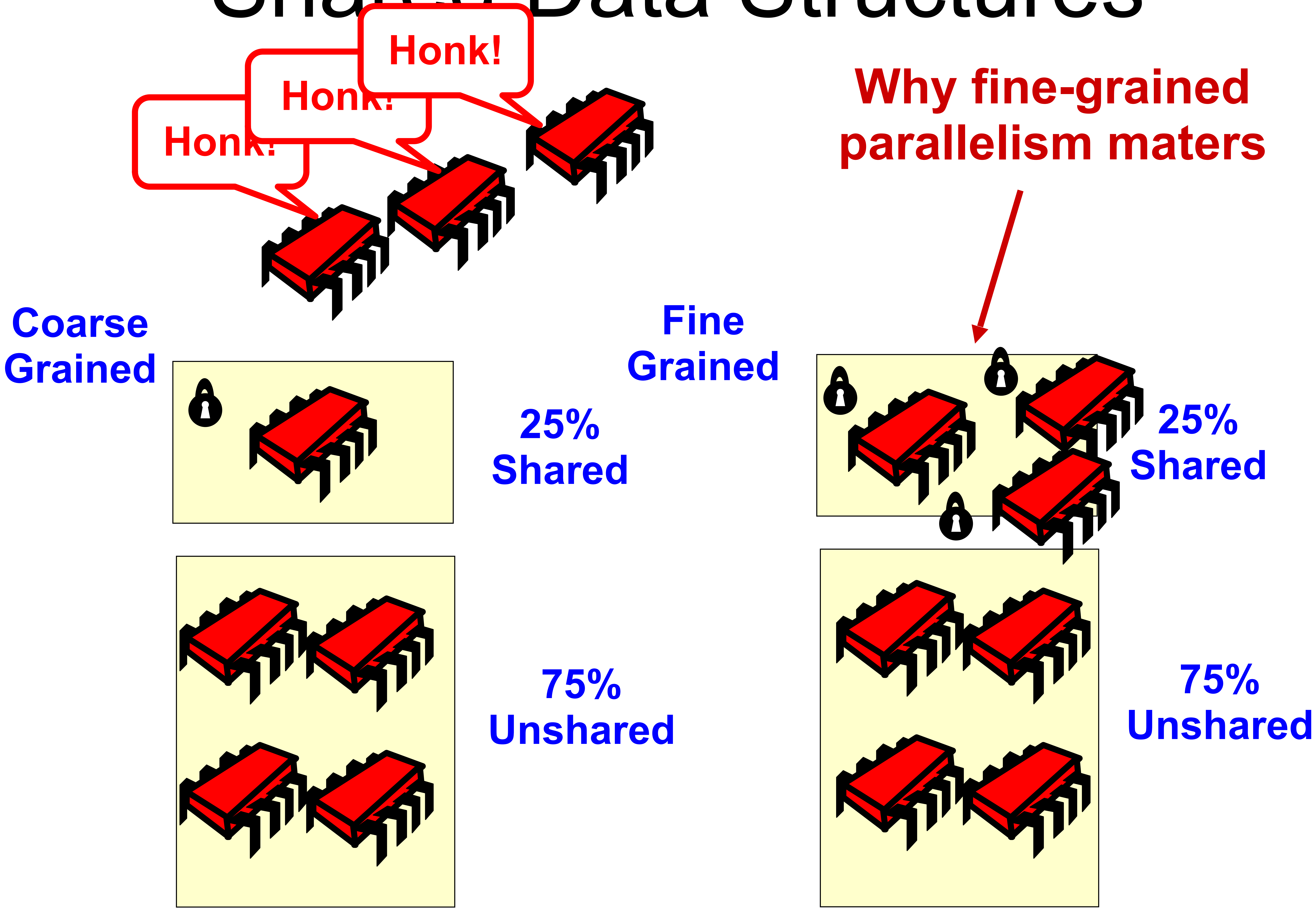
YSC4231: Parallel, Concurrent and Distributed Programming

Mutual Exclusion

Review: Amdahl's Law

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

Shared Data Structures



Example Synchronization Paradigms

- Mutual exclusion
- Readers-Writers
- Producer-Consumer

Mutual Exclusion

- We will clarify our understanding of mutual exclusion
- We will also see how to *reason* about various properties in an asynchronous concurrent setting

Mutual Exclusion

In his 1965 paper E. W. Dijkstra wrote:

"Given in this paper is a solution to a problem which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. [...]
Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved."

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

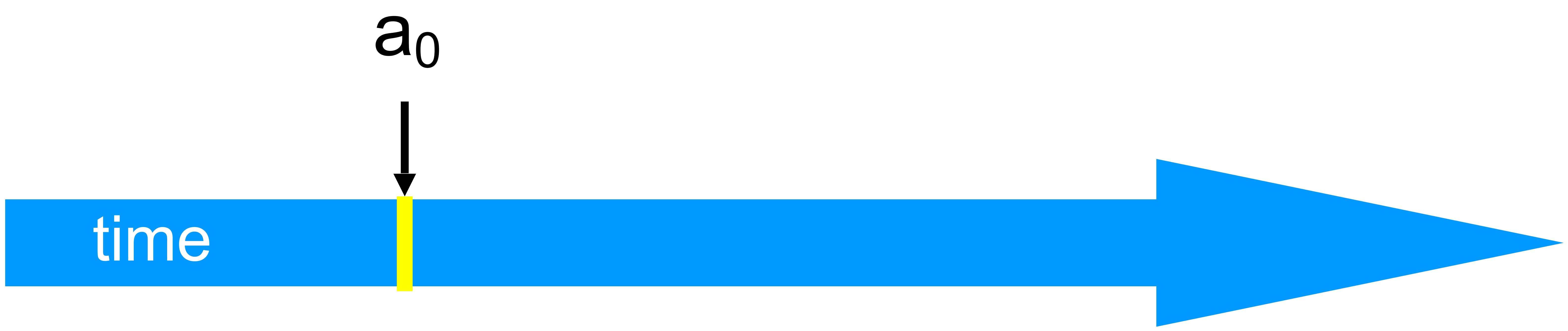
Time

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

- “*Time is what keeps everything from happening at once*.” (Ray Cummings, 1922)

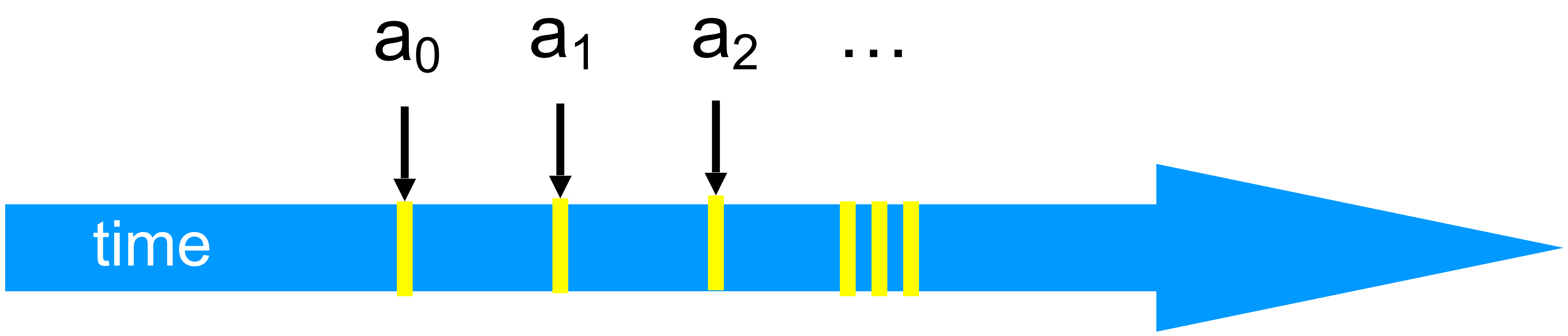
Events

- 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: $a_0 \rightarrow a_1$ indicates order

