

CS-206 Concurrency

Lecture 5

Event

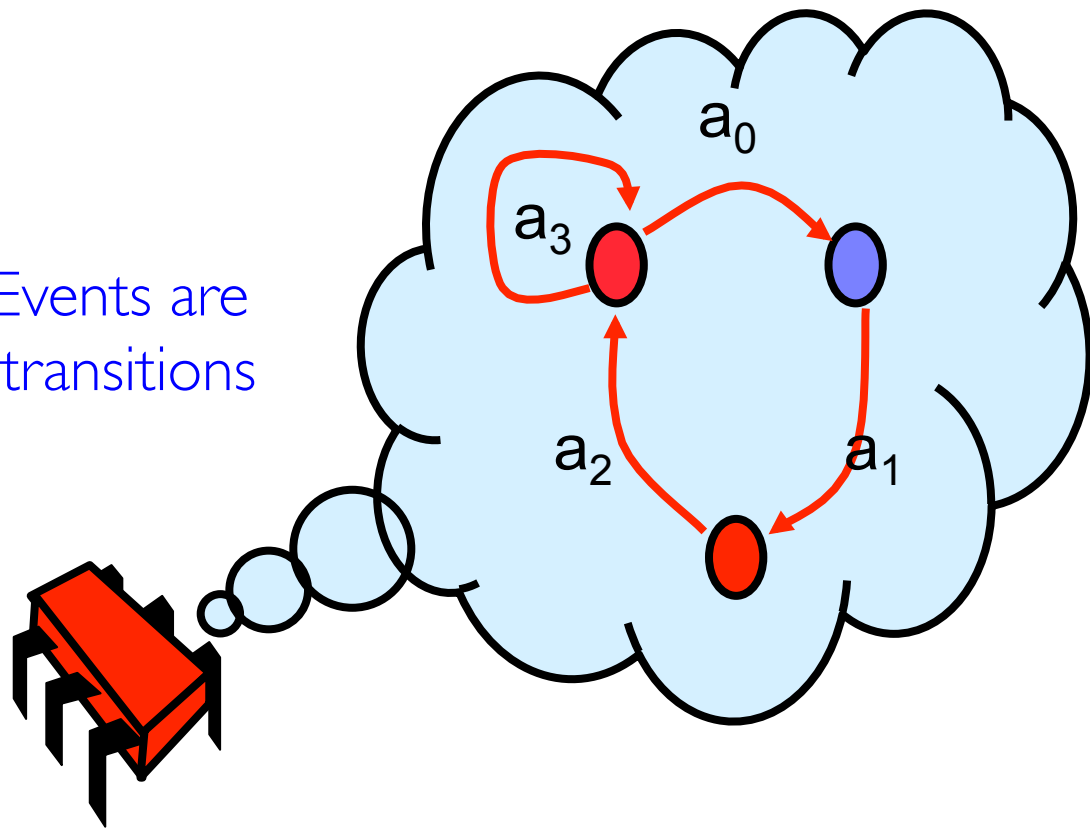
Ordering

Spring 2015

Prof. Babak Falsafi

parsa.epfl.ch/courses/cs206/

Events are
transitions



Adapted from slides originally developed by Maurice Herlihy and Nir Shavit from the Art of Multiprocessor Programming, and Babak Falsafi
EPFL Copyright 2015

Where are We?

Lecture & Lab				
M	T	W	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

▶ Event Ordering

▷ Formal definition

▶ Basic lock algorithms

▷ LockOne and LockTwo

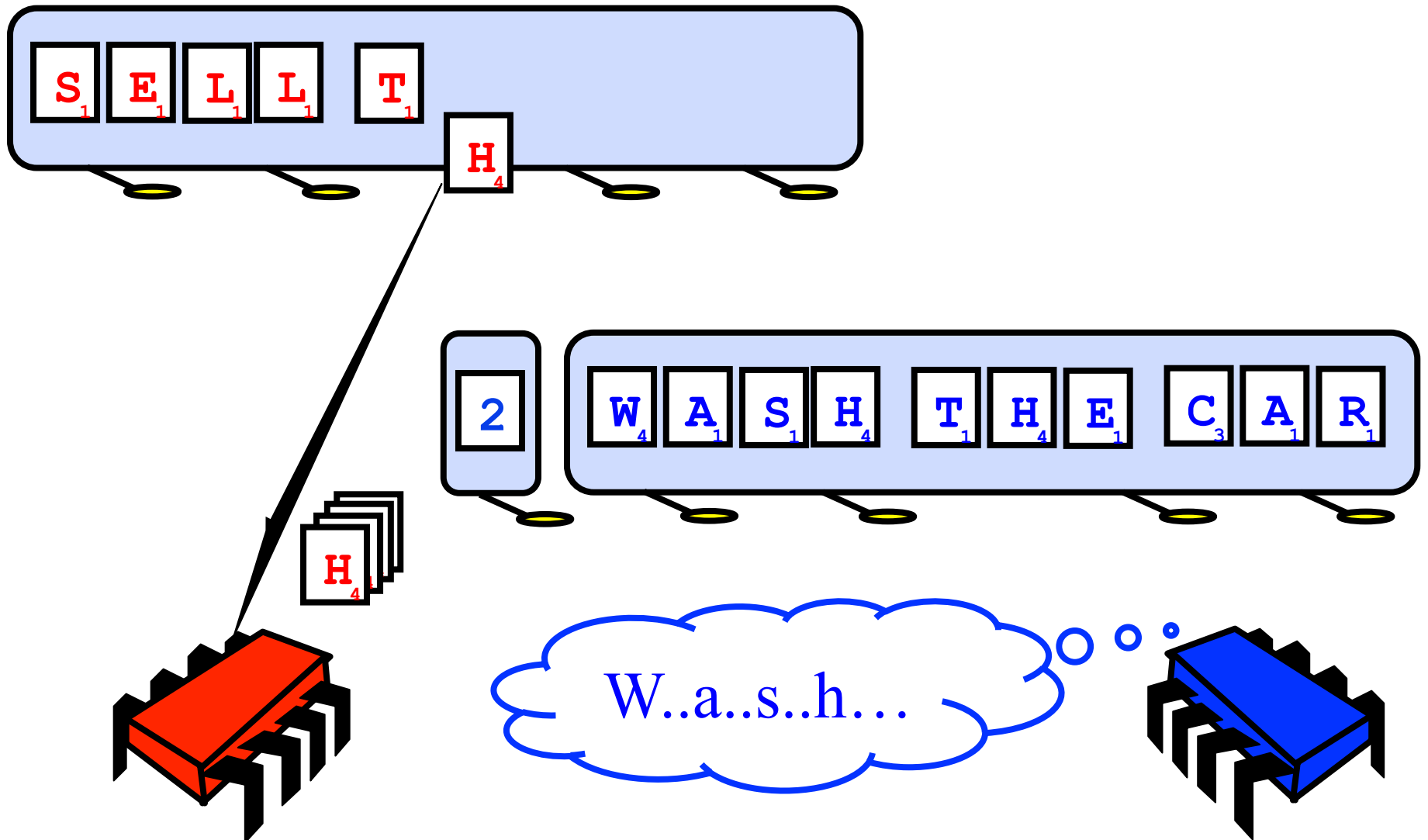
▶ Next week

▷ Advanced lock algorithms

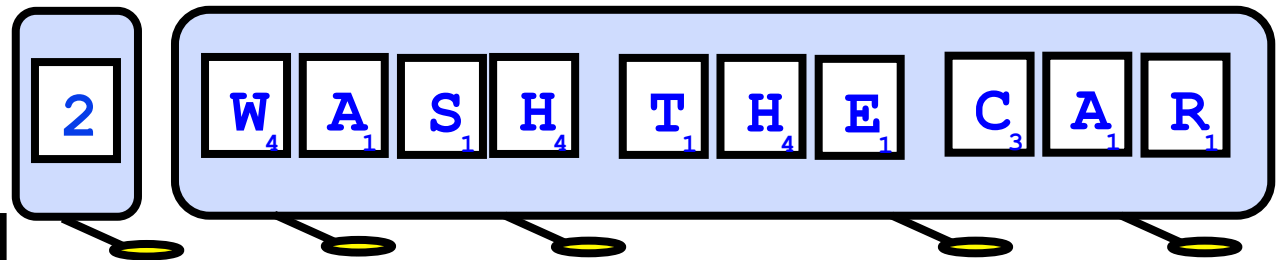
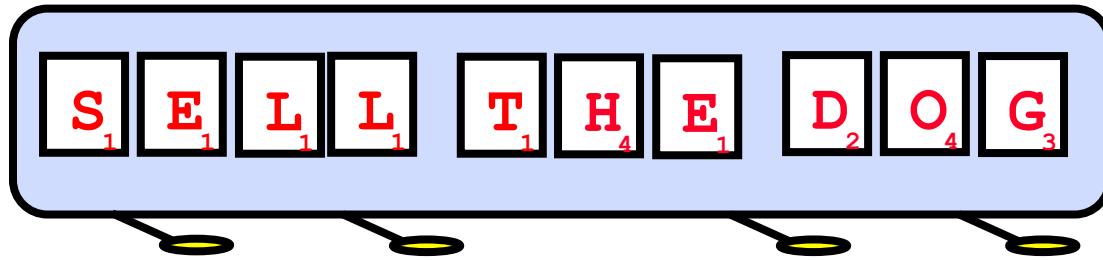
Recall: Parallelizing Readers/Writers

