# Concurrency:
# Multi-core Programming & Data Processing

## Mutual Exclusion


***Prof. P. Felber***
Pascal.Felber@unine.ch
http://iiun.unine.ch/

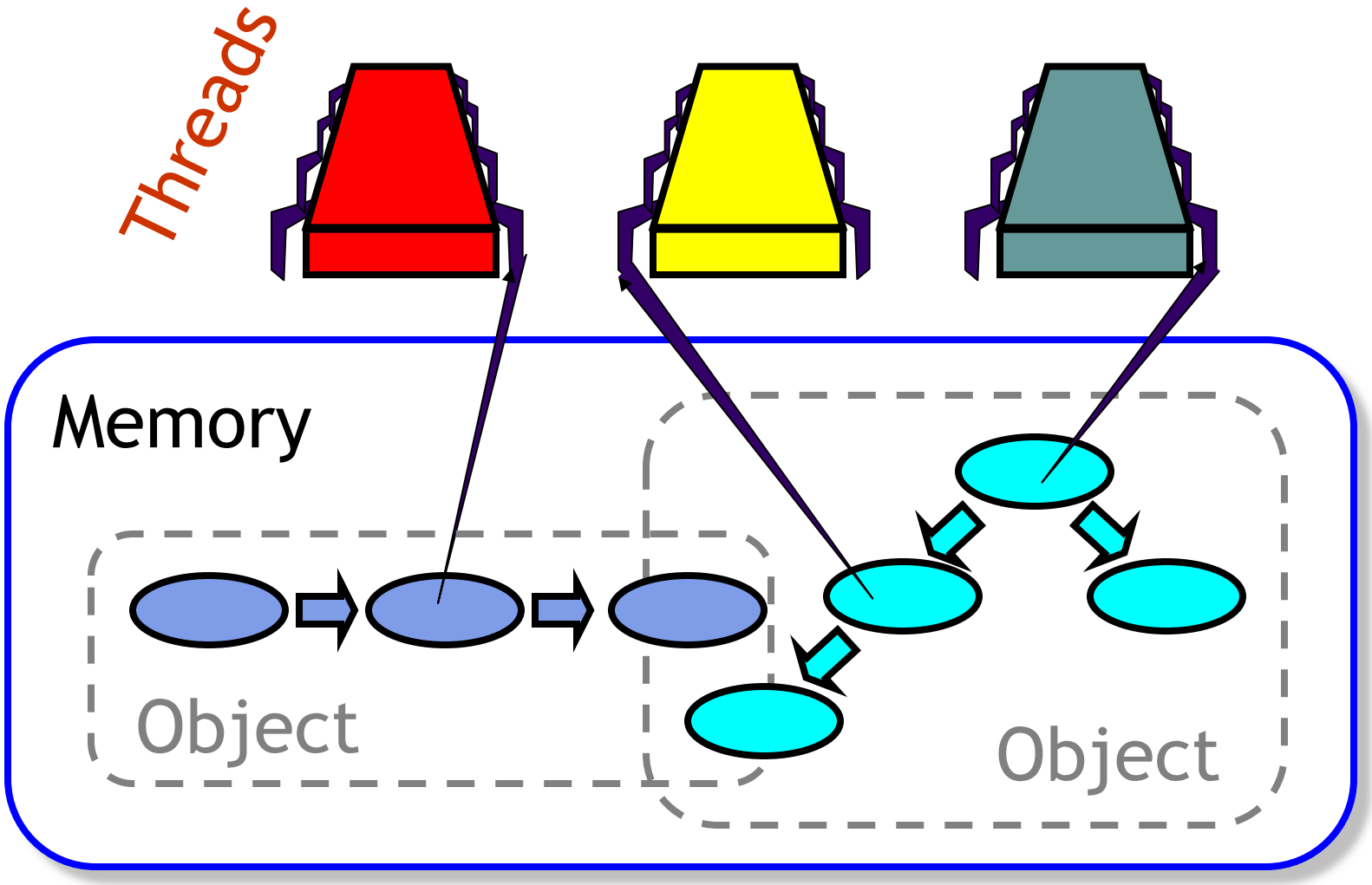

*Based on slides by **Maurice Herlihy** and **Nir Shavit***


Institut d'informatique
Université de Neuchâtel

# Review

- Model of computation
- Asynchrony
- Mutual exclusion

# Concurrent Computation



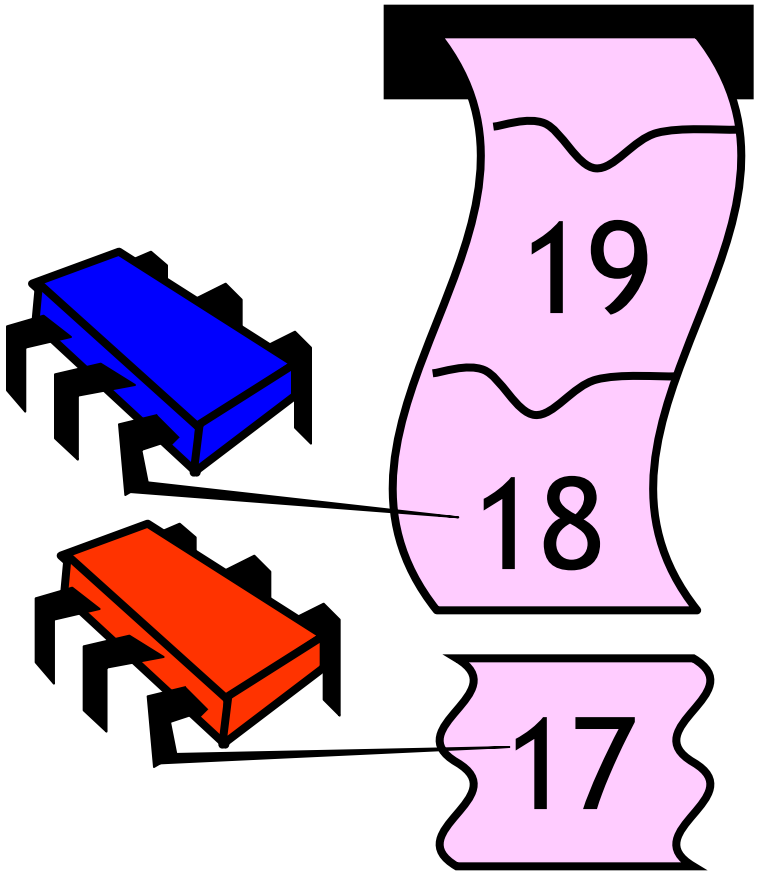
# Asynchrony

- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)