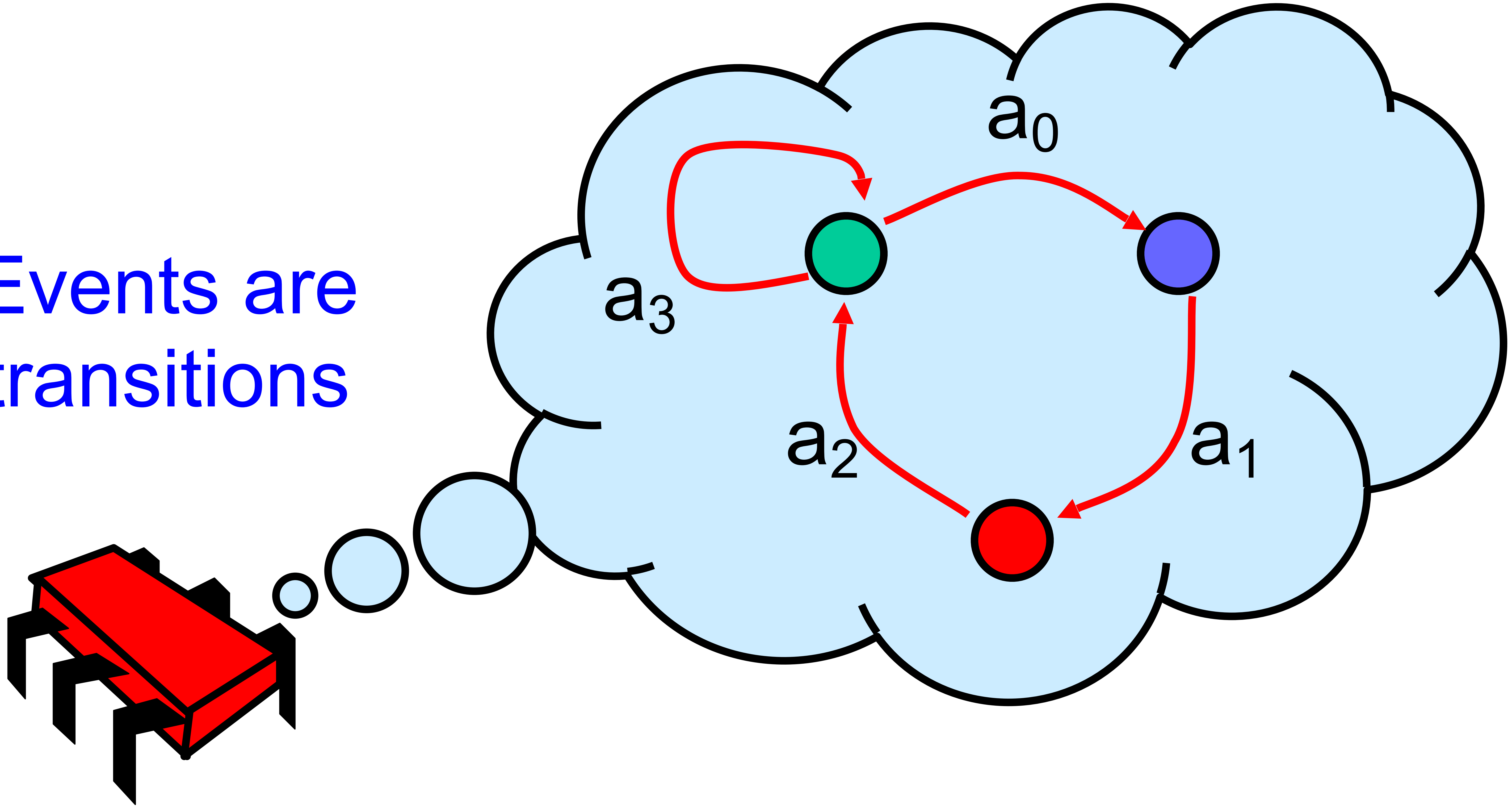


Example Thread Events

- 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

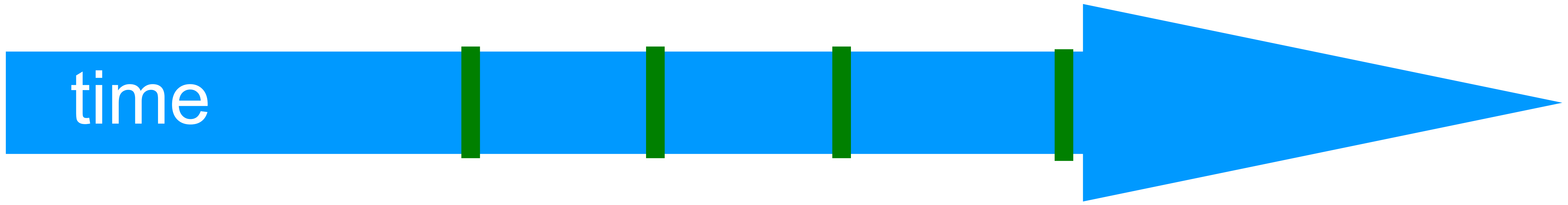
Concurrency

- Thread A

Concurrency

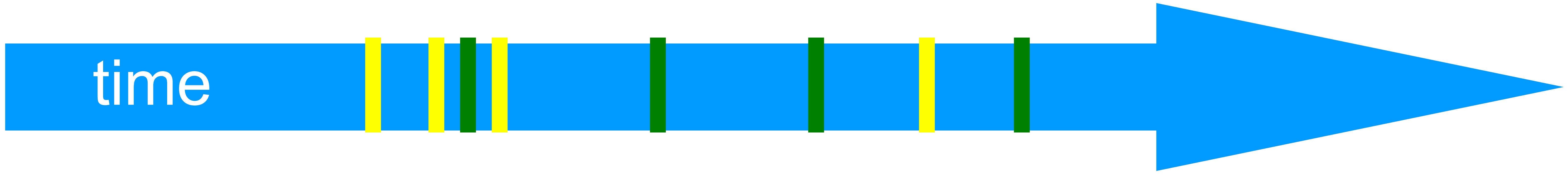
- Thread A

- Thread B