- ▶ Use two billboards
 - ▷ While Bob reads from one...
 - ▷ Alice writes to the other
- ▶ How do they know where to read/write?
 - ▷ Third billboard
 - ▷ Tells Bob which board to read

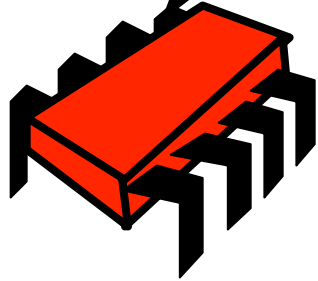
Wait-free protocol



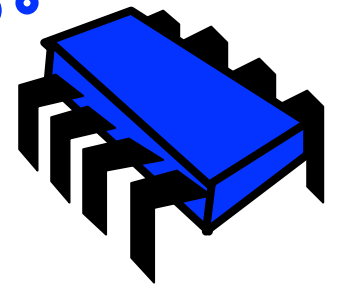
Wait-free protocol



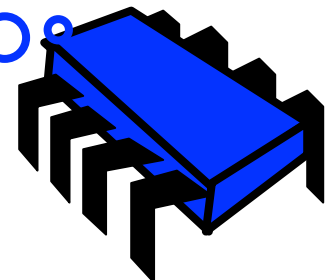
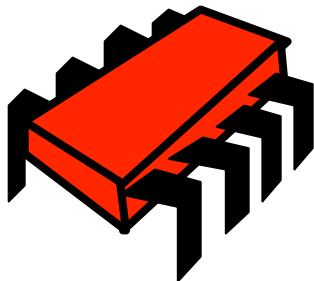
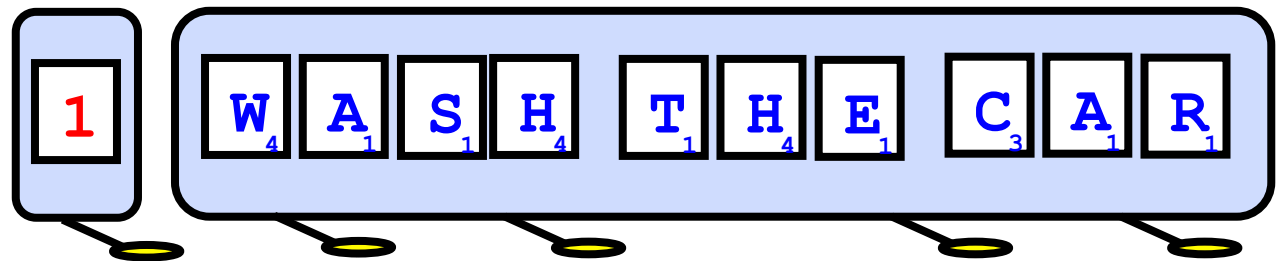
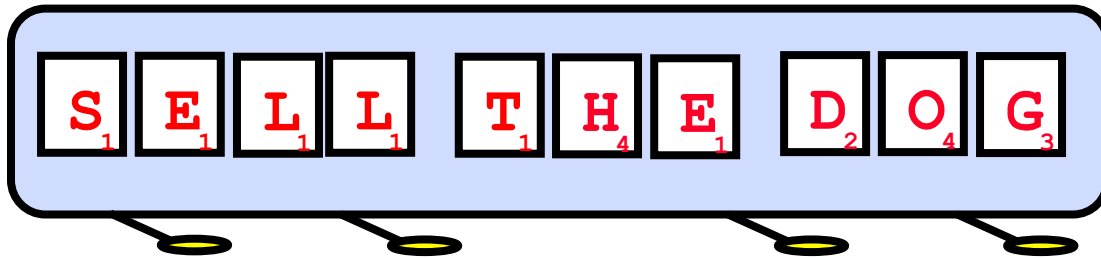
1



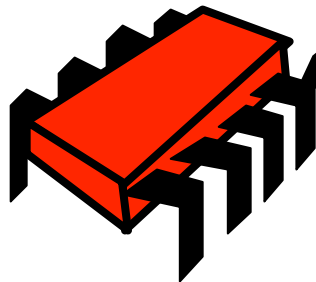
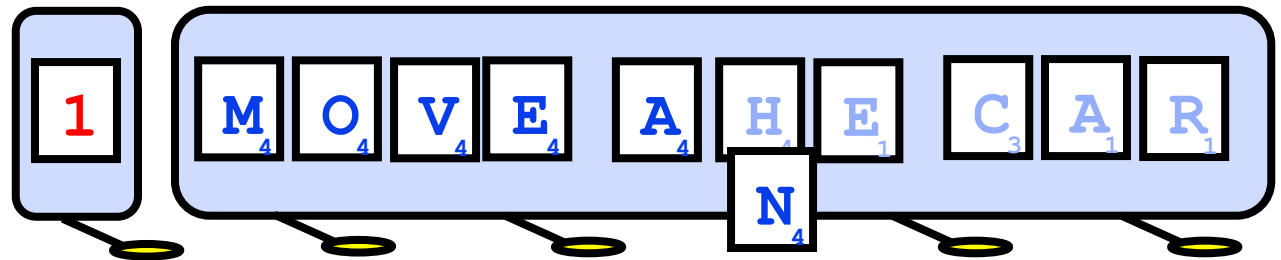
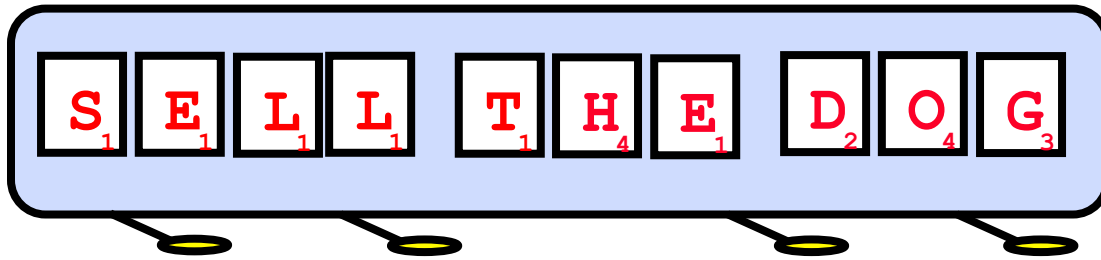
Wash the car! Got it!



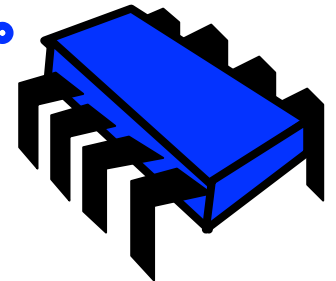
Wait-free protocol



Wait-free protocol



Sell the dog?
Ok!