# Parallel Primality Testing

- Challenge
  - Print primes from 1 to $10^{10}$
- Given
  - Ten-processor multiprocessor
  - One thread per processor
- Goal
  - Get ten-fold speedup (or close)

# Shared Counter




Each thread takes a number

# Procedure for Thread *i*

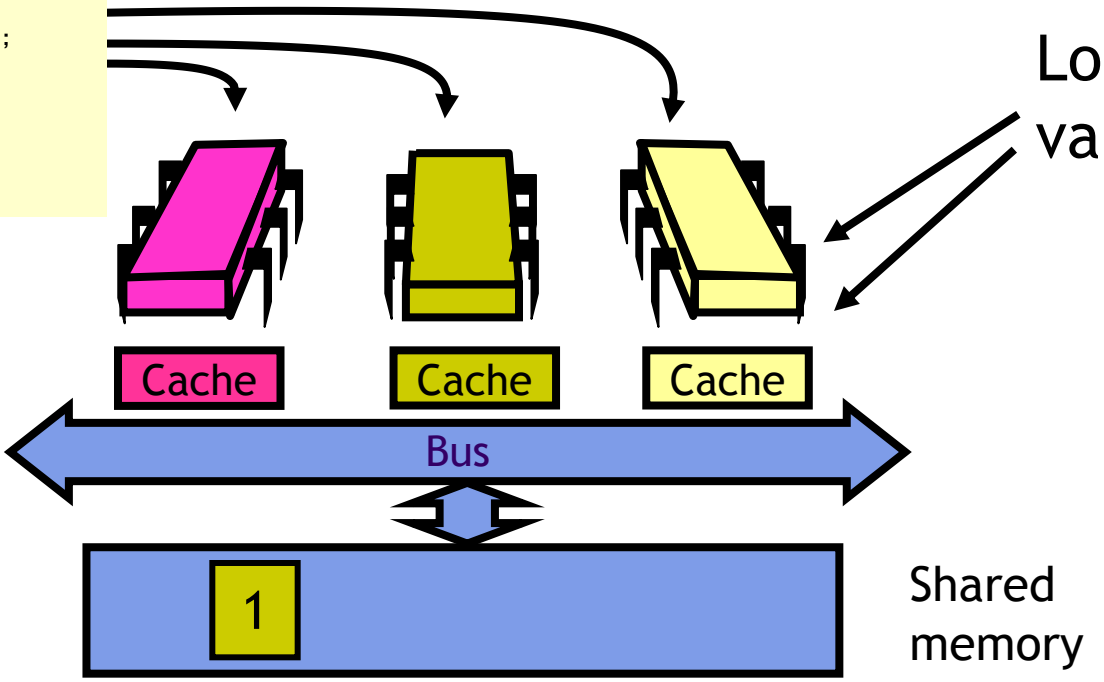
```
Counter counter = new Counter(1);

void primePrint {
  while (true) {
    long j = counter.getAndIncrement();
    if (j > 10^10)
      break;
    if (isPrime(j))
      print(j);
  }
}
```

# Where Things Reside

```
Counter counter = new Counter(1);
void primePrint {
  while (true) {
    long j = counter.getAndIncrement();
    if (j > 10^10)
      break;
    if (isPrime(j))
      print(j);
  }
}
```

