FACULDADE DE
CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Mutual Exclusion

lectures 03 & 04 (2025-03-17)

Master in Computer Science and Engineering

— Concurrency and Parallelism / 2024-25 —

João Lourenço <joao.lourenco@fct.unl.pt>

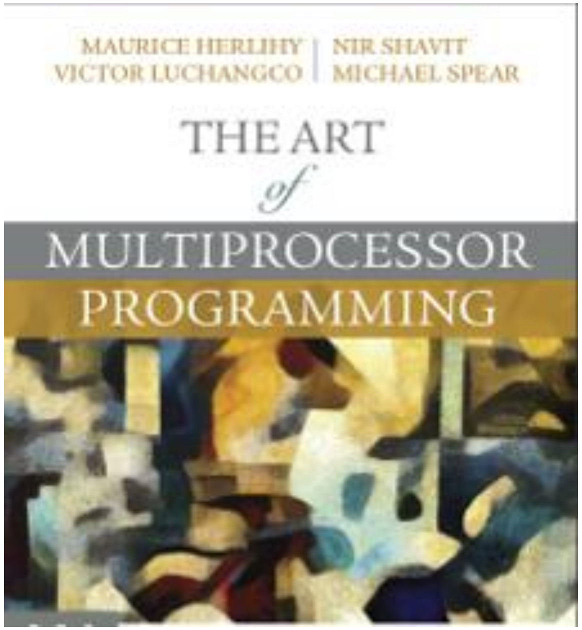
Based in slides the companion slides from “The Art of Multiprocessor Programming”

Outline

- Mutual exclusion
  - Time
  - Critical Sections
  - Locks
  - Fairness

  - Bibliography:
    - **Chapters 2** of book

      Herlihy M., Shavit N., Luchangco V., Spear M.;
      The Art of Multiprocessor Programming;
      Morgan Kaufmann (2020);   ISBN: 978-0-12-415950-1



Concurrent Objects

- We will clarify our understanding of mutual exclusion
- We will also show you 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

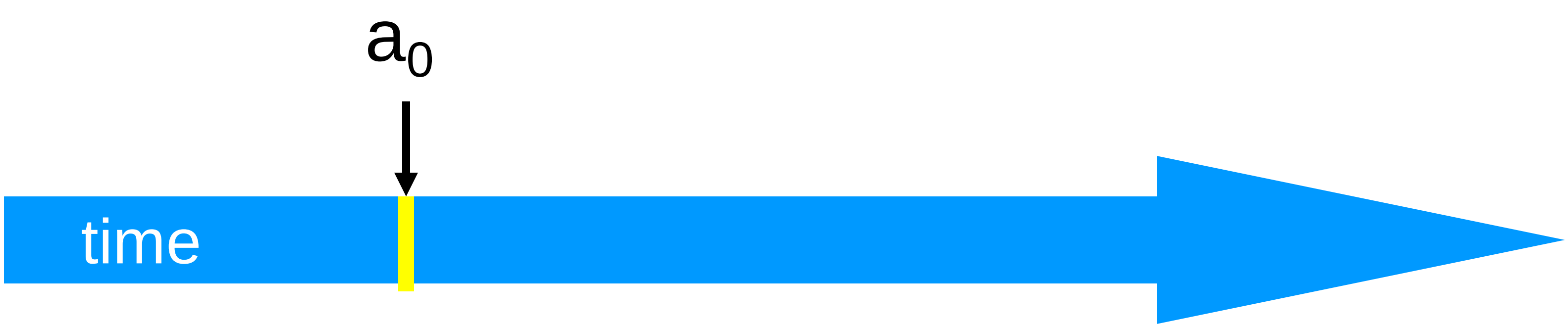
Time

- *“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)