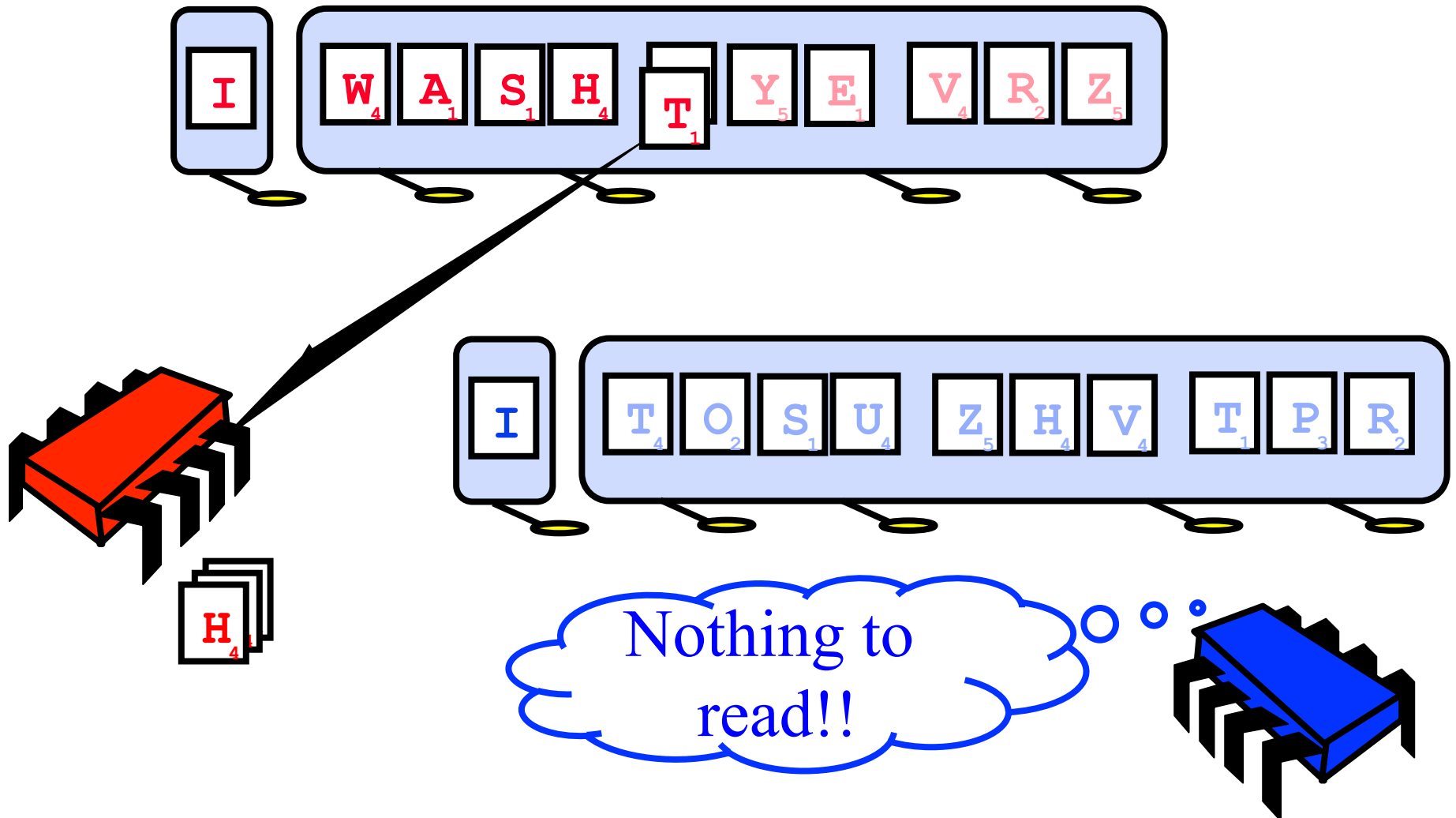
Wait-free protocol

- ▶ Is this protocol entirely wait-free?
- ▶ Is the protocol correct?
- ▶ How do you fix it?

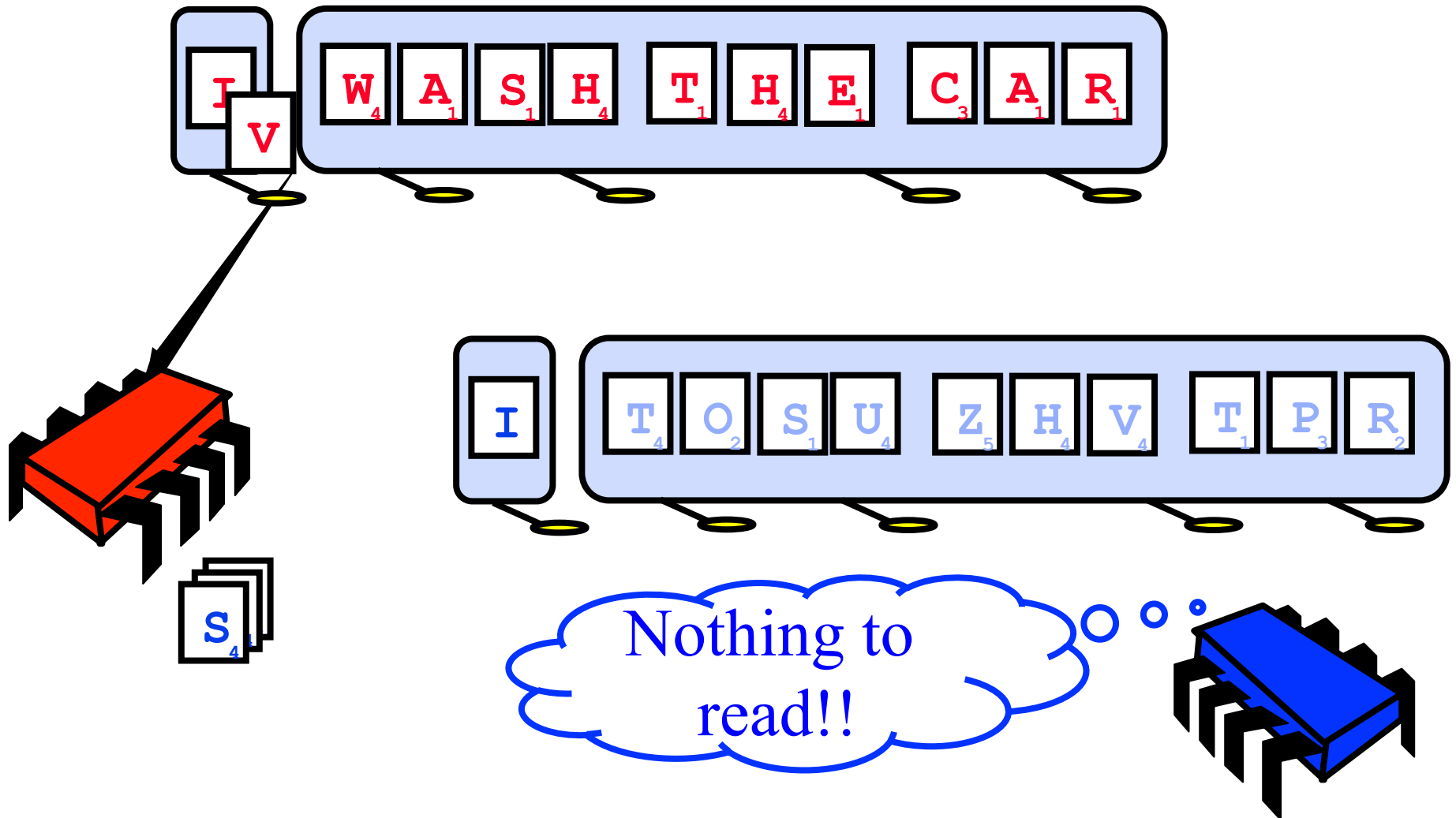
Fixing the protocol

- ▶ Add a valid/invalid flag to each billboard
- ▶ Alice only writes to invalid billboards
 - ▷ And marks them as valid afterwards
- ▶ Bob only reads from valid billboards
 - ▷ And marks them as invalid afterwards

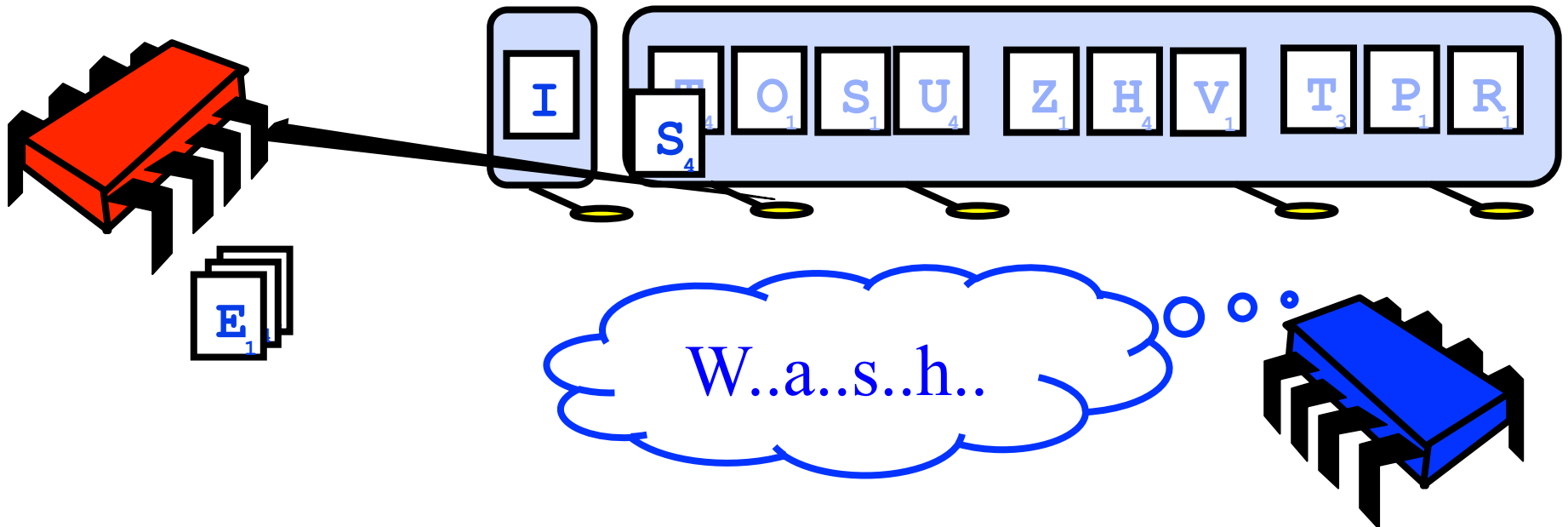
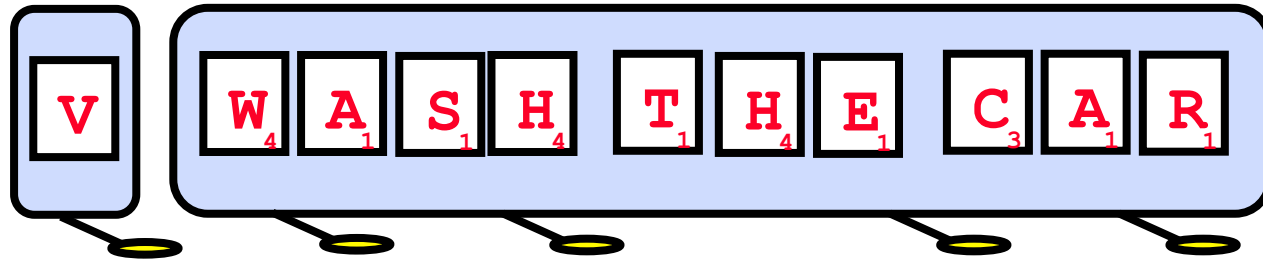
Wait-free protocol