Code




Shared counter

# Challenge

```
public class Counter {
  private long value;

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

Make these steps *atomic* (indivisible)

# 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

# What is Time?

- *"Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external."* **(I. Newton, 1689)**

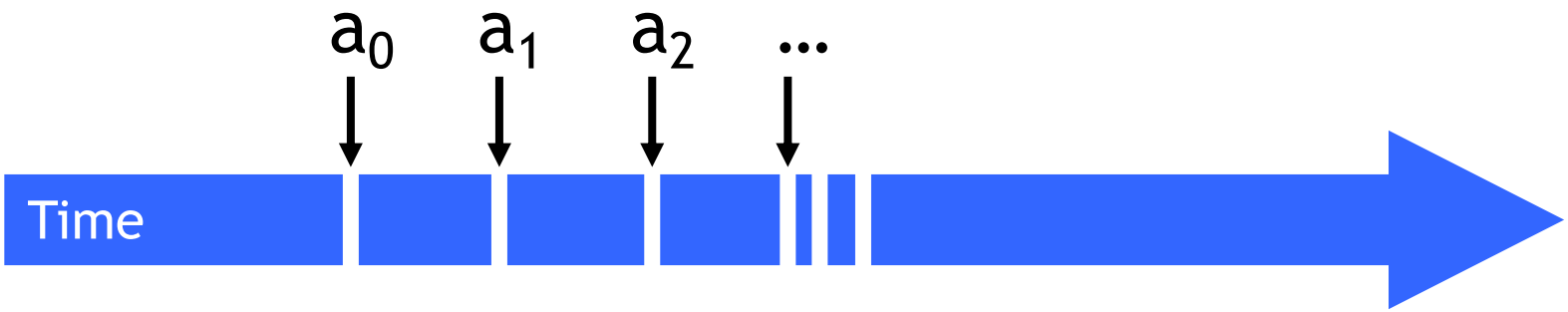
- *"Time is, like, Nature's way of making sure that everything doesn't happen all at once."* **(Anonymous, circa 1968)**

# Events

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

# Threads

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

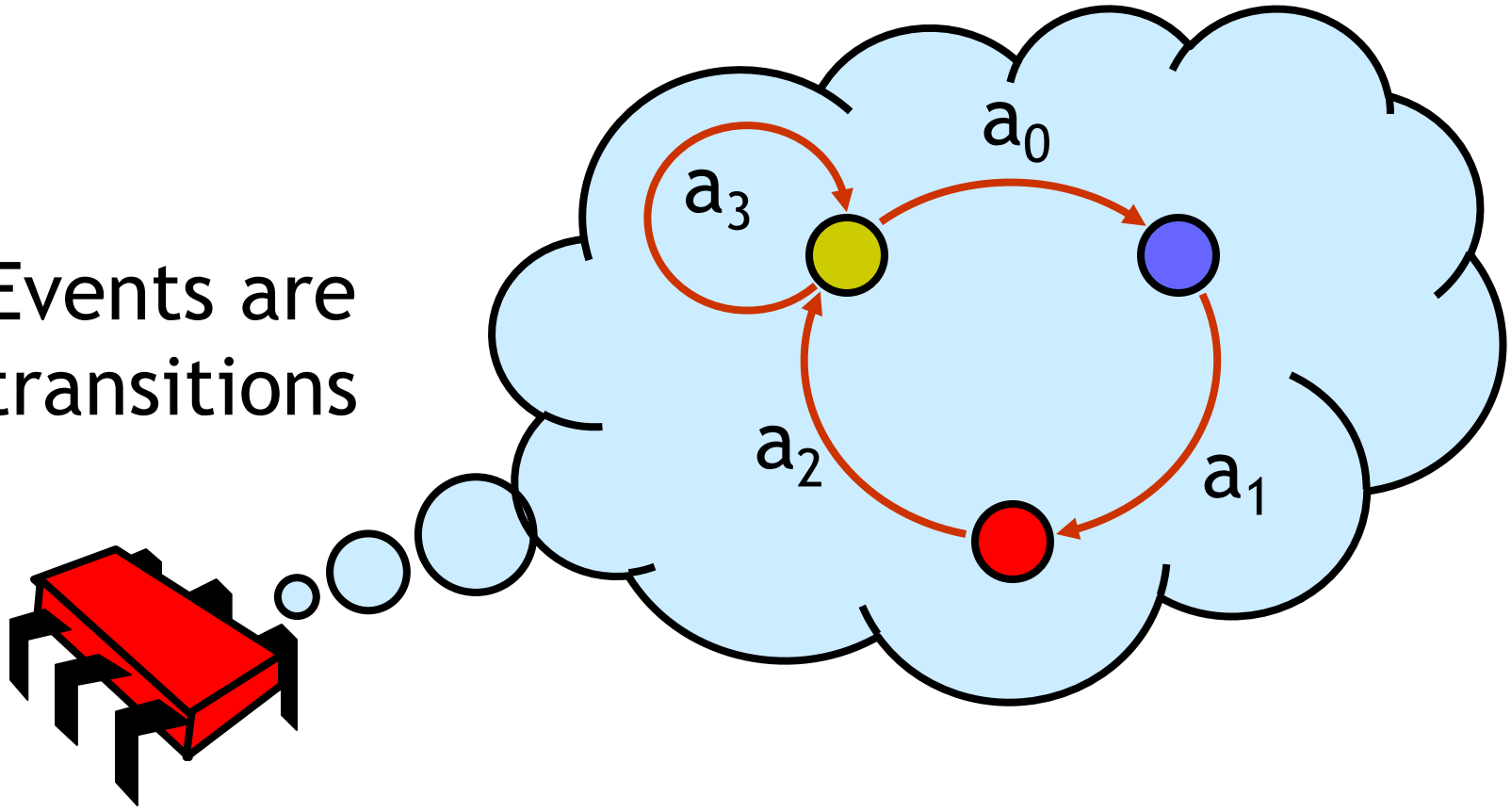


# Example Thread Events

- 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