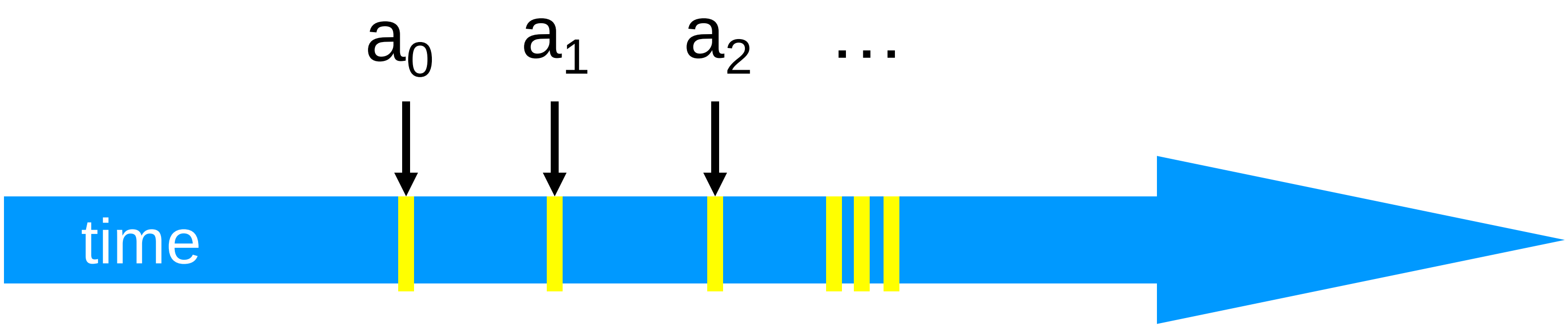
Events

- 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

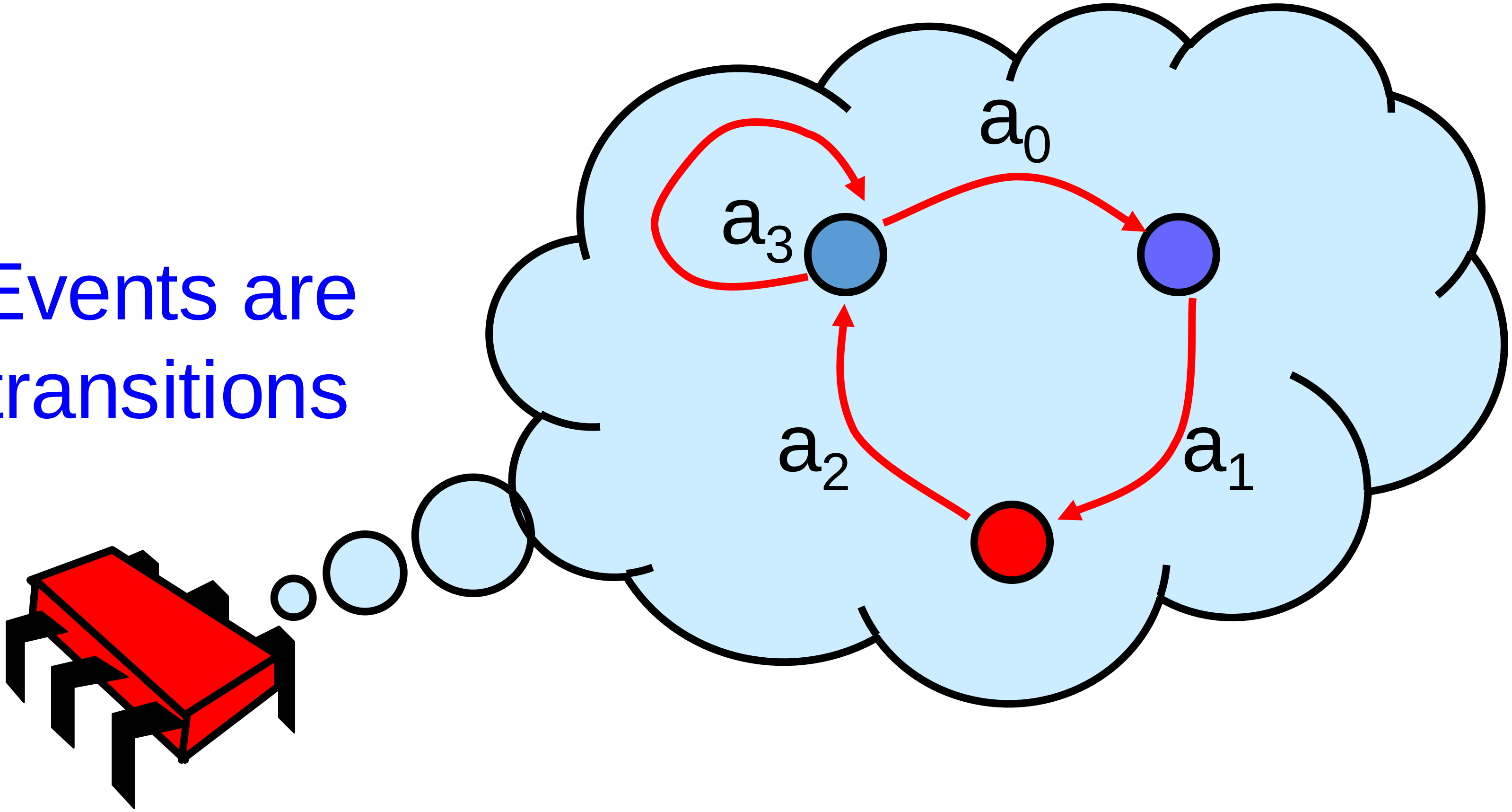


Example Thread Events

- 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

