Filter Lock satisfies properties:
 - Just like Peterson Alg at any level
 - So no one starves
- But what about fairness?
 - Threads can be overtaken by others

Bounded Waiting

- Want stronger fairness guarantees
- Thread not "overtaken" too much
- Need to adjust definitions

Bounded Waiting

- Divide `lock()` method into 2 parts:
 - Doorway interval:
 - Written D_A
 - always finishes in finite steps
 - Waiting interval:
 - Written W_A
 - may take unbounded steps

r -Bounded Waiting

- For threads A and B :
 - If $D_A^k \rightarrow D_B^j$
 - A 's k -th doorway precedes B 's j -th doorway
 - Then $CS_A^k \rightarrow CS_B^{j+r}$
 - A 's k -th critical section precedes B 's $(j+r)$ -th critical section
 - B cannot overtake A by more than r times
- First-come-first-served means $r = 0$.

Fairness Again

- Filter Lock satisfies properties:
 - No one starves
 - But very weak fairness
 - Not r -bounded for any r !
 - That's pretty lame...

Bakery Algorithm

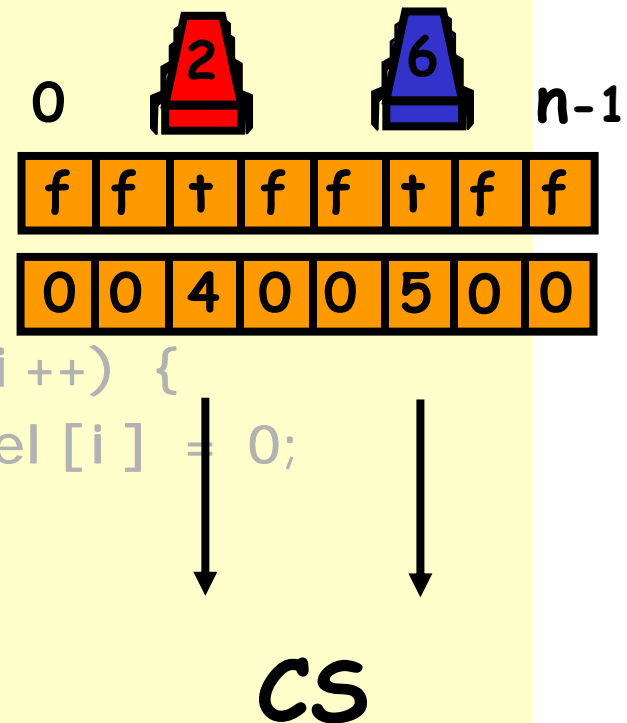
- Provides First-Come-First-Served
- How?
 - Take a "number"
 - Wait until lower numbers have been served
- Lexicographic order
 - $(a,i) > (b,j)$
 - If $a > b$, or $a = b$ and $i > j$

Bakery Algorithm

```
class Bakery implements Lock {
    volatile boolean[] flag;
    volatile 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 {  
    volatile boolean[] flag;  
    volatile 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 {  
    ...  
    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));  
    }  
}
```

Doorway

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 ( $\exists k$  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));  
    }
```

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

With lower (label,i)
in lexicographic order

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

No Deadlock

- There is always one thread with earliest label
- Ties are impossible (why?)

First-Come-First-Served

- If $D_A \rightarrow D_B$ then A's label is earlier
 - $\text{write}_A(\text{label}[A]) \rightarrow$
 $\text{read}_B(\text{label}[A]) \rightarrow$
 $\text{write}_B(\text{label}[B]) \rightarrow$
 $\text{read}_B(\text{flag}[A])$
- So B is locked out while $\text{flag}[A]$ is true

```
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));  
    }  
}
```

Mutual Exclusion

- Suppose **A** and **B** in CS together
- Suppose **A** has earlier label
- When **B** entered, it must have seen
 - `flag[A]` is *false*, or
 - `label[A] > label[B]`

```
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));  
    }  
}
```

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen `flag[A] == false`

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen $\text{flag}[A] == \text{false}$
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen $\text{flag}[A] == \text{false}$
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$
- Which contradicts the assumption that A has an earlier label

Bakery Y2³²K Bug

```
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));  
    }  
}
```

Bakery Y2³²K Bug