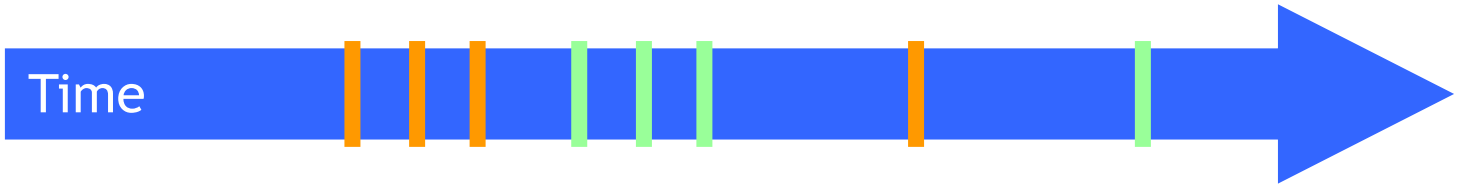
# Concurrency

- Thread A


- Thread B

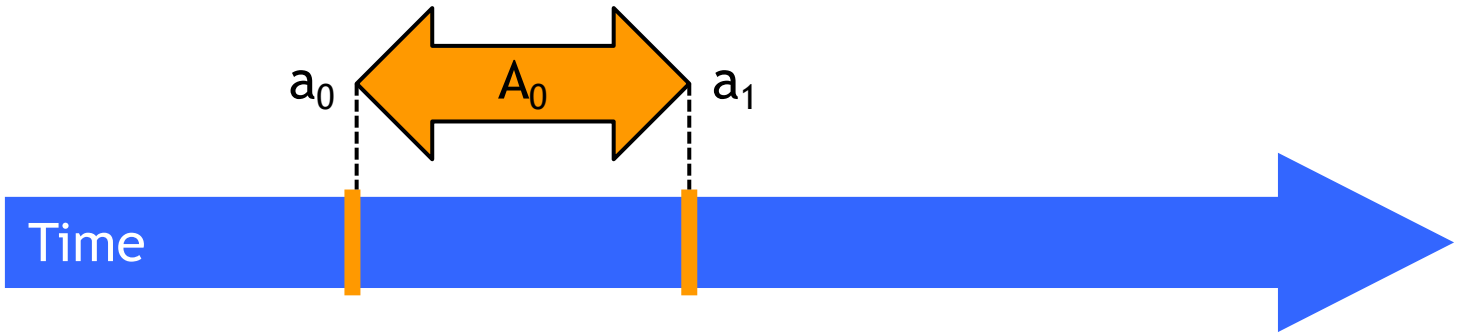
# Interleavings

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

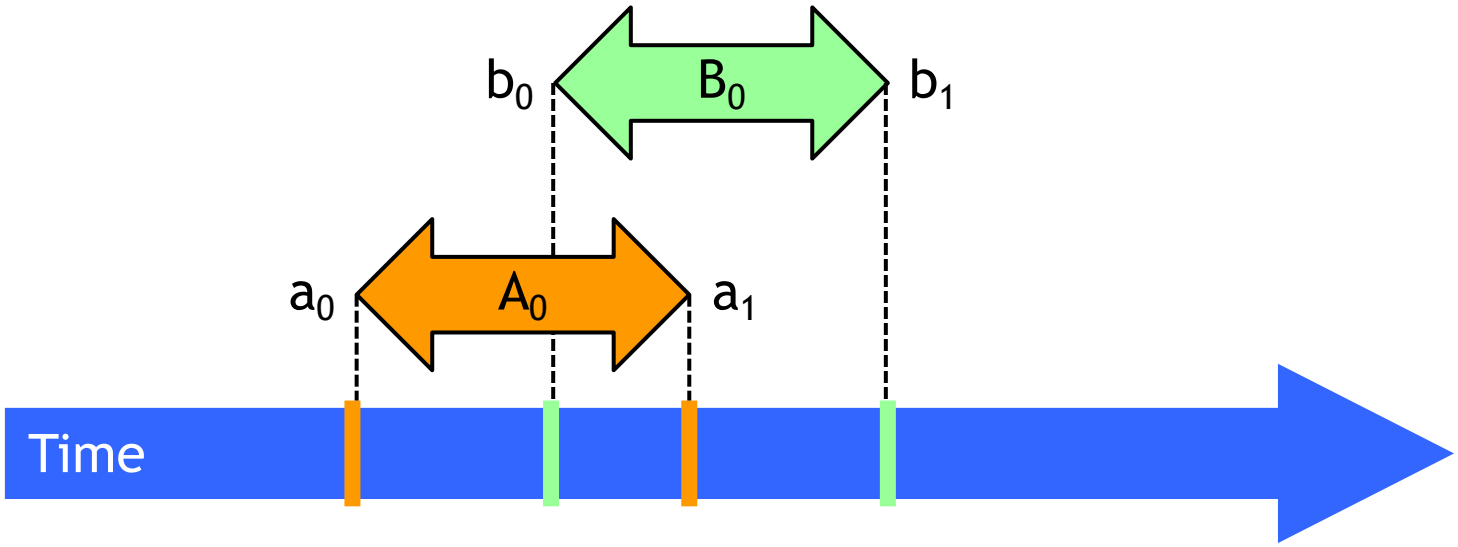


# Intervals

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

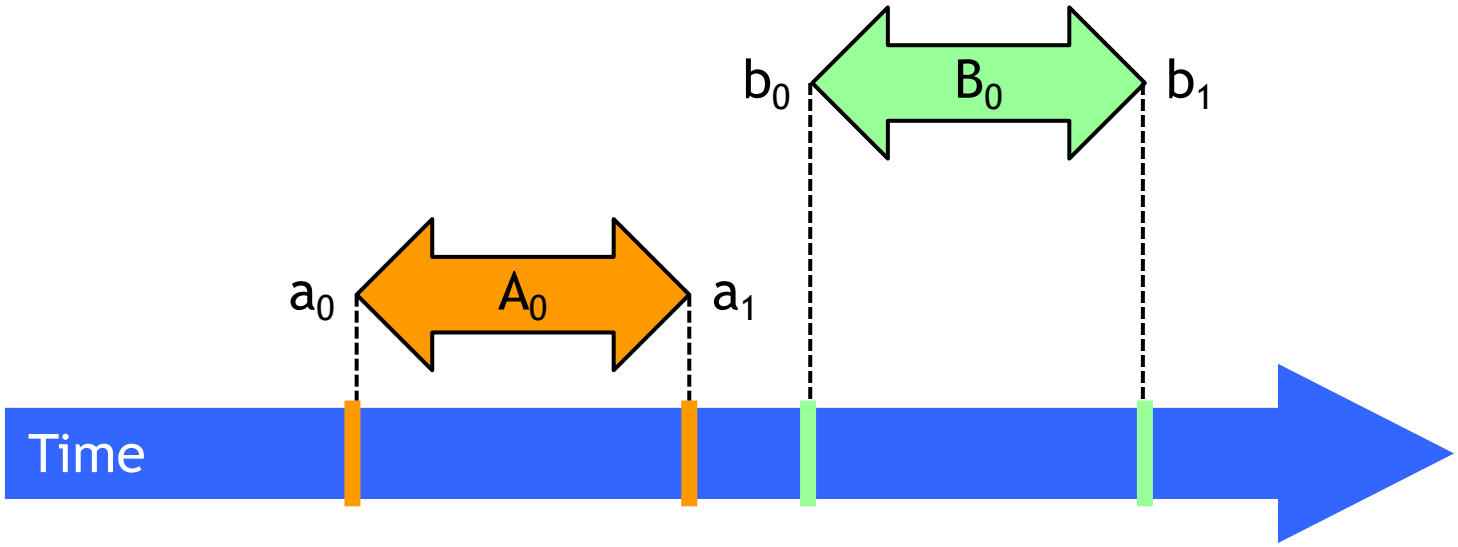


# Intervals may Overlap



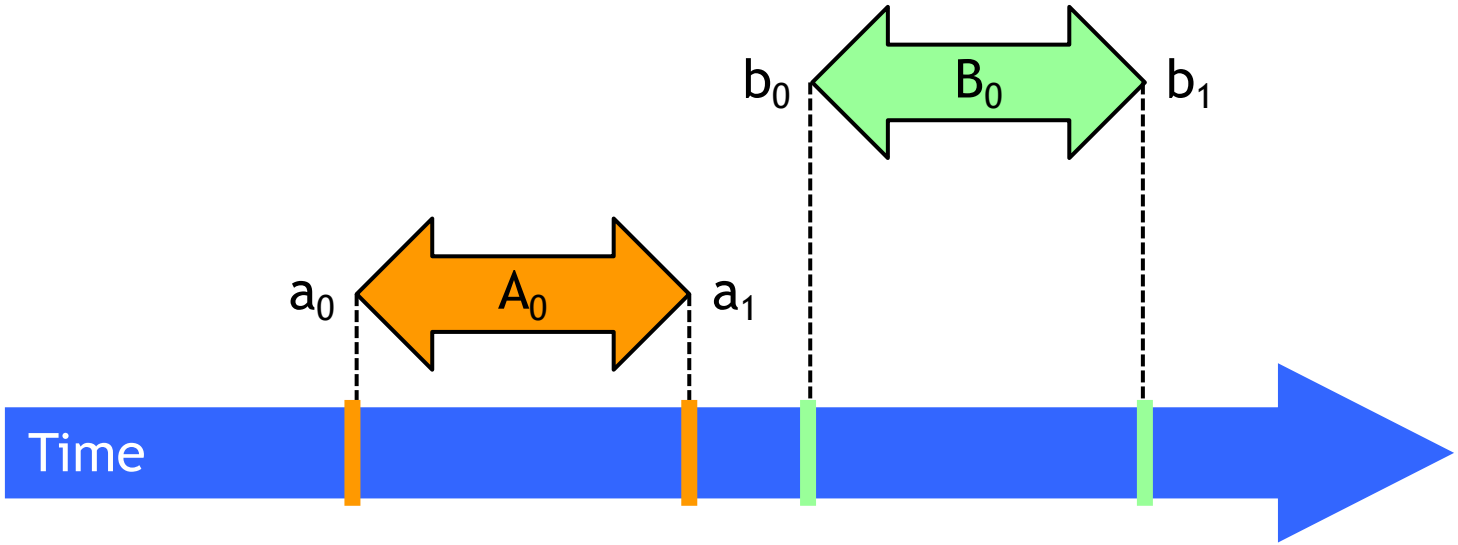
# Intervals may be Disjoint

- Precedence
  - Interval $A_0$ **precedes** interval $B_0$



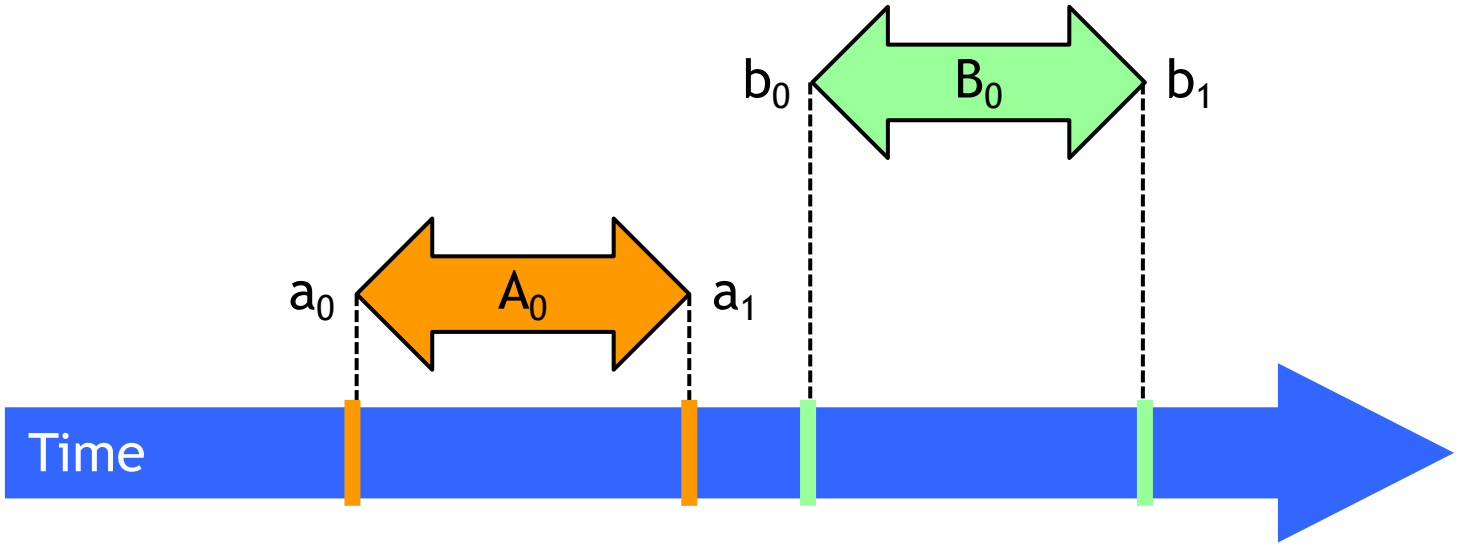
# Precedence

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



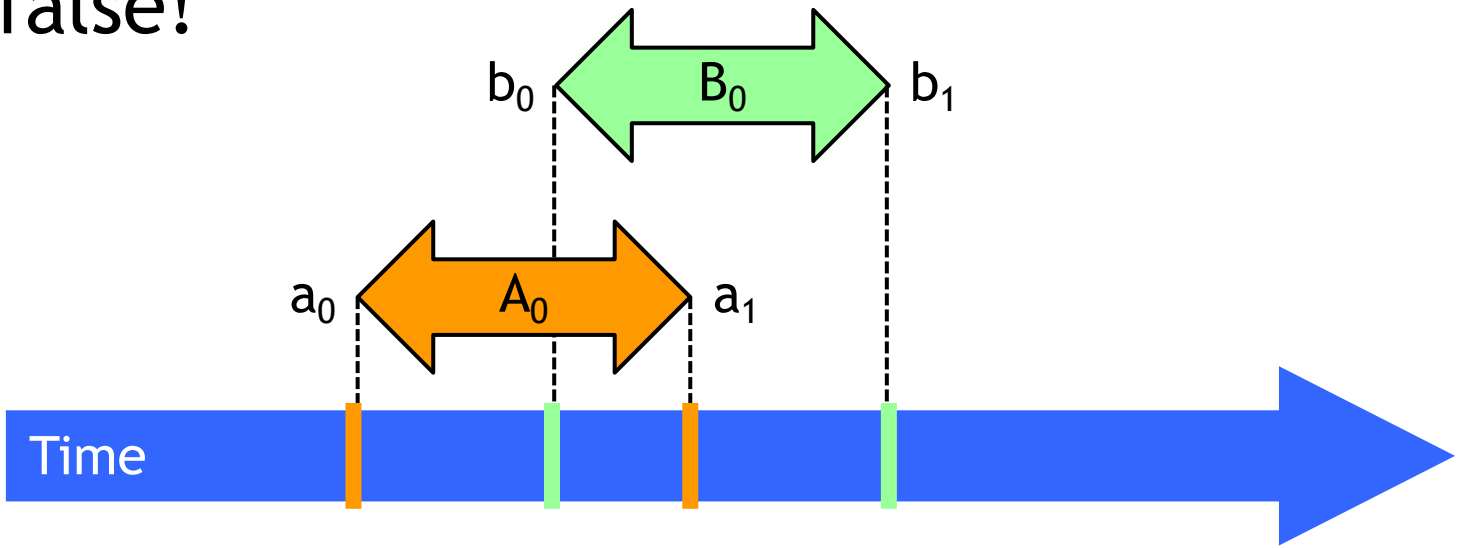