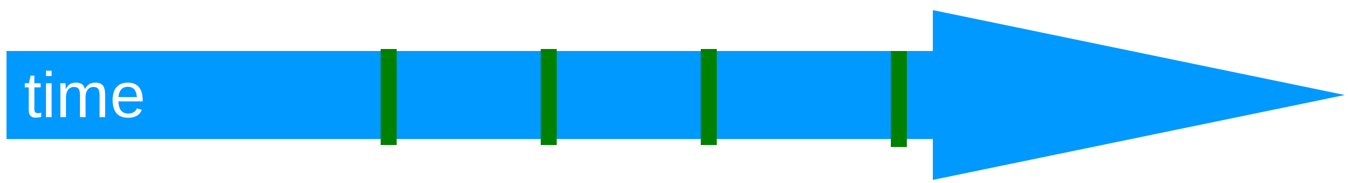
Concurrency

- Thread A

Concurrency

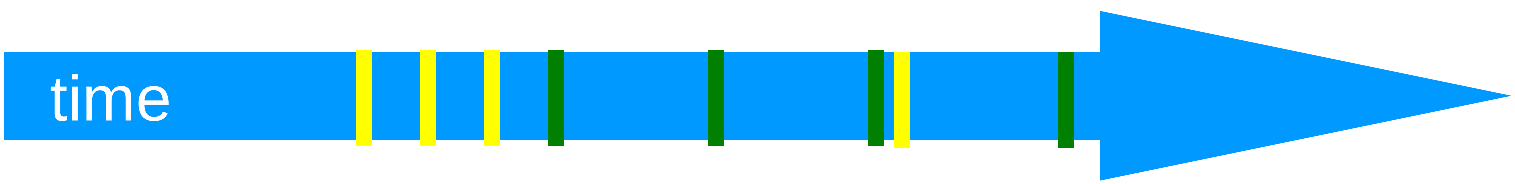
- Thread A

- Thread B



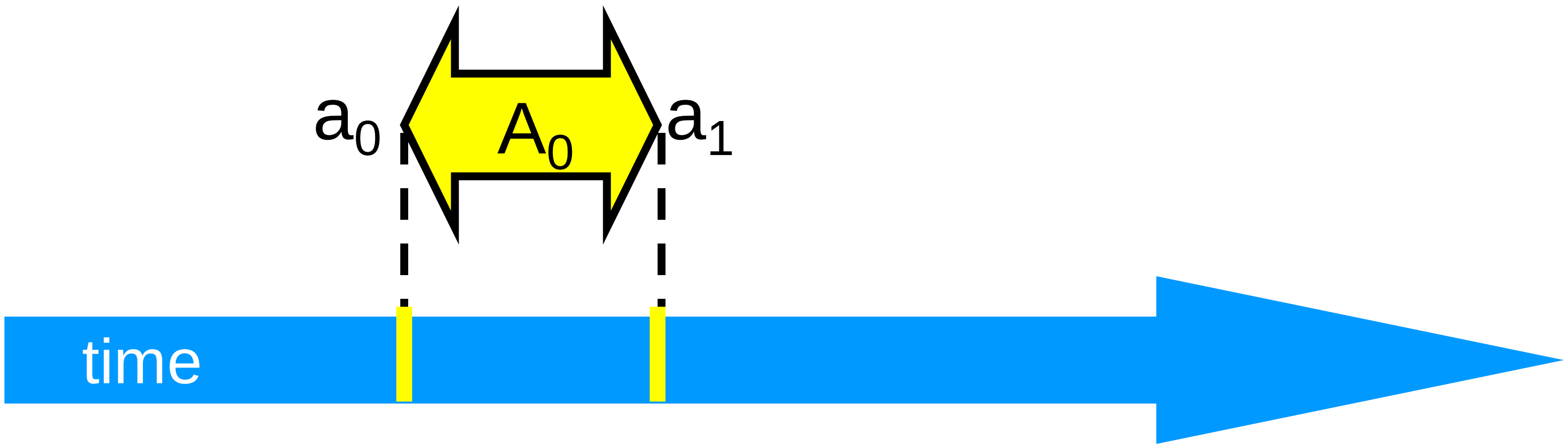
Concurrency

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

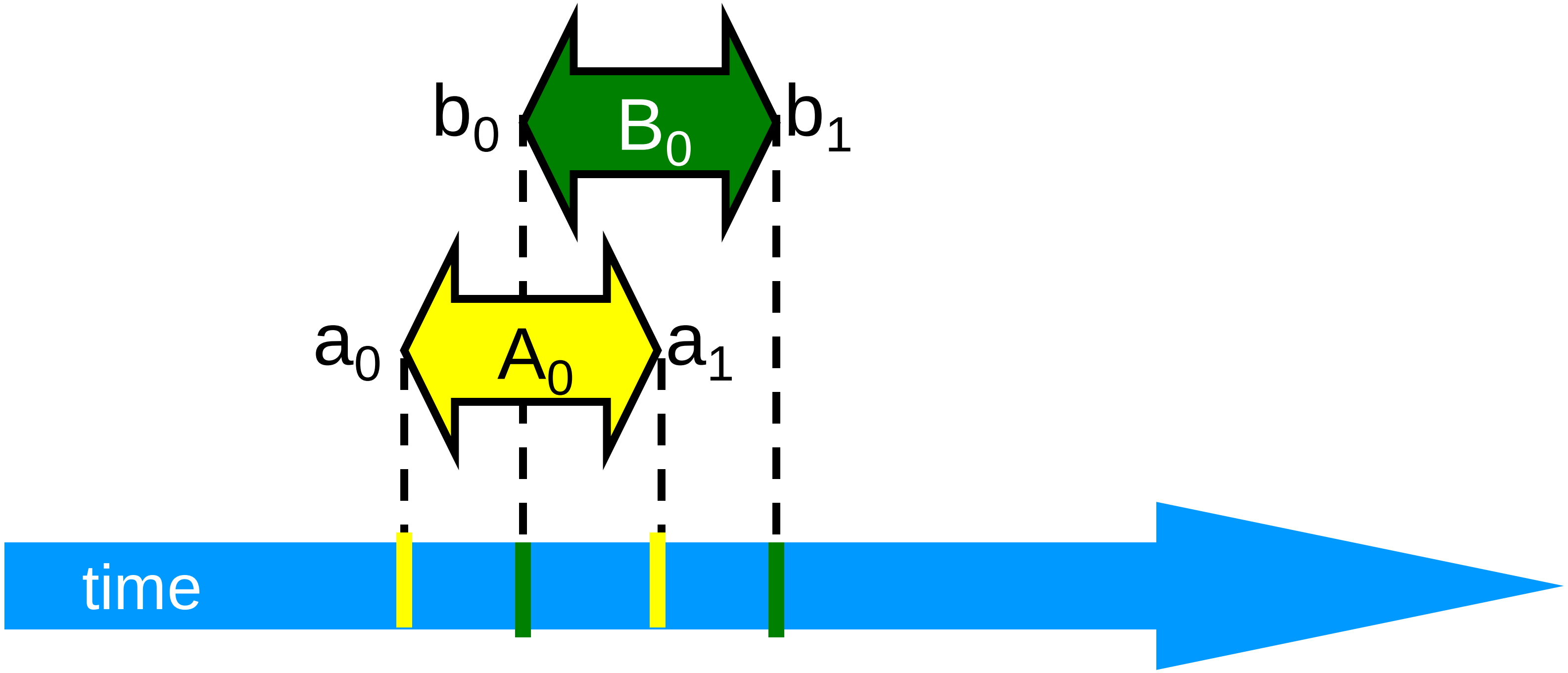


Intervals

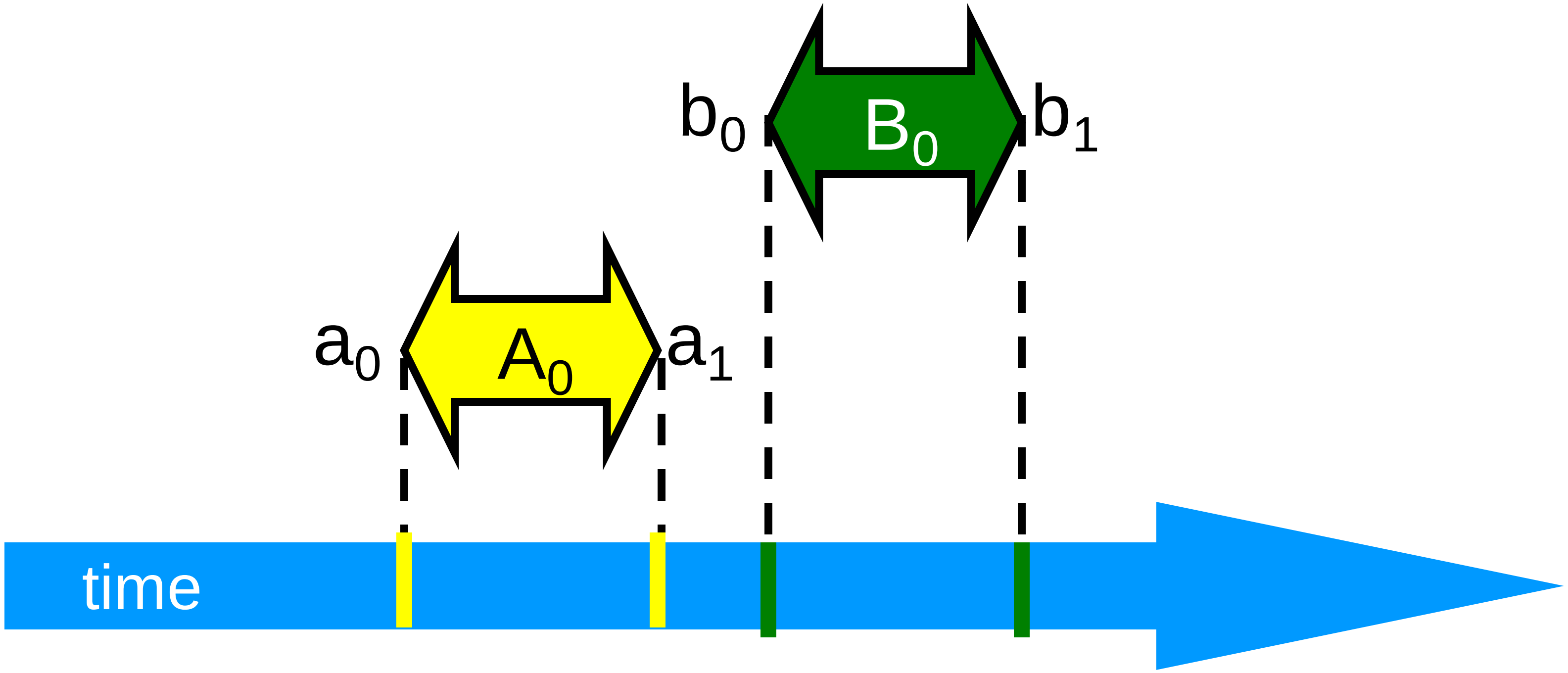
- An *interval* $A_0 = (a_0, a_1)$ is
  - Time between events a_0 and a_1



Intervals may Overlap

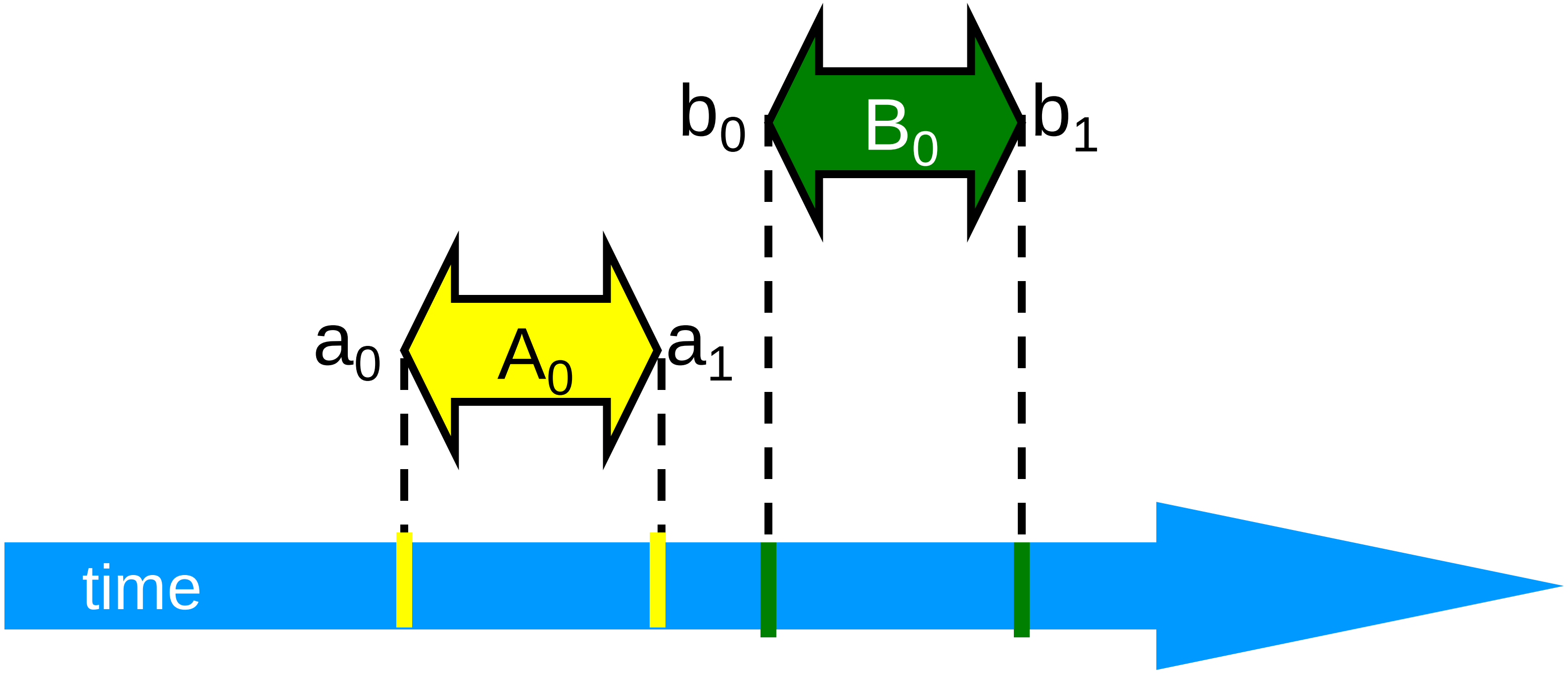


Intervals may be Disjoint



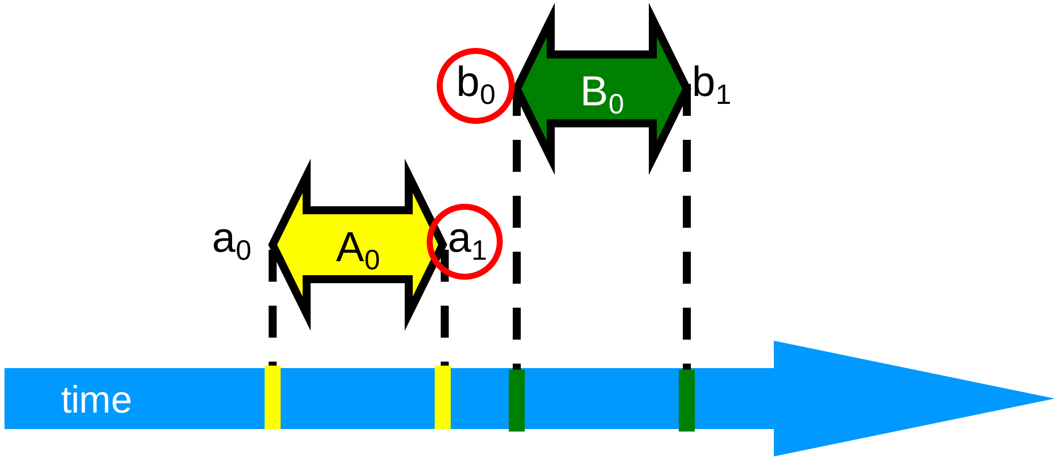
Precedence

- Interval A_0 precedes interval B_0



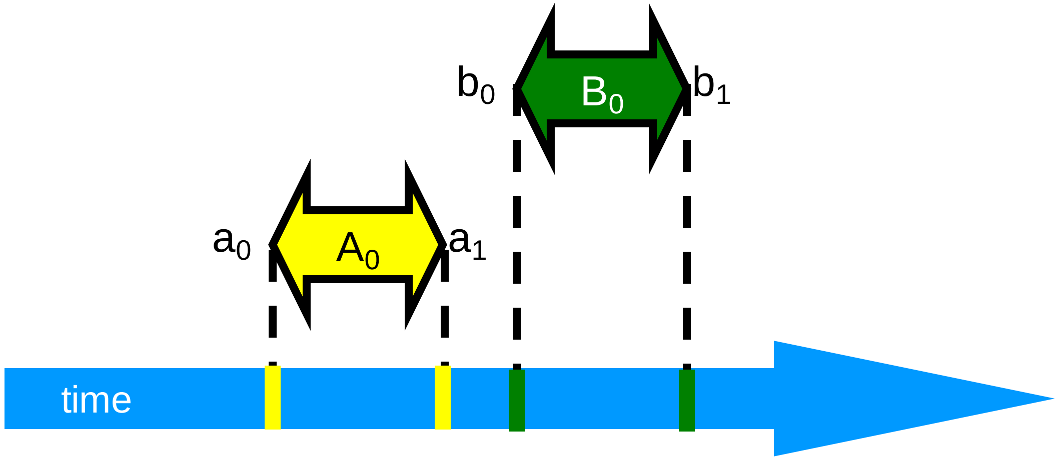
Precedence

- Notation: $A_0 \rightarrow B_0$
- Formally,
  - End event of A_0 (a_1) is before start event of B_0 (b_0)
  - Also called "happens before" or "precedes"



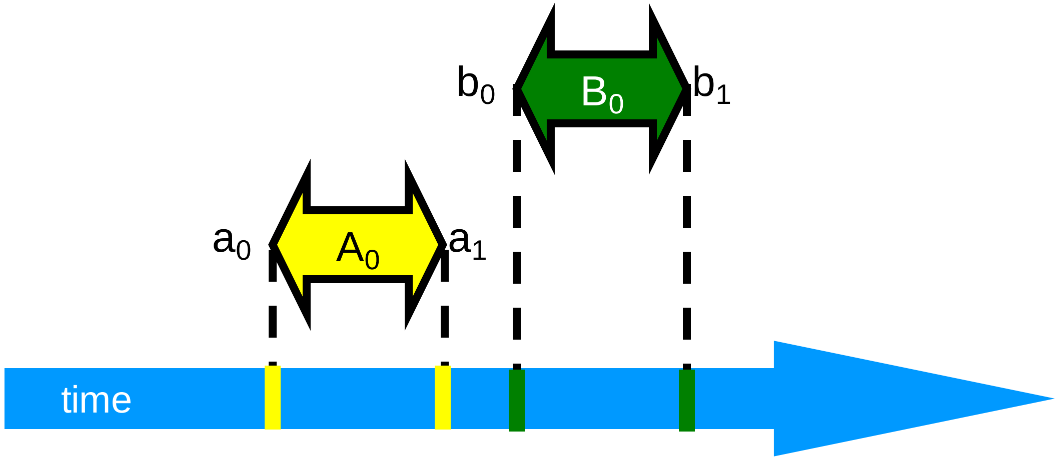
Precedence

- 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 $\rightarrow$ 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
(review)

- 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
(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

A_0^k

k-th occurrence of event a_0

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

Implementing a Counter

```
public class Counter {
  private long value;

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

Make these steps *indivisible*

Implementing a Counter (2)

- Would we need this?