```
class Bakery implements Lock { Mutex breaks if  
    ... label [i] overflows  
    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));  
    }
```

Does Overflow Actually Matter?

- Yes
 - Y2K
 - 18 January 2038 (Unix `time_t` rollover)
 - 16-bit counters
- No
 - 64-bit counters
- Maybe
 - 32-bit counters


Timestamps

- Label variable is really a **timestamp**
- Need ability to
 - Read others' timestamps
 - Compare them
 - Generate a **later** timestamp
- Can we do this without overflow?

The Good News

- One can construct a
 - Wait-free (no mutual exclusion)
 - Concurrent
 - Timestamping system
 - That never overflows

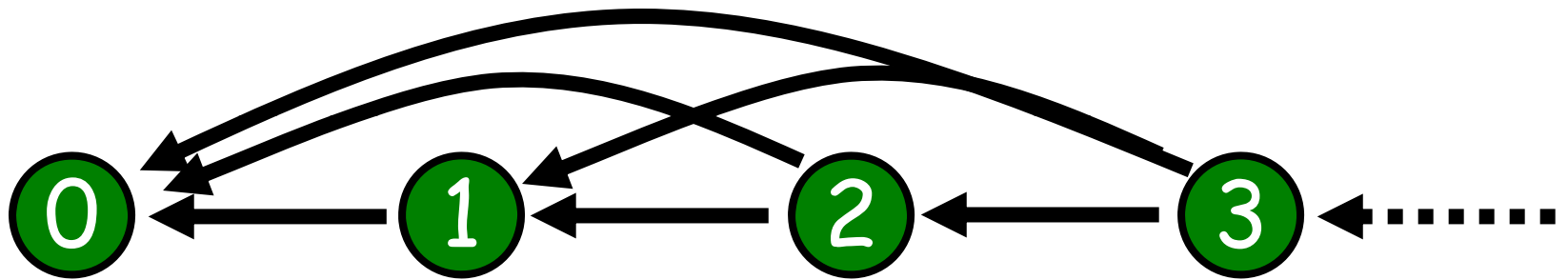
The **Bad** News

- One can construct a
 - Wait-free (no mutual exclusion)
 - Concurrent  This part is hard
 - Timestamping system
 - That never overflows