# Precedence Ordering

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



# Precedence Ordering

- Never true that A ➔ A
- If A ➔ B then not true that B ➔ A
- If A ➔ B and B ➔ C then A ➔ C
- Funny thing: A ➔ B and B ➔ A might both be false!



# Partial Orders
## (you may know this already)

- **Irreflexive**
  - Never true that A ➔ A
- **Antisymmetric**
  - If A ➔ B then not true that B ➔ A
- **Transitive**
  - If A ➔ B and B ➔ C then A ➔ C

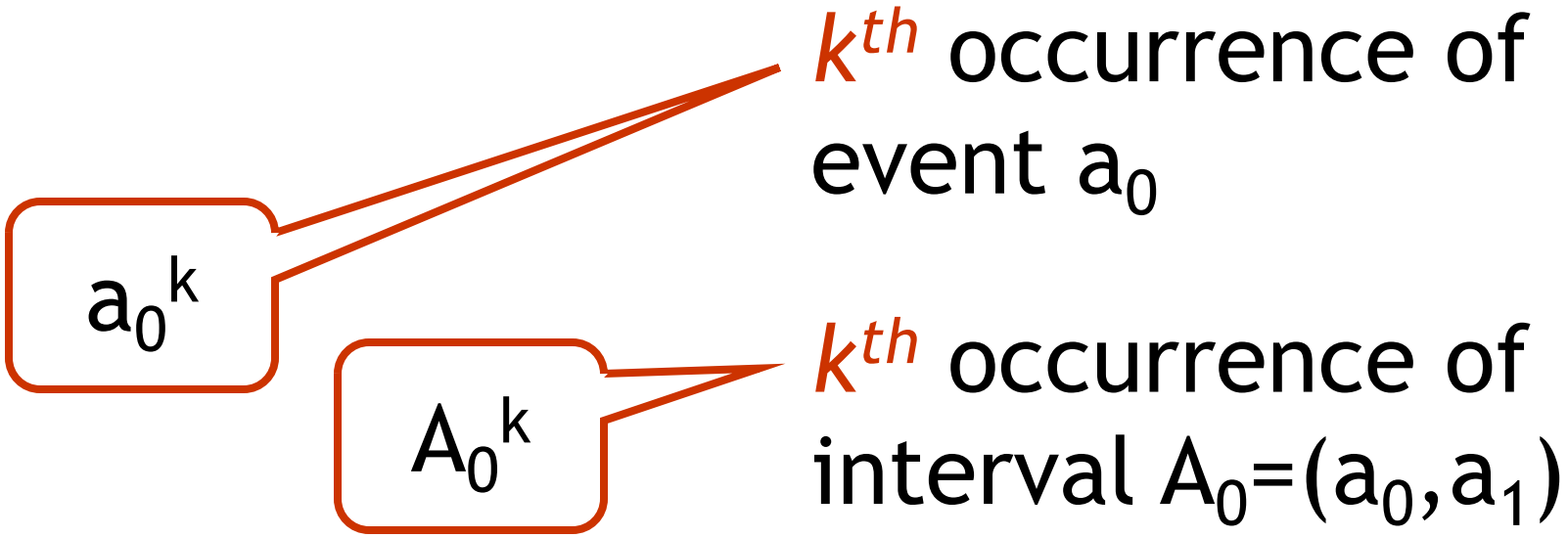
# Total Orders
## (you may know this already)

- Also
  - Irreflexive
  - Antisymmetric
  - Transitive

- Except that for every distinct A, B
  - Either A ➔ B or B ➔ A

# Repeated Events

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



# Implementing a Counter