```
public class Counter {
  private long value;

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

Make these steps *indivisible*

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

acquire lock

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() {
   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;
  }}
```

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 $CS_i^k \Leftrightarrow$ 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 $\Leftrightarrow$ be thread i's k-th critical section execution
- And CS_j^m $\Leftrightarrow$ be thread j's m-th critical section execution
- Then either
  - $\Leftrightarrow$ $\Leftrightarrow$ or $\Leftrightarrow$ $\Leftrightarrow$ (critical sections do not overlap!)

$CS_i^k \rightarrow CS_j^m$

$CS_j^m \rightarrow CS_i^k$

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

Starvation-Freedom => Deadlock-Freedom

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

LockOne

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

Each thread has flag

LockOne

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

Set my flag

LockOne

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

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

- **write_A(flag[A]=true) → read_A(flag[B]==false) →CS_A**
- **write_B(flag[B]=true) → read_B(flag[A]==false) → CS_B**

```
class LockOne implements Lock {
…
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:
  - $\mathbf{read_A(flag[B]{=}{=}false) \rightarrow write_B(flag[B]{=}true)}$
  - $\mathbf{read_B(flag[A]{=}{=}false) \rightarrow write_A(flag[A]{=}true)}$
- From the code
  - $\mathbf{write_A(flag[A]{=}true) \rightarrow read_A(flag[B]{=}{=}false)}$
  - $\mathbf{write_B(flag[B]{=}true) \rightarrow read_B(flag[A]{=}{=}false)}$

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

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

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

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

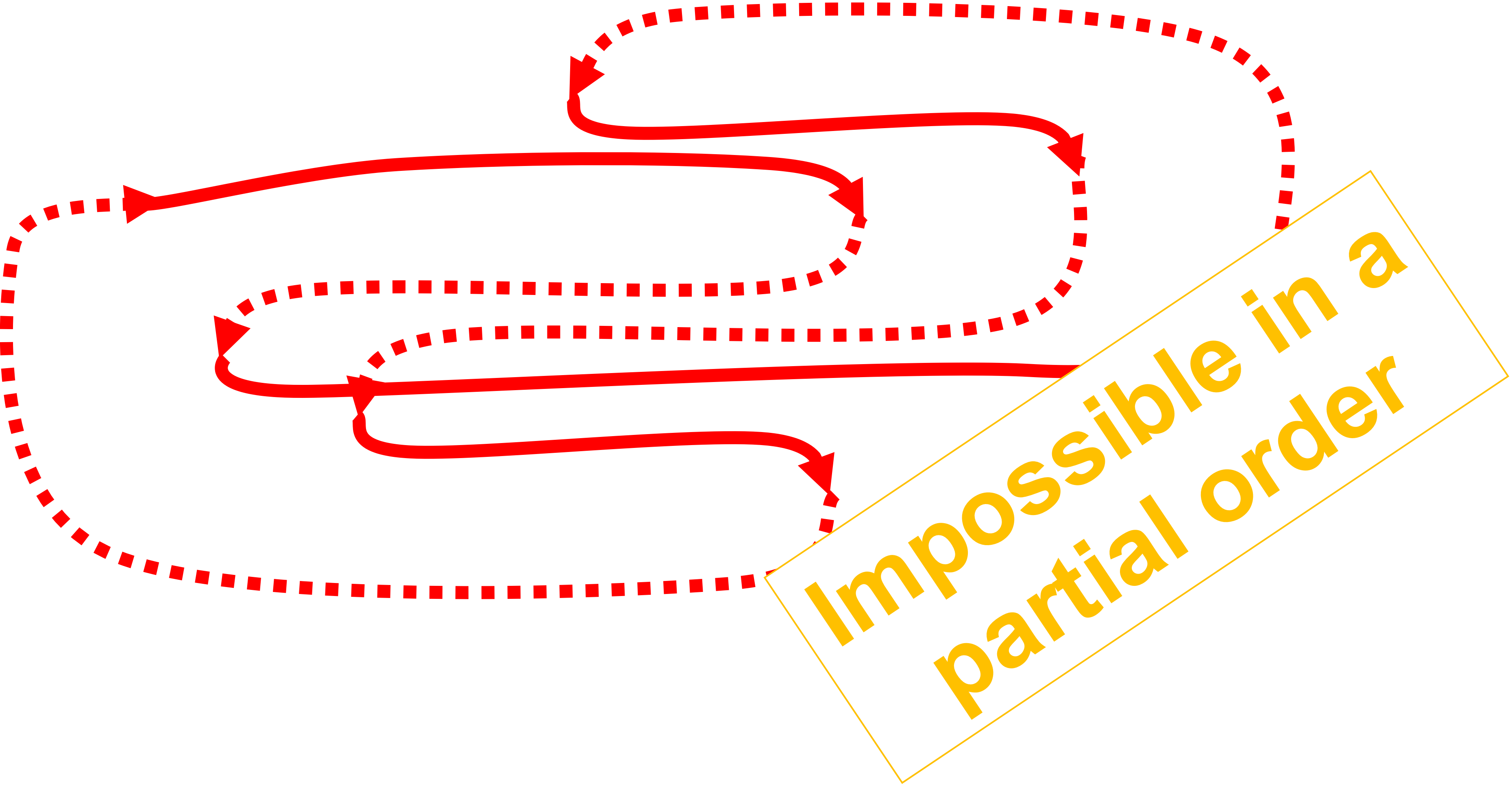
Combining

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

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

```
flag[i] = true;      flag[j] = true;
while (flag[j]){}    while (flag[i]){}
```

  - Sequential executions OK

LockTwo

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

  public void unlock() {}
}
```

LockTwo

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

  public void unlock() {}
}
```

Let other go first

LockTwo

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

  public void unlock() {}
}
```

Wait until other let me go

LockTwo

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

  public void unlock() {}
}
```

Nothing to do

LockTwo Claims

- Satisfies mutual exclusion
  - If thread i in CS
  - Then victim == j
  - victim cannot be both 0 and 1
- Not deadlock free
  - Sequential execution deadlocks
  - Concurrent execution does not

```
public void LockTwo() {
 victim = i;
 while (victim == i) {};
}
```

```
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

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

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

From the Code

Also from the Code

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

```
public void 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**

Combining Observations

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

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

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

Combining Observations

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

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

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

Combining Observations

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

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

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

➔ $\text{read}_A(\text{victim})$

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

Deadlock Free

```
public void 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
- One process solo: other's flag is false
- Both processes: 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

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

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

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

r-Bounded Waiting

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

What is “r” for Peterson's Algorithm?

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

Answer: r = 0

First-Come-First-Served

- For threads A and B:
  - If $\mathbf{D_A^k} \Rightarrow \mathbf{D_B^j}$
    - A's k-th doorway precedes B's j-th doorway
  - Then $\mathbf{CS_A^k} \Rightarrow \mathbf{CS_B^j}$
    - A's k-th critical section precedes B's j-th critical section
    - B cannot overtake A

Bakery Algorithm

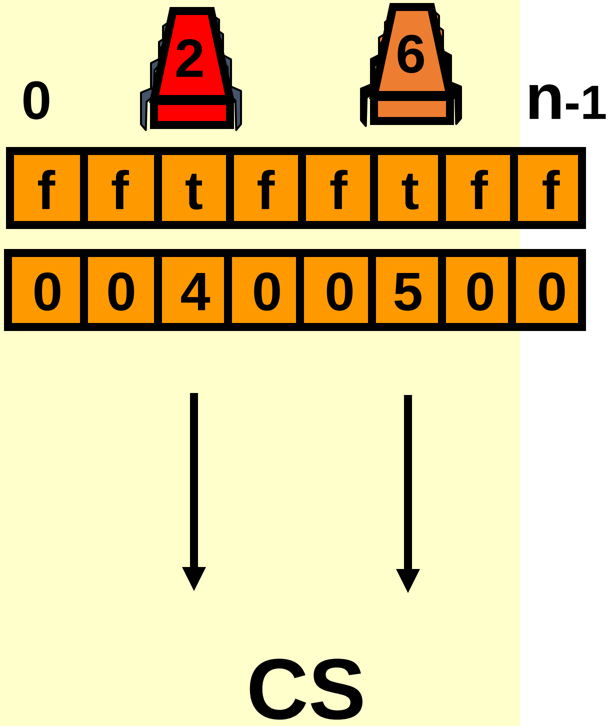
- Provides First-Come-First-Served for *n* threads
- How?
  - Take a “number”
  - Wait until lower numbers have been served
- Lexicographic order
  - (a,i) > (b,j)
    - If a > b, or a = b and i > j

Bakery Algorithm

```
class 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 {
  ...
  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

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

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

```
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));
}
```

- 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

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));
  }
  ...
}
```

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

- N registers such as flag[]…

Summary of Lecture

- In the 1960's several 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
  - But mathematically required …
- Need stronger hardware operations
  - that do not have the **“covering problem”**
- In later lectures: understand what these operations are…

The END