02 Mutual Exclusion

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

Mutual Exclusion

- We will clarify our understanding of mutual exclusion

Mutual Exclusion

- 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

Mutual Exclusion

- Formal problem definitions

Mutual Exclusion

- Formal problem definitions
- Solutions for ***2*** threads

Mutual Exclusion

- Formal problem definitions
- Solutions for ***2*** threads
- Solutions for ***n*** threads

Mutual Exclusion

- Formal problem definitions
- Solutions for ***2*** threads
- Solutions for ***n*** threads
- Fair solutions

Mutual Exclusion

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

Warning ⚠️

Warning ⚠️

- You will ***never*** use these protocols
  - Get over it

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 Concurrent Programming is Hard?

- Try preparing a seven-course banquet
  - By yourself

Why Concurrent Programming is Hard?

- Try preparing a seven-course banquet
  - By yourself
  - With one friend

Why Concurrent Programming is Hard?

- Try preparing a seven-course banquet
  - By yourself
  - With one friend
  - With twenty-seven friends …

Why Concurrent Programming is Hard?

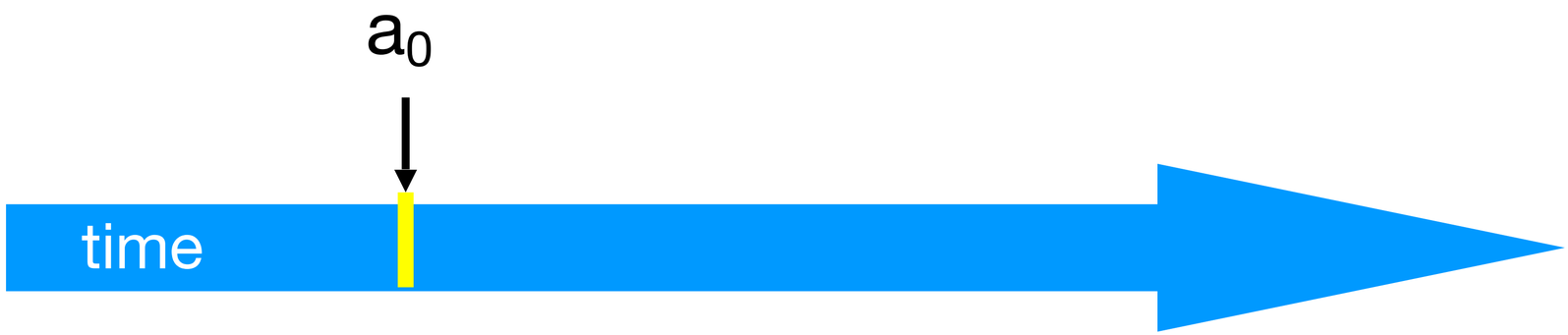
- Try preparing a seven-course banquet
  - By yourself
  - With one friend
  - With twenty-seven friends …
- Before we can talk about programs
  - Need a ***language***
  - Describing time and concurrency

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

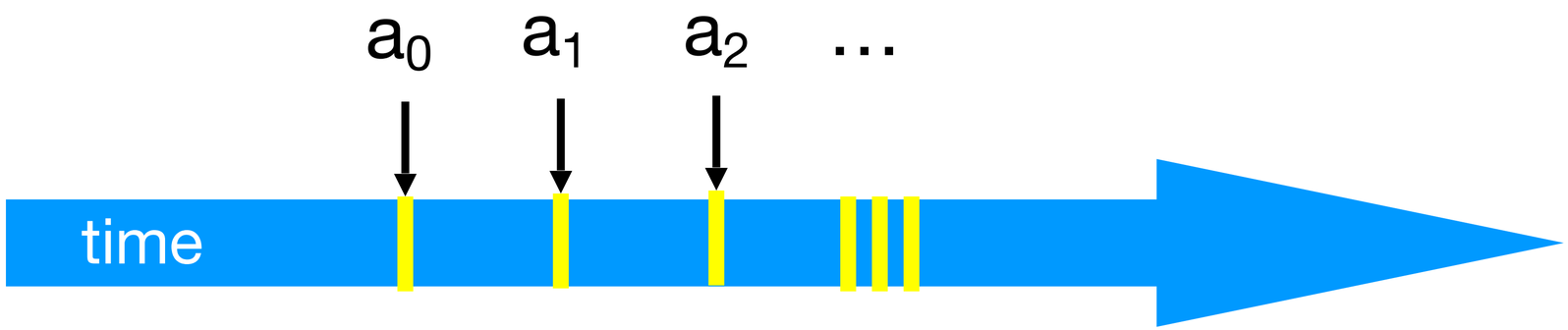
Events

- 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 ➔ a_1 indicates order

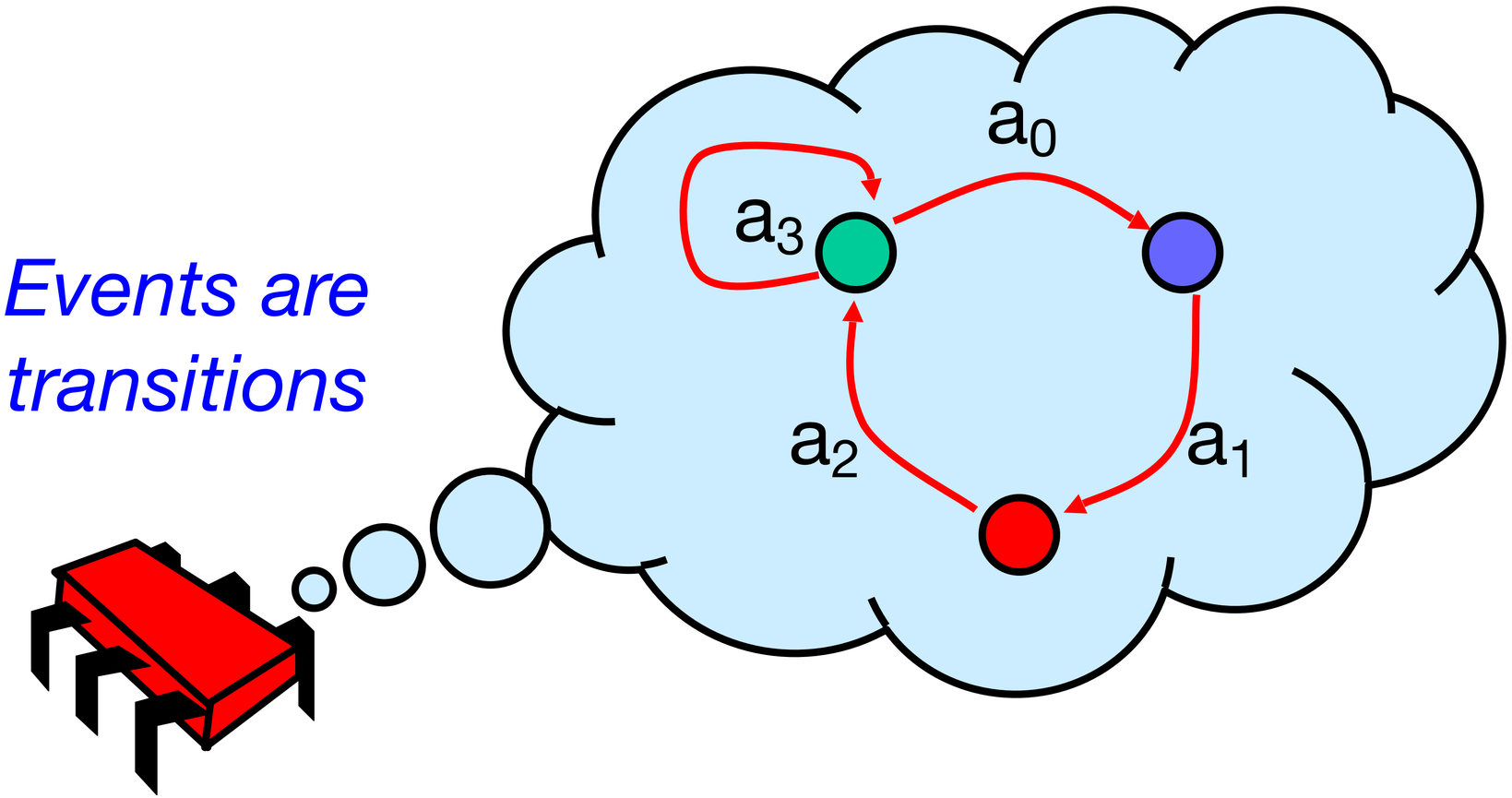


Examples Thread Events

- Assign to a shared variable
- Assign to a local variable
- Invoke method
- Return from a 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