```
public class Counter {
  private long value;

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

Make these steps *atomic* (indivisible)

# Locks (Mutual Exclusion)

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

Acquire lock

Release lock

# Using Locks

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

Acquire lock

Critical section

Release lock (no matter what)

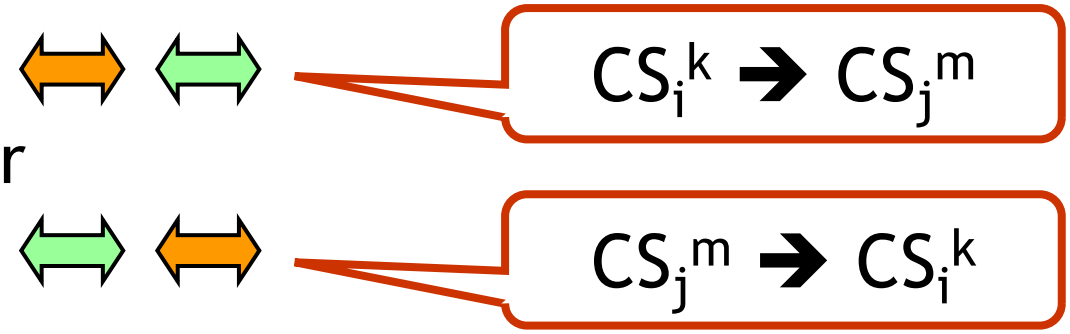
# Using Locks

```
private Lock lock;
…
while{mumble}{
   <non critical section>
   lock.lock();
   <critical section>
   lock.unlock();
}
…
```

- Only well formed executions!
- What are the required properties of locks?

# Mutual Exclusion

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



# 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

# Lockout-Free

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

# Two-Thread vs. *n*-Thread

- 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;
  …
  }
}
```

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

Set my flag

Wait for other flag to go false

No longer interested

# LockOne Satisfies Mutual Exclusion

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

# From the Code

$write_A(flag[A]=true) \rightarrow read_A(flag[B]==false) \rightarrow CS_A$

$write_B(flag[B]=true) \rightarrow read_B(flag[A]==false) \rightarrow CS_B$

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

# From the Assumption

$read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$

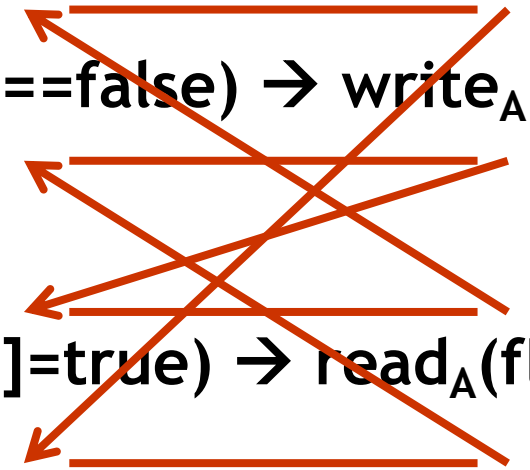
$read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$

# Combining

## Assumptions

$read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$

$read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$



## From the code

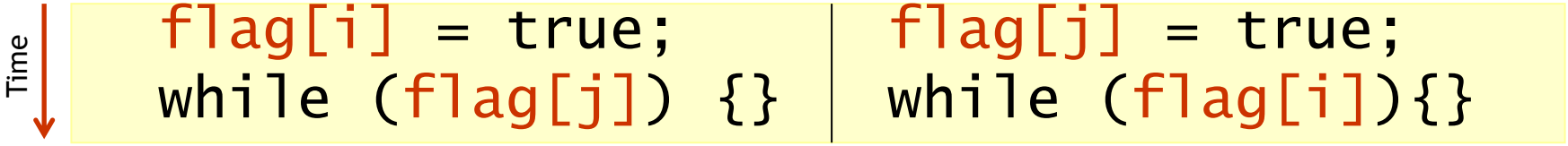
$write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$

$write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$

**Cycle!**

# Deadlock Freedom

- LockOne fails deadlock-freedom
  - Concurrent execution can deadlock




  - Sequential executions OK

# 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

Wait for permission

Nothing to do

# LockTwo Claims

- Satisfies mutual exclusion
  - If thread `i` in CS
  - Then `victim == j`
  - Cannot be both `i` and `j`
- Not deadlock free
  - Sequential execution deadlocks
  - Concurrent execution does not

```
{
  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;
}
```

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

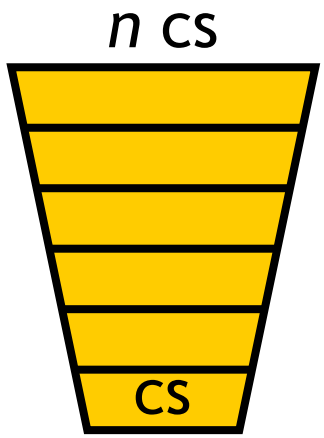
# Lockout Free

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

- Thread `i` blocked only if `j` repeatedly re-enters with `flag[j] == true` and `victim == i`
- When `j` re-enters, it sets `victim` to `j`, so `i` gets in

# The Filter Algorithm for *n* Threads

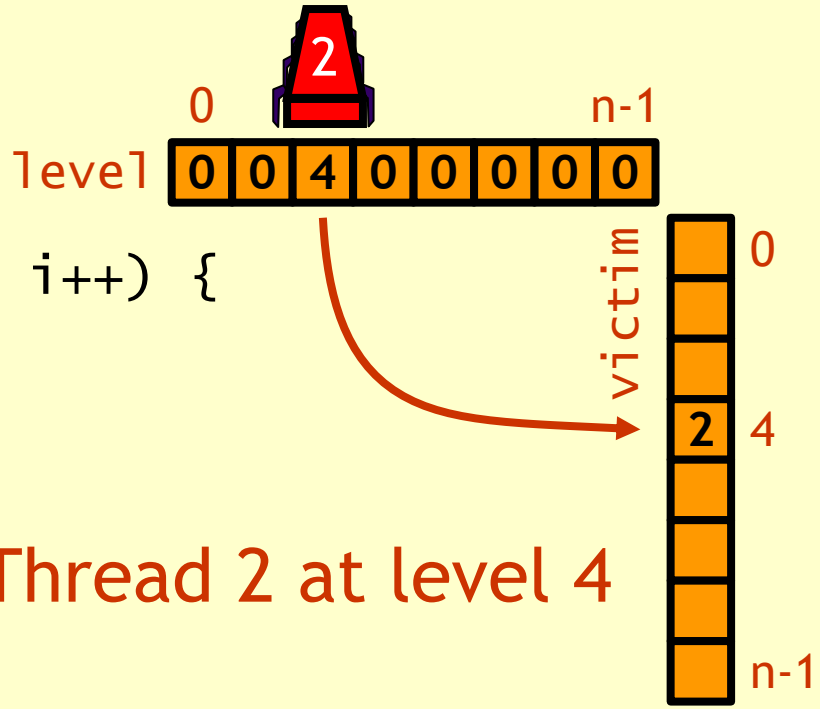
- There are *n-1* "waiting rooms" called levels
- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through



# 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 = 0; i < n; i++) {
       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 unlock() { level[i] = 0; }
}
```

One level at a time

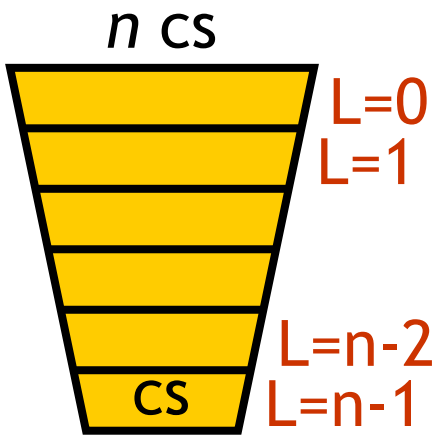
Announce intention to enter level L

Give priority to anyone but me

Wait as long as someone else is at same or higher level and I'm designated victim (enter level L when loop complete)

# Claim

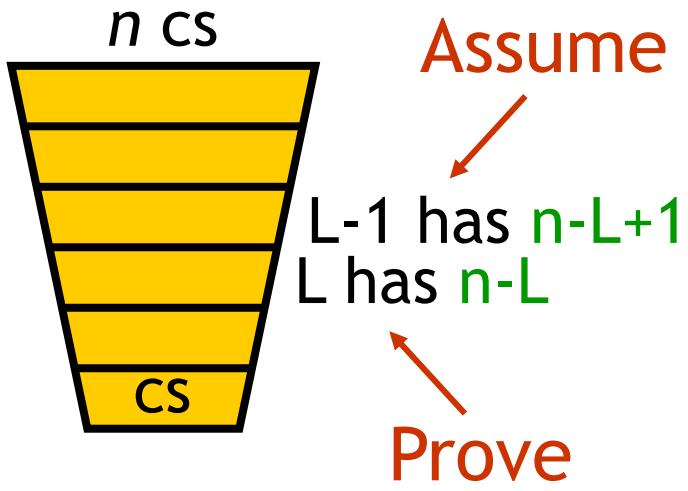
- Start at level L=0
- At most n-L threads enter level L
- Mutual exclusion at level L=n-1



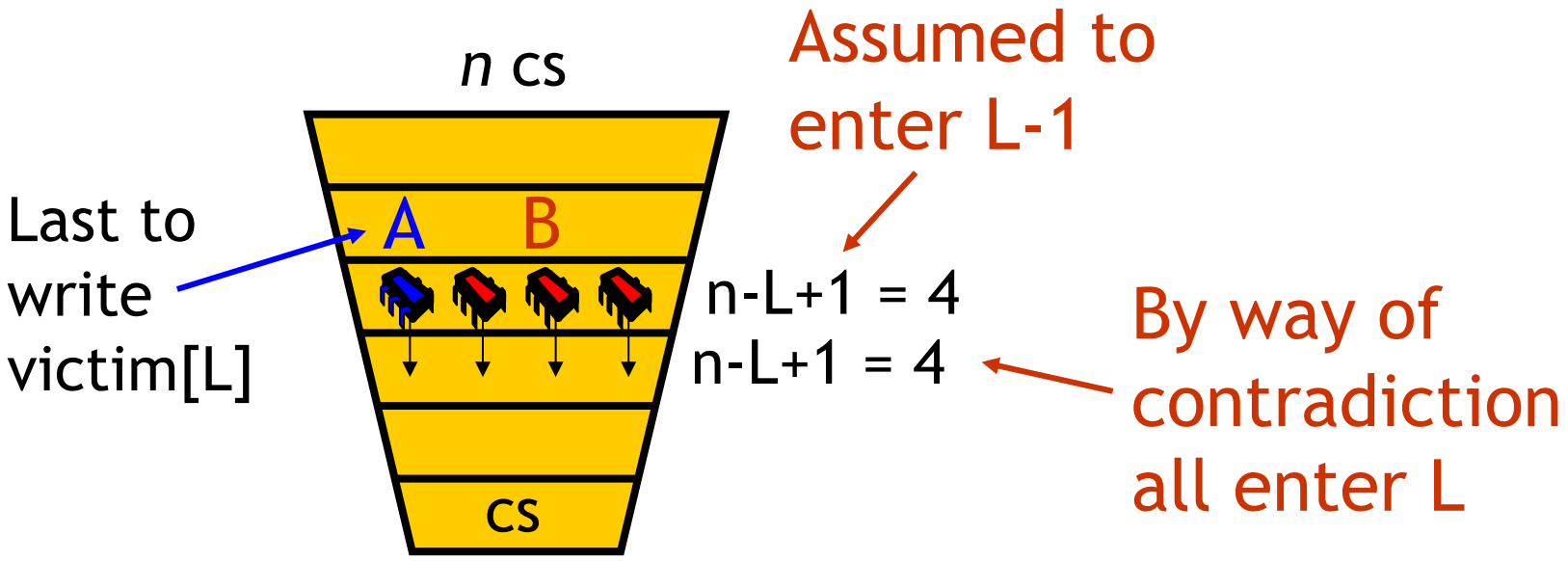
# Induction Hypothesis

- No more than n-L+1 threads 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 victim[L] == A could not have entered

# Observations

1. $write_B(level[B]=L) \rightarrow write_B(victim[L]=B)$ (code)
2. $write_A(victim[L]=A) \rightarrow read_A(level[B])$ (code)
3. $write_B(victim[L]=B) \rightarrow write_A(victim[L]=A)$

   (by assumption: A is the last thread to write victim[L])

```
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. $write_B(level[B]=L) \rightarrow write_B(victim[L]=B)$ (code)
3. $write_B(victim[L]=B) \rightarrow write_A(victim[L]=A)$ (ass.)
2. $write_A(victim[L]=A) \rightarrow read_A(level[B])$ (code)

Thus, A read level[B] ≥ L, A was last to write victim[L], so it could not have entered level L!

```
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) {}
  }
}
```

# No Lockout

- Filter Lock satisfies “no lockout” properties
  - Just like Peterson algorithm at any level
  - So no one starves (no lockout)
- 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…

- Divide `lock()` method into 2 parts
  - Doorway interval ($D_A$), always finishes in finite number of steps
  - Waiting interval ($W_A$), may take unbounded number of 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 if **r = 0**

# Fairness Again

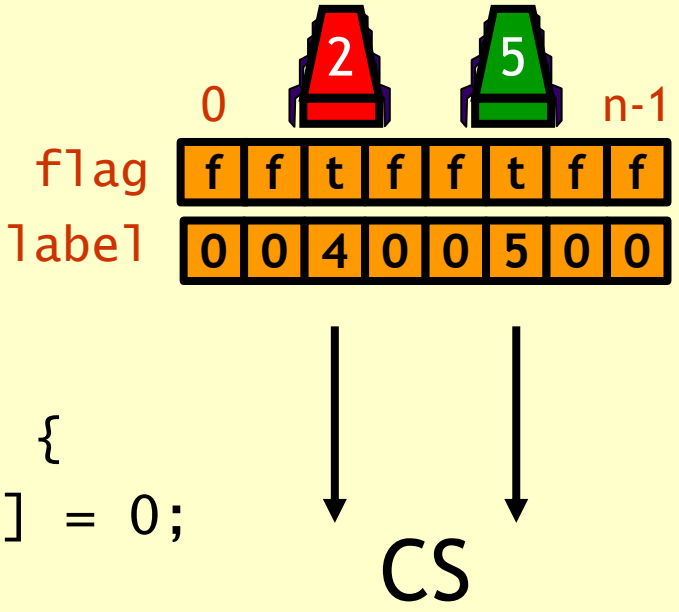
- Filter Lock satisfies some properties
  - No one starves (no lockout)
  - But very weak fairness
    - Not *r*-bounded for any *r*!
  - That's pretty lame…

# 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 int[] label;
  public Bakery (int n) {
    flag  = new boolean[n];
    label = new int[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)) {}
  }
  …
}
```

Doorway

Take increasing label (read labels in some arbitrary order)

I'm interested

Someone is interested…

…with lower label in lexicographic order

# Bakery Algorithm

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

No longer interested

# No Deadlock

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

# First-Come-First-Served

- If $D_A$ ➔ $D_B$ then A's label is earlier
  - $write_A(label[A])$ ➔ $read_B(label[A])$ ➔ $write_B(label[B])$ ➔ $read_B(flag[A])$
- So B is locked out while flag[A] == 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] == 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
  - $\text{labeling}_B$ ➔ $\text{read}_B(\text{flag}[A])$ ➔ $\text{write}_A(\text{flag}[A])$ ➔ $\text{labeling}_A$
- Which contradicts the assumption that A has an earlier label

# Deep Philosophical Question

- The Bakery Algorithm is
  - Succinct
  - Elegant
  - 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, like `flag[]`) registers are needed to solve deadlock-free mutual exclusion

# Summary of Lecture

- In the 1960's many incorrect solutions to lockout-free mutual exclusion using RW-registers were published…
- Today we know how to solve FIFO *n* thread mutual exclusion using *2n* RW-registers