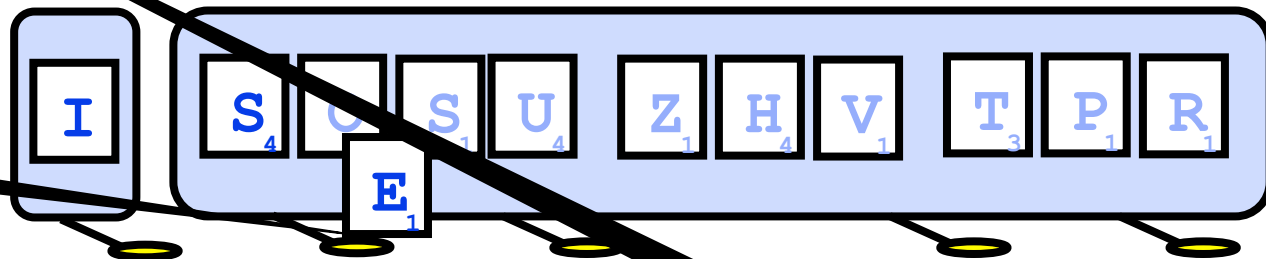
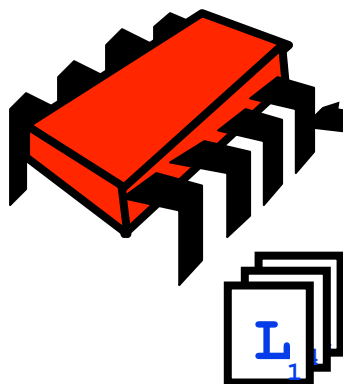
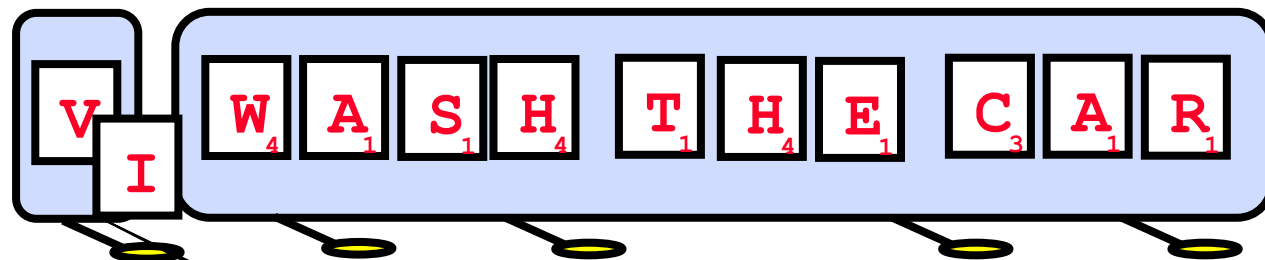
Wait-free protocol



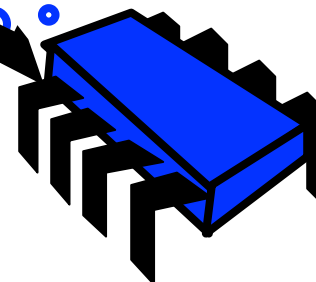
Wait-free protocol



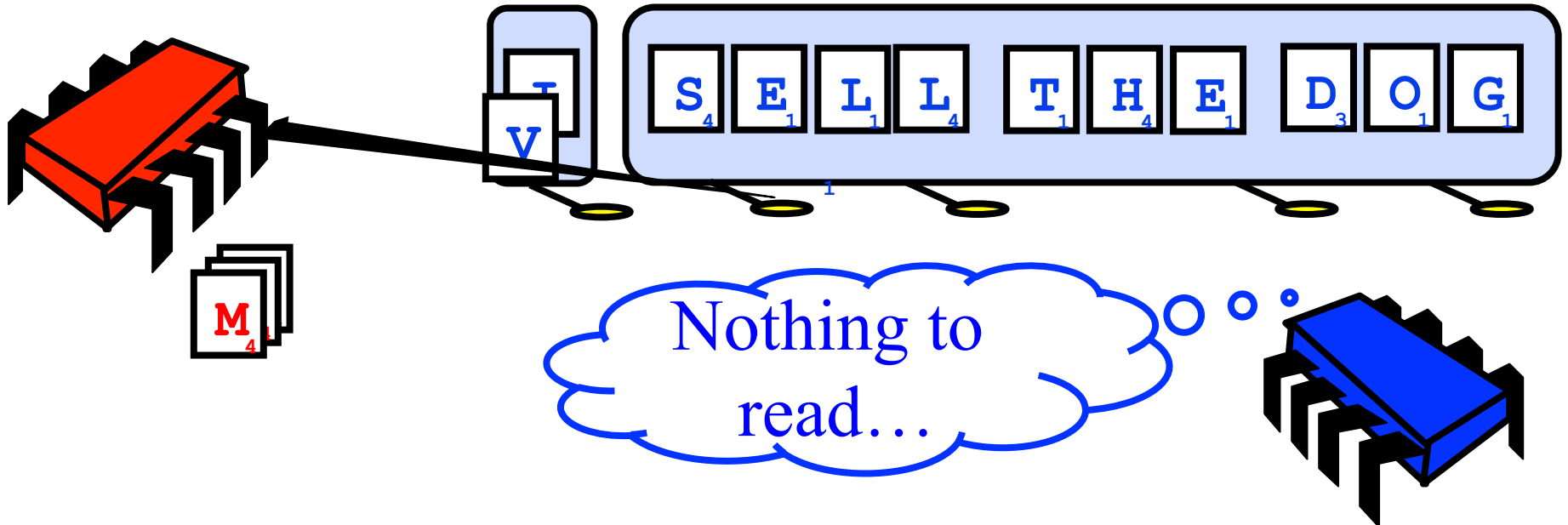
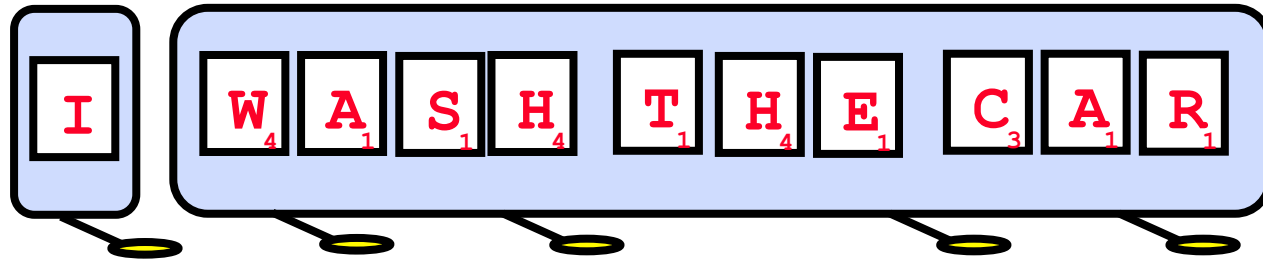
Wait-free protocol