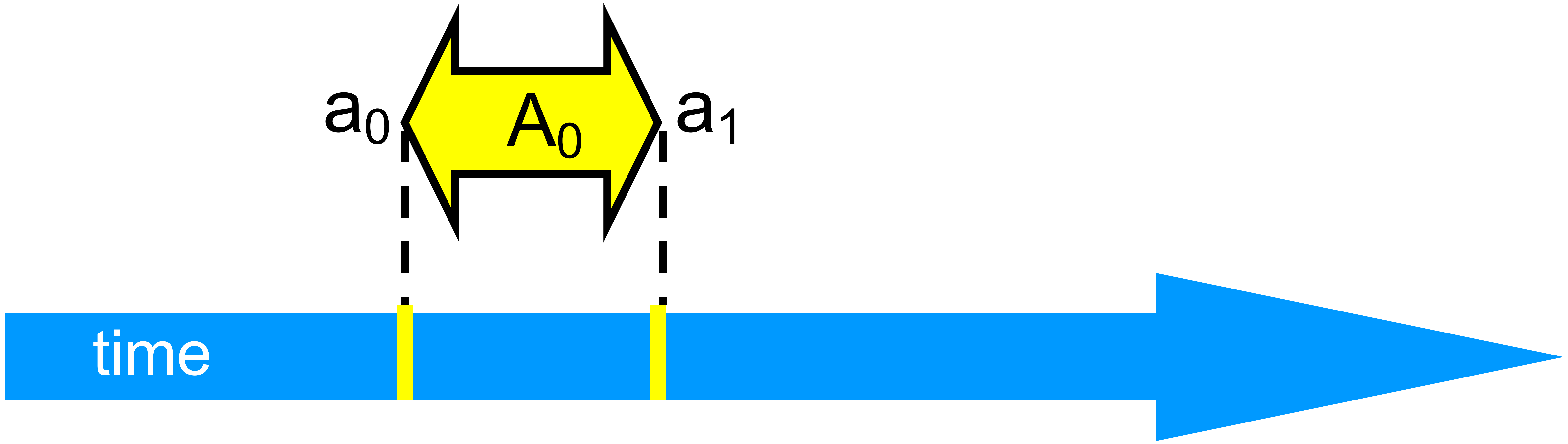
Interleavings

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

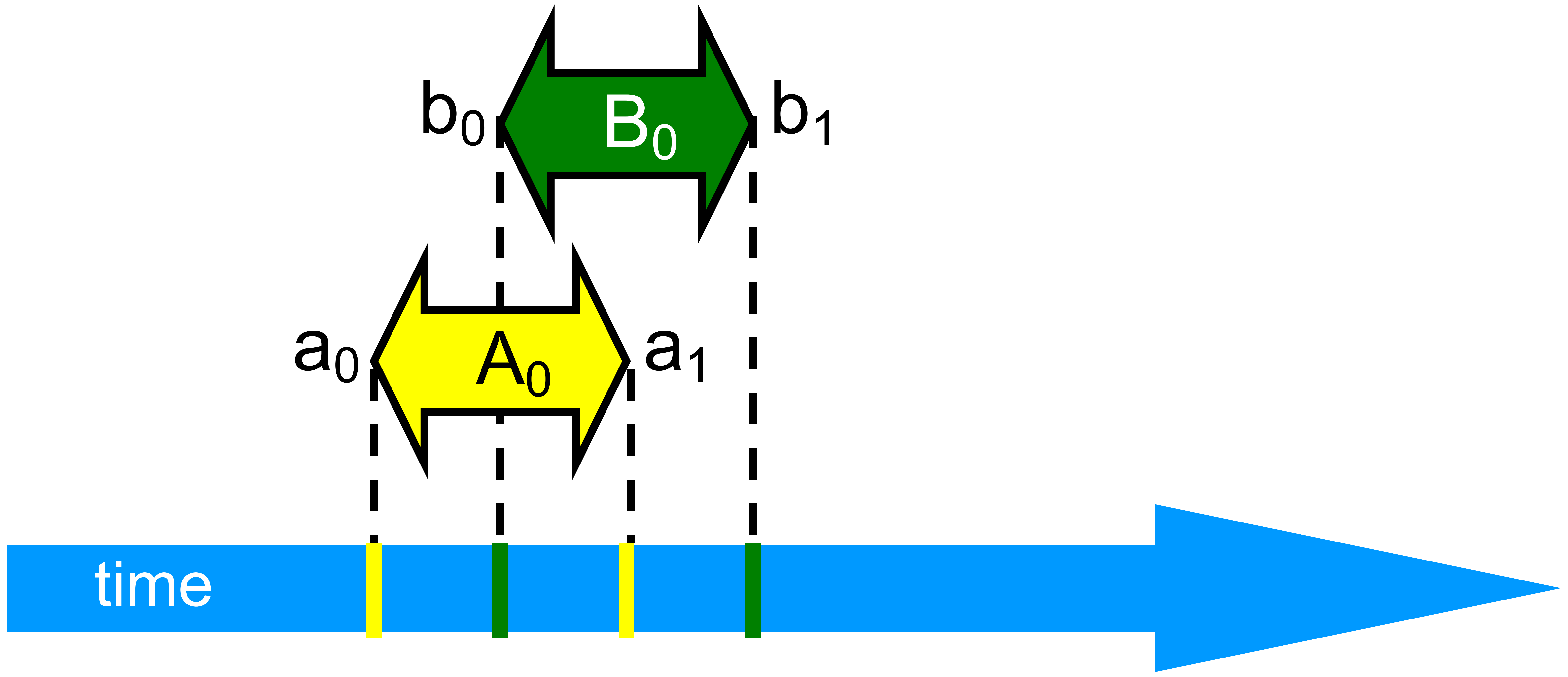


Intervals

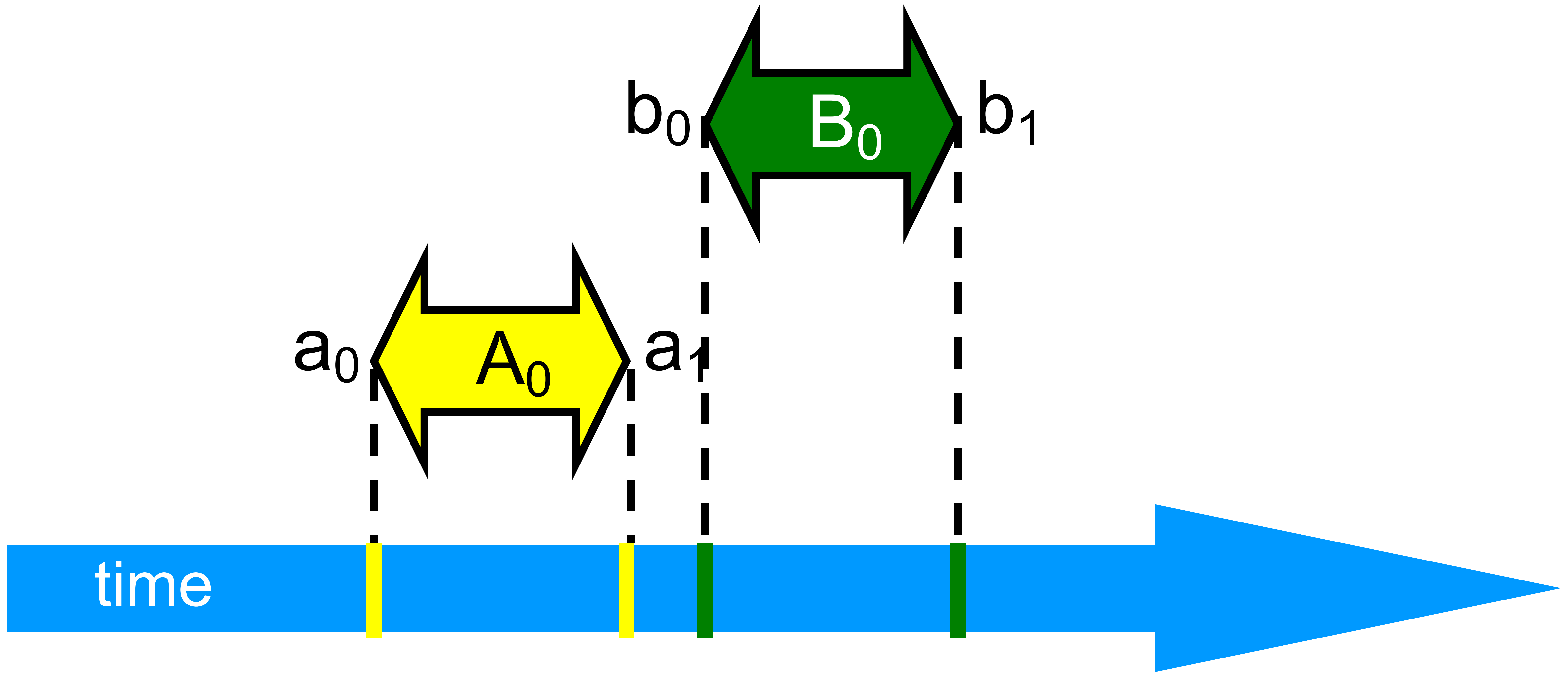
- An *interval* $A_0 = (a_0, a_1)$ is
  - Time between events a_0 and a_1