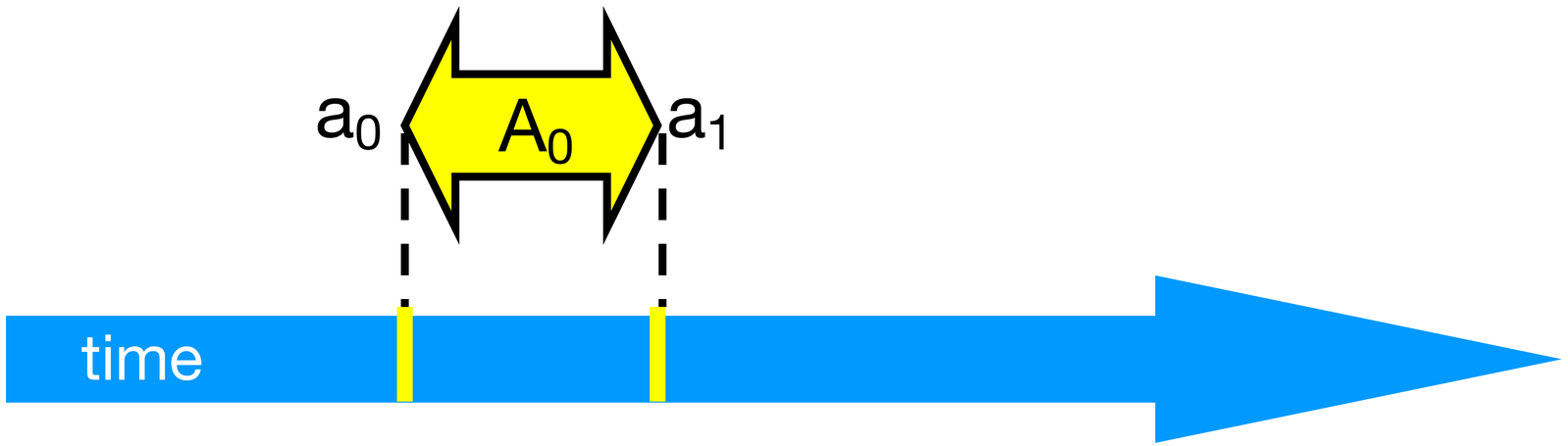
- Thread B

Interleavings

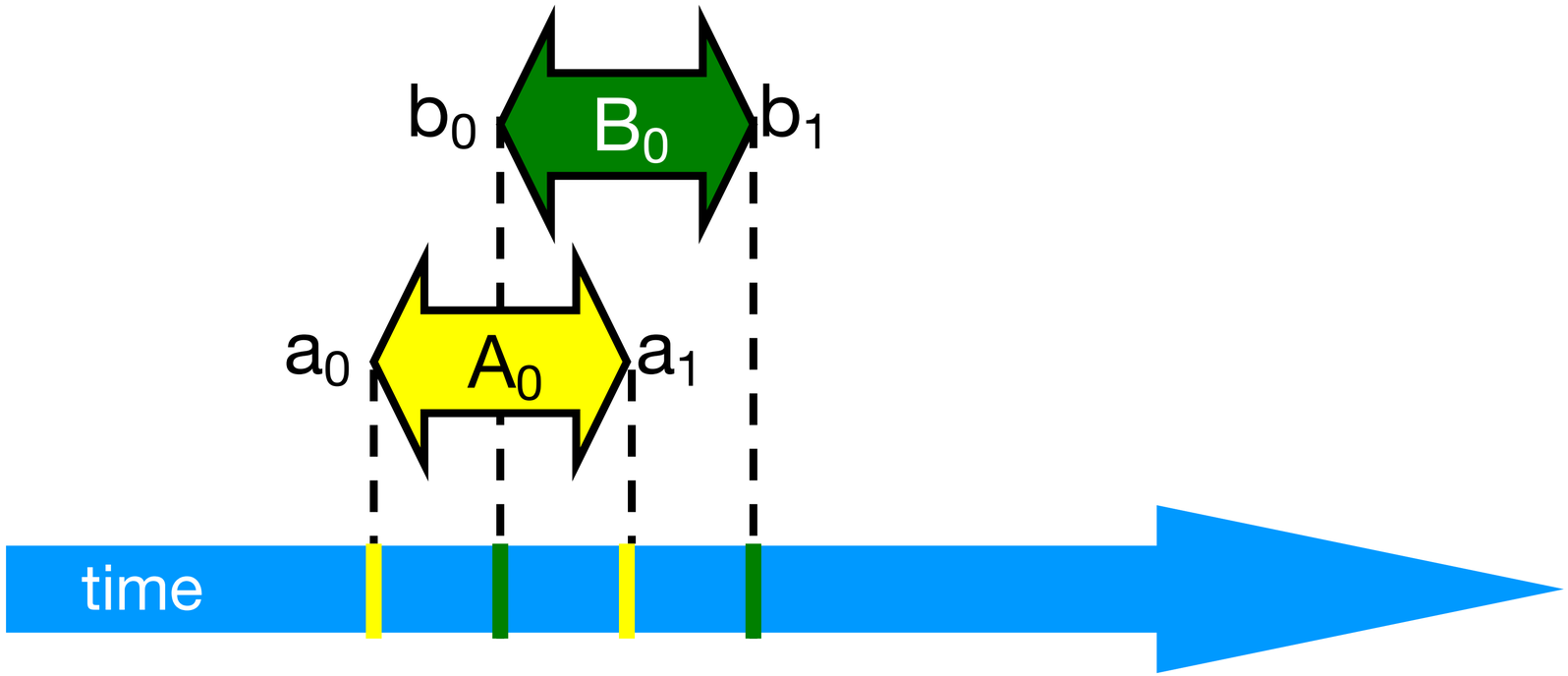
- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)

Intervals

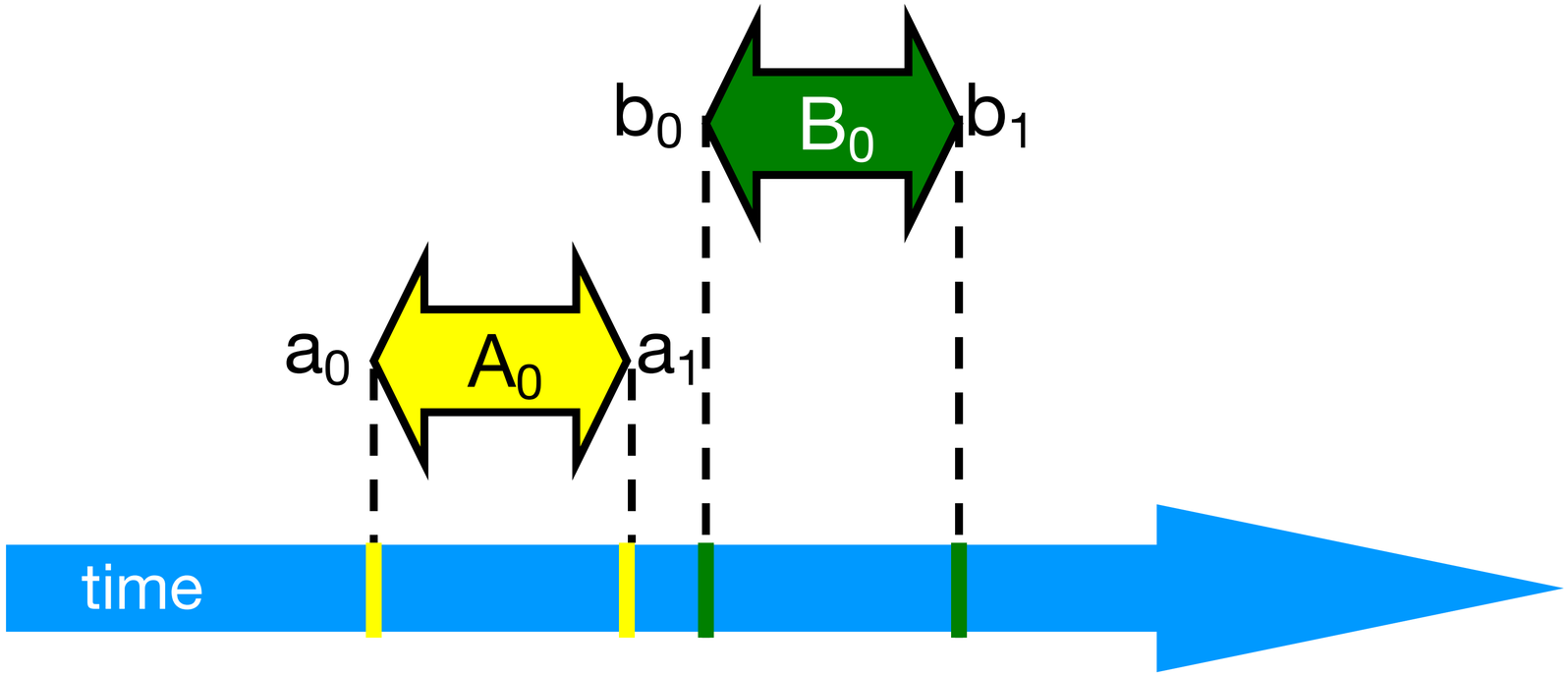
- An ***interval*** $A_0 = (a_0, a_1)$ is
- Time between events a_0 and a_1



Intervals may overlap

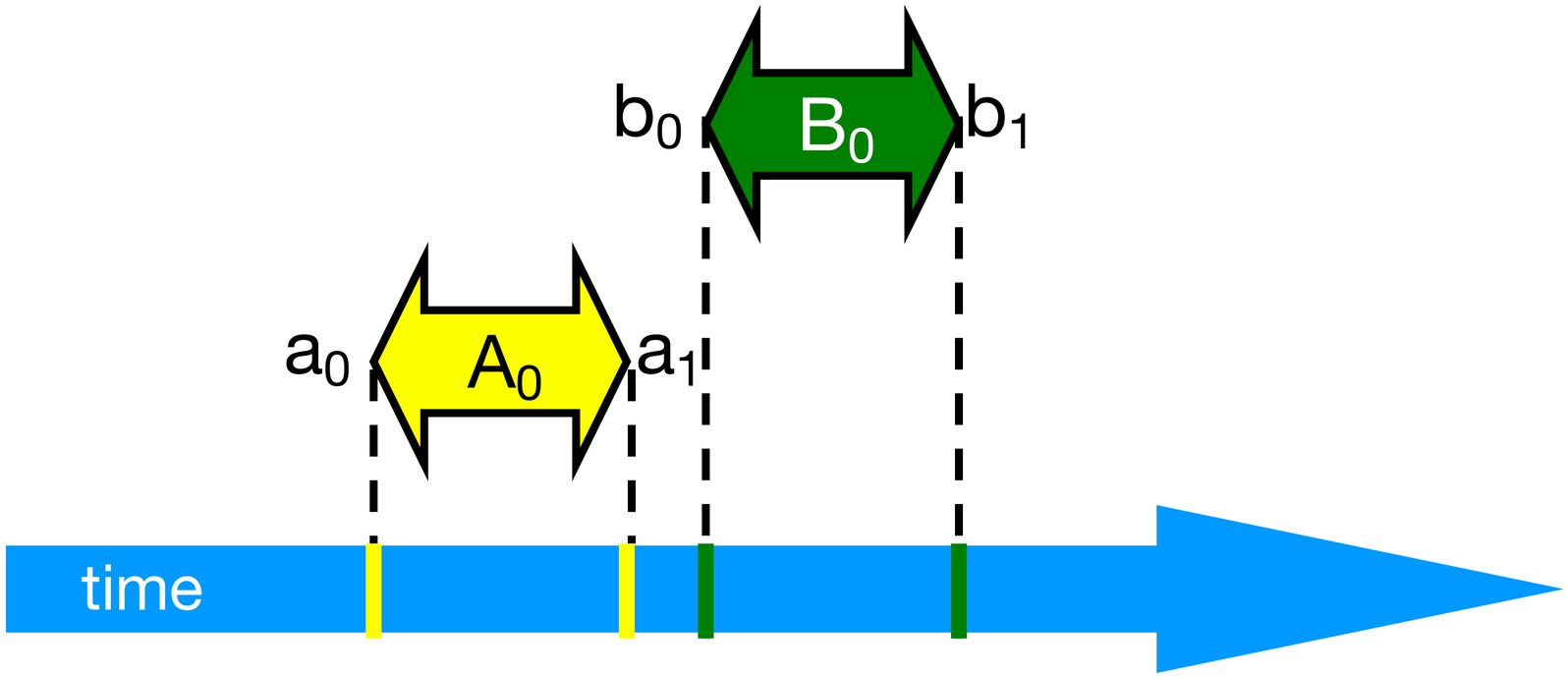


Intervals may be disjoint



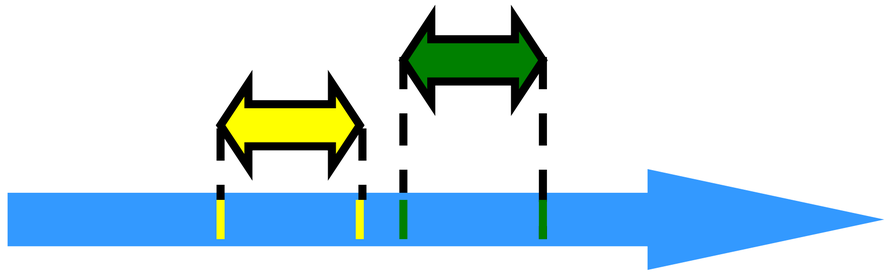
Precedence

- Interval A_0 ***precedes*** interval B_0

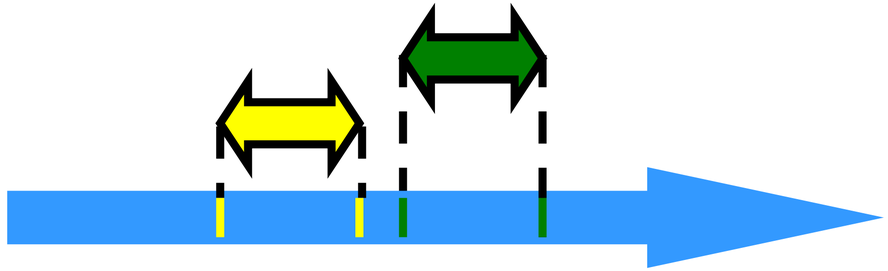


Precedence

- Notation: A_0 → B_0
- Formally,
  - End event of A_0 before start event of B0
  - Also called ***“happens before”*** or ***“precedes”***

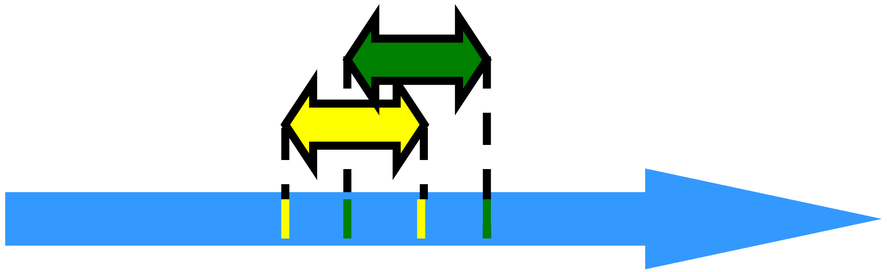


Precedence Ordering



- 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 & B ➔ C then A ➔ C
- **Funny thing:** A ➔ B & B ➔ A might both be false!

Precedence Orders (review)

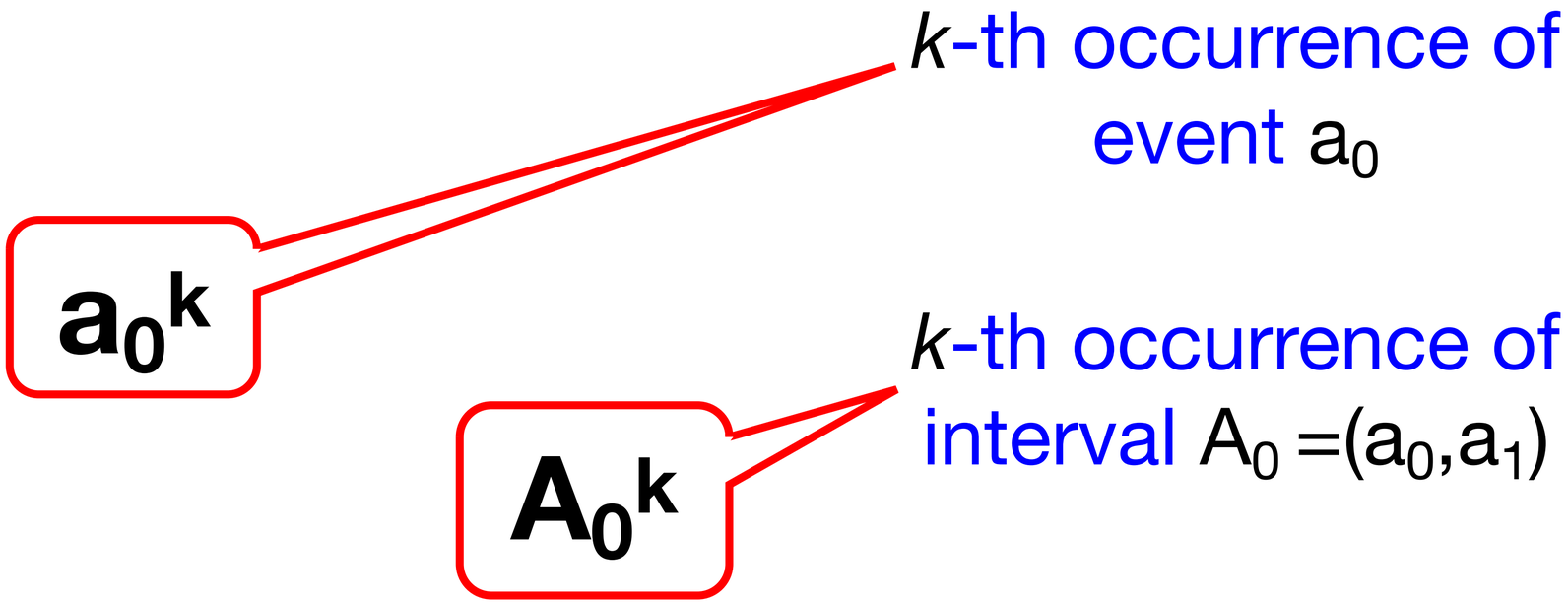
- **Irreflexive**:
  - Never true that A ➔ A
- **Antisymmetric**:
  - If A ➔ B, then not true that B ➔ A
- **Transitive**:
  - If A ➔ B & B ➔ C then A ➔ C

Total Orders (review)

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

Repeated Events

while cond do
$a_0; a_1$
done



Challenge

```
(* A counter is just a reference *)
let create_counter initial_value =
  ref initial_value

let get_and_increment counter =
  let v = !counter in
  counter := v + 1;
  v
```

Make this atomic (indivisible)

Mutex

Module Mutex

```
module Mutex: sig .. end
```

Locks for mutual exclusion.

Mutexes (mutual-exclusion locks) are used to implement critical sections and protect shared mutable data structures against concurrent accesses. The typical use is (if `m` is the mutex associated with the data structure `D`):

```
Mutex.lock m;
(* Critical section that operates over D *);
Mutex.unlock m
```

https://ocaml.org/manual/5.4/api/Mutex.html

Mutex

type `t`

The type of mutexes.

val `create : unit -> t`

Return a new mutex.

val `lock : t -> unit`

Lock the given mutex. Only one thread can have the mutex locked at any time. A thread that attempts to lock a mutex already locked by another thread will suspend until the other thread unlocks the mutex.

Before 4.12 `Sys_error` was not raised for recursive locking (platform-dependent behaviour)
Raises `Sys_error` if the mutex is already locked by the thread calling `Mutex.lock`.

val `try_lock : t -> bool`

Same as `Mutex.lock`, but does not suspend the calling thread if the mutex is already locked: just return **`false`** immediately in that case. If the mutex is unlocked, lock it and return **`true`**.

val `unlock : t -> unit`

Unlock the given mutex. Other threads suspended trying to lock the mutex will restart. The mutex must have been previously locked by the thread that calls `Mutex.unlock`.

Before 4.12 `Sys_error` was not raised when unlocking an unlocked mutex or when unlocking a mutex from a different thread.
Raises `Sys_error` if the mutex is unlocked or was locked by another thread.

https://ocaml.org/manual/5.4/api/Mutex.html

Implementing an atomic counter

```
module type COUNTER = sig
  (** Abstract counter type *)
  type t

  (** Create a new counter initialized to 0 *)
  val create : unit -> t

  (** Atomically get current value and increment, returns old value *)
  val get_and_increment : t -> int

  (** Get current count *)
  val get_count : t -> int
end
```

Implementing an atomic counter

```
module Counter : COUNTER = struct
  type t = { counter : int ref; mutex : Mutex.t }

  let create () = { counter = ref 0; mutex = Mutex.create () }

  let get_and_increment t =
    Mutex.lock t.mutex;
    let old_value = !(t.counter) in
    t.counter := old_value + 1;
    Mutex.unlock t.mutex;
    old_value

  let get_count t =
    Mutex.lock t.mutex;
    let count = !(t.counter) in
    Mutex.unlock t.mutex;
    count
end
```

Implementing an atomic counter

```
module Counter : COUNTER = struct
  type t = { counter : int ref; mutex : Mutex.t }

  let create () = { counter = ref 0; mutex = Mutex.create () }

  let get_and_increment t =
    Mutex.lock t.mutex;
    let old_value = !(t.counter) in
    t.counter := old_value + 1;
    Mutex.unlock t.mutex;
    old_value

  let get_count t =
    Mutex.lock t.mutex;
    let count = !(t.counter) in
    Mutex.unlock t.mutex;
    count
end
```

Critical Section

Implementing an atomic counter

```
module Counter : COUNTER = struct
  type t = { counter : int ref; mutex : Mutex.t }

  let create () = { counter = ref 0; mutex = Mutex.create () }

  let get_and_increment t =
    Mutex.lock t.mutex;
    let old_value = !(t.counter) in
    t.counter := old_value + 1;
    Mutex.unlock t.mutex;
    old_value

  let get_count t =
    Mutex.lock t.mutex;
    let count = !(t.counter) in
    Mutex.unlock t.mutex;
    count
end
```

Critical Section

Demo

Desirable properties of locks

- Mutual exclusion
- Deadlock freedom
- Starvation freedom

Mutual Exclusion

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

Mutual Exclusion

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

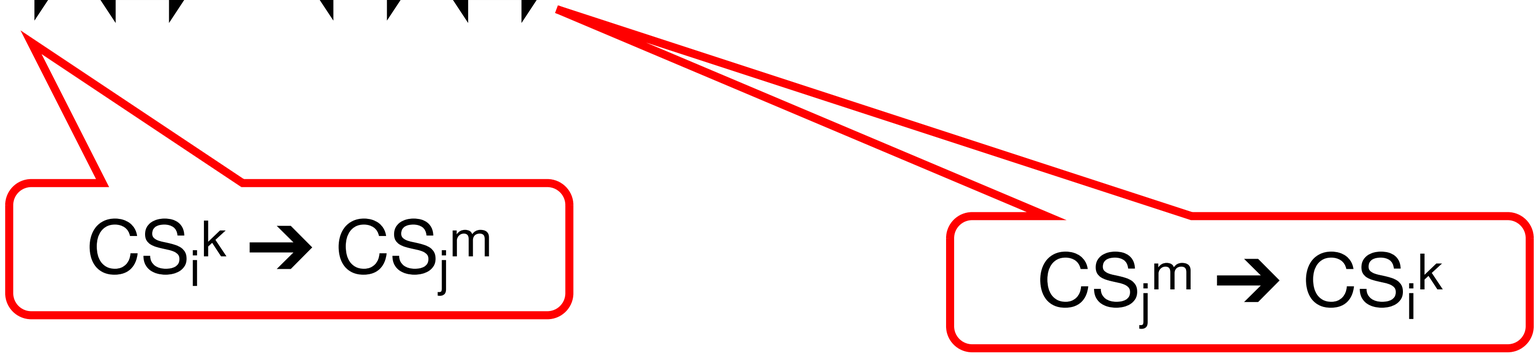
Mutual Exclusion

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



Deadlock Freedom

- 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 freedom

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

Two-Thread vs n-Thread solutions

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

Two-Thread Conventions

```
let lock () =
  let i = (Domain.self () :> int) - 1 in
  (* ^^ Returns 0 or 1 if you spawn only two domains *)
  let j = 1 - i in
  (* Other thread: if i=0 then j=1, if i=1 then j=0 *)
```

*For the discussion, the current thread will be **i** and the other thread will be **j***

LockOne

```
module LockOne : LOCK = struct
  (* Two boolean flags, one per thread *)
  let flag = [| false; false |]

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    (* Other thread: if i=0 then j=1, if i=1 then j=0 *)
    flag.(i) <- true;
    (* Wait while the other thread wants to enter critical section *)
    while flag.(j) do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

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

LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

A

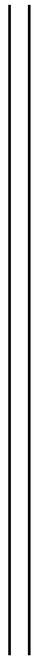
B

LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

Initially $\text{flag}[A] = \text{flag}[B] = \text{false}$

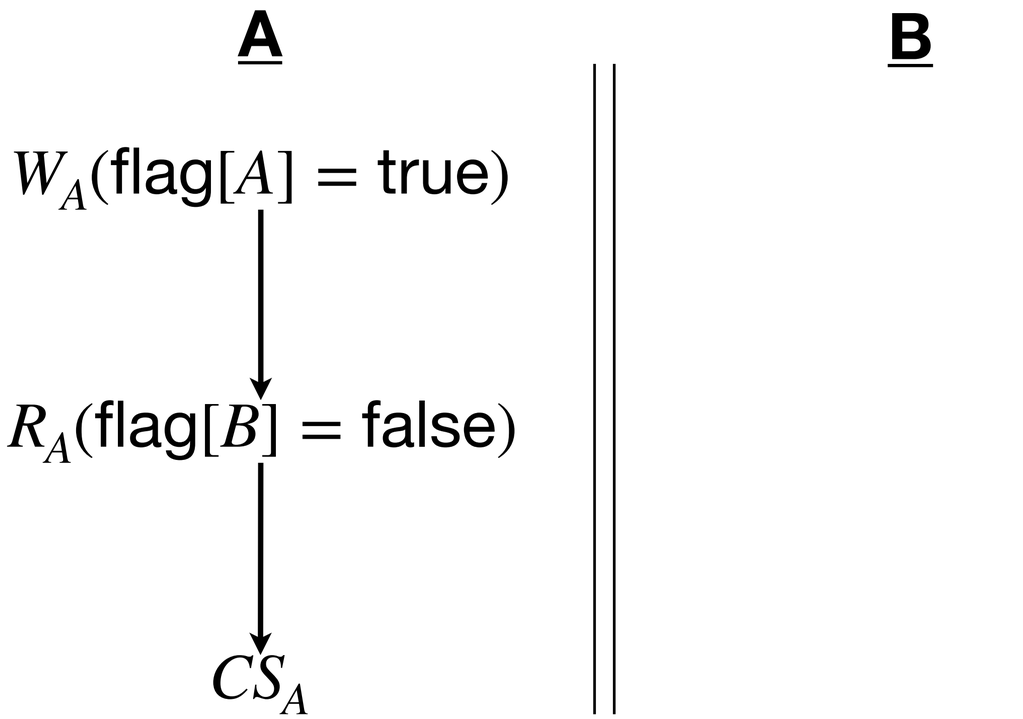
A **B**



LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

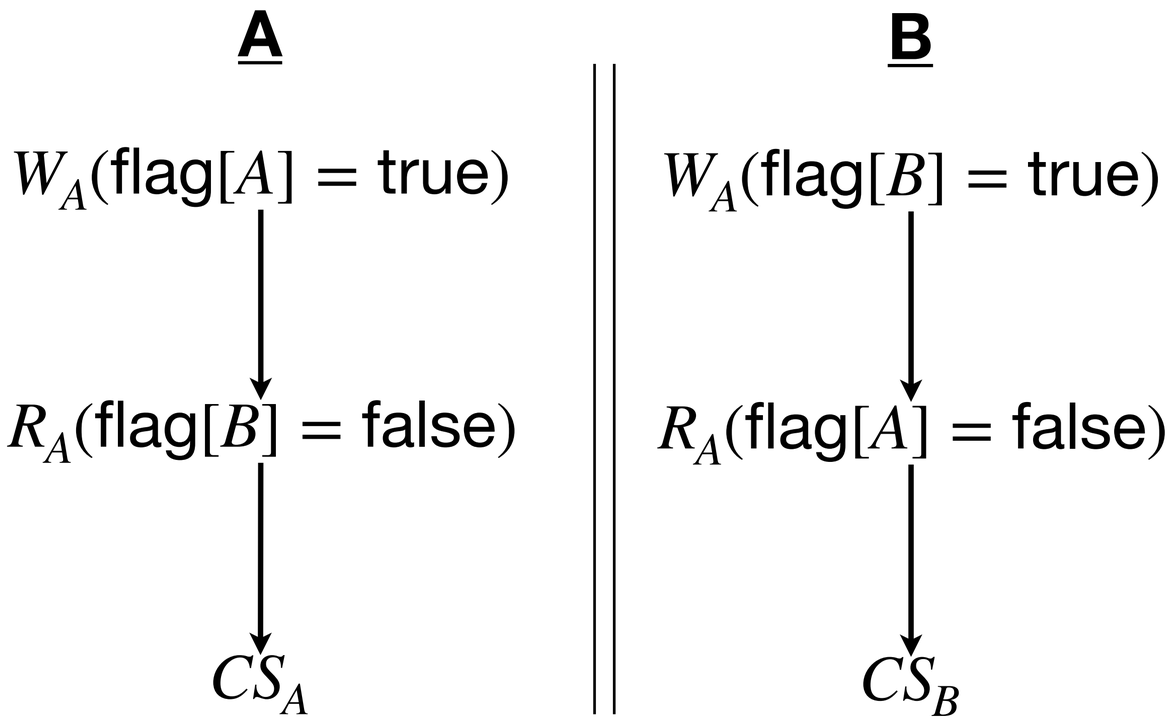
Initially $\text{flag}[A] = \text{flag}[B] = \text{false}$



LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

Initially $\text{flag}[A] = \text{flag}[B] = \text{false}$

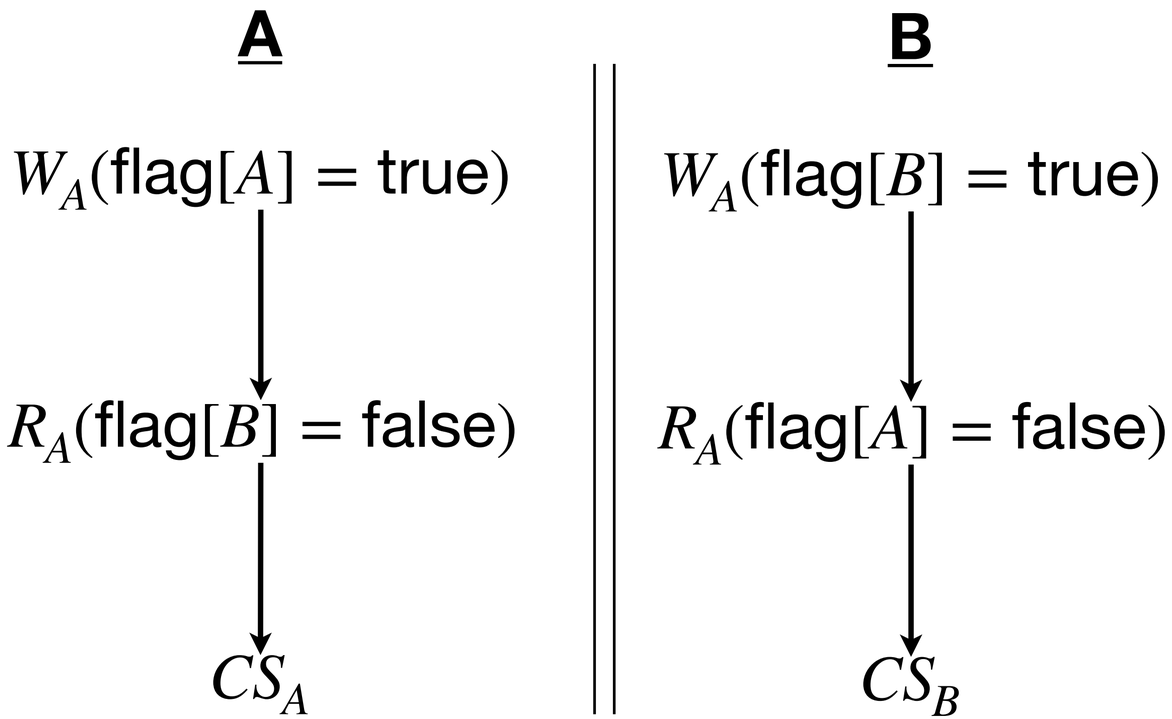


LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

- The ***events*** are ***interleaved***
  - Construct a ***total order*** from the ***partial order***

Initially $\text{flag}[A] = \text{flag}[B] = \text{false}$

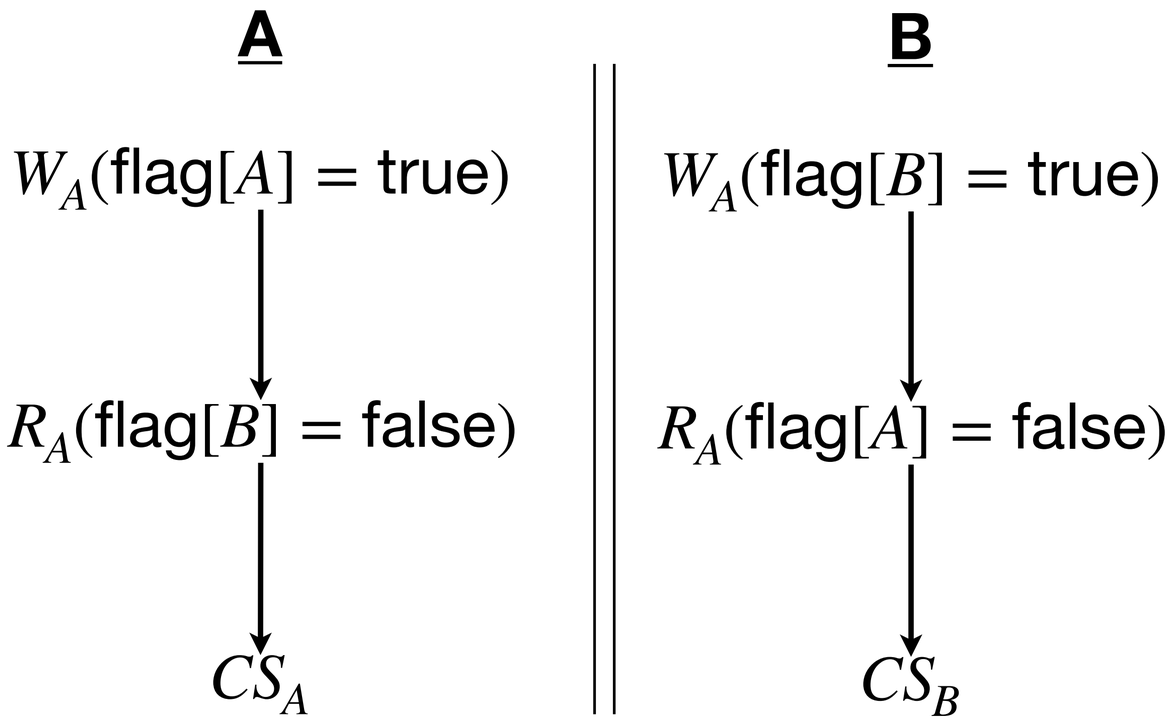


LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

- The ***events*** are ***interleaved***
  - Construct a ***total order*** from the ***partial order***
- Without loss of generality, assume that $W_A(\textbf{\textit{flag}}[A] = \textbf{\textit{true}})$ happened first.

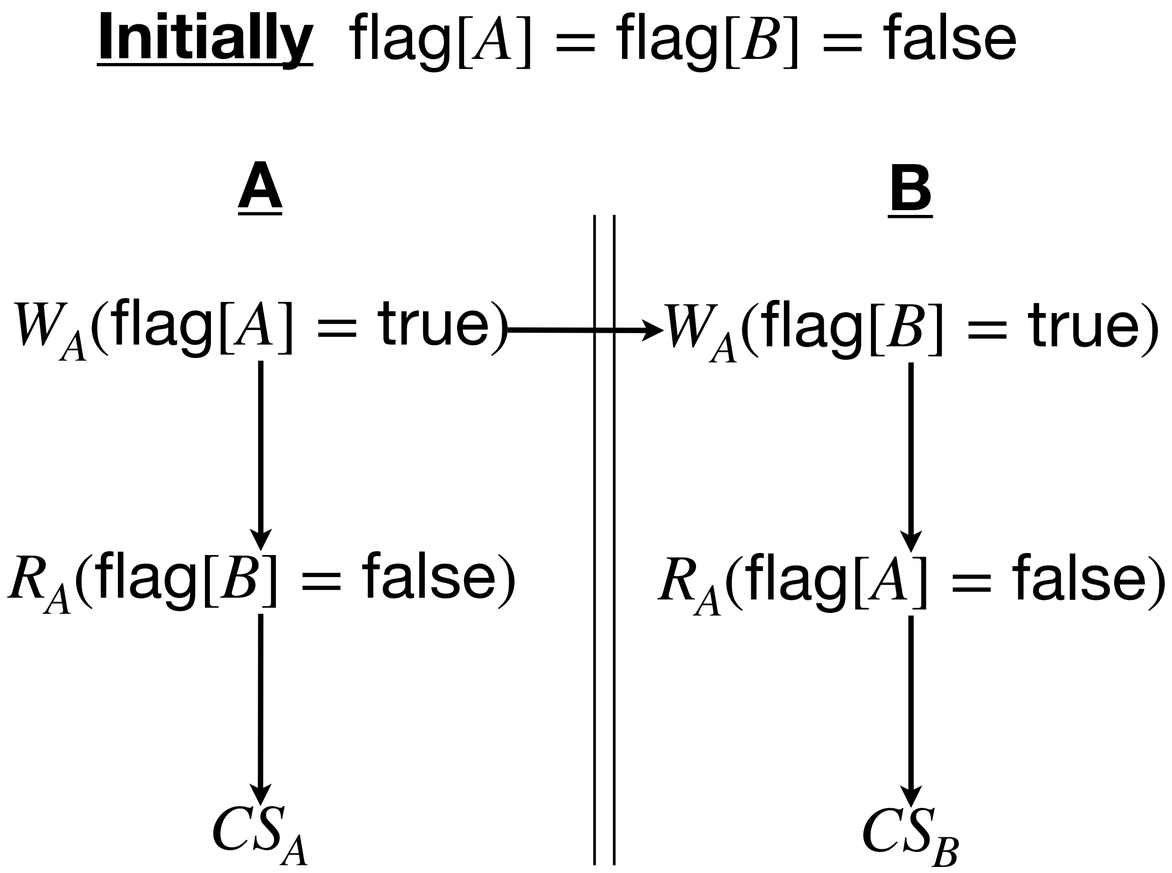
Initially $\text{flag}[A] = \text{flag}[B] = \text{false}$



LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

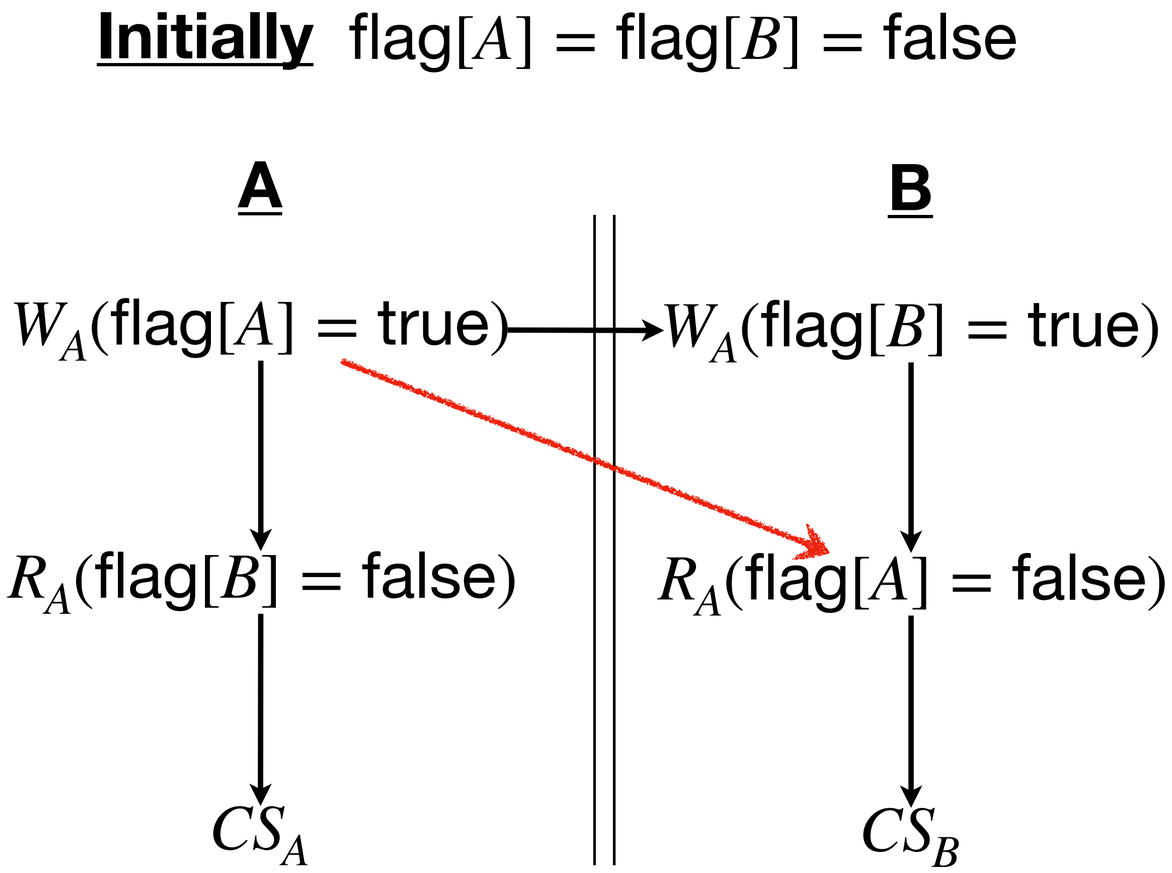
- The ***events*** are ***interleaved***
  - Construct a ***total order*** from the ***partial order***
- Without loss of generality, assume that $W_A(\textbf{\textit{flag}}[A] = \textbf{\textit{true}})$ happened first.



LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

- The ***events*** are ***interleaved***
  - Construct a ***total order*** from the ***partial order***
- Without loss of generality, assume that $W_A(\textbf{\textit{flag}}[A] = \textbf{\textit{true}})$ happened first.

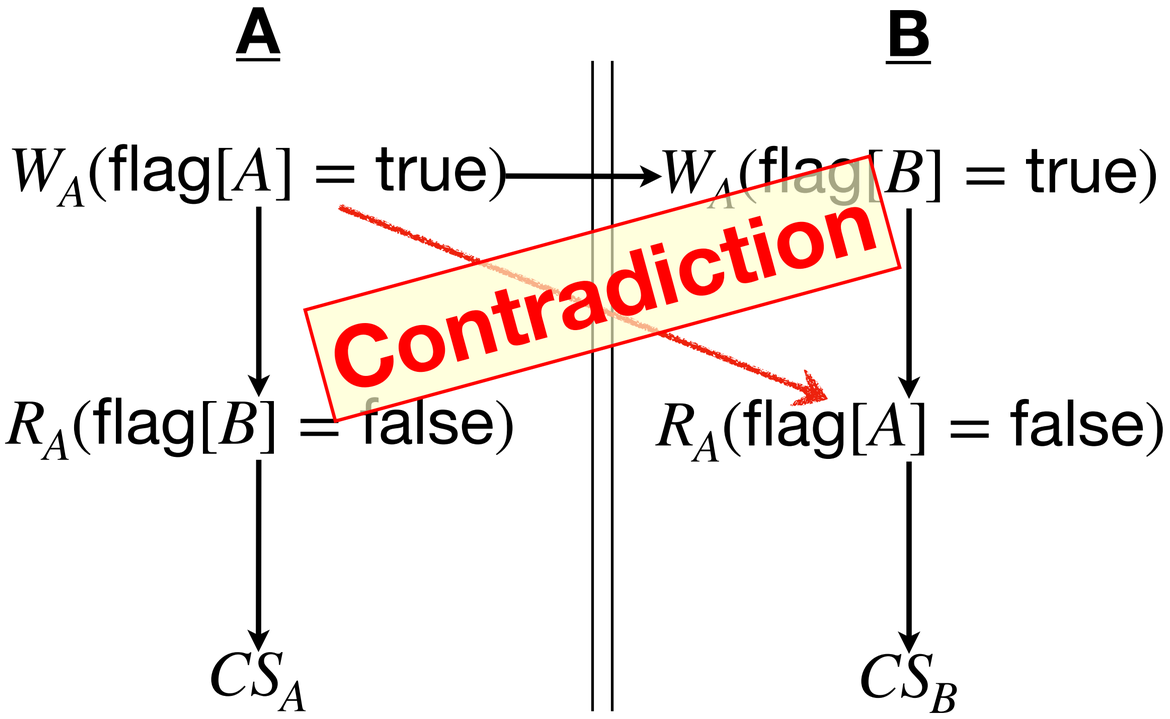


LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

- The ***events*** are ***interleaved***
  - Construct a ***total order*** from the ***partial order***
- Without loss of generality, assume that $W_A(\textbf{\textit{flag}}[A] = \textbf{\textit{true}})$ happened first.

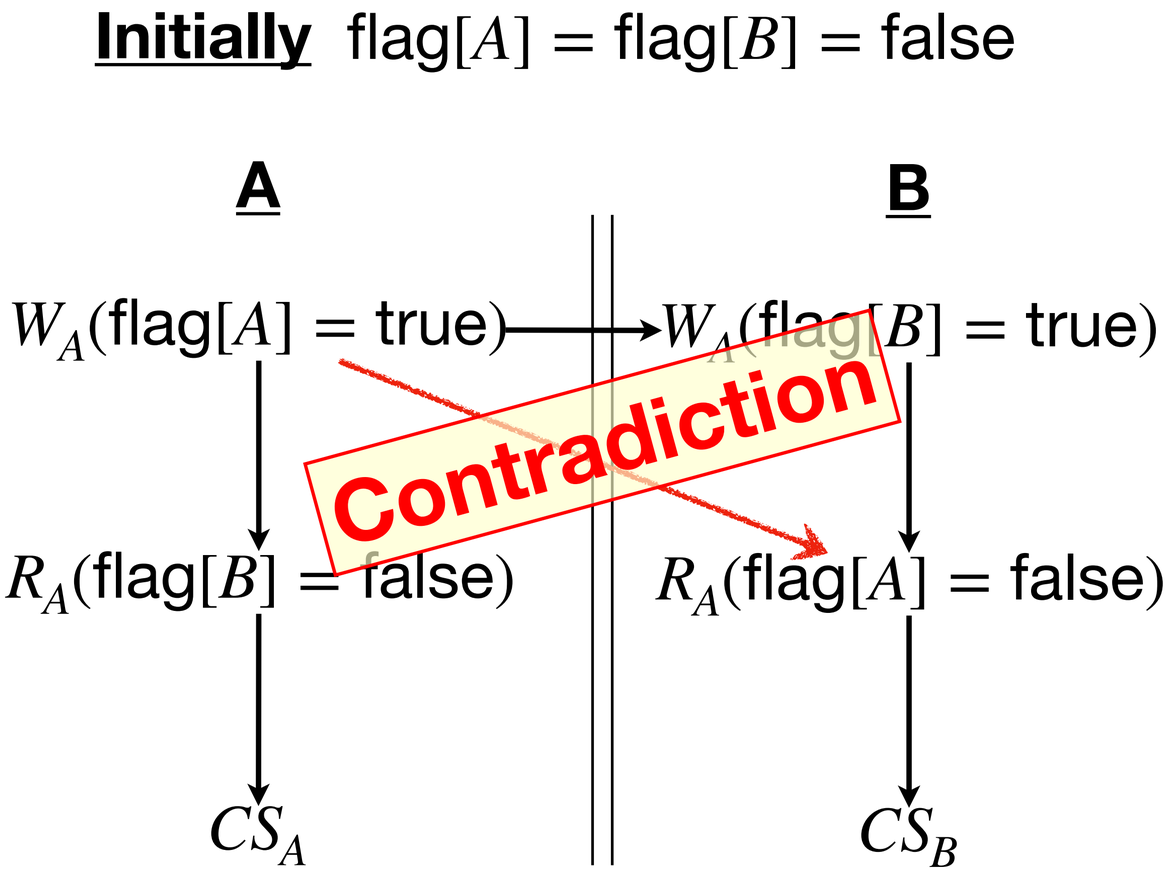
Initially $\text{flag}[A] = \text{flag}[B] = \text{false}$



LockOne *satisfies* Mutual Exclusion

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   (* Other thread: if i=0 then j=1,
                    if i=1 then j=0 *)
   flag.(i) <- true;
   (* Wait while the other thread wants
      to enter critical section *)
   while flag.(j) do
     ()
   done
```

- The ***events*** are ***interleaved***
  - Construct a ***total order*** from the ***partial order***
- Without loss of generality, assume that $W_A(\textbf{\textit{flag}}[A] = \textbf{\textit{true}})$ happened first.
- Can derive a ***contradiction*** on the other side as well



LockOne *fails* Deadlock freedom

- Concurrent executions deadlock

Thread 0

```
flag.(0) <- true;
 (* Wait while the other thread wants
    to enter critical section *)
while flag.(1) do
  ()
done
```

Thread 1

```
flag.(1) <- true;
 (* Wait while the other thread wants
    to enter critical section *)
while flag.(0) do
  ()
done
```

- Sequential executions are OK

LockOne *fails* Deadlock freedom

- Concurrent executions deadlock

Thread 0

```
flag.(0) <- true;
 (* Wait while the other thread wants
    to enter critical section *)
while flag.(1) do
  ()
done
```

Thread 1

```
flag.(1) <- true;
 (* Wait while the other thread wants
    to enter critical section *)
while flag.(0) do
  ()
done
```

- Sequential executions are OK

Demo

LockTwo

```
module LockTwo : LOCK = struct
  (* single shared 'victim' variable *)
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    victim := i;
    (* let the other go first *)
    while !victim = i do
      () (* wait for permission *)
    done

  let unlock () = () (* Nothing to do *)
end
```

LockTwo claims

```
module LockTwo : LOCK = struct
  (* single shared 'victim' variable *)
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    victim := i;
    (* let the other go first *)
    while !victim = i do
      () (* wait for permission *)
    done

  let unlock () = () (* Nothing to do *)
end
```

LockTwo claims

- Satisfies mutual exclusion
  - If thread **`i`** in CS
  - Then **`victim == j`**
  - Cannot be both 0 and 1

```
module LockTwo : LOCK = struct
  (* single shared 'victim' variable *)
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    victim := i;
    (* let the other go first *)
    while !victim = i do
      () (* wait for permission *)
    done

  let unlock () = () (* Nothing to do *)
end
```

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***

```
module LockTwo : LOCK = struct
  (* single shared 'victim' variable *)
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    victim := i;
    (* let the other go first *)
    while !victim = i do
      () (* wait for permission *)
    done

  let unlock () = () (* Nothing to do *)
end
```

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***

```
module LockTwo : LOCK = struct
  (* single shared 'victim' variable *)
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    victim := i;
    (* let the other go first *)
    while !victim = i do
      () (* wait for permission *)
    done

  let unlock () = () (* Nothing to do *)
end
```

Demo

Peterson’s Algorithm

- Combine ideas from LockOne and LockTwo

```
module Peterson : LOCK = struct
  (* Two boolean flags (from LockOne) and one victim variable (from LockTwo) *)
  let flag = [| false; false |]
  let victim = ref 0

  let lock () =
    let i = (Domain.self () :> int) - 1 in
    let j = 1 - i in
    flag.(i) <- true; (* Announce I’m interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants to enter AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    let i = (Domain.self () :> int) - 1 in
    flag.(i) <- false
end
```

Peterson’s satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I’m interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

Peterson’s satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I’m interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

Peterson’s satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I’m interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B] := \text{true}) \rightarrow \text{write}_B(\text{victim} := B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

Peterson's satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B] := \text{true}) \rightarrow \text{write}_B(\text{victim} := B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

WLOG assume that A wrote to victim last

Peterson's satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B] := \text{true}) \rightarrow \text{write}_B(\text{victim} := B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

WLOG assume that A wrote to victim last

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

Peterson's satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B] := \text{true}) \rightarrow \text{write}_B(\text{victim} := B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

WLOG assume that A wrote to victim last

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

Peterson's satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

Victim == A

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B] := \text{true}) \rightarrow \text{write}_B(\text{victim} := B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

WLOG assume that A wrote to victim last

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

Peterson's satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

flag[B] == false, Since A entered CS

Victim == A

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B] := \text{true}) \rightarrow \text{write}_B(\text{victim} := B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

WLOG assume that A wrote to victim last

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

Peterson's satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

flag[B] == false, Since A entered CS

Victim == A

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B] := \text{true}) \rightarrow \text{write}_B(\text{victim} := B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

WLOG assume that A wrote to victim last

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

Peterson's satisfies Mutual Exclusion

- Suppose it does not

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

Contradiction

flag[B] == false, Since A entered CS

Victim == A

$$\text{write}_A(\text{flag}[A] := \text{true}) \rightarrow \text{write}_A(\text{victim} := A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \quad (1)$$

$$\text{write}_B(\text{flag}[B] := \text{true}) \rightarrow \text{write}_B(\text{victim} := B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B \quad (2)$$

WLOG assume that A wrote to victim last

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

Peterson’s satisfies Deadlock Freedom

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I’m interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

Peterson’s satisfies Deadlock Freedom

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I’m interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

- Thread blocked
  - only at **`while`** loop
  - only if other's flag is true
  - only if it is the victim

Peterson’s satisfies Deadlock Freedom

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I’m interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

- 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

Peterson’s satisfies Deadlock Freedom

```
let lock () =
   let i = (Domain.self () :> int) - 1 in
   let j = 1 - i in
   flag.(i) <- true; (* Announce I’m interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done
```

- 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

Peterson’s lock is Starvation Free

- ***Thread i*** is blocked only if ***Thread j repeatedly re-enters*** CS

```
 let lock () =
    ...
    flag.(i) <- true; (* Announce I’m interested *)
    victim := i; (* Defer to the other *)
    (* Wait while the other thread wants
       to enter AND we're the victim *)
    while flag.(j) && !victim = i do
      ()
    done

  let unlock () =
    ...
    flag.(i) <- false
end
```

Peterson’s lock is Starvation Free

- ***Thread i*** is blocked only if ***Thread j repeatedly re-enters*** CS
- Requires $\text{flag}[j] = \text{true} \wedge \text{victim} = i$

```
let lock () =
   ...
   flag.(i) <- true; (* Announce I’m interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants
      to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done

 let unlock () =
   ...
   flag.(i) <- false
end
```

Peterson's lock is Starvation Free

- ***Thread i*** is blocked only if ***Thread j repeatedly re-enters*** CS
- Requires $\text{flag}[j] = \text{true} \wedge \text{victim} = i$
- But, when ***Thread j*** reenters, it sets victim to j
  - So, ***Thread i*** will enter the CS

```
let lock () =
   ...
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants
      to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done

 let unlock () =
   ...
   flag.(i) <- false
end
```

Peterson's lock is Starvation Free

- ***Thread i*** is blocked only if ***Thread j repeatedly re-enters*** CS
- Requires $\text{flag}[j] = \text{true} \wedge \text{victim} = i$
- But, when ***Thread j*** reenters, it sets victim to j
  - So, ***Thread i*** will enter the CS

```
let lock () =
   ...
   flag.(i) <- true; (* Announce I'm interested *)
   victim := i; (* Defer to the other *)
   (* Wait while the other thread wants
      to enter AND we're the victim *)
   while flag.(j) && !victim = i do
     ()
   done

 let unlock () =
   ...
   flag.(i) <- false
end
```

Demo

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 (FCFS)

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

Bakery Algorithm

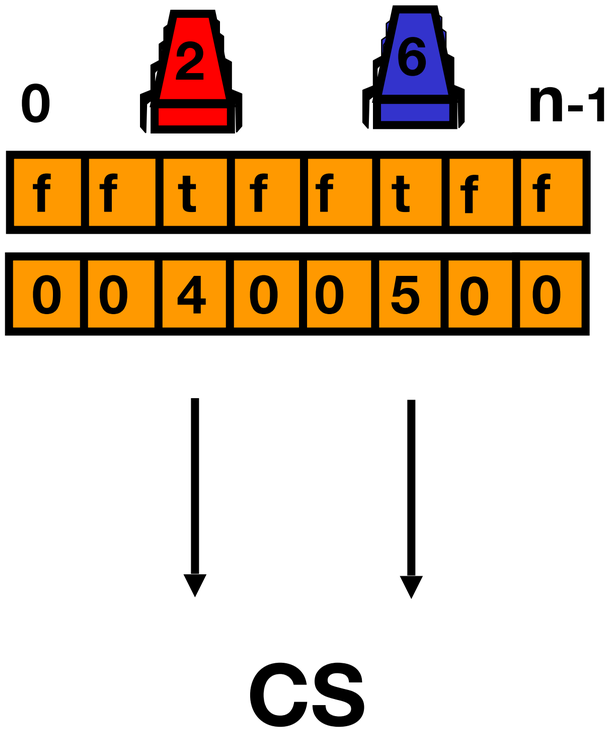
- Provides First-Come-First-Served for *n* threads
- How?
  - Take a ***"number"***
  - Wait until lower numbers have been served
- ***Lexicographic order***
  - (a,i) > (b,j)
    - If a > b, or a = b and i > j

Bakery Algorithm

```
module Bakery : BAKERY = struct

  type t = { flag : bool array;
             label : int array }

  let create n_threads =
    { flag = Array.make n_threads false;
      label = Array.make n_threads 0 }
```



Bakery Algorithm

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Bakery Algorithm

Doorway

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

```
let lock bakery =
  let thread_id = (Domain.self () :> int) - 1 in
  bakery.flag.(thread_id) <- true;
  let max = max_counter bakery.label in
  let my_label = max + 1 in
  bakery.label.(thread_id) <- my_label;
  (* Wait while there's a conflict *)
  while conflict bakery thread_id my_label do
    ()
  done

let unlock bakery =
  let thread_id = (Domain.self () :> int) - 1 in
  bakery.flag.(thread_id) <- false
```

Bakery Algorithm

I’m interested

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Bakery Algorithm

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Pick a label that's greater than all seen labels

Bakery Algorithm

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Pick a label that's greater than all seen labels

⚠️ *Two threads may pick the same label… that's ok!*

Bakery Algorithm

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

Someone other than me is interested

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Bakery Algorithm

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

Someone other than me is interested

Whose (label,id) is lexicographically smaller than mine

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Bakery Algorithm

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Not interested

Note: *Labels strictly increasing!*

No Deadlock

- There is always one thread with the earliest label
- Ties are impossible. Why?

First-Come-First-Served

- If $D_A \rightarrow D_B$ then
  - A's label is smaller

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

First-Come-First-Served

- If $D_A \rightarrow D_B$ then
  - A's label is smaller

Doorway

```
let lock bakery =
  let thread_id = (Domain.self () :> int) - 1 in
  bakery.flag.(thread_id) <- true;
  let max = max_counter bakery.label in
  let my_label = max + 1 in
  bakery.label.(thread_id) <- my_label;
  (* Wait while there's a conflict *)
  while conflict bakery thread_id my_label do
    ()
  done

let unlock bakery =
  let thread_id = (Domain.self () :> int) - 1 in
  bakery.flag.(thread_id) <- false
```

First-Come-First-Served

- If $D_A \rightarrow D_B$ then
  - A's label is smaller
- And:
  - $write_A(label[A]) \rightarrow read_B(label[A]) \rightarrow$

    $write_B(label[B]) \rightarrow read_B(flag[A])$

Doorway

```
let lock bakery =
  let thread_id = (Domain.self () :> int) - 1 in
  bakery.flag.(thread_id) <- true;
  let max = max_counter bakery.label in
  let my_label = max + 1 in
  bakery.label.(thread_id) <- my_label;
  (* Wait while there's a conflict *)
  while conflict bakery thread_id my_label do
    ()
  done

let unlock bakery =
  let thread_id = (Domain.self () :> int) - 1 in
  bakery.flag.(thread_id) <- false
```

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

Doorway

```
let lock bakery =
  let thread_id = (Domain.self () :> int) - 1 in
  bakery.flag.(thread_id) <- true;
  let max = max_counter bakery.label in
  let my_label = max + 1 in
  bakery.label.(thread_id) <- my_label;
  (* Wait while there's a conflict *)
  while conflict bakery thread_id my_label do
    ()
  done

let unlock bakery =
  let thread_id = (Domain.self () :> int) - 1 in
  bakery.flag.(thread_id) <- false
```

Mutual Exclusion

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Mutual Exclusion

- Suppose
  - A and B in CS together, and
  - A has an earlier label than B

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Mutual Exclusion

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

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Mutual Exclusion

- Suppose
  - A and B in CS together, and
  - A has an earlier label than B
- When B entered CS, it must have seen
  - flag[A] == *false*, or
  - label[A] > label[B]
- But labels are strictly increasing
  - So B must have seen flag[A] == *false*

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Mutual Exclusion

- Suppose
  - A and B in CS together, and
  - A has an earlier label than B
- When B entered CS, it must have seen
  - flag[A] == *false*, or
  - label[A] > label[B]
- But labels are strictly increasing
  - So B must have seen flag[A] == *false*
- Labeling_B ➔ read_B(flag[A] == false) ➔ write_A(flag[A] == true) ➔ Labeling_A

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Mutual Exclusion

- Suppose
  - A and B in CS together, and
  - A has an earlier label than B
- When B entered CS, it must have seen
  - flag[A] == *false*, or
  - label[A] > label[B]
- But labels are strictly increasing
  - So B must have seen flag[A] == *false*
- Labeling_B ➔ read_B(flag[A] == false) ➔ write_A(flag[A] == true) ➔ Labeling_A
- Which contradicts the assumption that A has an earlier label

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Mutual Exclusion

- Suppose
  - A and B in CS together, and
  - A has an earlier label than B
- When B entered CS, it must have seen
  - flag[A] == *false*, or
  - label[A] > label[B]
- But labels are strictly increasing
  - So B must have seen flag[A] == *false*
- Labeling_B ➔ read_B(flag[A] == false) ➔ write_A(flag[A] == true) ➔ Labeling_A
- Which contradicts the assumption that A has an earlier label

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Demo

Bakery Y2^{32}K Problem

```
(* Helper: find the maximum counter value across all labels *)
let max_counter labels = Array.fold_left max 0 labels

(* Helper: check if there's a conflict with any other thread *)
let conflict bakery me my_label =
  exists2i
    (fun i flag label ->
      i <> me && flag &&
      (label < my_label || (label = my_label && i < me)))
    bakery.flag bakery.label
```

```
let lock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- true;
   let max = max_counter bakery.label in
   let my_label = max + 1 in
   bakery.label.(thread_id) <- my_label;
   (* Wait while there's a conflict *)
   while conflict bakery thread_id my_label do
     ()
   done

 let unlock bakery =
   let thread_id = (Domain.self () :> int) - 1 in
   bakery.flag.(thread_id) <- false
```

Labels may overflow

- Bakery Algorithm assumes labels are ***strictly increasing***
  - Breaks when labels overflow

Does Overflow Actually Matter?

- Yes!
  - Y2K problem
    - Two-digit years (00–99) wrapped from 99 → 00.
    - Not a theoretical bug — it broke real systems.
  - 18 January 2038 (Unix **`time_t`** rollover)
    - Signed 32-bit seconds since 1970 overflow at $2^{31} – 1$.
    - This will happen on systems that still use 32-bit time_t.
  - 16-bit counters
    - Max value: 65,535
    - 1 increment/sec → wraps in 18 hours
    - 1 MHz hardware counter → wraps in 65 ms
- 64-bit counters (No), 32-bit counters (Maybe)

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 flavours
  - 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[]`**...

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…