Instead ...

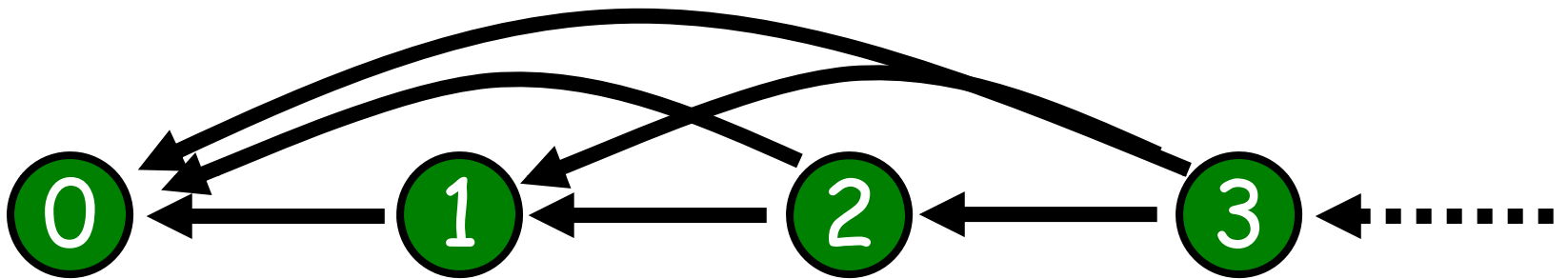
- We construct a **Sequential** timestamping system
 - Same basic idea
 - But simpler
- Uses mutex to read & write atomically
- No good for building locks
 - But useful anyway

Precedence Graphs



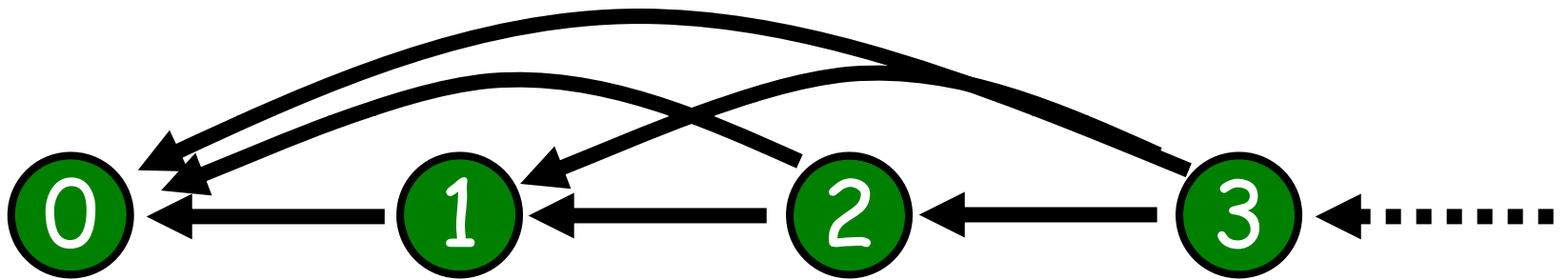
- Timestamps form directed graph
- Edge x to y
 - Means x is later timestamp
 - We say x **dominates** y

Unbounded Counter Precedence Graph

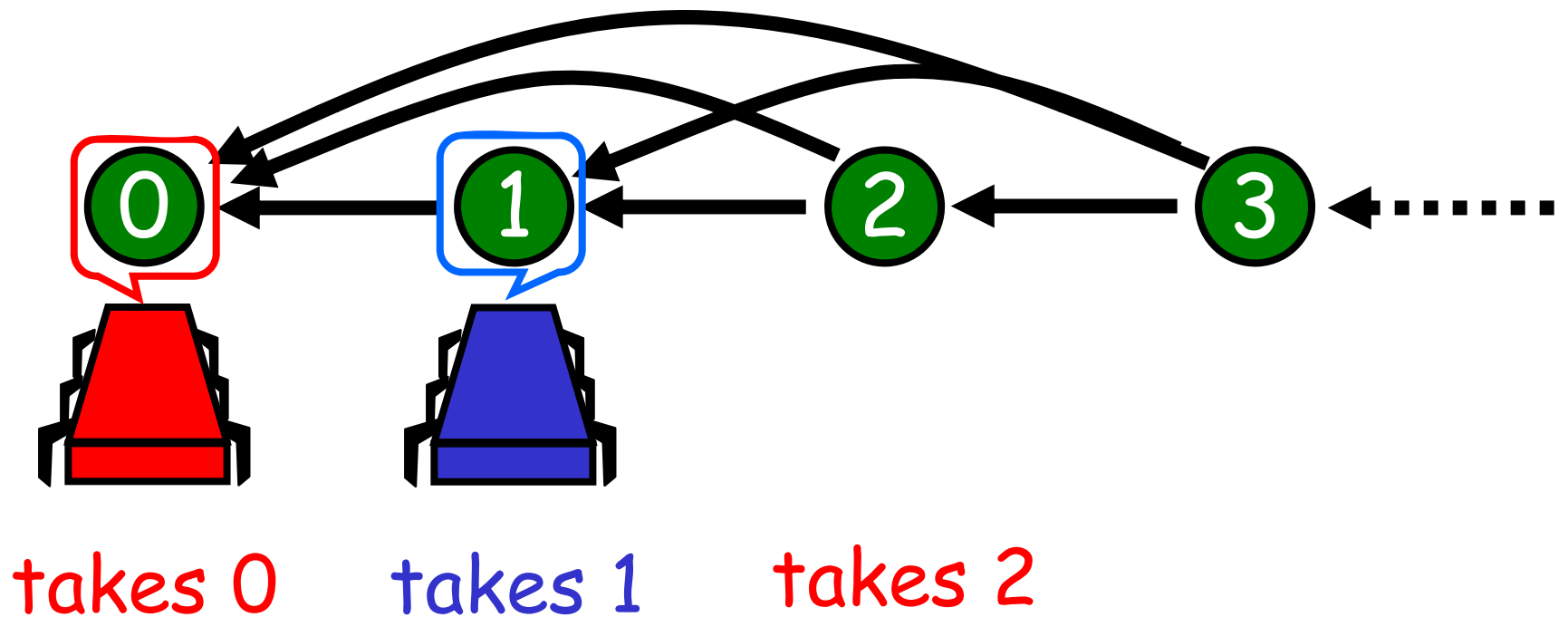


- Timestamping = move tokens on graph
- Atomically
 - read others' tokens
 - move mine
- Ignore tie-breaking for now

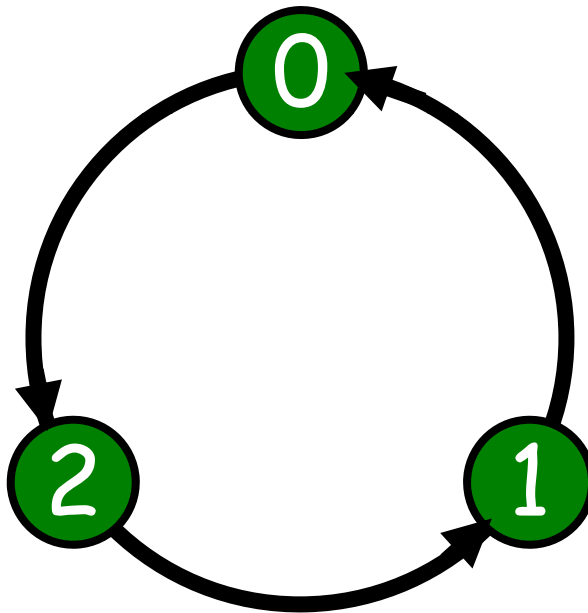
Unbounded Counter Precedence Graph



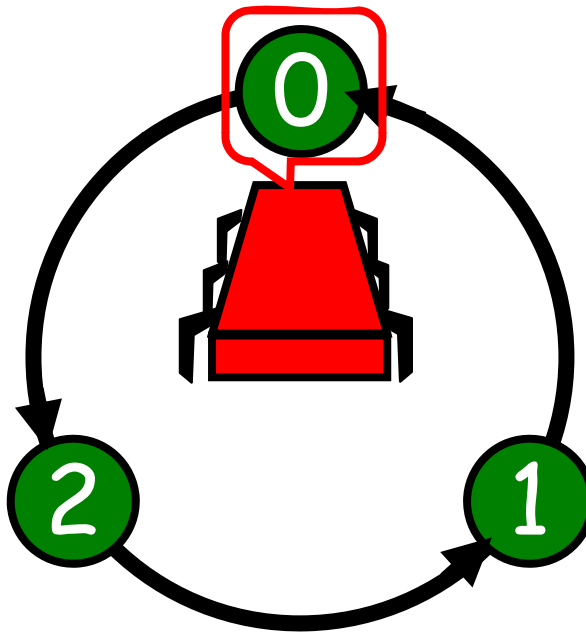
Unbounded Counter Precedence Graph



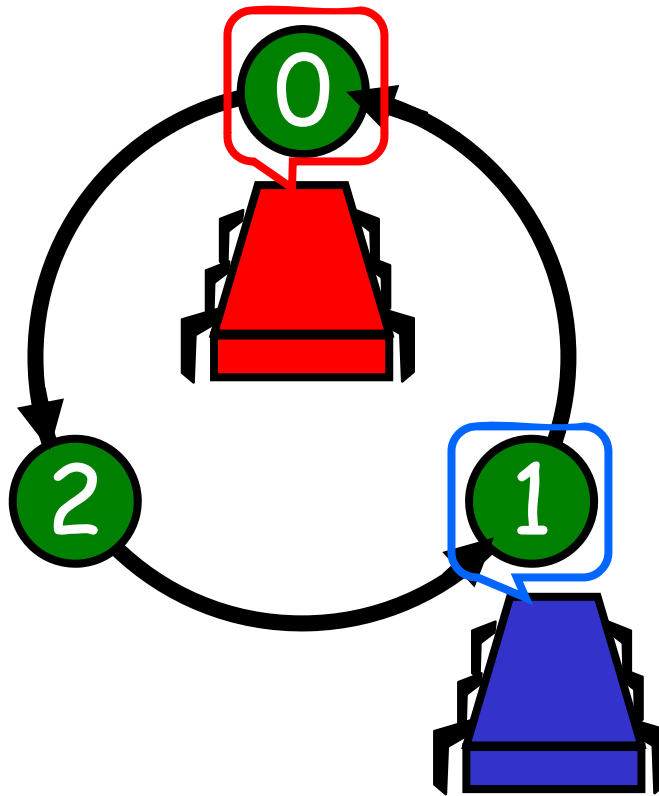
Two-Thread Bounded Precedence Graph



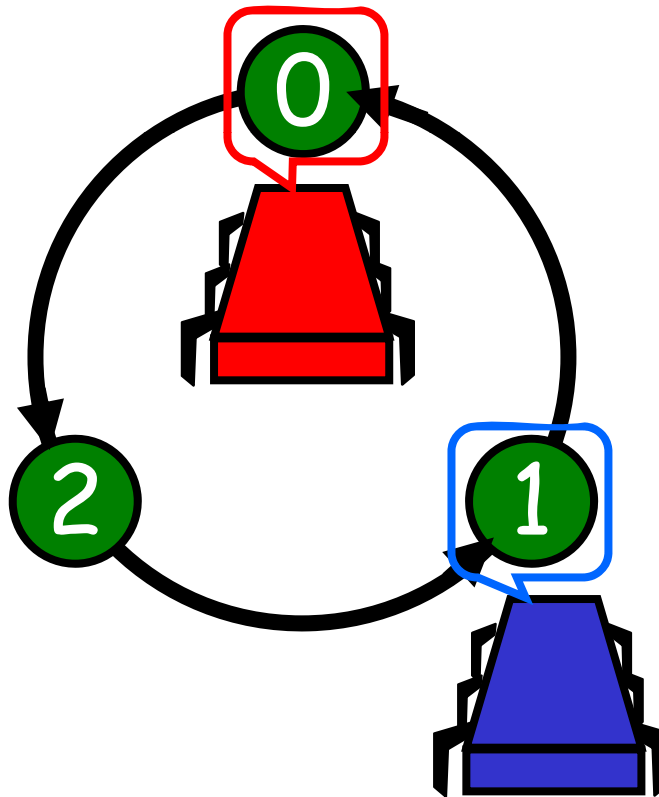
Two-Thread Bounded Precedence Graph



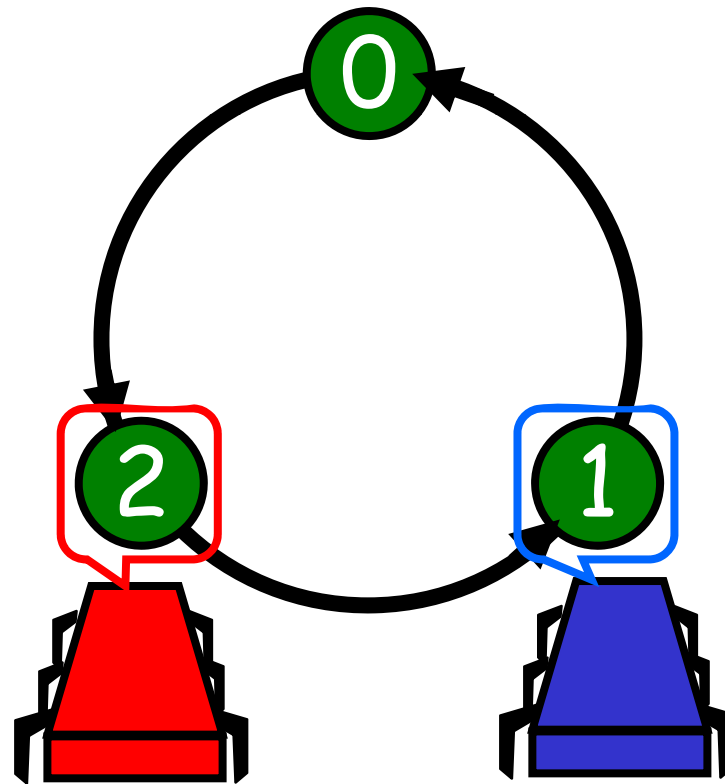
Two-Thread Bounded Precedence Graph



Two-Thread Bounded Precedence Graph

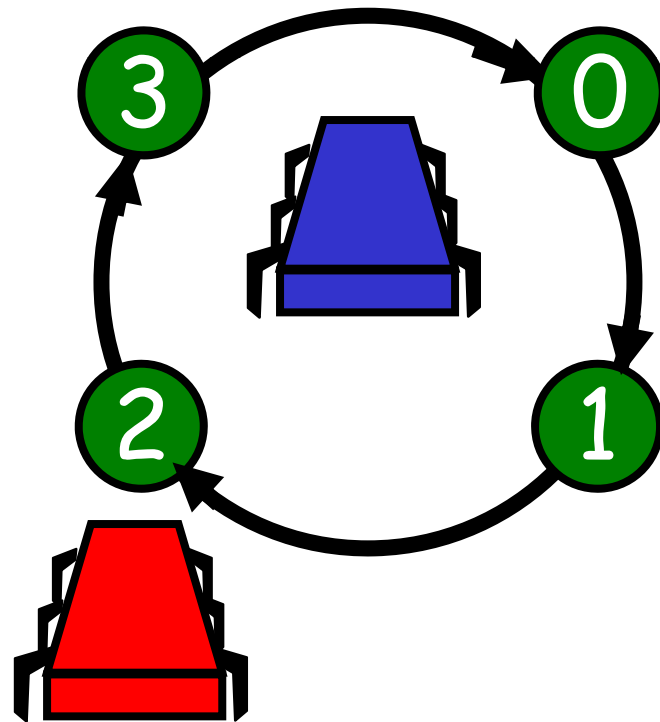


Two-Thread Bounded Precedence Graph T^2

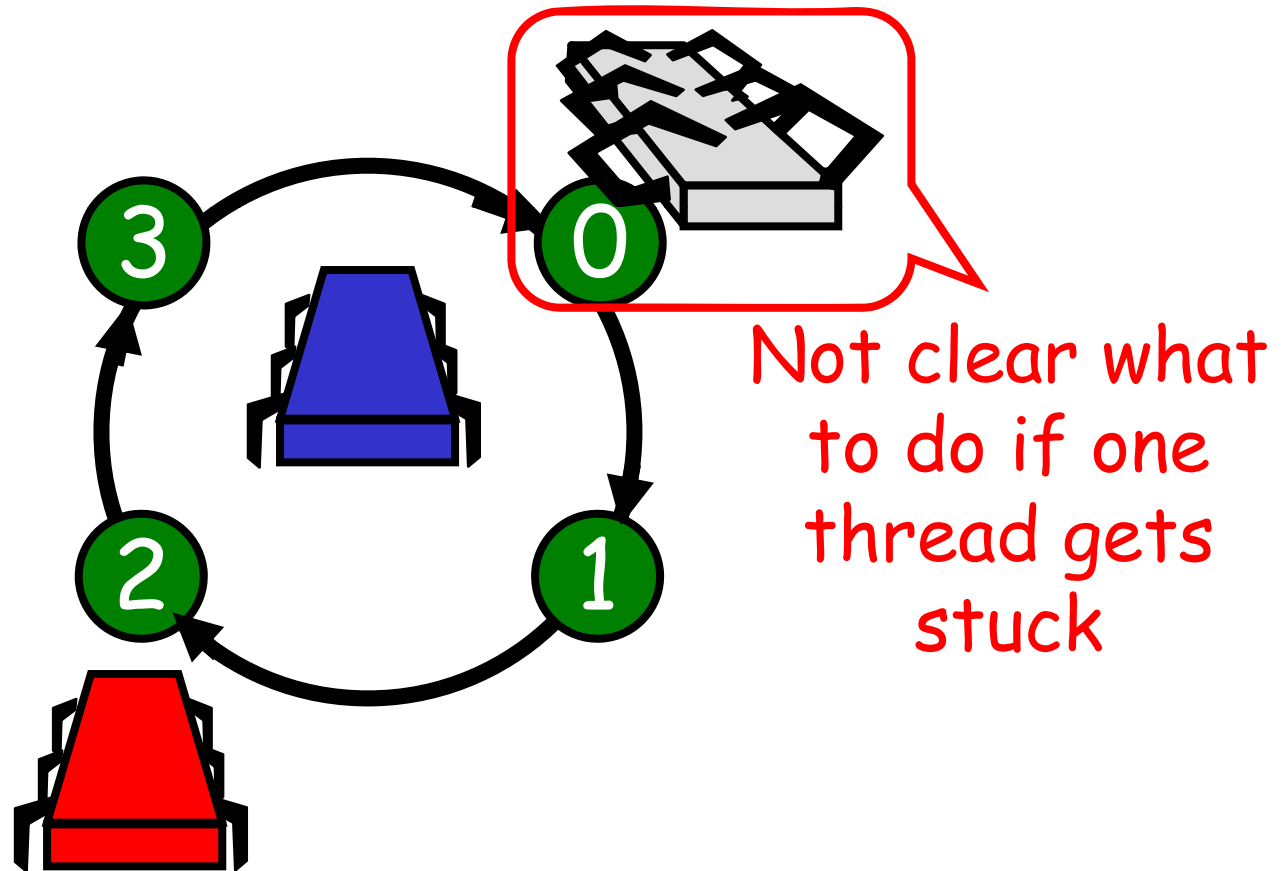


and so on ...

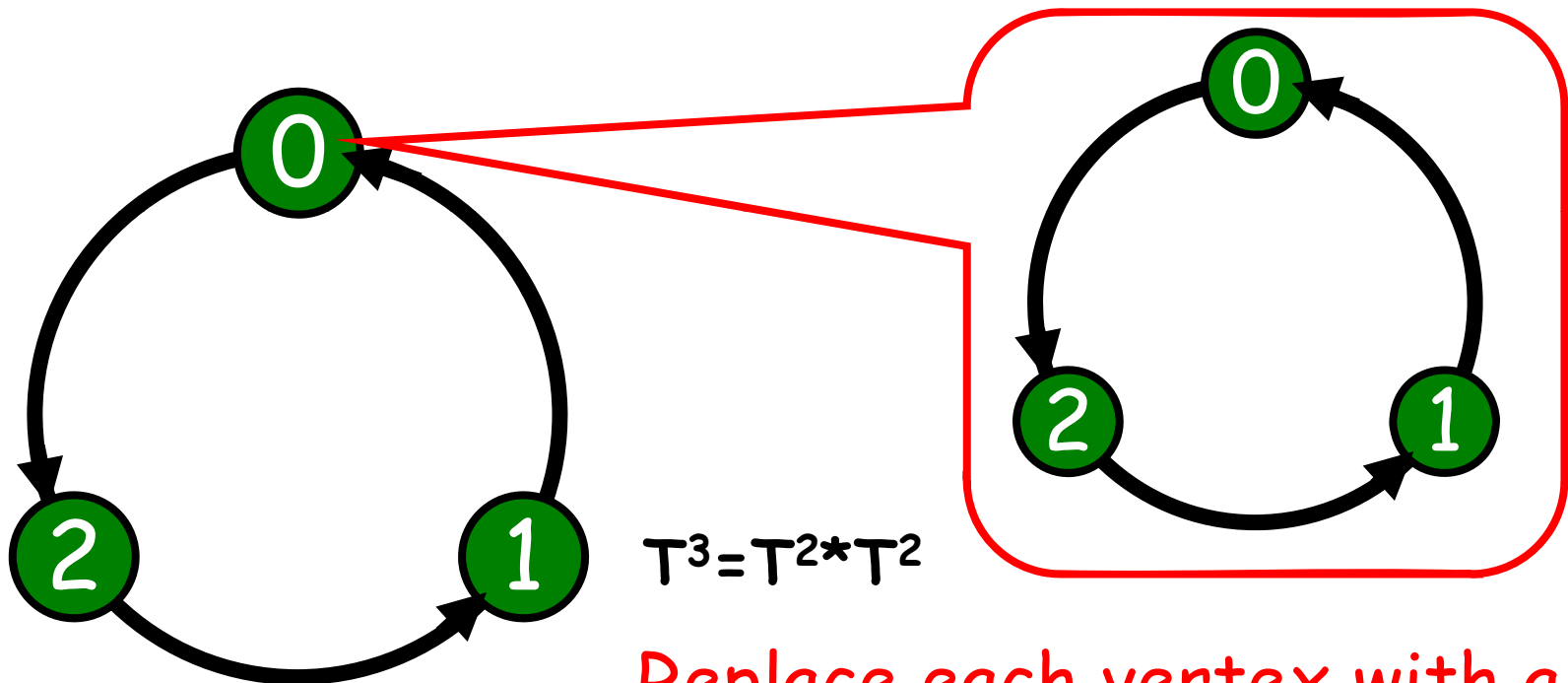
Three-Thread Bounded Precedence Graph?



Three-Thread Bounded Precedence Graph?

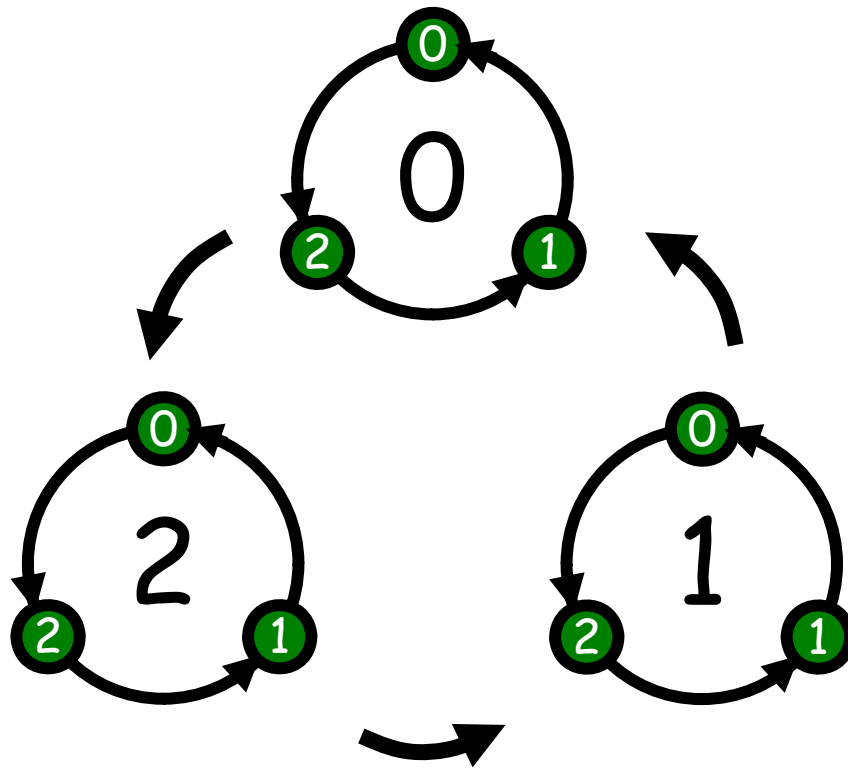


Graph Composition

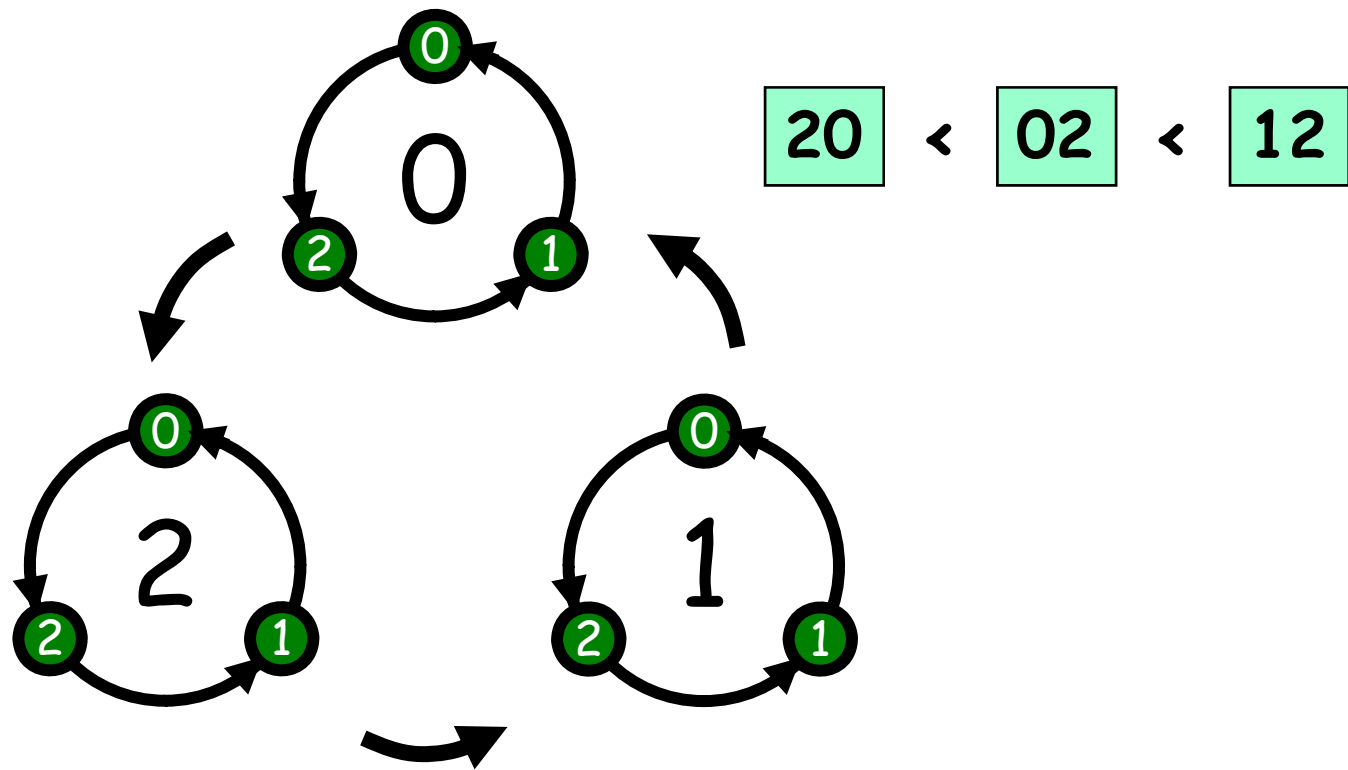


Replace each vertex with a copy of the graph

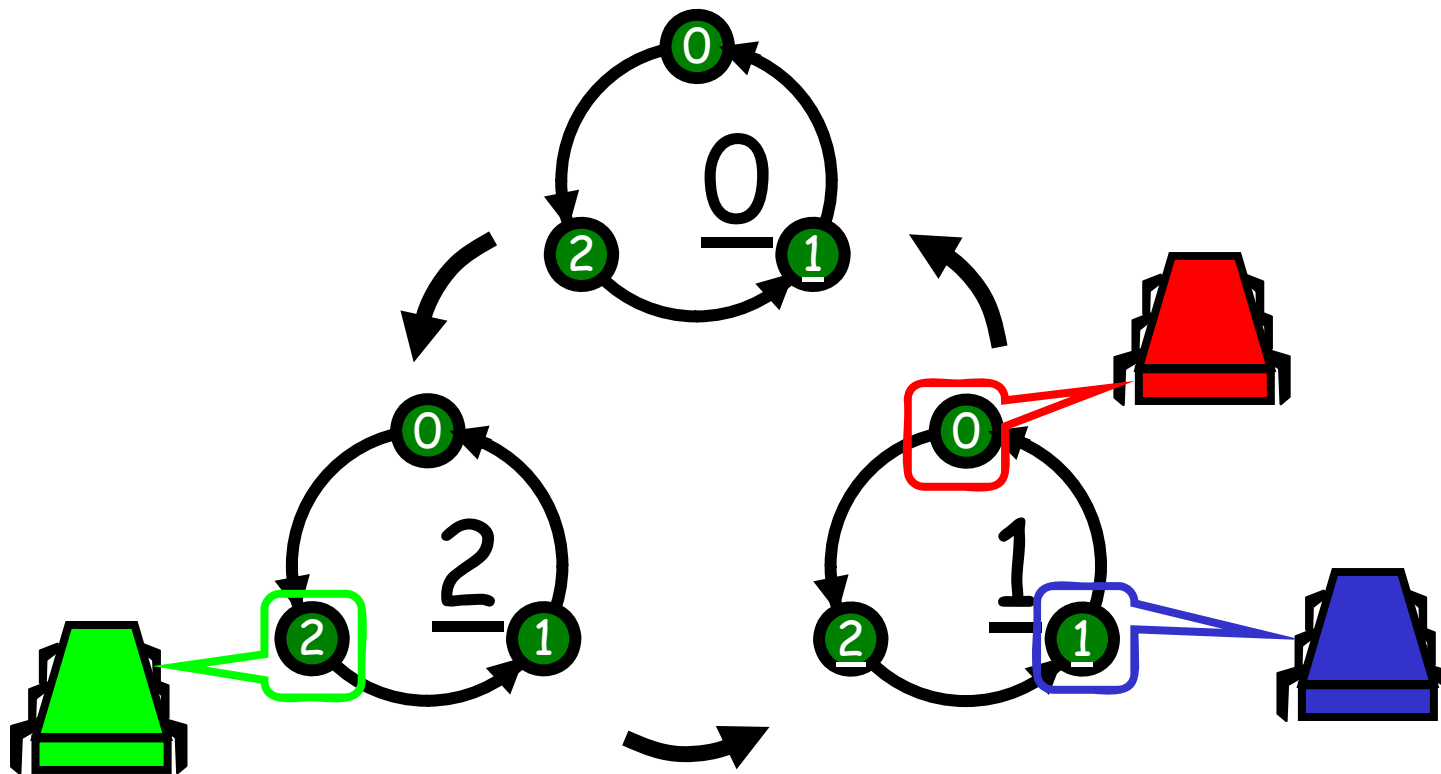
Three-Thread Bounded Precedence Graph T^3



Three-Thread Bounded Precedence Graph T^3



Three-Thread Bounded Precedence Graph T^3



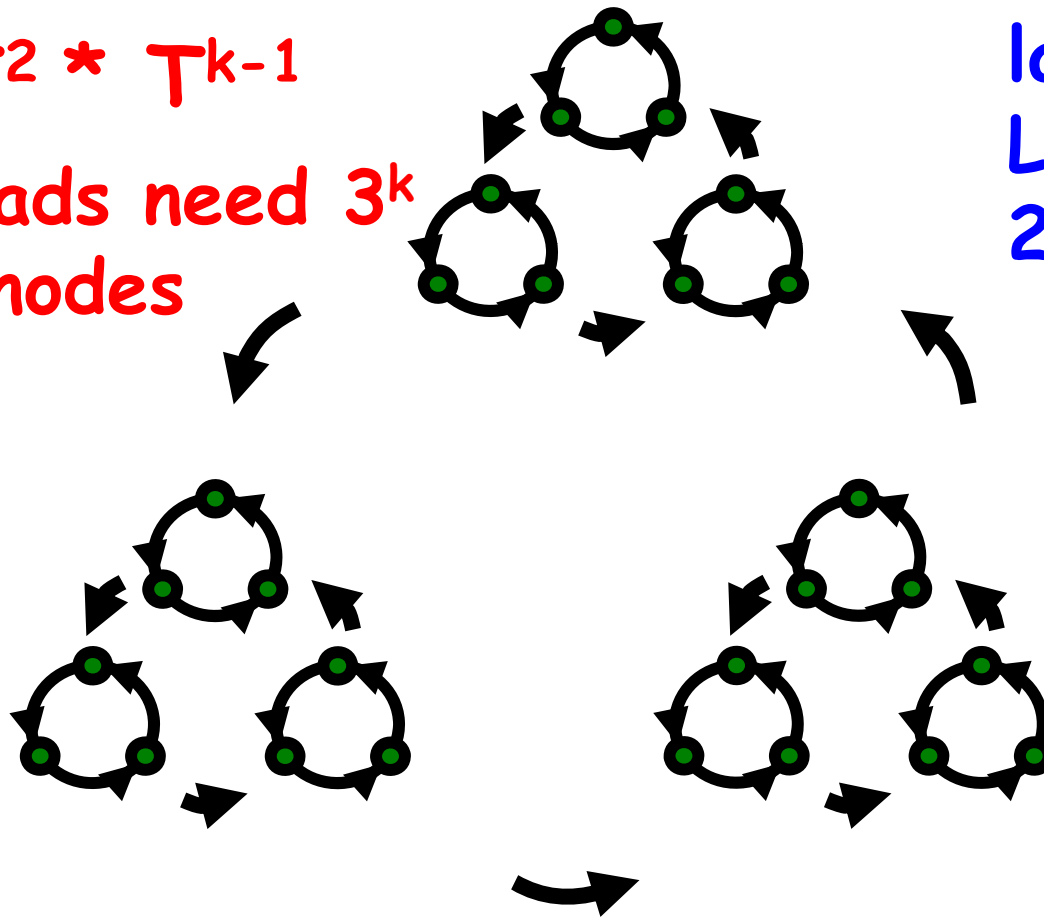
and so on...

In General

$$T^k = T^2 * T^{k-1}$$

K threads need 3^k
nodes

$$\text{label size} = \log_2(3^k) = 2n$$



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

Shared Memory

- Shared read/write memory locations called **Registers** (historical reasons)
- Come in different flavors
 - Multi-Reader-Single-Writer (Flag[])
 - Multi-Reader-Multi-Writer (**Victim**[])
 - Not interesting: SRMW and SRSW

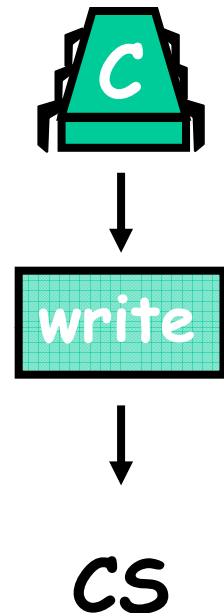
Theorem

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

N registers like `Flag[]`...

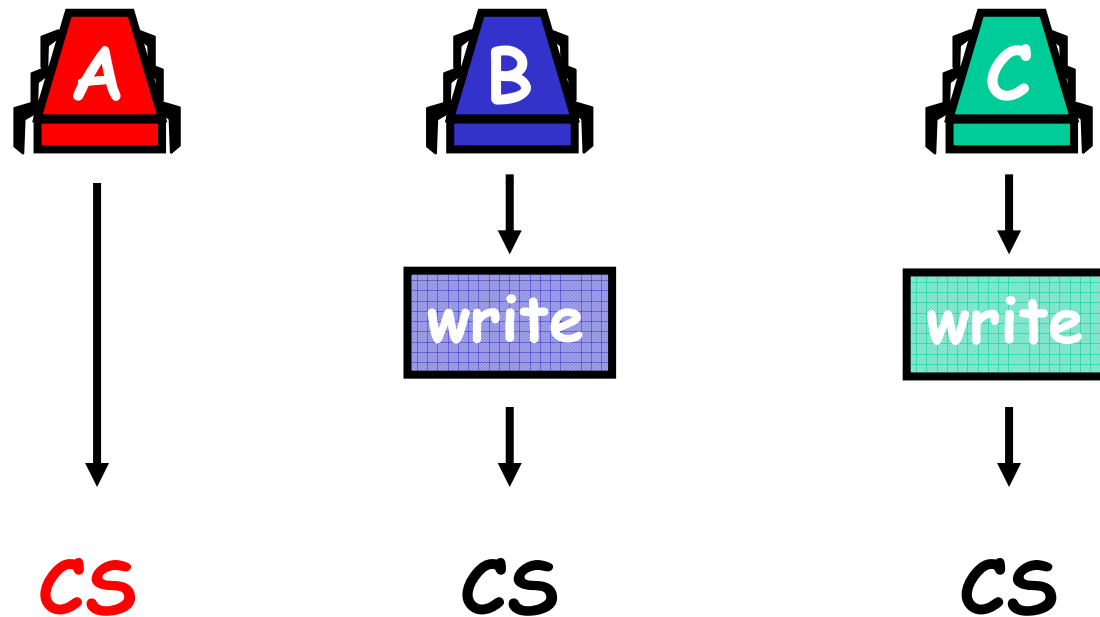
Proving Algorithmic Impossibility

- To show no algorithm exists:
 - assume by way of contradiction one does,
 - show a **bad execution** that violates properties:
 - in our case assume an alg for deadlock free mutual exclusion using $< N$ registers



Proof: Need N-MRSW Registers

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.

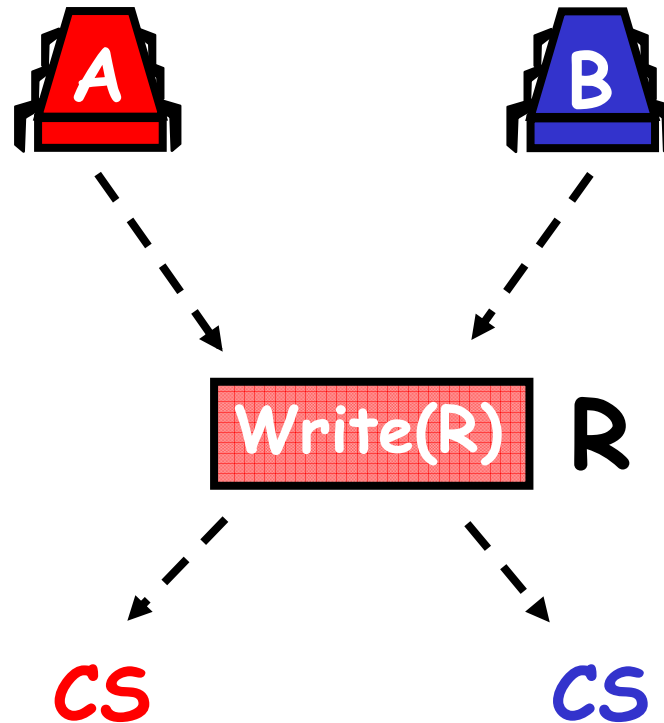
(So multiple writers don't help)

Theorem (First 2-Threads)

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

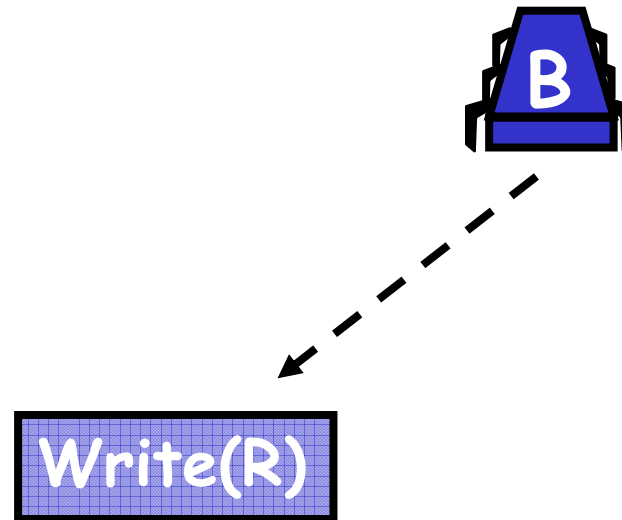
Proof: assume one register suffices and derive a contradiction

Two Thread Execution



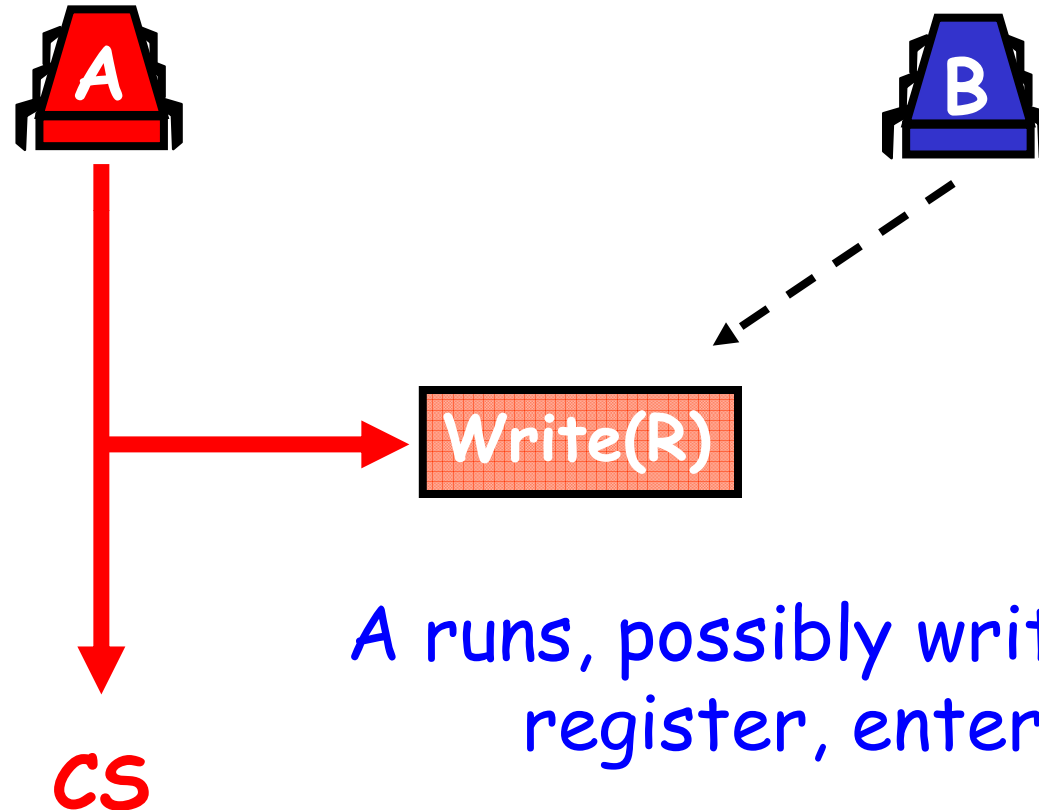
- Threads run, reading and writing R
- Deadlock free so at least one gets in

Covering State for One Register

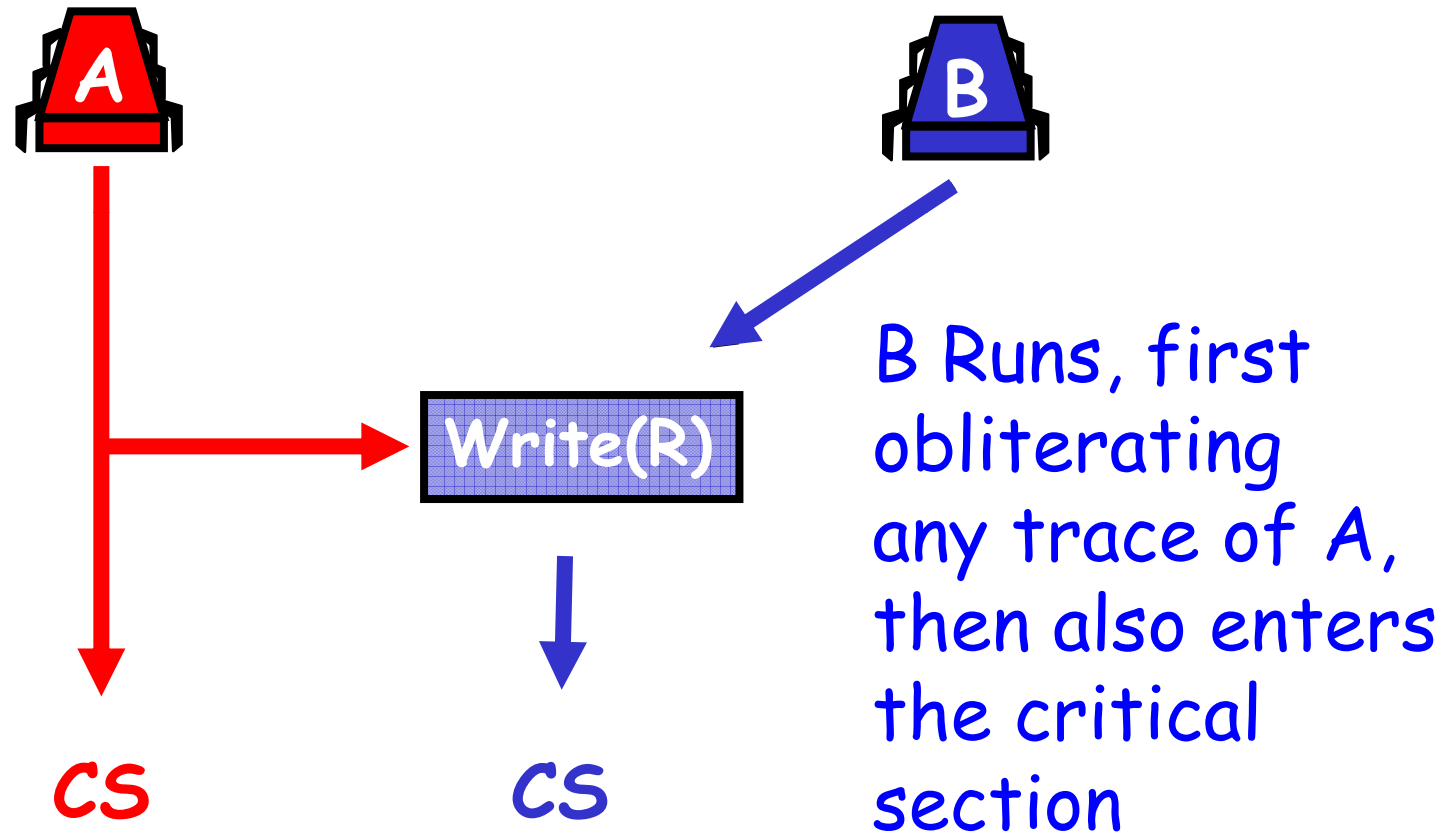


B has to write to the register
before entering CS, so stop it just
before

Proof: Assume Cover of 1



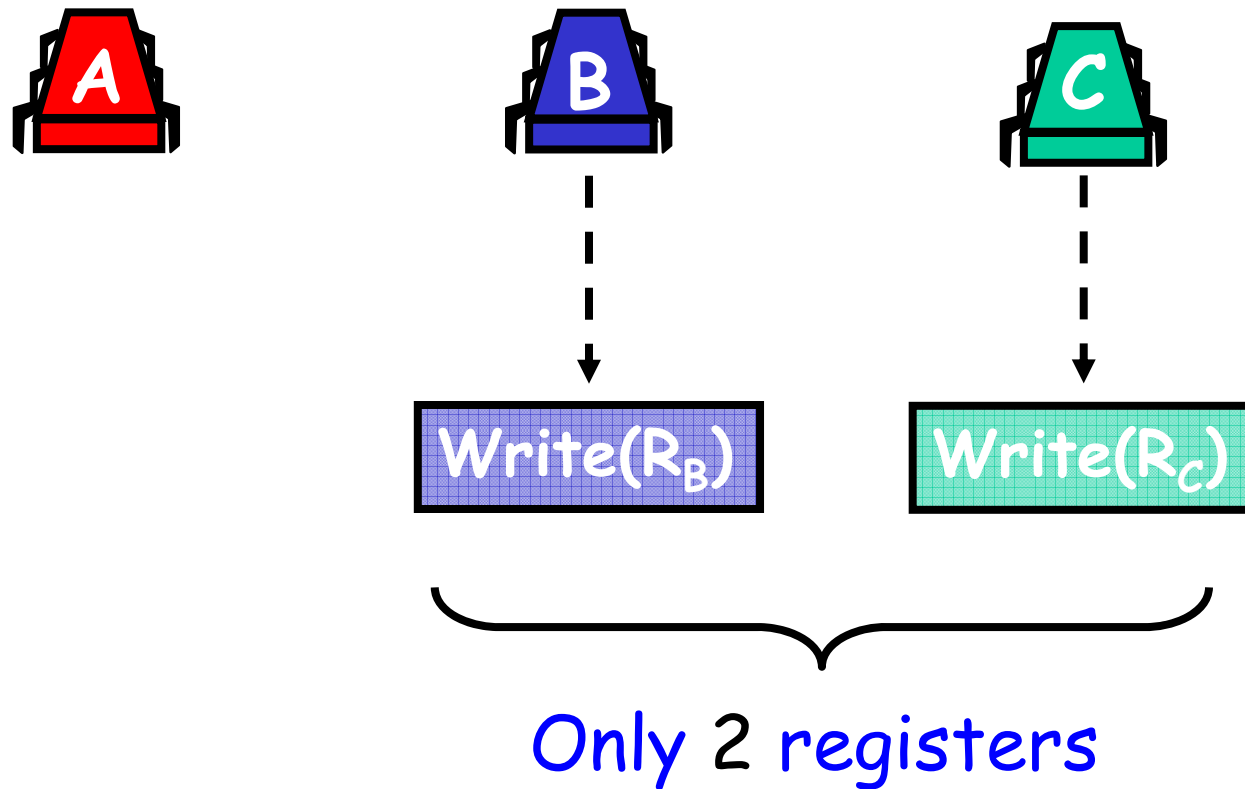
Proof: Assume Cover of 1



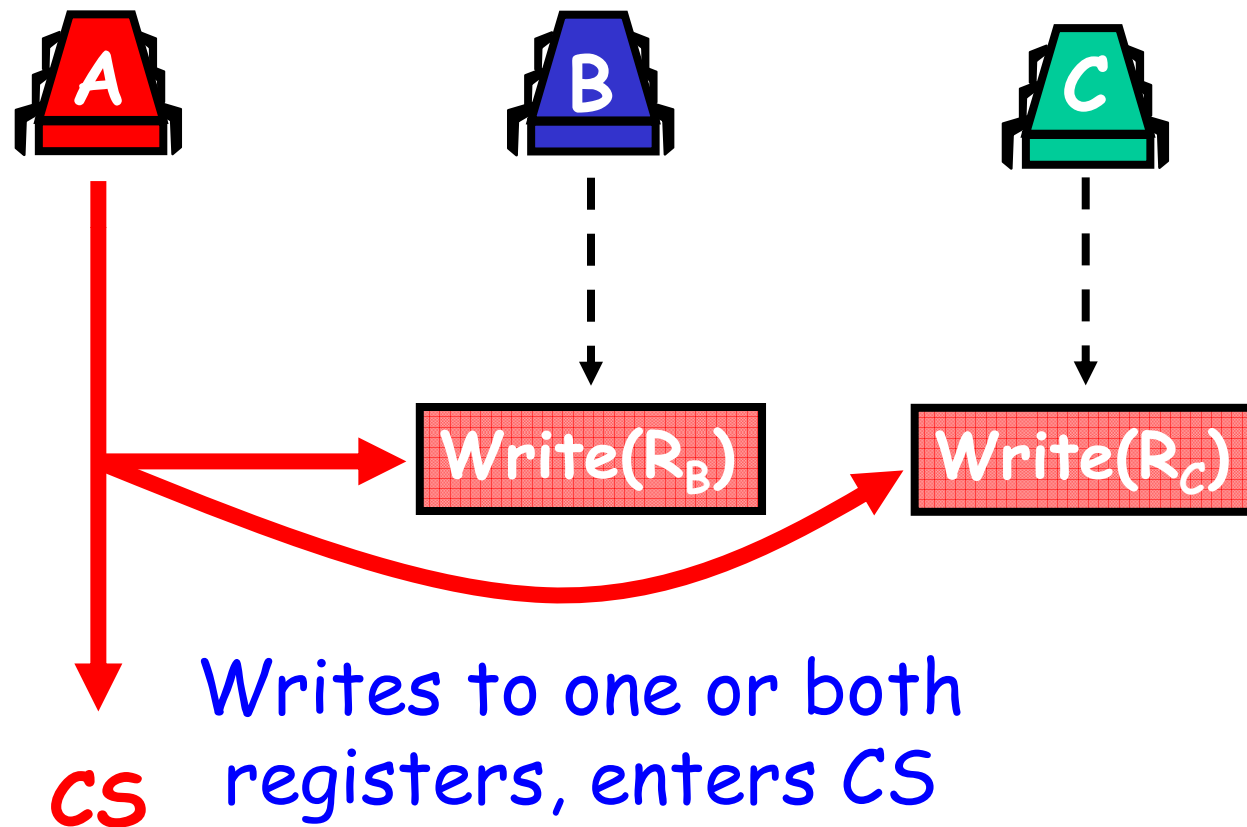
Theorem

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

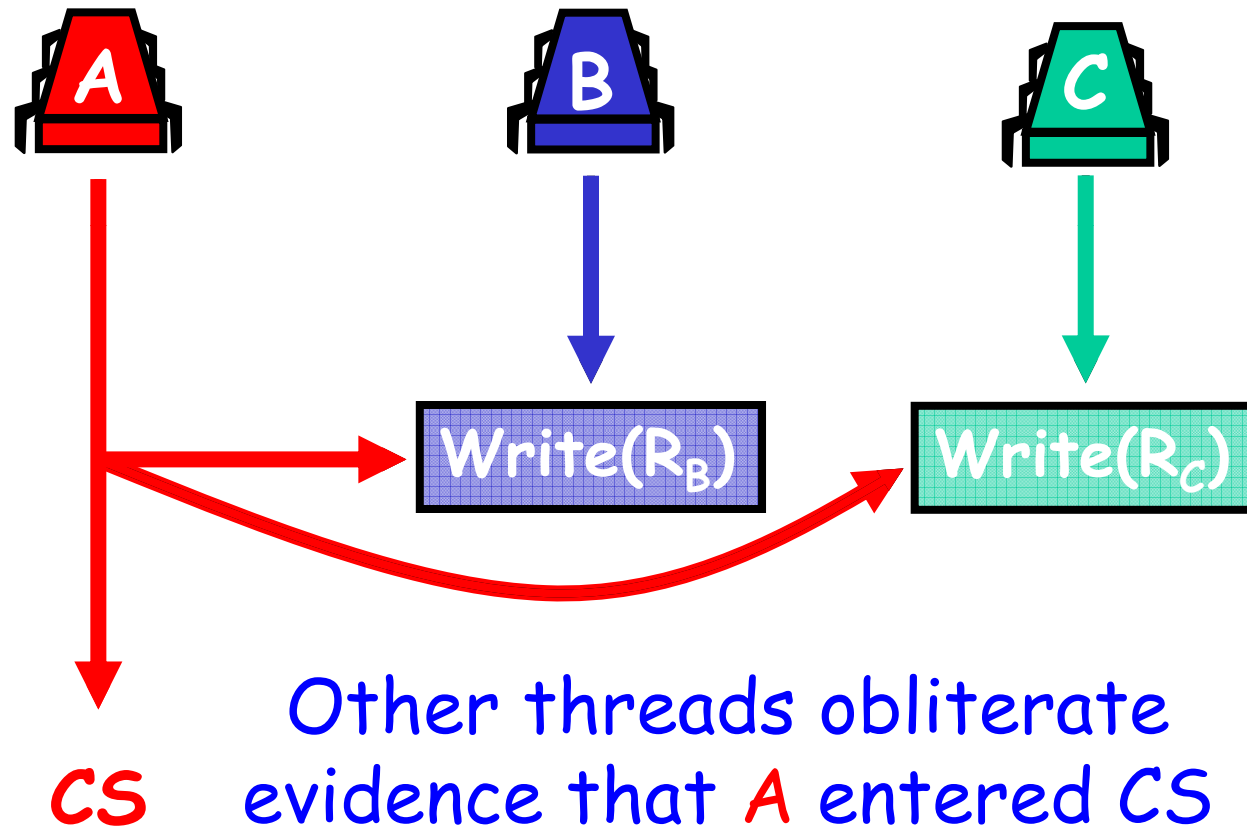
Proof: Assume Cover of 2



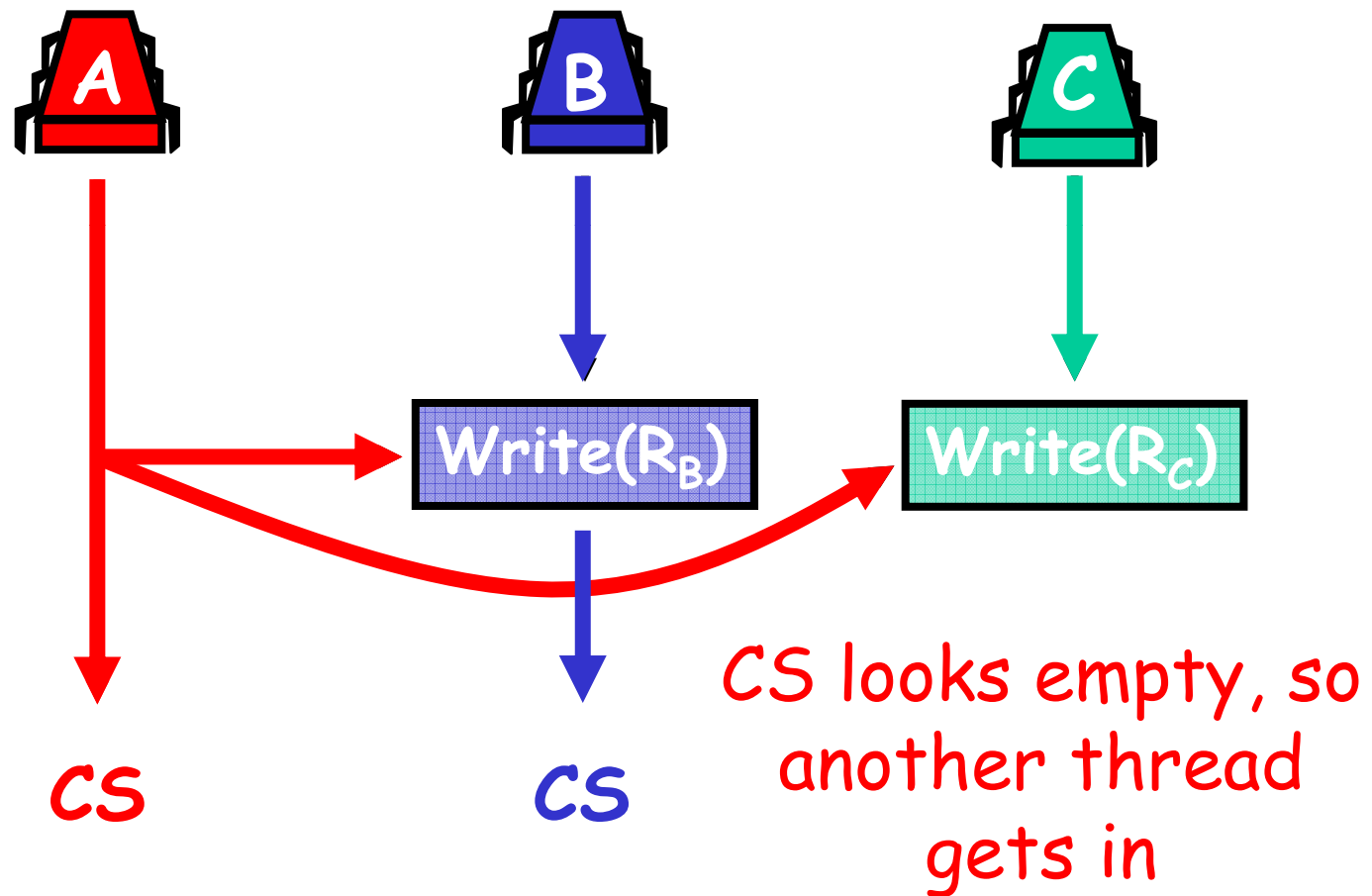
Run A Solo



Obliterate Traces of A



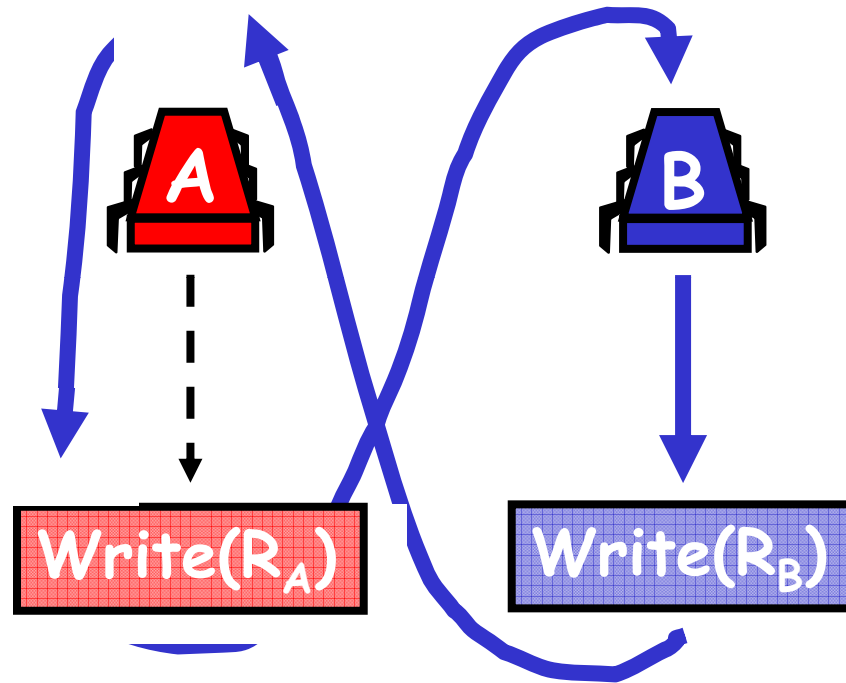
Mutual Exclusion Fails



Proof Strategy

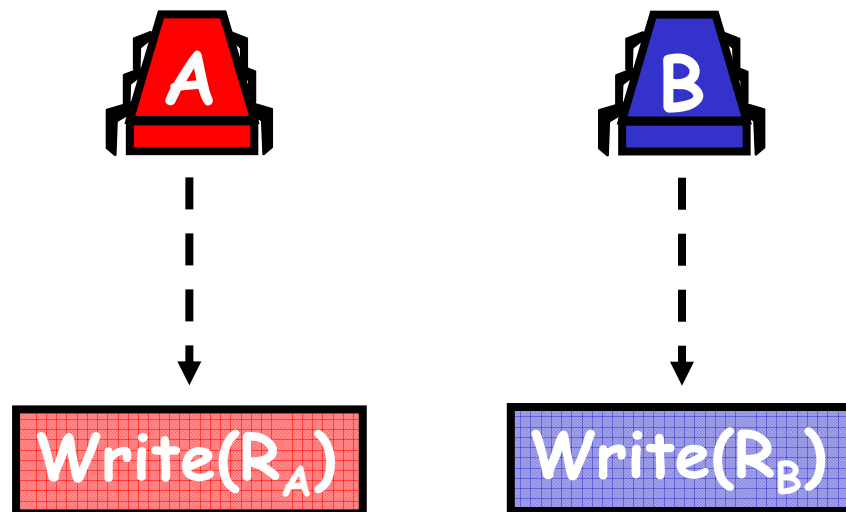
- Proved: a contradiction starting from a covering state for 2 registers
- Claim: a covering state for 2 registers is reachable from any state where CS is empty

Covering State for Two



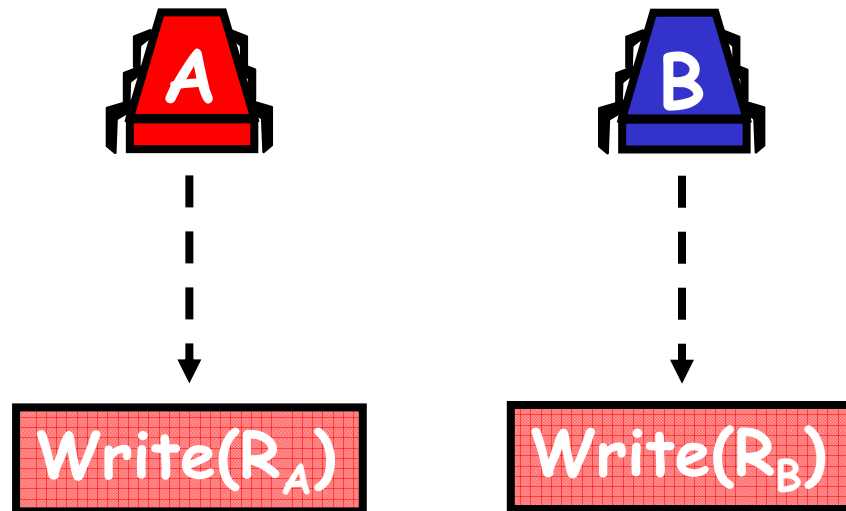
- If we run **B** through CS 3 times, **B** must return twice to cover some register, say **R_B**

Covering State for Two



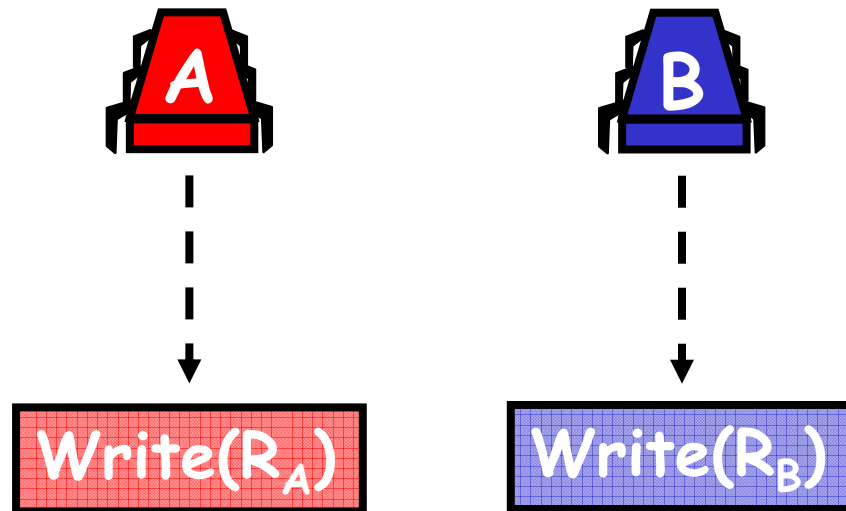
- Start with **B** covering register **R_B** for the 1st time
- Run **A** until it is about to write to uncovered **R_A**
- Are we done?

Covering State for Two



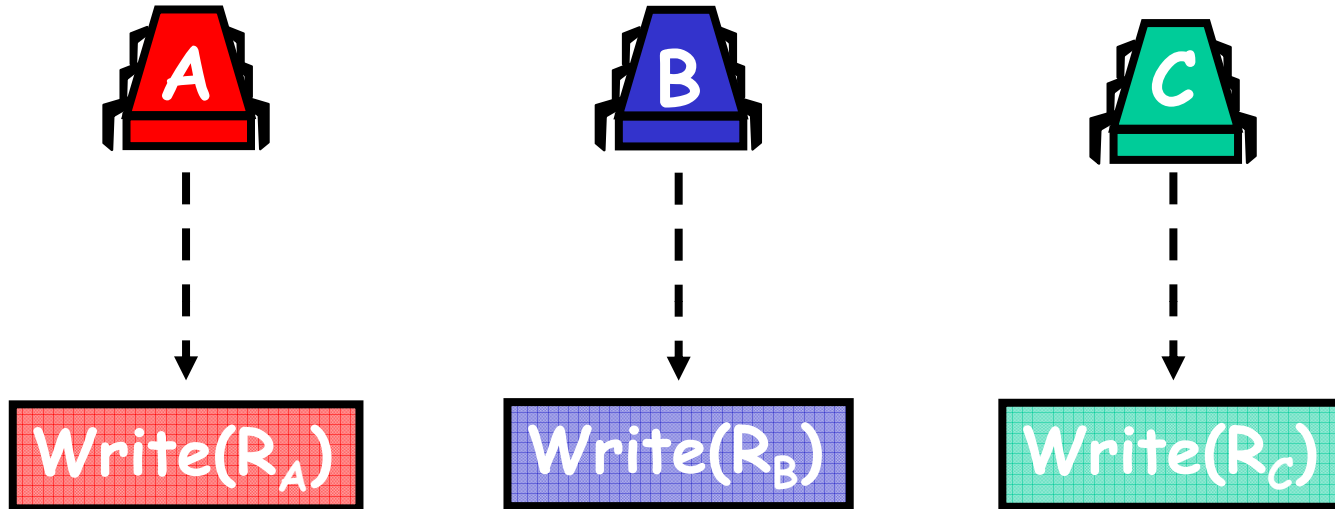
- **NO!** A could have written to R_B
- So CS no longer looks empty

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**
- 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

Summary of Lecture

- N RW-Registers inefficient
 - Because writes "cover" older writes
- Need stronger hardware operations
 - do not have the "covering problem"
- In next lectures - understand what these operations are...



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/2.5/).

- **You are free:**
 - **to Share** — to copy, distribute and transmit the work
 - **to Remix** — to adapt the work
- **Under the following conditions:**
 - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
 - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
 - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.