Intervals may Overlap

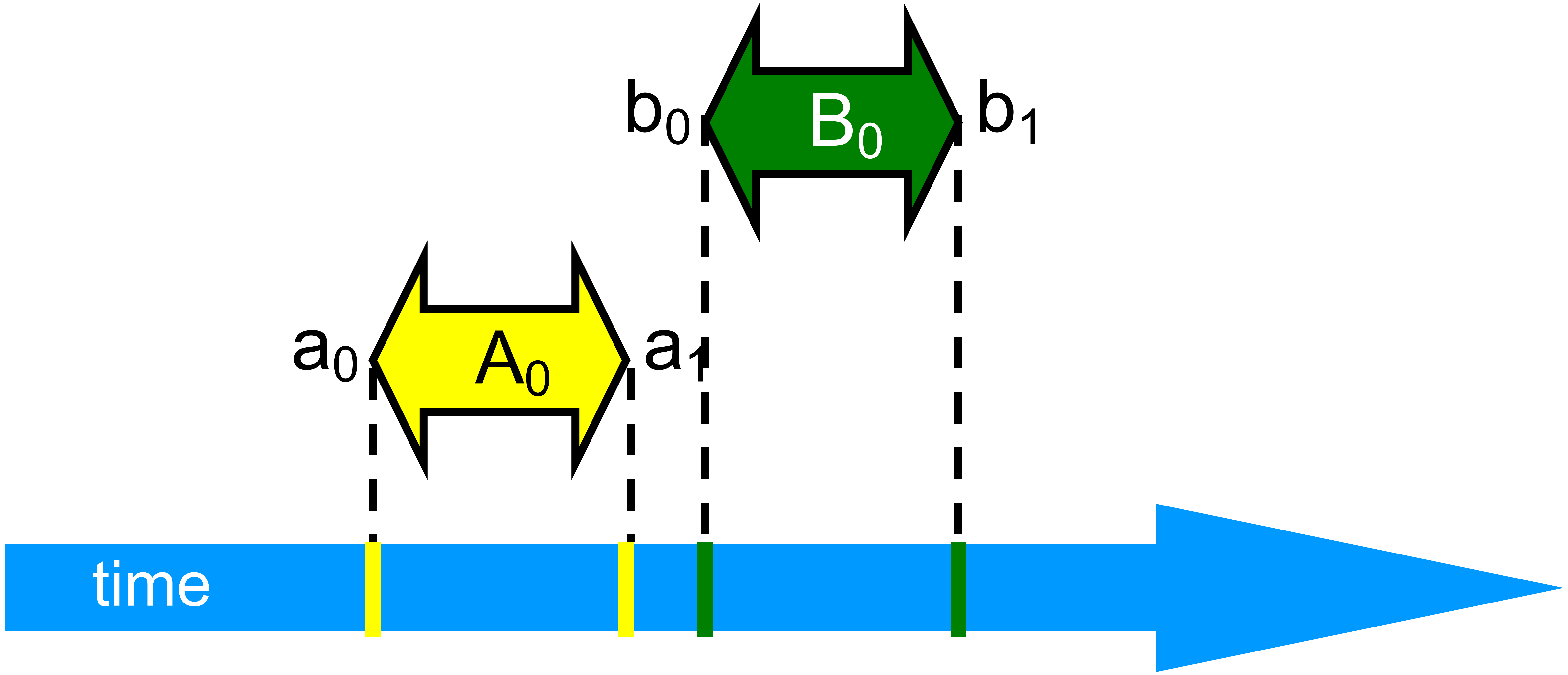


Intervals may be Disjoint

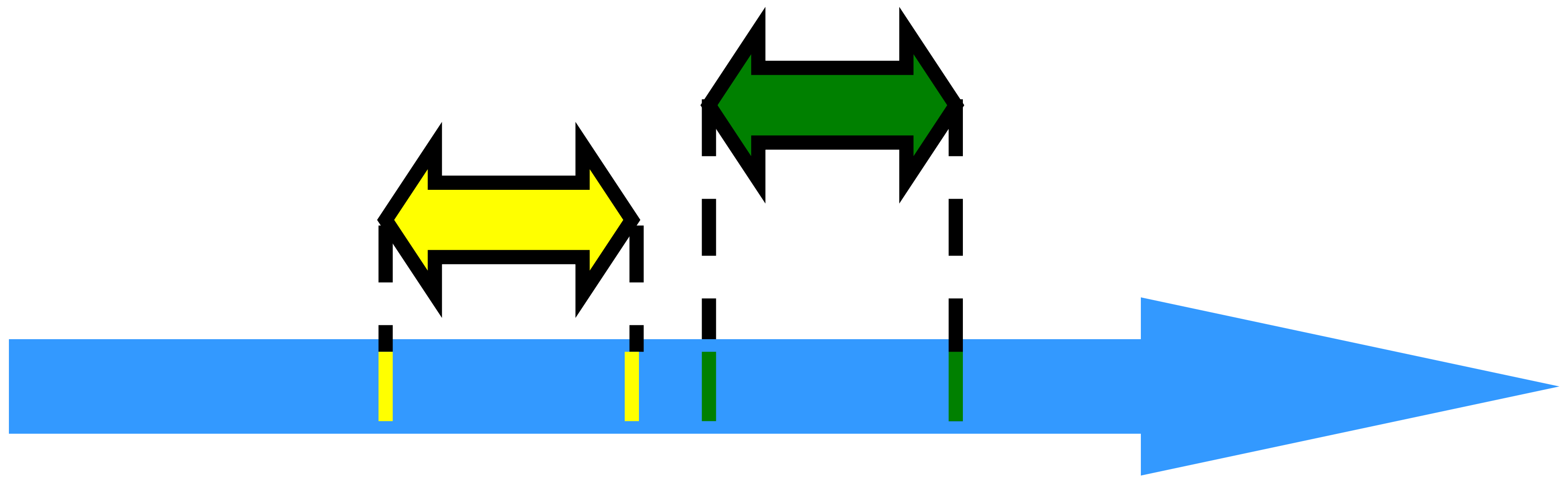


Precedence

Interval A_0 precedes interval B_0

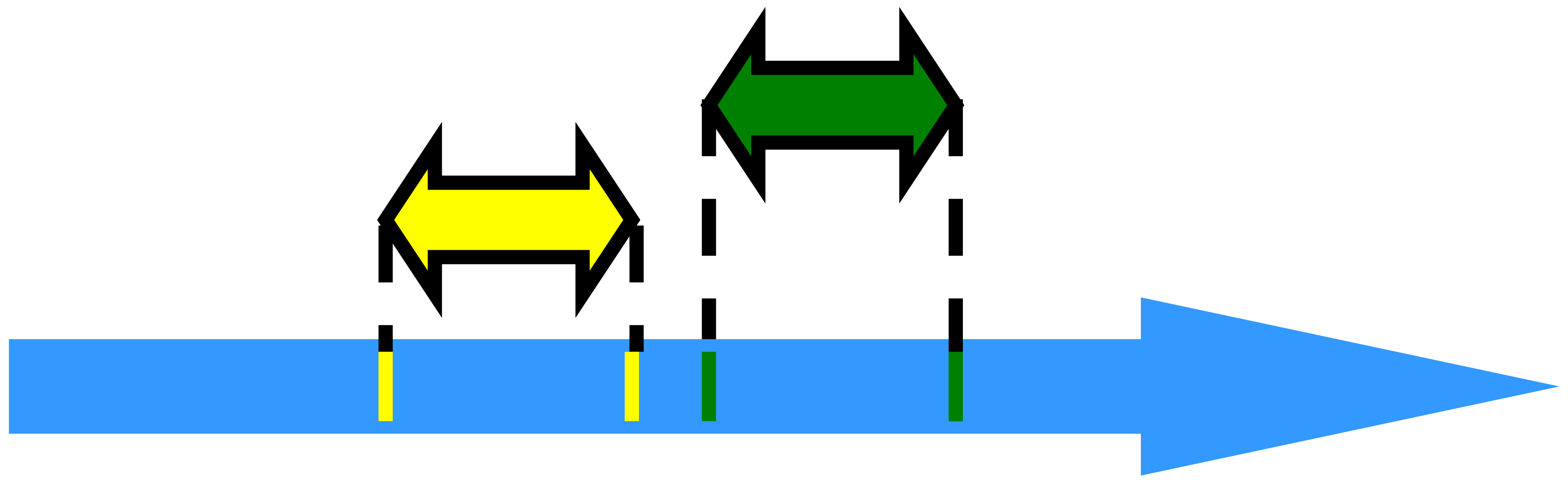


Precedence



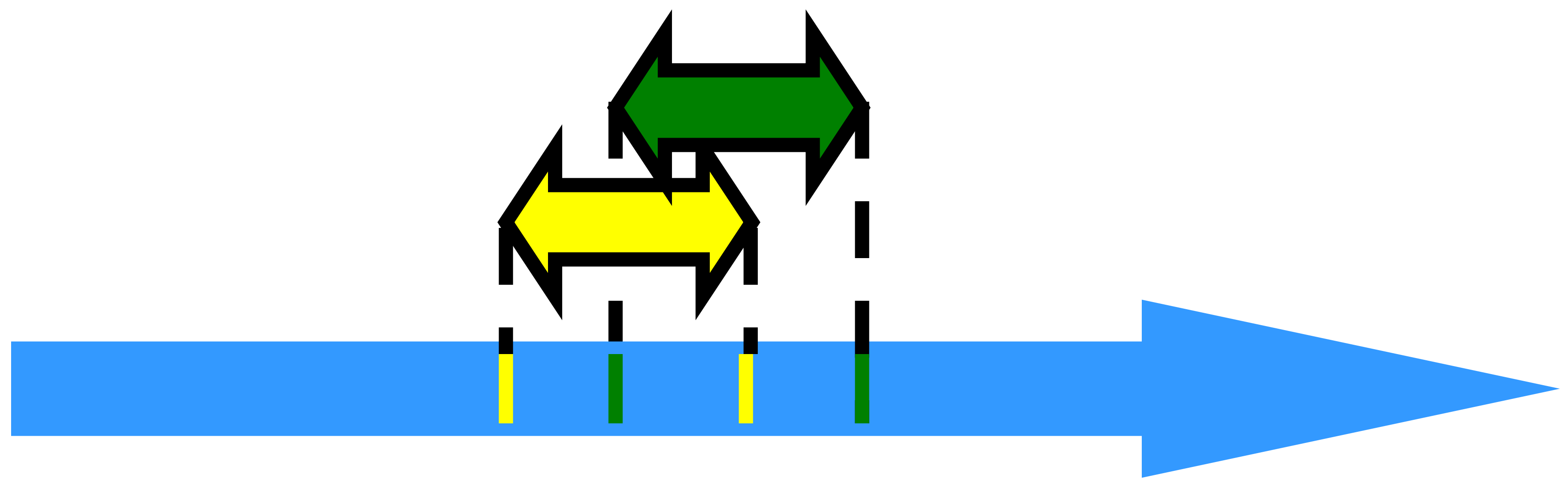
- Notation: $A_0 \rightarrow B_0$
- Formally,
  - End event of A_0 before start event of B_0
  - Also called "happens before" or "precedes"

Precedence Ordering



- Remark: $A_0 \rightarrow B_0$ is just like saying
  - 1066 AD → 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 & B →C then A →C
- Funny thing: A →B & B →A might both be false!

Partial Orders

(review)

- Irreflexive:
  - Never true that A → A
- Antisymmetric:
  - If A → B then not true that B → A
- Transitive:
  - If A → B & B → C then A → C

Total Orders

(review)

- Also
  - Irreflexive
  - Antisymmetric
  - Transitive
- Except that for every distinct A, B,
  - Either A → B or B → A

Repeated Events

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

a_0^k

k-th occurrence of event a_0

A_0^k

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

Implementing a Counter

```
class Counter {
  private var count = 0

  def getAndIncrement: Int = {
    val tmp = count
    count = tmp + 1
    tmp
  }
}
```

Make these steps *indivisible* using locks

Locks (Mutual Exclusion)

```
trait Lock {

 def lock(): Unit

 def unlock(): Unit
}
```

Locks (Mutual Exclusion)

```
trait Lock {

 def lock(): Unit

 def unlock(): Unit
}
```

acquire lock

Locks (Mutual Exclusion)

```
trait Lock {
  def lock(): Unit
  def unlock(): Unit
}
```

acquire lock

release lock

Using Locks

```
class Counter {
  private var count = 0
  private val lock : Lock = …

  def getAndIncrement: Int = {
    var tmp = 0
    lock.lock()
    try {
      tmp = count
      count = tmp + 1
    } finally {
      lock.unlock()
      tmp
    }
  }
}
```

Using Locks

```
class Counter {
  private var count = 0
  private val lock : Lock = …

  def getAndIncrement: Int = {
    var tmp = 0
    lock.lock()
    try {
      tmp = count
      count = tmp + 1
    } finally {
      lock.unlock()
      tmp
    }
  }
}
```

acquire Lock

Using Locks

```
class Counter {
  private var count = 0
  private val lock : Lock = …

  def getAndIncrement: Int = {
    var tmp = 0
    lock.lock()
    try {
      tmp = count
      count = tmp + 1
    } finally {
      lock.unlock()
      tmp
    }
  }
}
```

Release lock
(no matter what)

Using Locks

```
class Counter {
  private var count = 0
  private val lock : Lock = …

  def getAndIncrement: Int = {
    var tmp = 0
    lock.lock()
    try {
      tmp = count
      count = tmp + 1
    } finally {
      lock.unlock()
      tmp
    }
  }
}
```

critical section

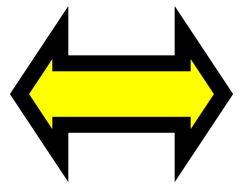
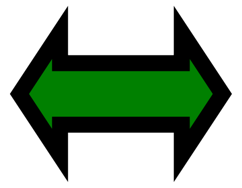
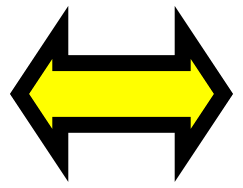
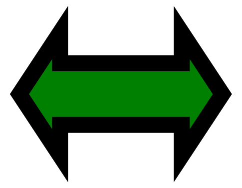
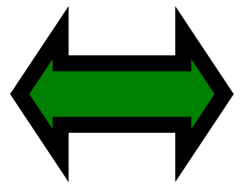
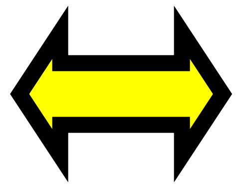
Mutual Exclusion

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

Mutual Exclusion

- Let CS_i^k $\Leftrightarrow$ be thread i's k-th critical section execution
- And CS_j^m $\Leftrightarrow$ 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 ⇔ ⇔

$CS_i^k \rightarrow CS_j^m$

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 ⇔ ⇔

$CS_i^k \rightarrow CS_j^m$

$CS_j^m \rightarrow CS_i^k$

Deadlock-Free

- If some thread calls `lock()`
  - And never returns (fails to acquire the lock)
  - 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

- 2-thread solutions first
  - Illustrate most basic ideas
  - Fits on one slide
- Then *n*-thread solutions

Two-Thread Conventions

```
class … extends Lock {
  …
  // thread-local index, 0 or 1
  def lock(): Unit = {
    val i = ThreadID.get();
    val j = 1 - i;
  …
  }
}
```

Two-Thread Conventions

```
class … extends Lock {
  …
  // thread-local index, 0 or 1
  def lock(): Unit = {
    val i = ThreadID.get();
    val j = 1 - i;
  …
  }
}
```

Henceforth: i is current thread, j is other thread

LockOne

```
class LockOne extends Lock {
  private val flag: Array[Boolean] = new Array(2)

  override def lock(): Unit = {
    val i = ThreadID.get
    val j = 1 - i
    flag(i) = true
    while (flag(j)) {} // spin
  }
…
```

LockOne

```
class LockOne extends Lock {
  private val flag: Array[Boolean] = new Array(2)

  override def lock(): Unit = {
    val i = ThreadID.get
    val j = 1 - i
    flag(i) = true
    while (flag(j)) {} // spin
  }
…
```

Each thread has flag

* In JVM reality, using an array this way is not quite right, but we'll gloss over it for now…

LockOne

```
class LockOne extends Lock {
  private val flag: Array[Boolean] = new Array(2)

  override def lock(): Unit = {
    val i = ThreadID.get
    val j = 1 - i
    flag(i) = true
    while (flag(j)) {} // spin
  }
…
```

Set my flag

LockOne

```
class LockOne extends Lock {
  private val flag: Array[Boolean] = new Array(2)

  override def lock(): Unit = {
    val i = ThreadID.get
    val j = 1 - i
    flag(i) = true
    while (flag(j)) {} // spin
  }
…
```

Wait for other flag to become 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 **`lock()`** before entering
- Derive a contradiction

From the Code

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

```
override def lock(): Unit = {
  …
  flag(i) = true
  while (flag(j)) {} // spin
}
```

From the Assumption

Since A is in the CS it did not see B's flag and vice versa.

- **$read_A$(flag[B]==false) → $write_B$(flag[B]=true)**

- **$read_B$(flag[A]==false) → $write_A$(flag[A]=true)**

```
override def lock(): Unit = {
  …
  flag(i) = true
  while (flag(j)) {} // spin
}
```

Combining

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

```
override def lock(): Unit = {
  …
  flag(i) = true
  while (flag(j)) {} // spin
}
```

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

```
override def lock(): Unit = {
  …
  flag(i) = true
  while (flag(j)) {} // spin
}
```

Combining

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

```
override def lock(): Unit = {
  …
  flag(i) = true
  while (flag(j)) {} // spin
}
```

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

```
override def lock(): Unit = {
  …
  flag(i) = true
  while (flag(j)) {} // spin
}
```