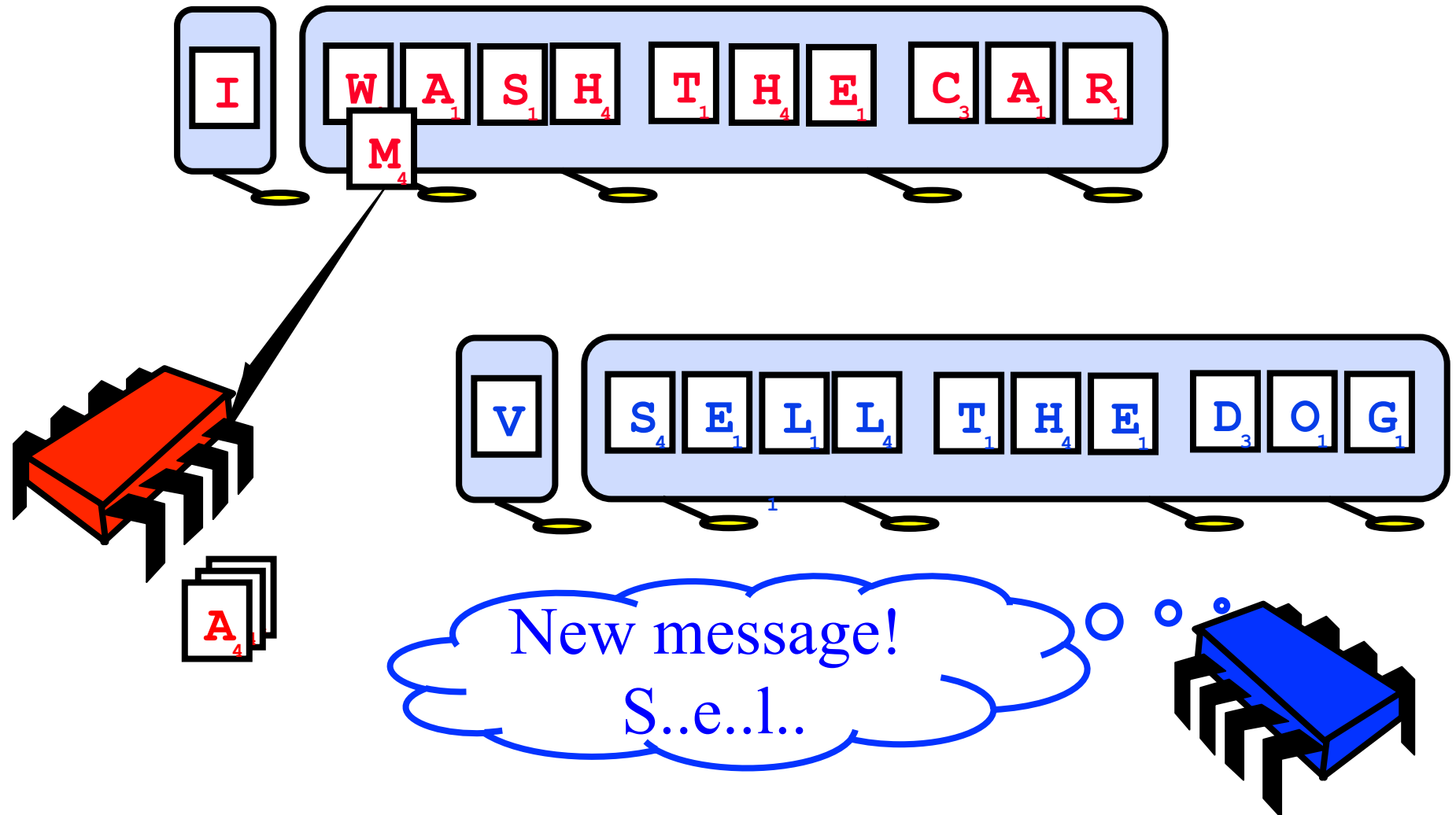
Wash the car. Got it!



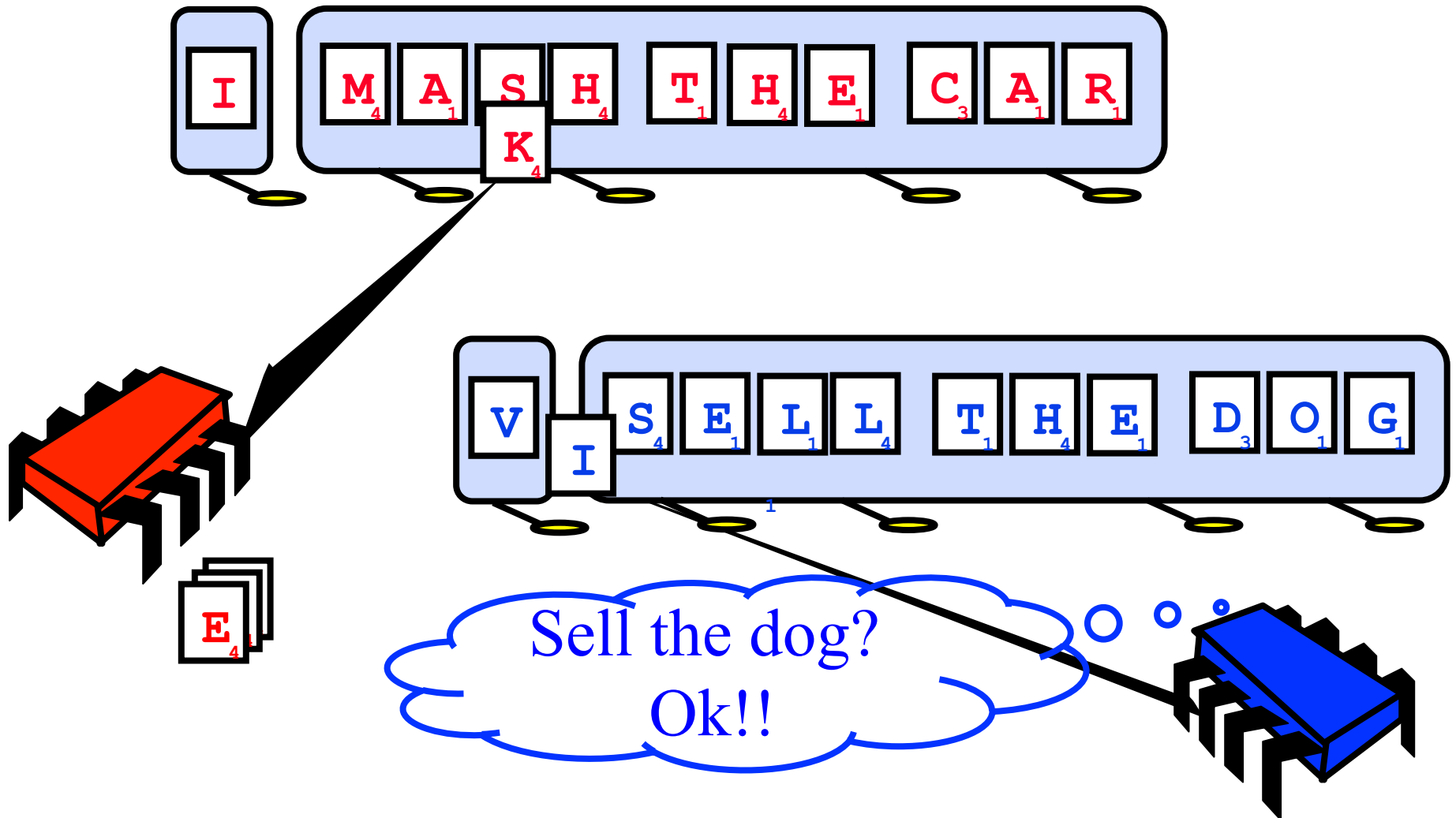
Wait-free protocol



Wait-free protocol



Wait-free protocol



Wait-free protocol

- ▶ Bob only waits if there is no new message
- ▶ Alice only waits if both billboards are written
- ▶ They can read/write in parallel
 - ▷ With locks, they could not
- ▶ But we have to use more billboards (memory)
- ▶ There is always such a trade-off
 - ▷ We can use more memory
 - ▷ Force less common operations to be slow
 - ▷ Lock resources and risk long waiting times
 - ▷ ...

Recall: Event Ordering Properties

Correctness:

- ▶ Safety
- ▶ Liveness

Quality:

- ▶ Fairness
- ▶ Performance

Need to formalize the problem to reason about correctness

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



- ▶ 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

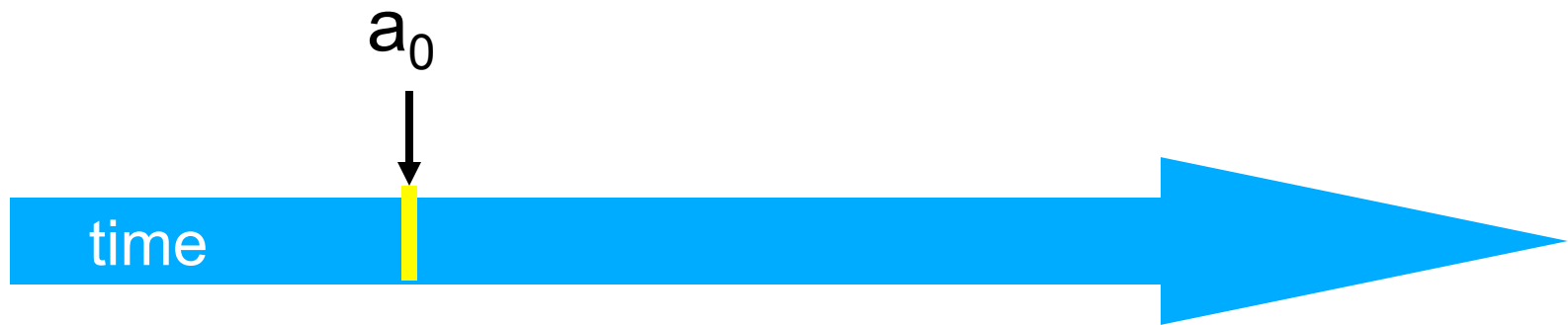
- ▶ “Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external.” (I. Newton, 1689)
- ▶ “Time is, like, Nature’s way of making sure that everything doesn’t happen all at once.” (Anonymous, circa 1968)



time

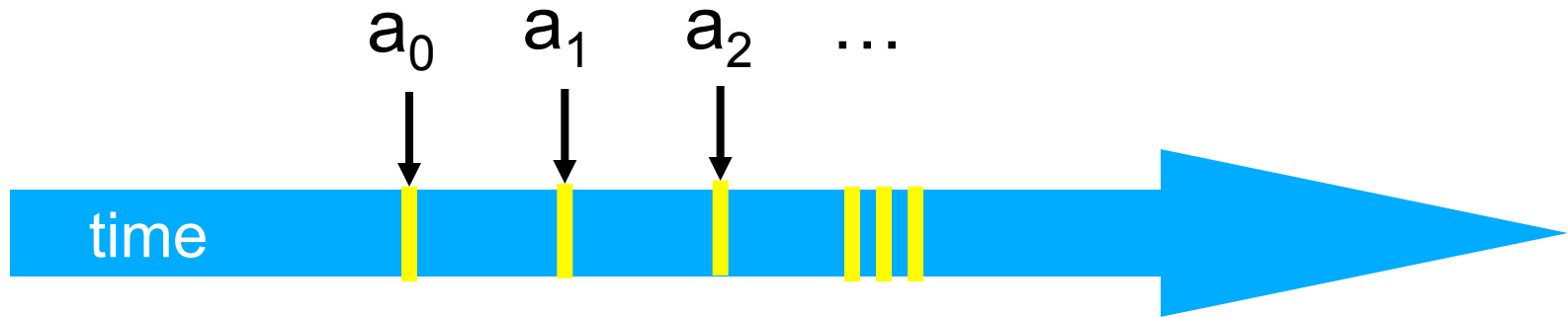
Events

- ▶ 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, \dots of events
 - ▷ “Trace” model
 - ▷ Notation: $a_0 \rightarrow a_1$ indicates order

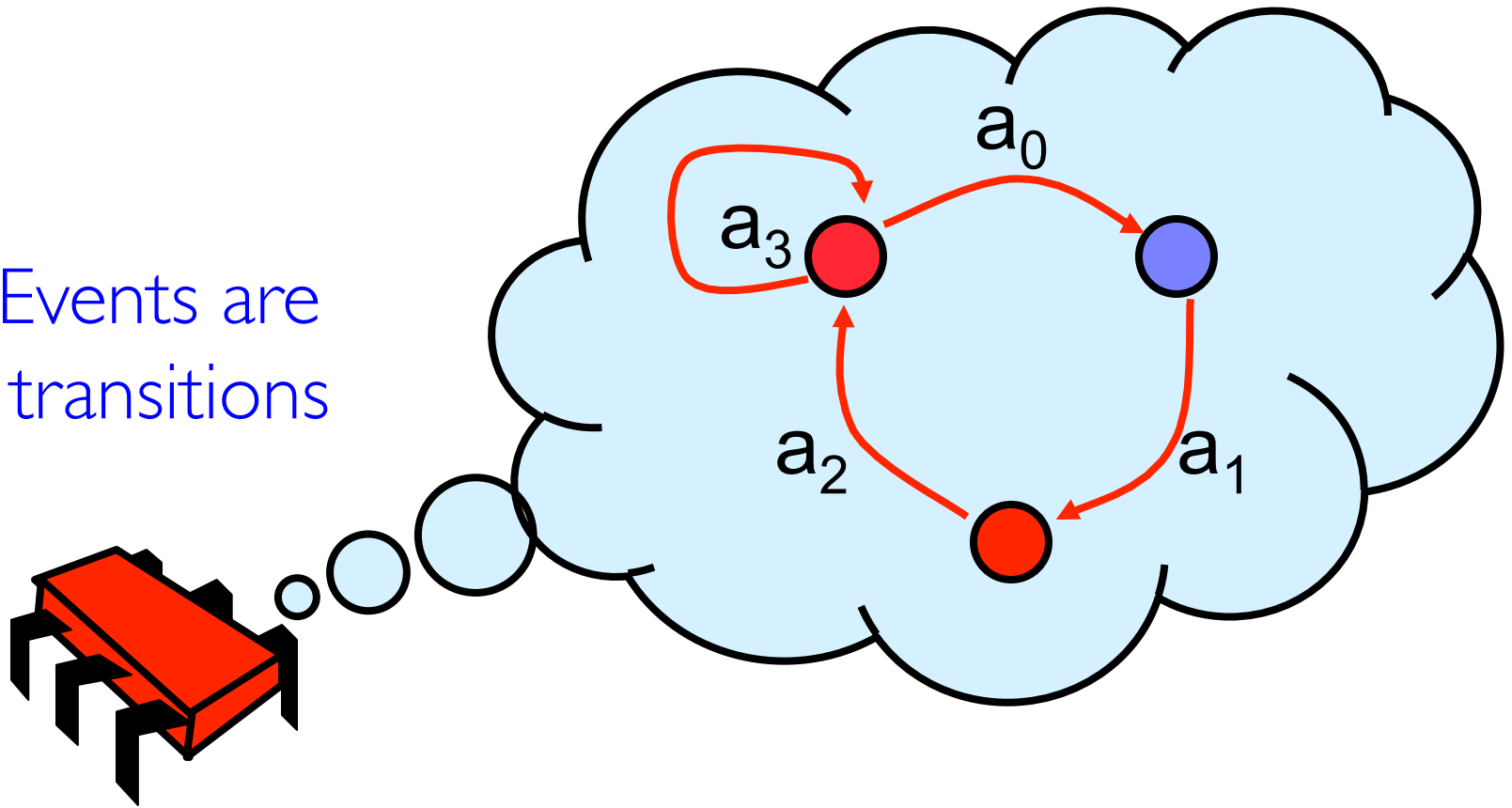


Example Thread Events

- ▶ 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

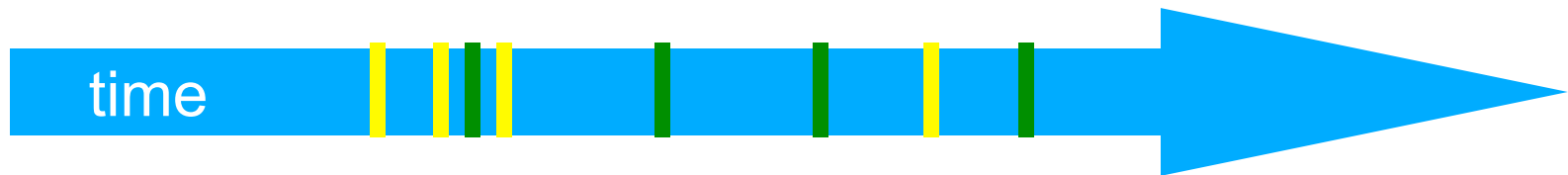


Critical Section

- ▶ A **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

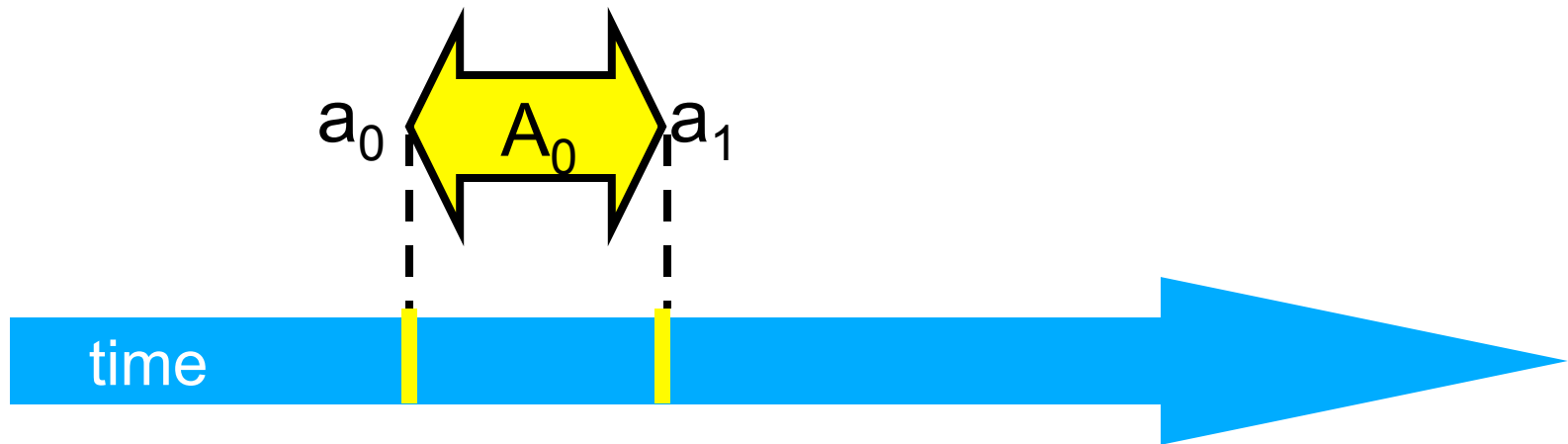
Interleavings

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

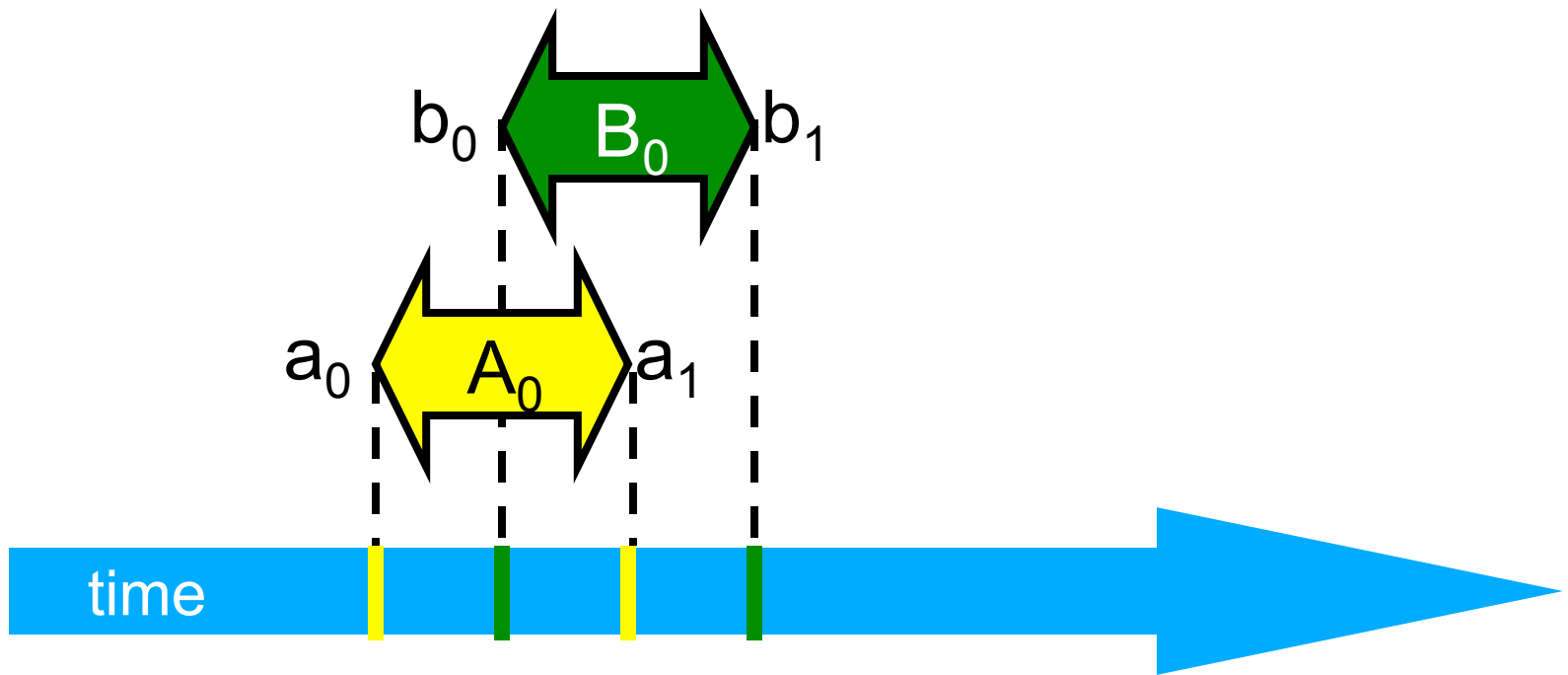


Intervals

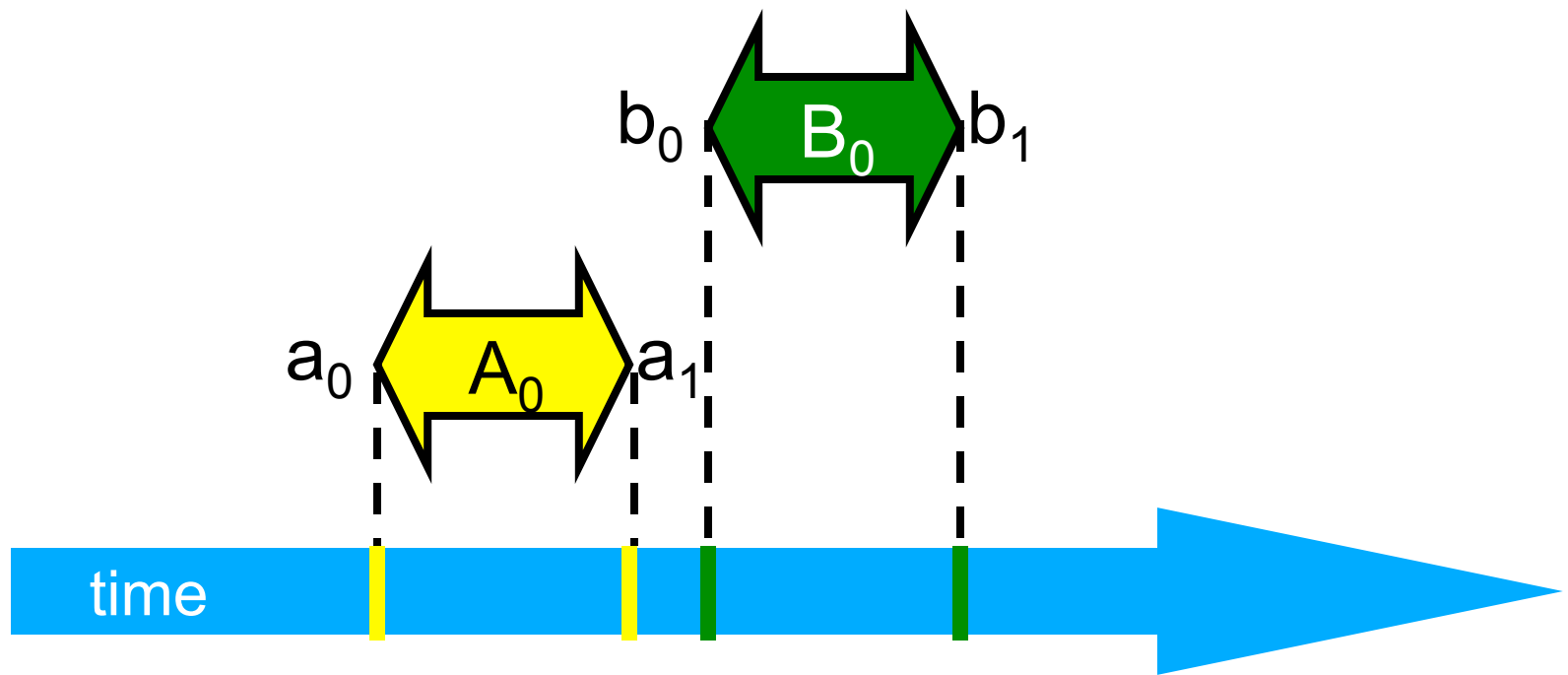
- ▶ An *interval* $A_0 = (a_0, a_1)$ is
 - ▷ Time between events a_0 and a_1



Intervals may Overlap

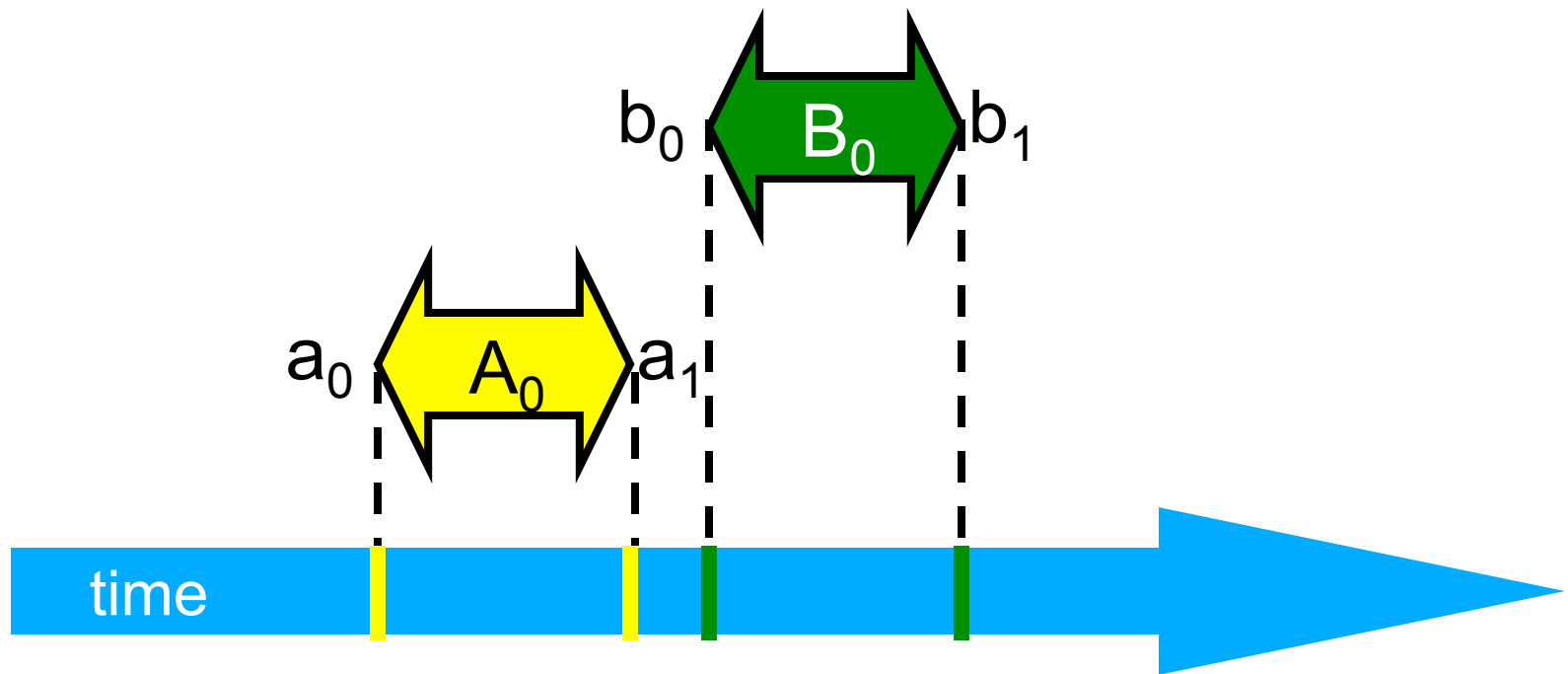


Intervals may be Disjoint

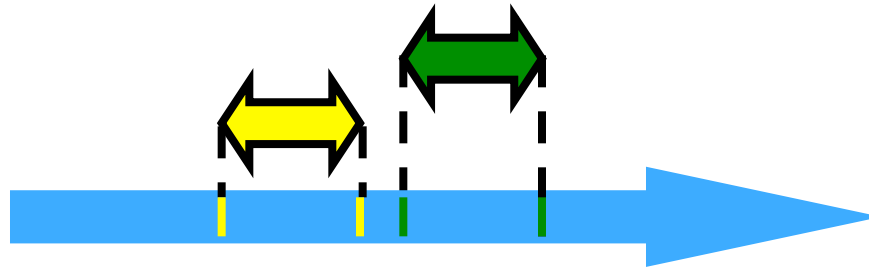


Precedence

Interval A_0 precedes interval B_0

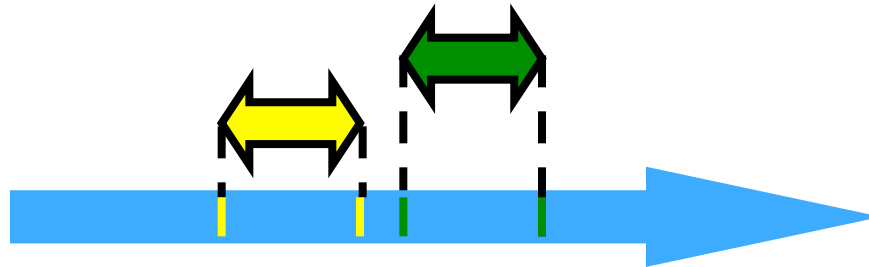


Precedence



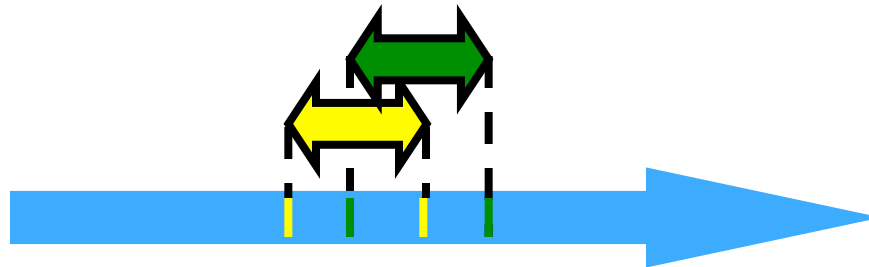
- ▶ Notation: $A_0 \rightarrow B_0$
- ▶ Formally,
 - ▷ End event of A_0 before start event of B_0
 - ▷ Also called “happens before” or “precedes”

Precedence Ordering



- ▶ Remark: $A_0 \rightarrow B_0$ is just like saying
 - ▷ 1066 AD \rightarrow 1492 AD,
 - ▷ Middle Ages \rightarrow 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 $A \rightarrow B$ or $B \rightarrow A$

Repeated Events

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

k -th occurrence of
event a_0

a_0^k

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

A_0^k

Implementing a Counter

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

Make these steps
indivisible using locks

Locks (Mutual Exclusion)

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

Locks (Mutual Exclusion)

```
public interface Lock {
```

```
    public void lock();
```

acquire lock

```
    public void unlock();
```

```
}
```

Locks (Mutual Exclusion)

```
public interface Lock {
```

```
    public void lock();
```

acquire lock

```
    public void unlock();
```

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  be thread i's k-th critical section execution



Mutual Exclusion

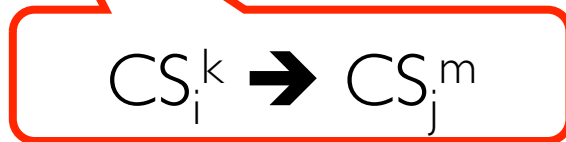
- ▶ Let CS_i^k  be thread i's k-th critical section execution
- ▶ And CS_j^m  be thread j's m-th critical section execution

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



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



$CS_i^k \rightarrow CS_j^m$

$CS_j^m \rightarrow CS_i^k$

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

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]) {}
}
```

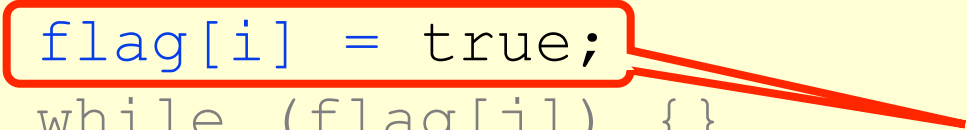
LockOne

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

Each thread has flag

LockOne

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



Set my flag

LockOne

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

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 the `lock()` method before entering
- ▶ Derive a contradiction
 - ▷ Assume the two enter critical section together
 - ▷ Show it is not possible!

From the Code

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

```
class LockOne implements Lock {  
    ...  
    public void lock() {  
        ...  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

From the Assumption

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

Combining

▶ Assumptions:

▷ $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

▷ $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

▶ From the code

▷ $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

▷ $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

▶ Assumptions:

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

▶ From the code

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

Combining

▶ Assumptions:

▶ $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

▶ $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

▶ From the code

▶ $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

▶ $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

▶ Assumptions:

▶ $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

▶ $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

▶ From the code

▶ $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

▶ $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

▶ Assumptions:

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

▶ From the code

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

Combining

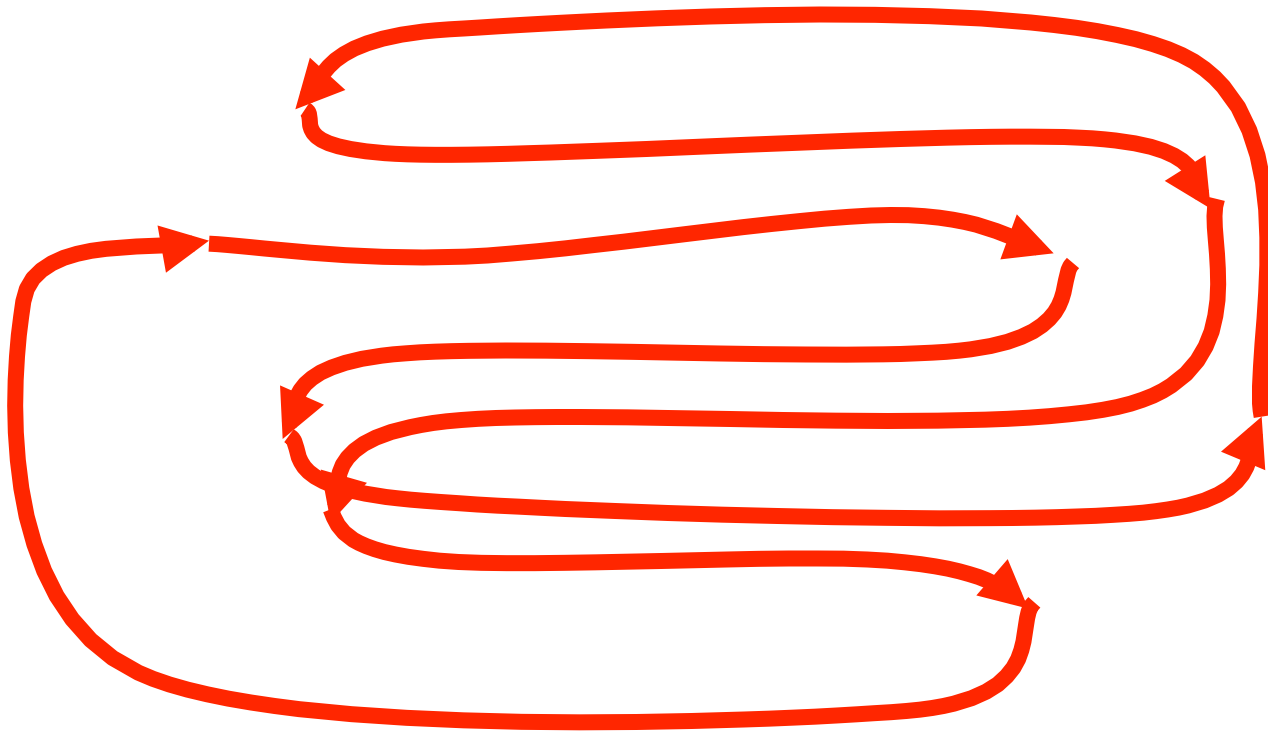
► