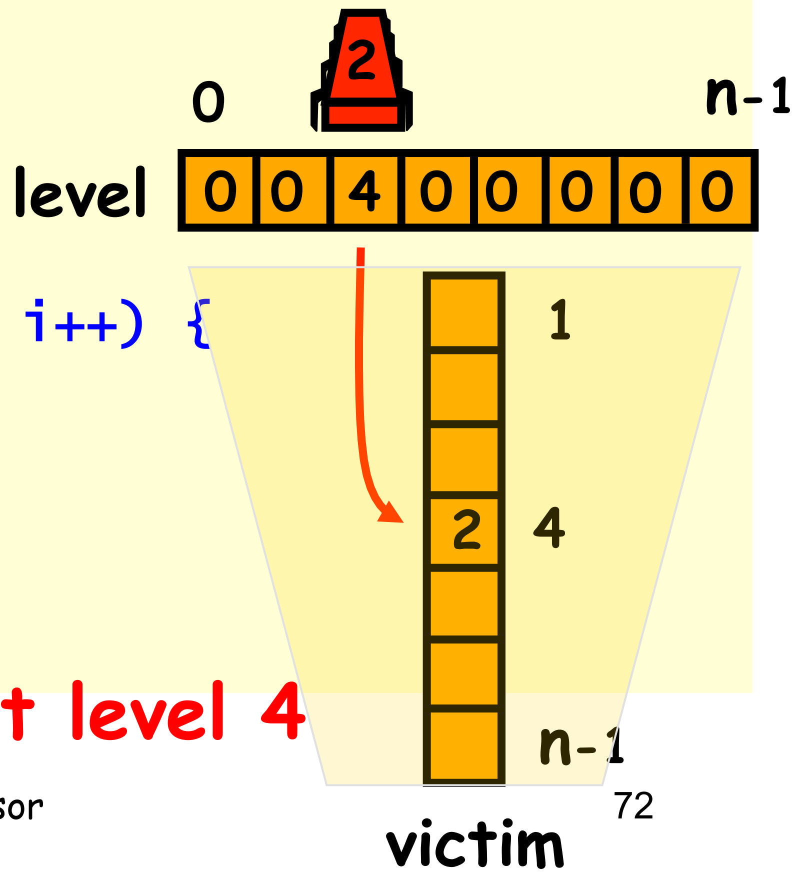
Mutual Exclusion

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

Filter

```
class Filter implements Lock {
   int[] level;  // level[i] for thread i
   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 ((∃ k != 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 ((∃ k != 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 ((∃ k != 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 ((∃ k != 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 ((∃ k != i) level[k] >= L) &&
           victim[L] == i);
  }}
public void release(int i) {
  level[i] = 0;
}}
```

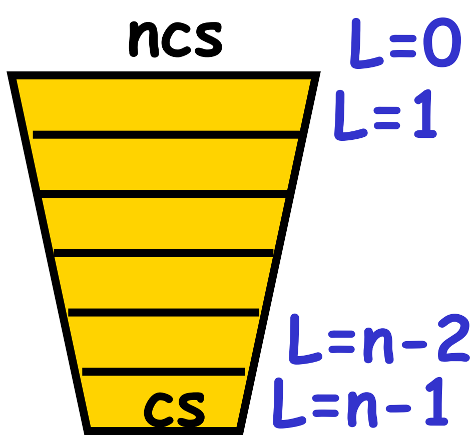
Filter

```
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 ((∃ k != i) level[k] >= L) &&
             victim[L] == i);
    }}
```

Thread enters level L when it completes the loop

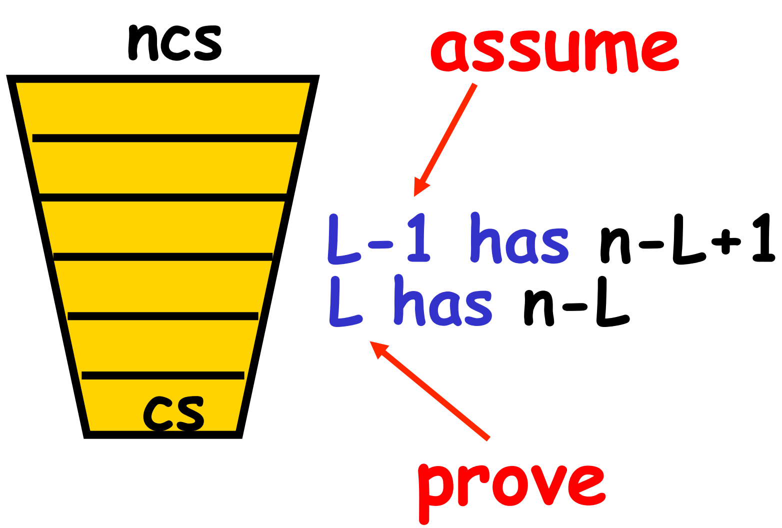
Claim

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

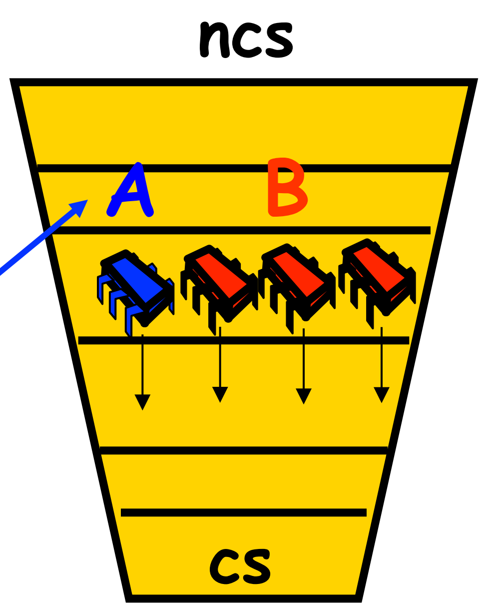


Induction Hypothesis

- 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



Assumed to enter L-1

n-L+1 = 4

n-L+1 = 4

Last to write victim[L]

By way of contradiction all enter L

Show that A must have seen B in level[L] and since victim[L] == A could not have entered

From the Code

(1) $write_B(level[B]=L) \rightarrow write_B(victim[L]=B)$

```
public void lock() {
 for (int L = 1; L < n; L++) {
   level[i] = L;
   victim[L]  = i;
   while ((∃ k != 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 ((∃ k != 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) $write_B(level[B]=L) \rightarrow$

(3) $write_B(victim[L]=B) \rightarrow write_A(victim[L]=A)$

(2) $\rightarrow read_A(level[B])$

```
public void lock() {
 for (int L = 1; L < n; L++) {
   level[i]  = L;
   victim[L] = i;
   while ((∃ k != 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 level[B] ≥ L,
A was last to write 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

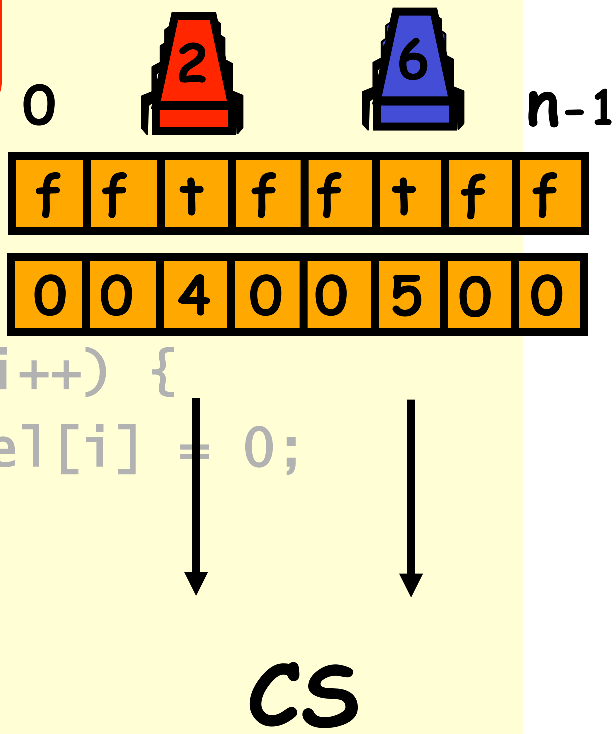
- Provides First-Come-First-Served
- How?
  - Take a "number"
  - Wait until lower numbers have been served
- Lexicographic order
  - $(a,i) > (b,j)$
    - If $a > b$, or $a = b$ and $i > j$

Bakery Algorithm

```
class Bakery implements Lock {
   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;
    }
  }
 …
```



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));
  }
```

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));
 }
```

Doorway

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));
 }
```

I'm interested

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));
 }
```

Take increasing label (read labels in some arbitrary order)

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 (∃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 smaller
- And:
  - write$_A$(label[A]) → read$_B$(label[A]) → write$_B$(label[B]) → read$_B$(flag[A])
- So B is locked out while flag[A] is true

```
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

- 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 flag[A] == 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 flag[A] == 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 Y2^{32}K 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 $Y2^{32}K$ 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));
 }
```

Mutex breaks if label[i] overflows

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 ~~Good~~ Bad News

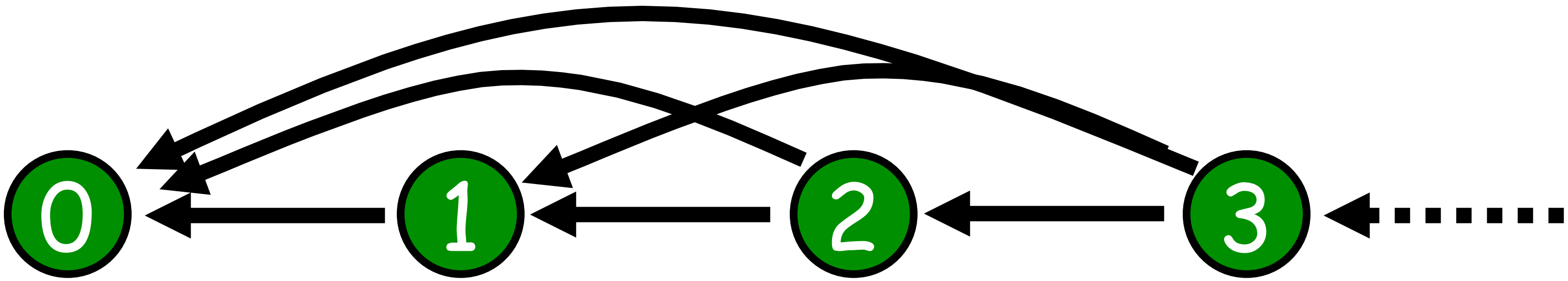
- One can construct a
  - Wait-free (no mutual exclusion)
  - Concurrent
  - Timestamping system
  - That never overflows

This part is hard

Instead …

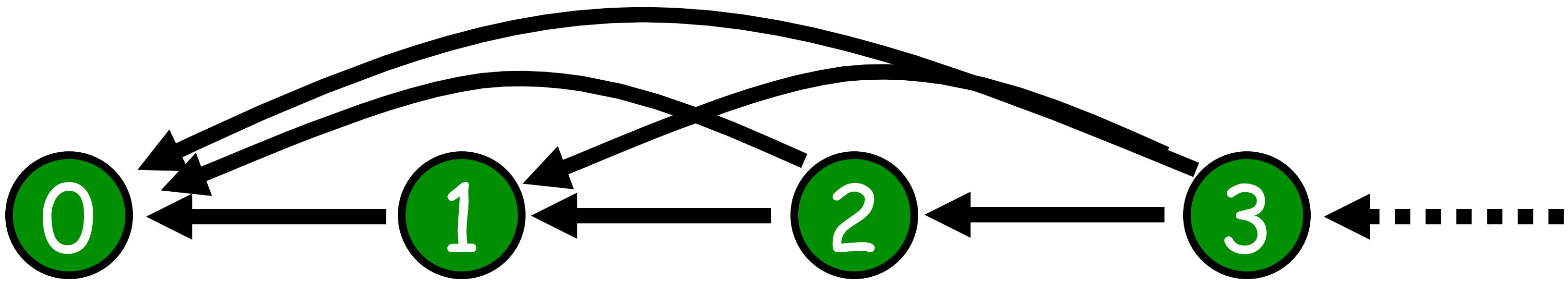
- We construct a Sequential timestamping system
  - Same basic idea
  - But simpler
- Uses mutex to read & write atomically
- No good for building locks
  - But useful anyway

Precedence Graphs



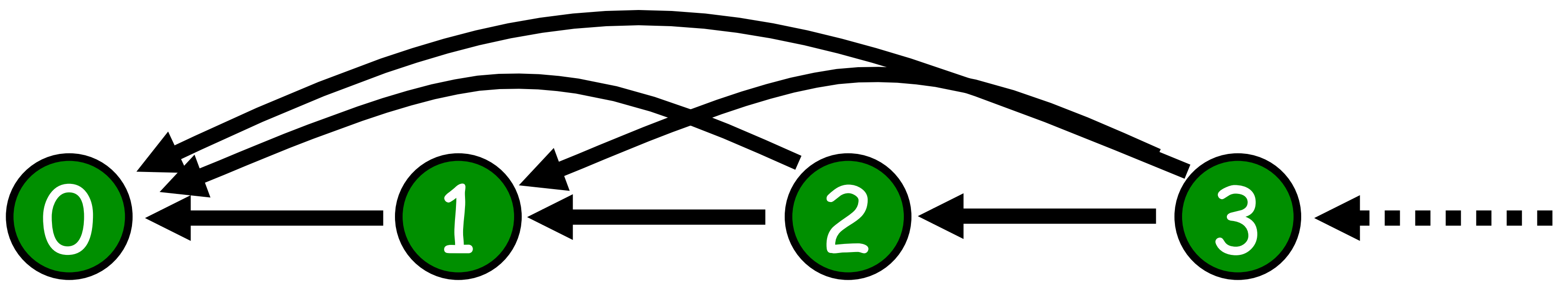
- Timestamps form directed graph
- Edge x to y
  - Means x is later timestamp
  - We say x dominates y

Unbounded Counter Precedence Graph

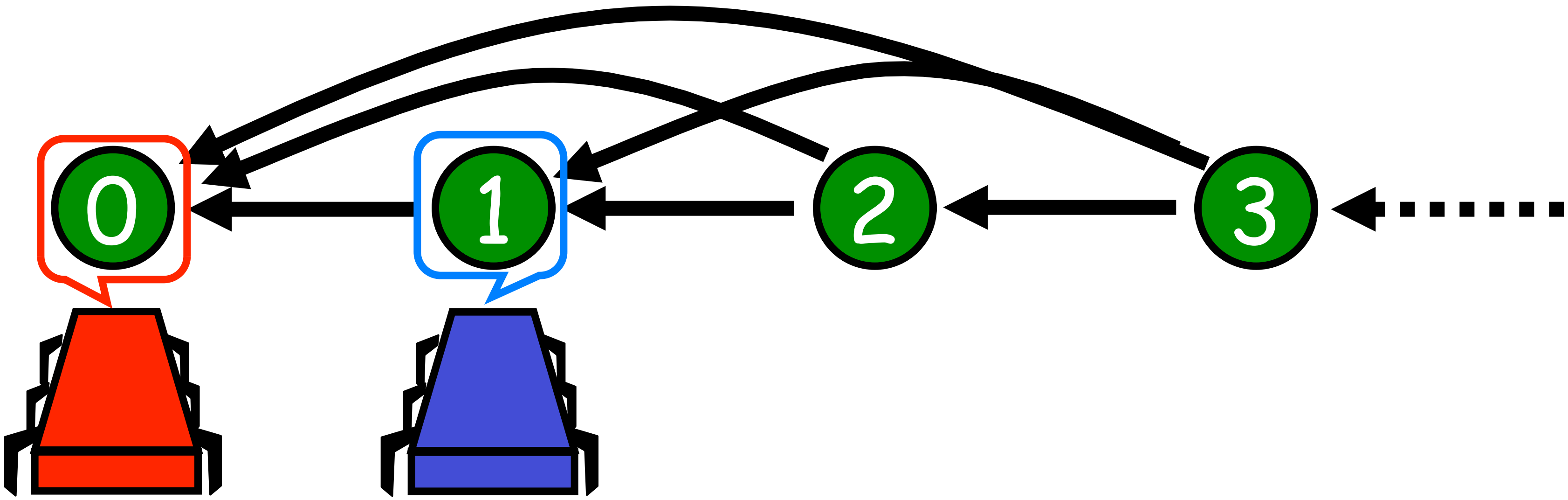


- Timestamping = move tokens on graph
- Atomically
  - read others' tokens
  - move mine
- Ignore tie-breaking for now

Unbounded Counter Precedence Graph

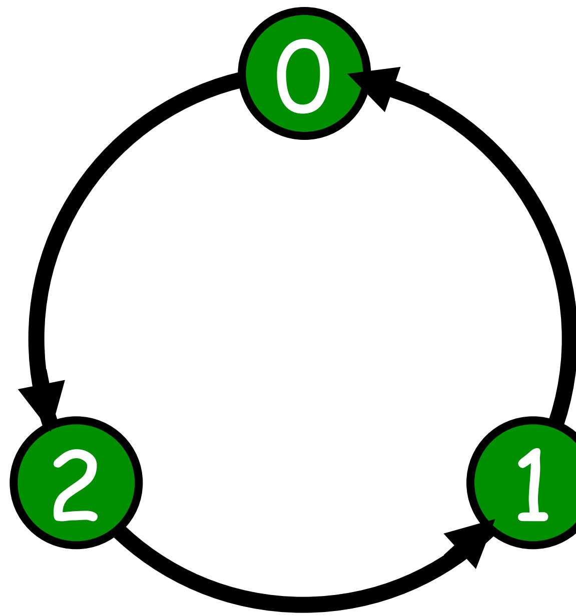


Unbounded Counter Precedence Graph

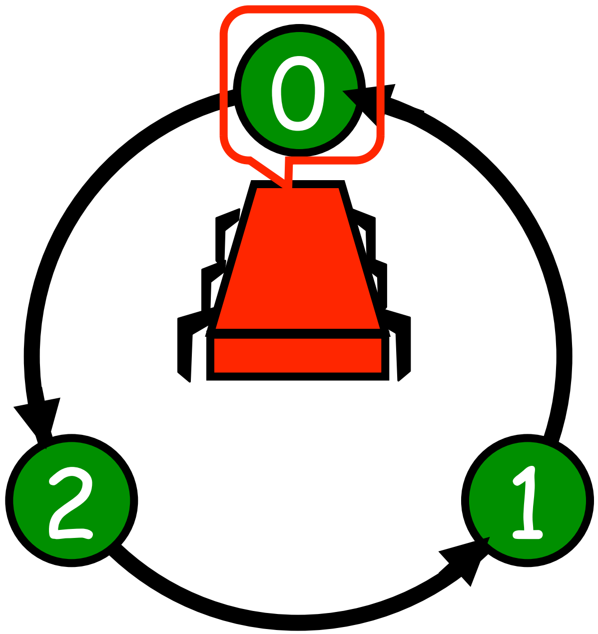


takes 0 takes 1 takes 2

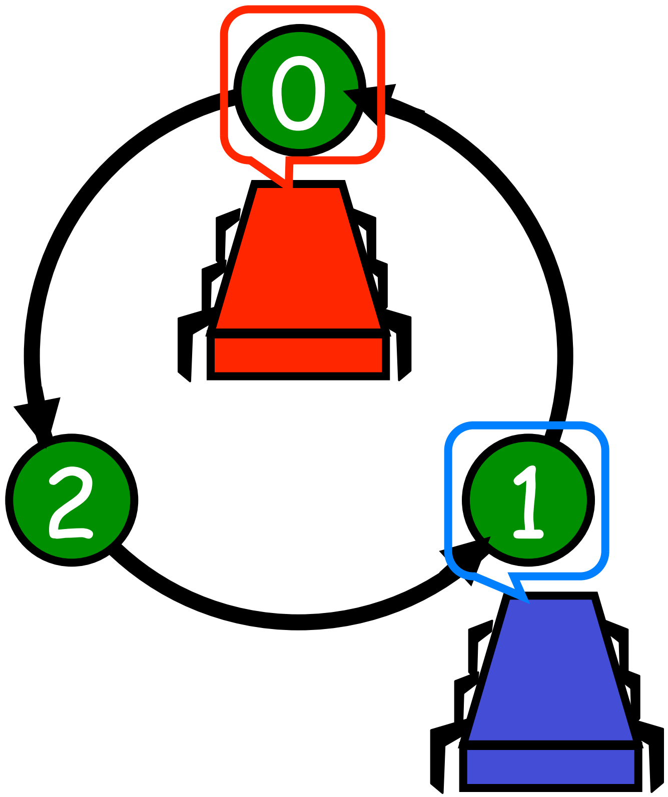
Two-Thread Bounded Precedence Graph



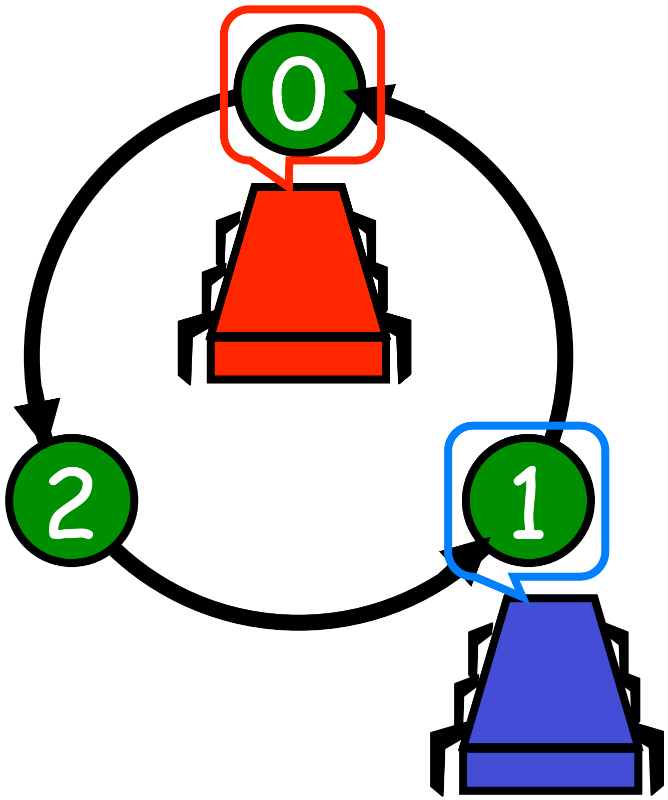
Two-Thread Bounded Precedence Graph



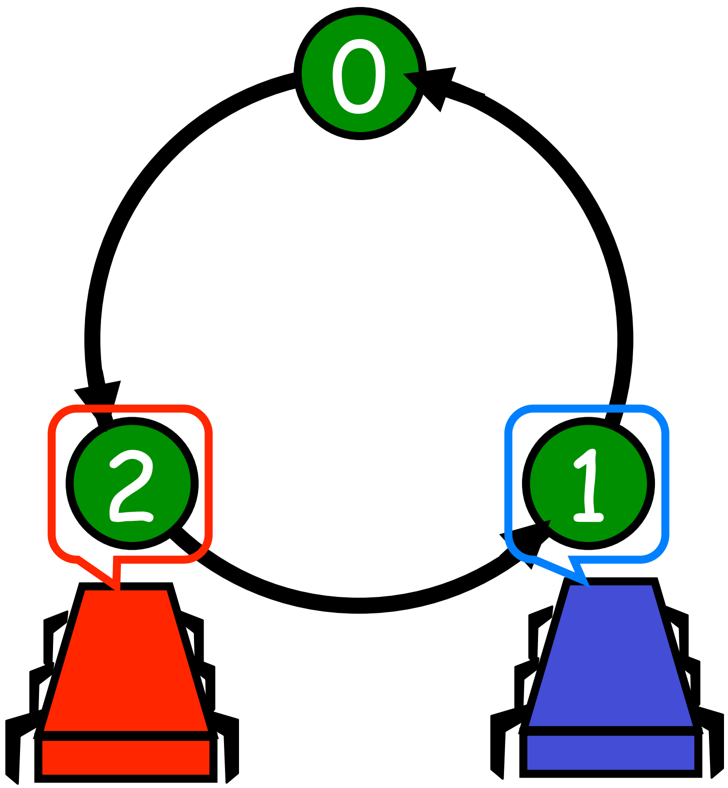
Two-Thread Bounded Precedence Graph



Two-Thread Bounded Precedence Graph

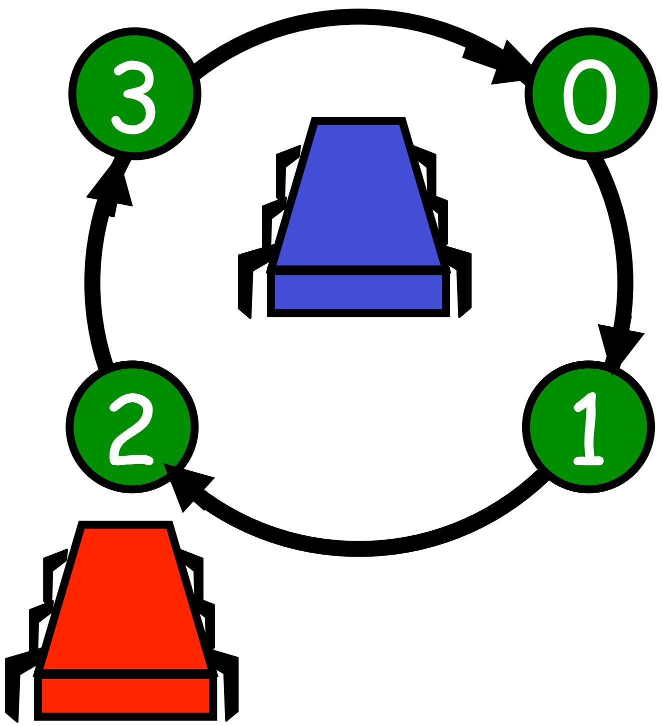


Two-Thread Bounded Precedence Graph T^2

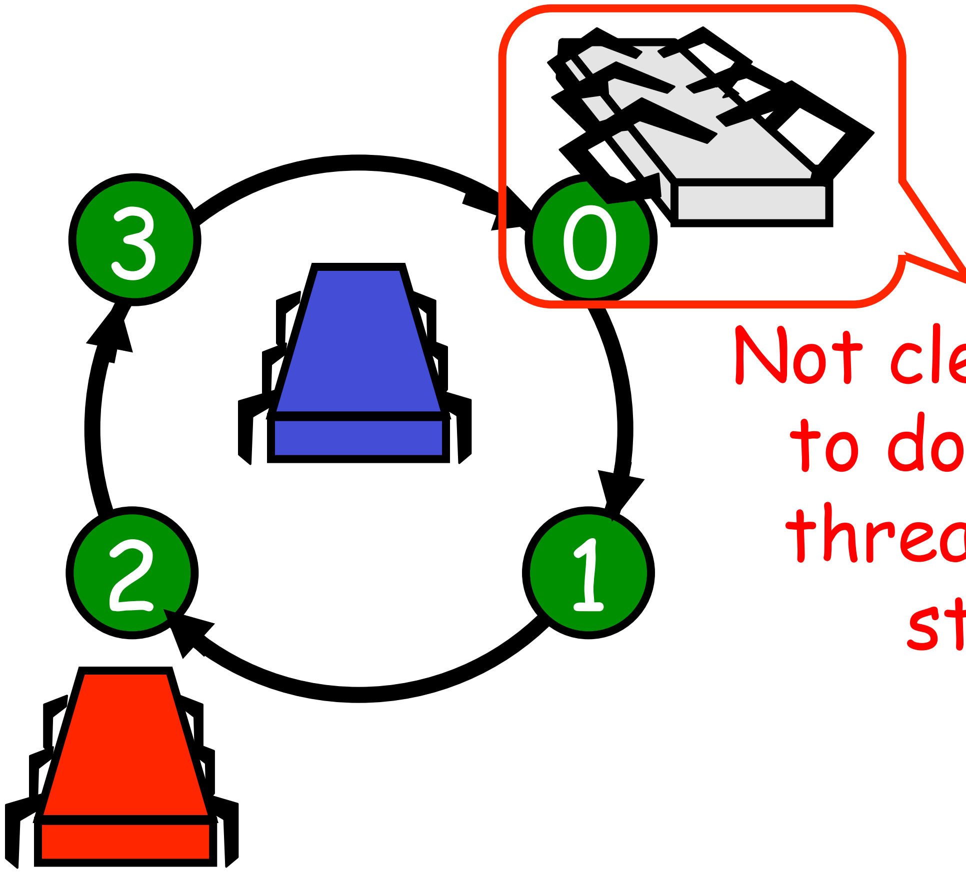


and so on ...

Three-Thread Bounded Precedence Graph?

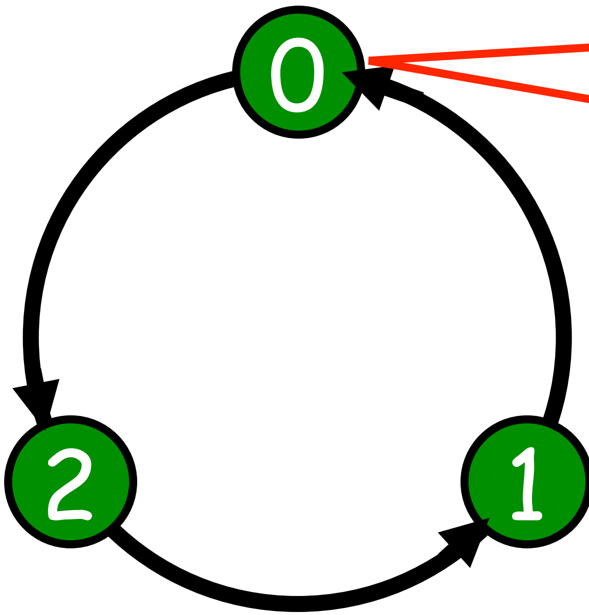


Three-Thread Bounded Precedence Graph?

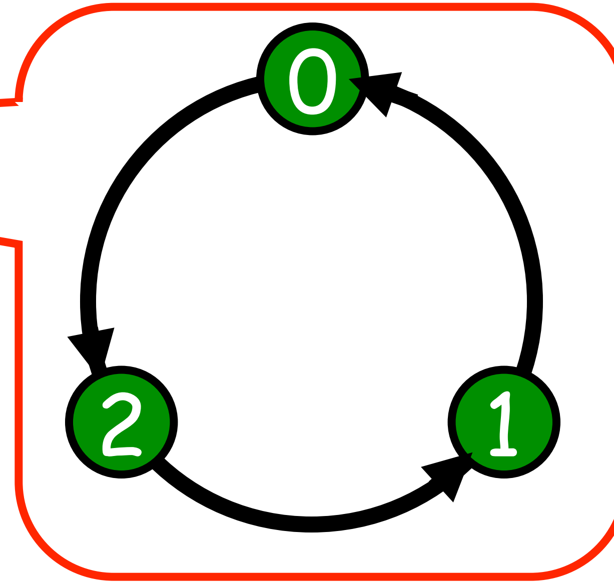


Not clear what to do if one thread gets stuck

Graph Composition

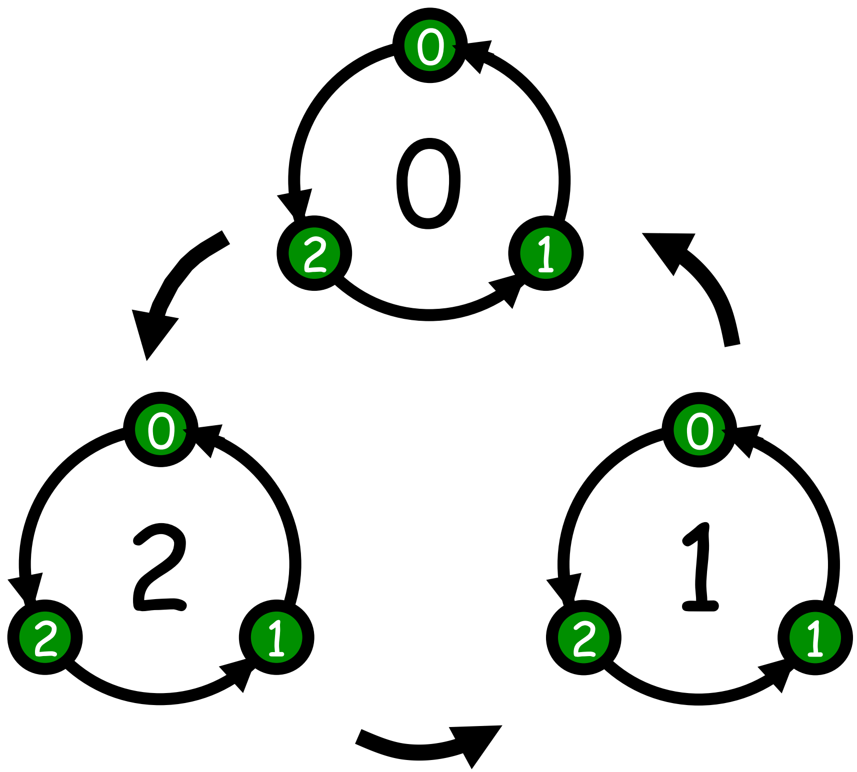


$T^3 = T^2 * T^2$

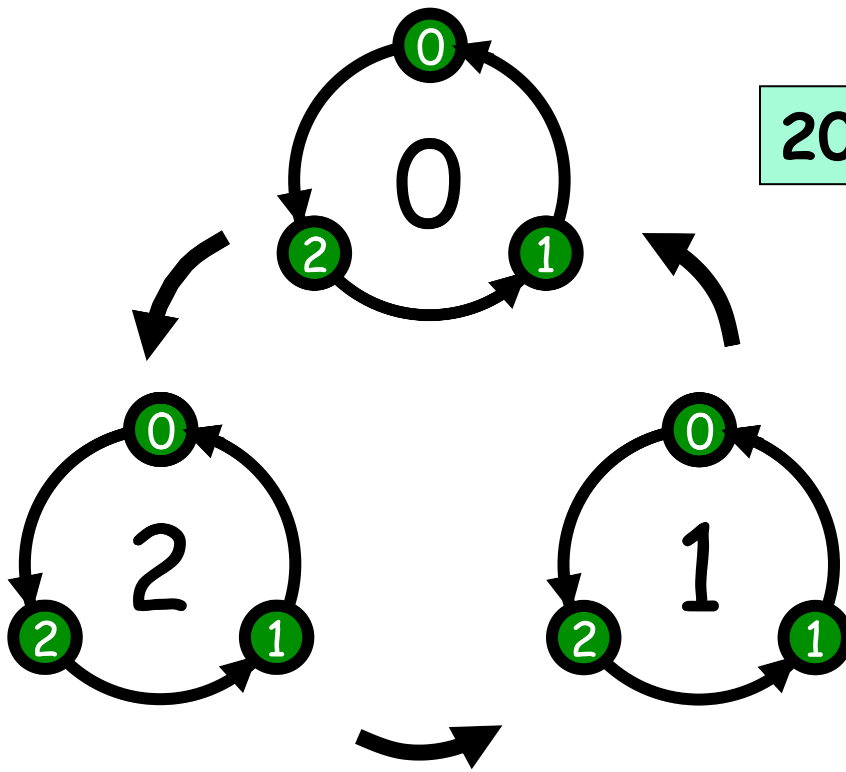


Replace each vertex with a copy of the graph

Three-Thread Bounded Precedence Graph T^3

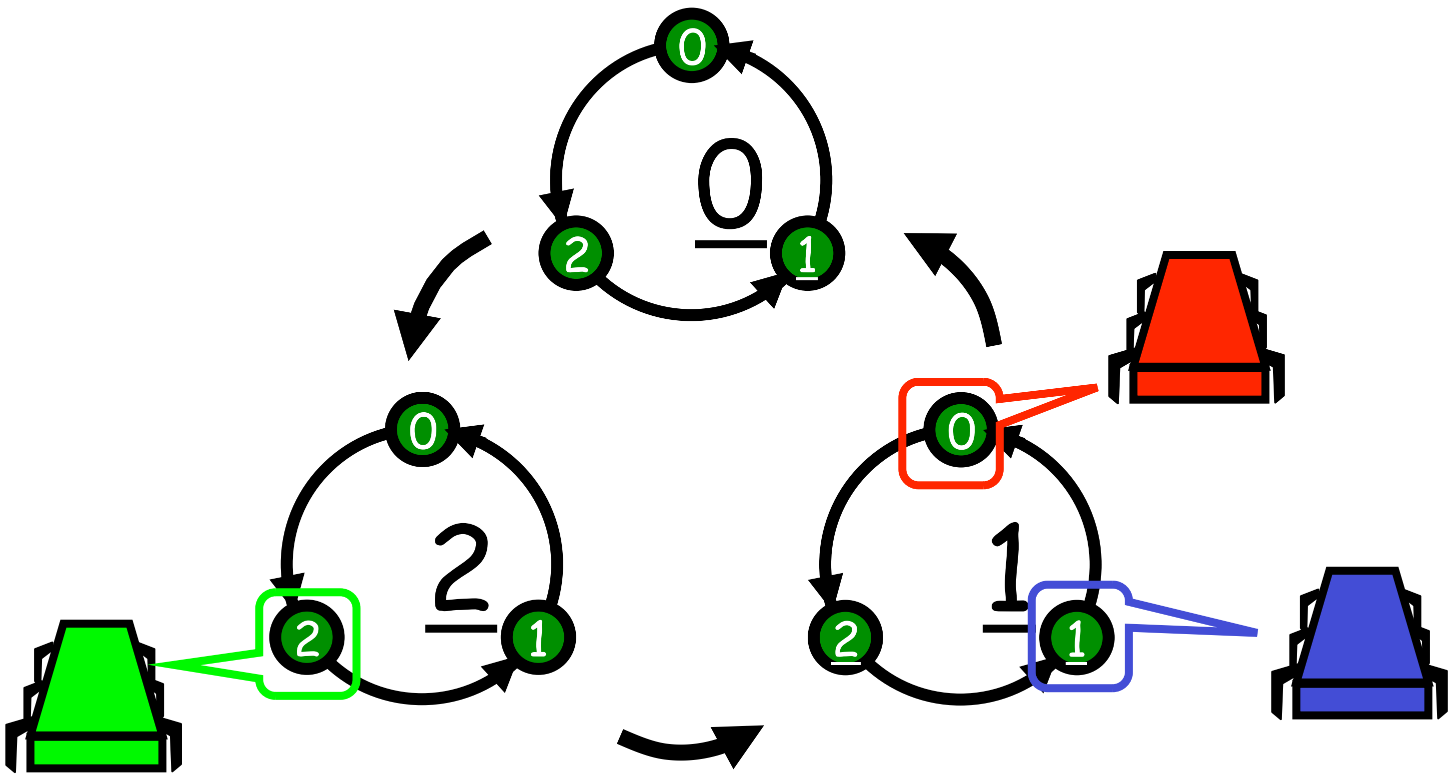


Three-Thread Bounded Precedence Graph T^3



20	<	02	<	12

Three-Thread Bounded Precedence Graph T^3



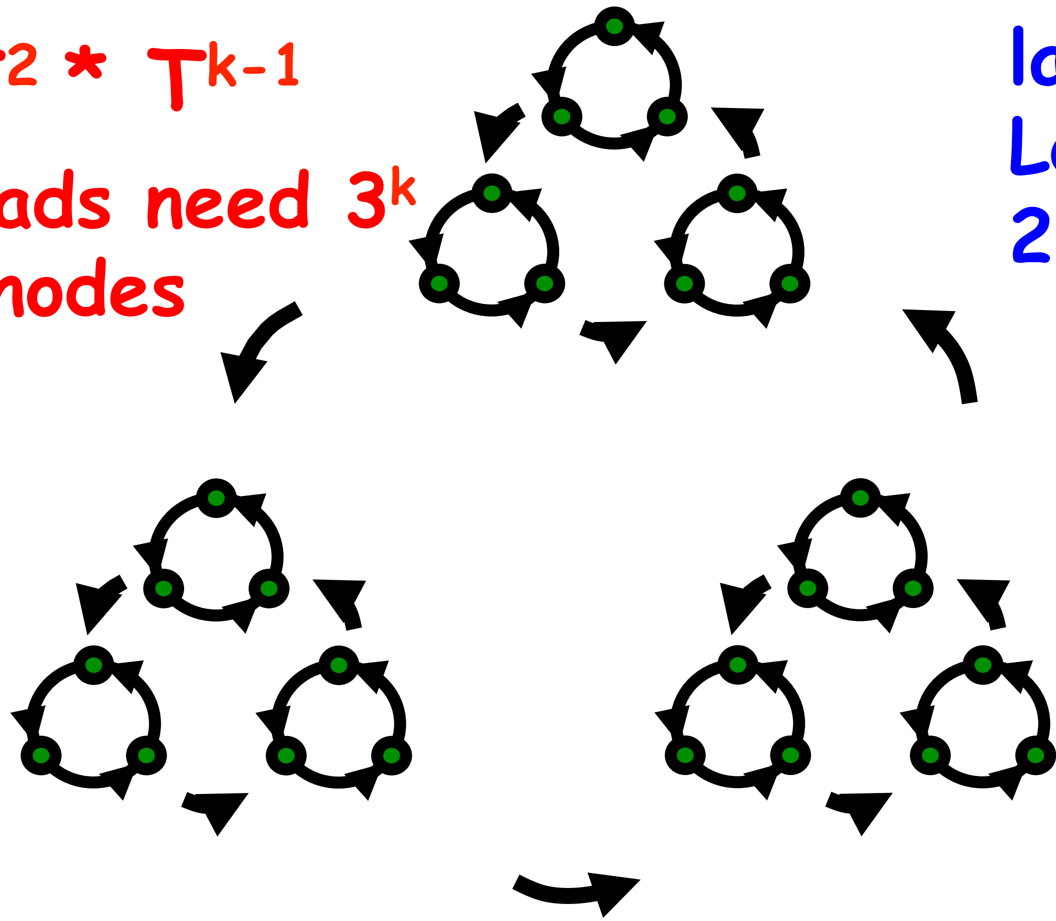
and so on…

In General

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

K threads need 3^k nodes

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 that interesting: SRMW and SRSW

Theorem

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

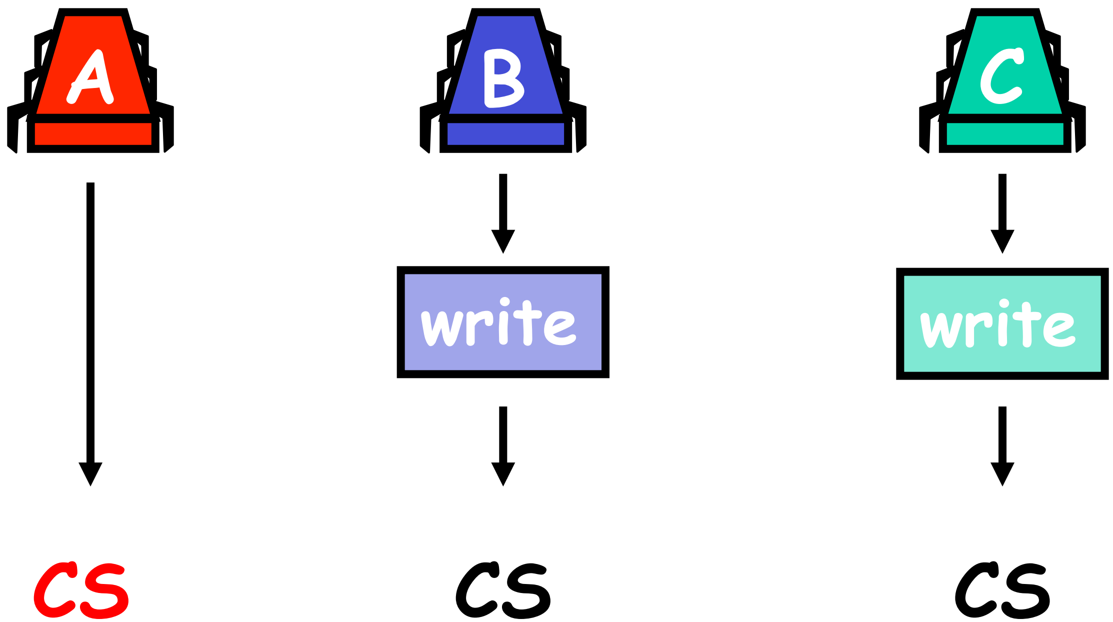
N registers like Flag[]…

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.

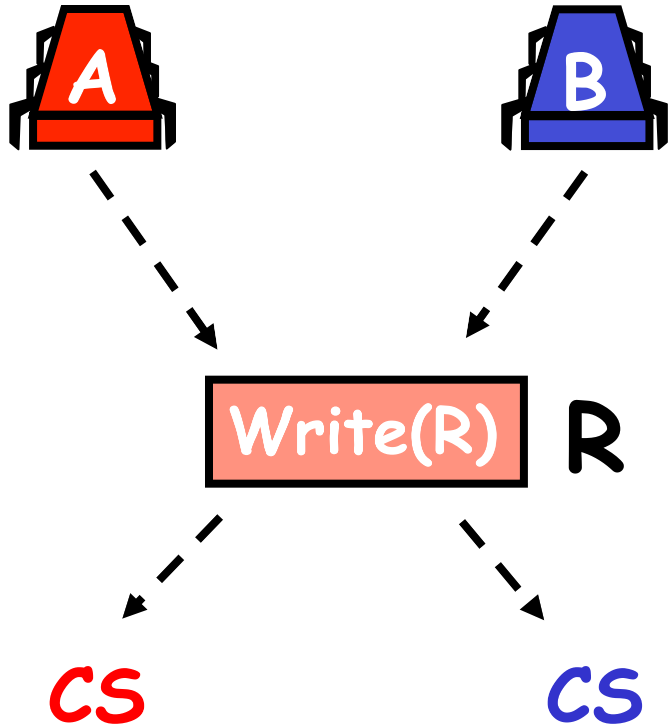
(So multiple writers don't help)

Theorem (First 2-Threads)

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

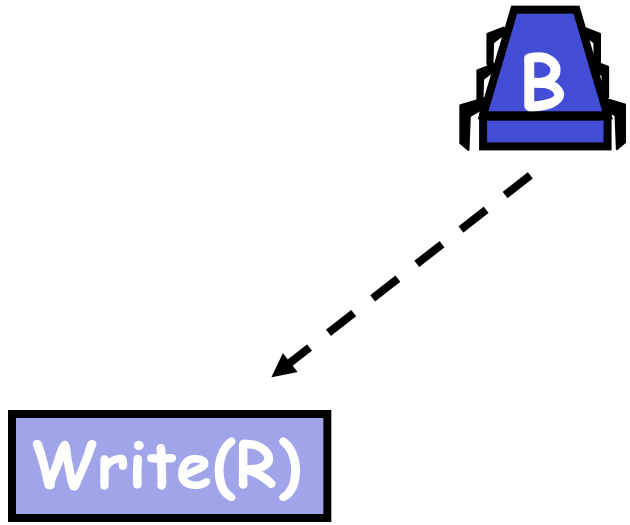
Proof: assume one register suffices and derive a contradiction

Two Thread Execution



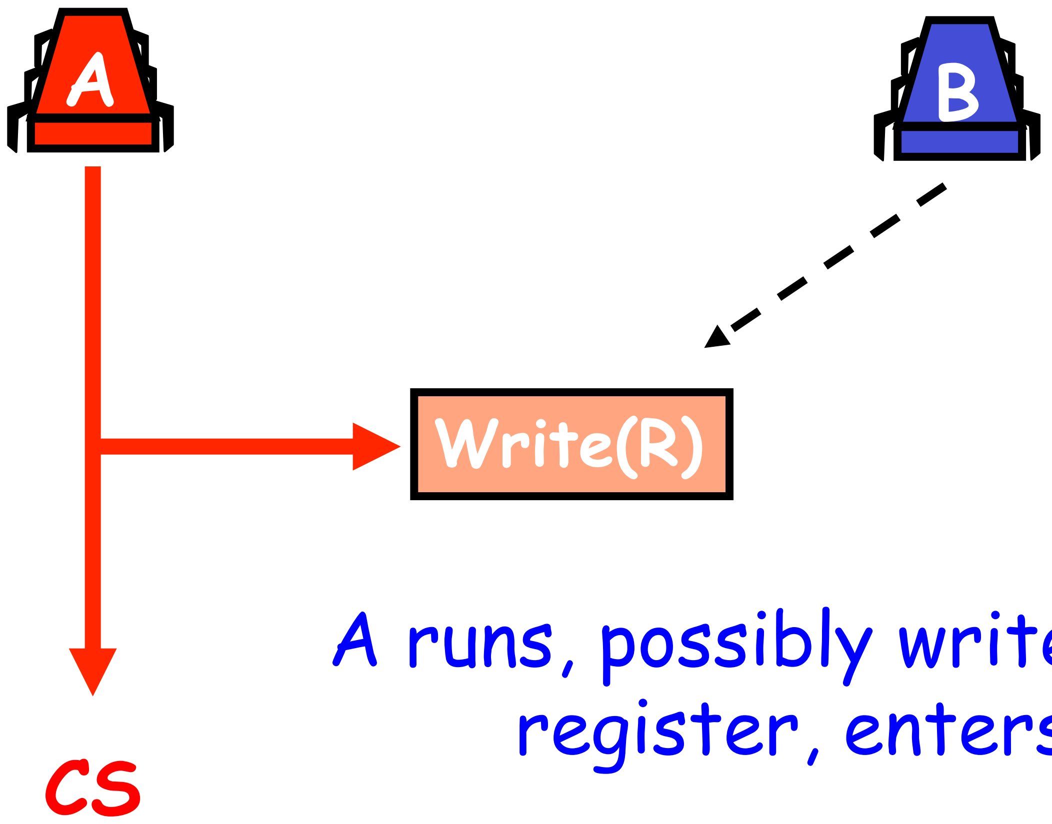
- Threads run, reading and writing R
- Deadlock free so at least one gets in

Covering State for One Register Always Exists



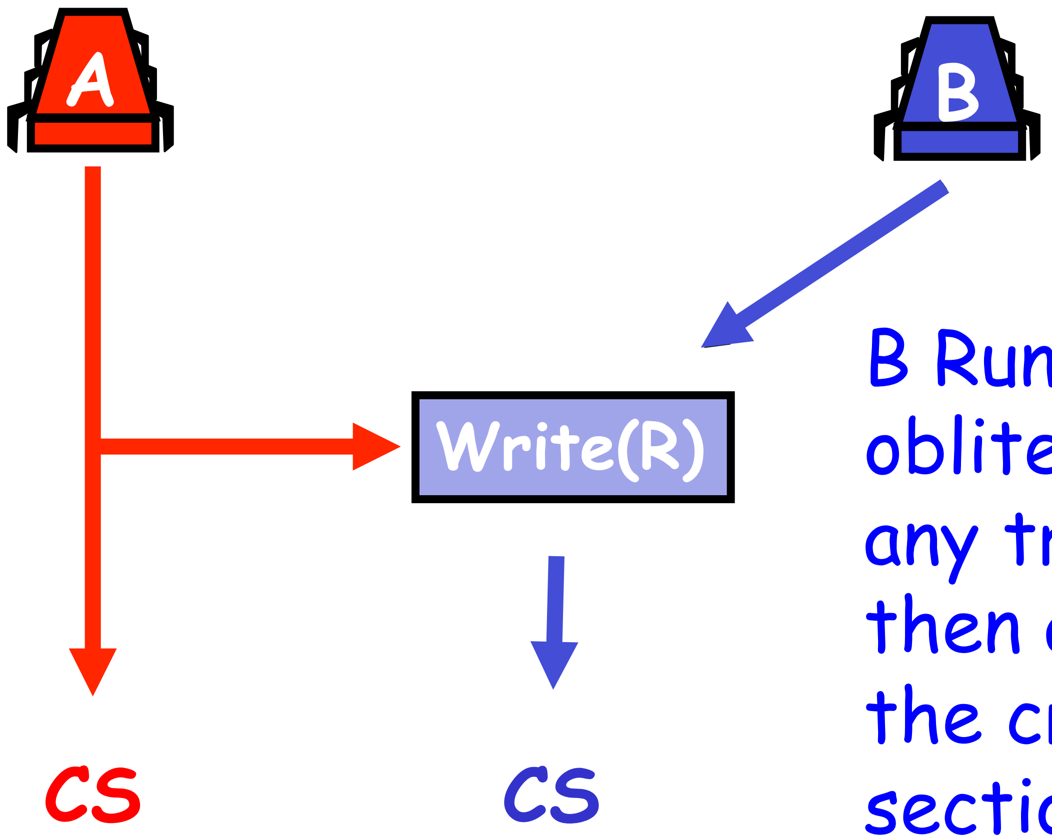
In any protocol B has to write to the register before entering CS, so stop it just before

Proof: Assume Cover of 1



A runs, possibly writes to the register, enters CS

Proof: Assume Cover of 1



B Runs, first obliterating any trace of A, then also enters the critical section

Theorem

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