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

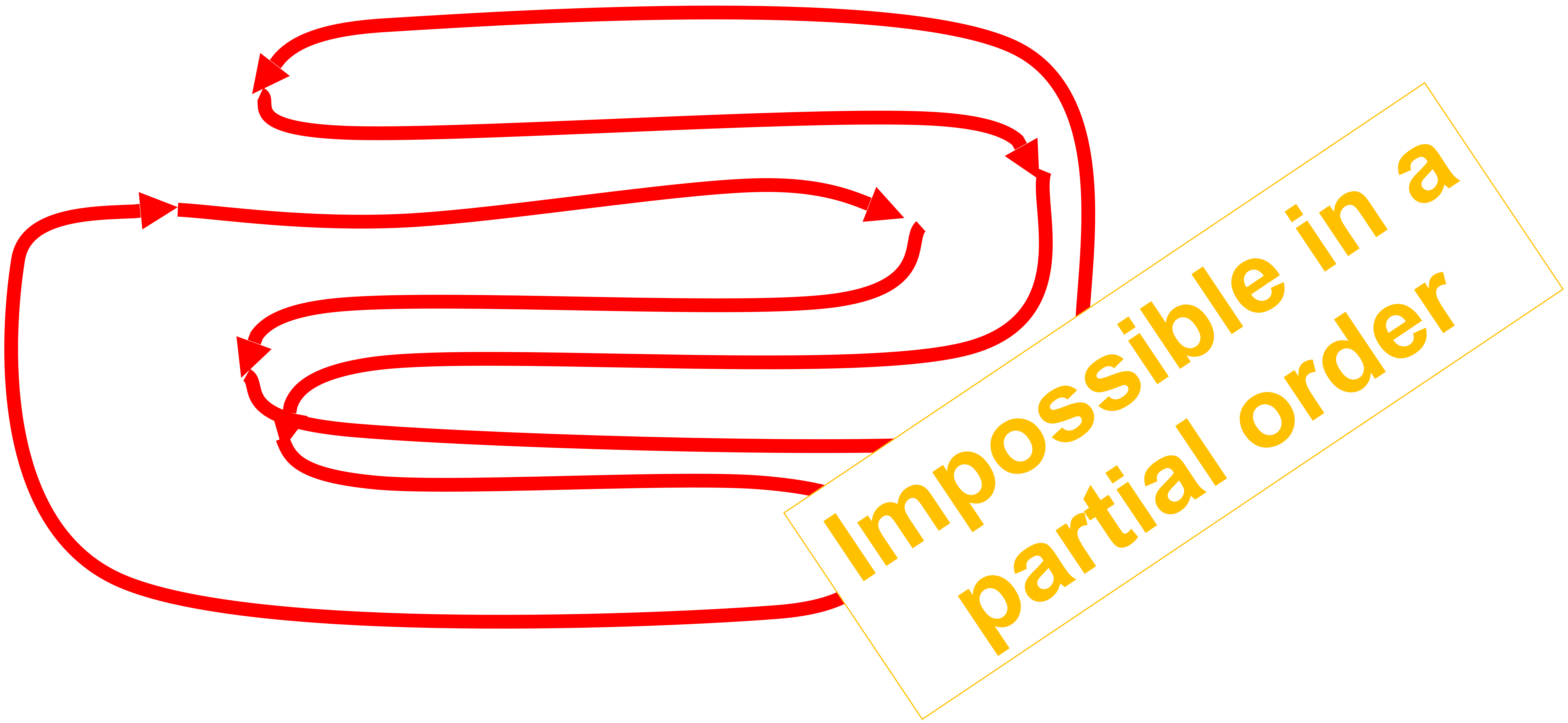
```
override def lock(): Unit = {
  …
  flag(i) = true
  while (flag(j)) {} // spin
}
```

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

```
override def lock(): Unit = {
  …
  flag(i) = true
  while (flag(j)) {} // spin
}
```

Cycle!



Demo: Testing Locks

Deadlock Freedom

- LockOne Fails deadlock-freedom
  – Concurrent execution can deadlock

```
flag(i) = true        flag(j) = true
while (flag(j)){}     while (flag(i)){}
```

  – Sequential executions OK

LockTwo

```
class LockTwo extends Lock {
  private var victim: Int = 0

  override def lock(): Unit = {
    val i = ThreadID.get
    victim = i
    while (victim == i) {}
 }
…
```

LockTwo

```
class LockTwo extends Lock {
  private var victim: Int = 0

  override def lock(): Unit = {
    val i = ThreadID.get
    victim = i
    while (victim == i) {}
  }
…
```

Let other go first

LockTwo

```
class LockTwo extends Lock {
  private var victim: Int = 0

  override def lock(): Unit = {
    val i = ThreadID.get
    victim = i
    while (victim == i) {}
  }
…
```

Wait for permission

LockTwo

```
class LockTwo extends Lock {
  private var victim: Int = 0

  override def lock(): Unit = {
    val i = ThreadID.get
    victim = i
    while (victim == i) {}
 }

override def unlock(): Unit = {}
…
```

Nothing to do

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

```
def lock() {
  victim = i; // my ThreadID
  while (victim == i) {};
}
```

Mid-lecture break

Peterson's Algorithm

```
def lock(): Unit = {
  val i = ThreadID.get
  val j = 1 - i
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}

def unlock(): Unit = {
  val i = ThreadID.get
  flag(i) = false
}
```

Peterson's Algorithm

```
def lock(): Unit = {
  val i = ThreadID.get
  val j = 1 - i
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}

def unlock(): Unit = {
  val i = ThreadID.get
  flag(i) = false
}
```

Announce I'm interested

Peterson's Algorithm

```
def lock(): Unit = {
  val i = ThreadID.get
  val j = 1 - i
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}

def unlock(): Unit = {
  val i = ThreadID.get
  flag(i) = false
}
```

Announce I'm interested

Defer to other

Peterson's Algorithm

```
def lock(): Unit = {
  val i = ThreadID.get
  val j = 1 - i
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}

def unlock(): Unit = {
  val i = ThreadID.get
  flag(i) = false
}
```

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

Peterson's Algorithm

```
def lock(): Unit = {
  val i = ThreadID.get
  val j = 1 - i
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}

def unlock(): Unit = {
  val i = ThreadID.get
  flag(i) = false
}
```

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

No longer interested

Mutual Exclusion

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

```
def lock() {
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}
```

From the Code

Also from the Code

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

```
def lock() {
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}
```

Assumption

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

W.L.O.G. assume A is the last thread to write **victim**

```
def lock() {
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}
```

Combining Observations

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

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

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

$\rightarrow \quad \text{read}_A(\text{victim})$

```
def lock() {
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}
```

Combining Observations

(1) $\text{write}_B(\text{flag[B]=true}) \rightarrow$

(3) $\text{write}_B(\text{victim=B}) \rightarrow$

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

```
def lock() {
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}
```

Combining Observations

```
def lock() {
  flag(i) = true
  victim = i
  while (flag(j) && victim == i) {}
}
```

(1) $\text{write}_B(\text{flag[B]=true}) \rightarrow$

(3) $\text{write}_B(\text{victim=B}) \rightarrow$

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

$\rightarrow \text{read}_A(\text{victim})$

A read flag[B] == true and victim == A, so it could not have entered the CS **(QED)**

Deadlock Free

```
def lock() {
  …
  while (flag(j) && victim == i) {};
```

- Thread blocked
  - only at **`while`** loop
  - only if other's flag is true
  - only if it is the victim
- Solo: other's flag is false
- Both: one or the other not 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

```
def lock() {
  flag(i) = true
  victim    = i
  while (flag(j) && victim == i) {};
}

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

Demo: Peterson Lock

Bounded Waiting

- Want *stronger* fairness guarantees
- Thread not "overtaken" too much
- If A starts before B, then A enters before B?
- But what does "start" mean?
- Need to adjust definitions ....

Bounded Waiting

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

First-Come-First-Served

- 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 $CS_A^k \rightarrow CS_B^j$
    - A's k-th critical section precedes B's j-th critical section
    - B cannot overtake A

Bakery Algorithm

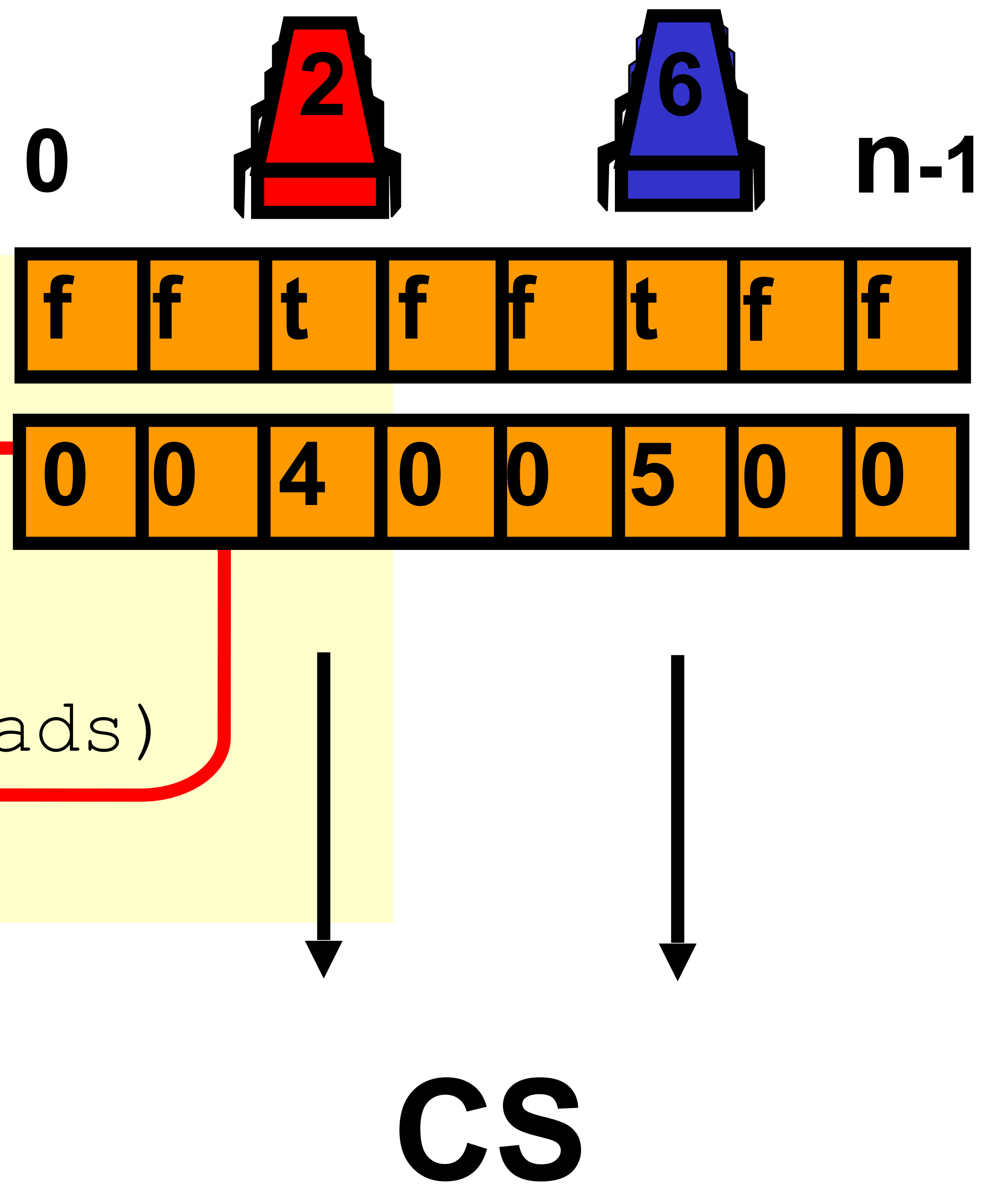
- Provides First-Come-First-Served for *n* threads
- How?
  - Take a "number"
  - Wait until lower numbers have been served
- Lexicographic order
  - $(a,i) > (b,j)$
    - If $a > b$, or $a = b$ and $i > j$

Bakery Algorithm

```
class BakeryLock(val threads: Int) extends Lock {

  private val label: Array[Label] =
        Array.fill(threads)(new Label())

  private val flag: Array[Boolean] = new Array(threads)
```

Bakery Algorithm

```
class BakeryLock(val threads: Int) extends Lock {

  private val label: Array[Label] =
        Array.fill(threads)(new Label())

  private val flag: Array[Boolean] = new Array(threads)
```



Bakery Algorithm

```
class BakeryLock extends Lock {
  …
 override def 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 BakeryLock extends Lock {
  …
 override def 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 BakeryLock extends Lock {
  …
 override def 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

Take increasing label (read labels in some arbitrary order)

```
class BakeryLock extends Lock {
  …
 override def 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 BakeryLock extends Lock {
  …
 override def 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 BakeryLock extends Lock {
  …
 override def 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

… whose (label,i) in lexicographic order is lower

Bakery Algorithm

```
class BakeryLock extends Lock {

    …

 override def unlock() {
   flag(i) = false;
 }
}
```

Bakery Algorithm

```
class BakeryLock extends Lock {

    …

 override def 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:
  - $write_A(label[A]) \rightarrow$
  - $read_B(label[A]) \rightarrow$
  - $write_B(label[B]) \rightarrow read_B(flag[A])$
- So B sees
  - smaller label for A
  - locked out while flag[A] is true

```
class BakeryLock extends Lock {
  …
 override def 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 BakeryLock extends Lock {
  …
 override def lock() {
  flag(i)   = true
  label(i) = max(label(0), …,label(n-1))+1

  while (∃k flag(k)
           && (label(i),i) > (label(k),k)) {}
 }
```

```
class BakeryLock extends Lock {
  …
 override def 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

```
class BakeryLock extends Lock {
  …
 override def 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
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$

Mutual Exclusion

```
class BakeryLock extends Lock {
  …
 override def lock() {
  flag(i)   = true
  label(i) = max(label(0), …,label(n-1))+1

  while (∃k flag(k)
           && (label(i),i) > (label(k),k)) {}
 }
```

- 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

Any issues with BackeryLock?

Bakery Y2^{32}K Bug

```
class BakeryLock extends Lock {
  …
 override def 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 BakeryLock extends Lock {
  …
 override def 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

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 such as **`flag()`**…

Real-Life Implementations

- Demo

Summary of the last two lectures

- We have seen several impractical examples of implementing *mutual exclusion* algorithms.

- We learned how to reason about their claims: *mutual exclusion, deadlock freedom, etc.*

- Today we know how to solve the *First-Come-First-Served* N thread mutual exclusion using 2N RW-Registers

Summary of the last two lectures

- N RW-Registers inefficient
  - Because writes **“cover”** older writes
- Need stronger hardware operations
  - that do not have the **“covering problem”**
- In next lectures - understand what these operations are…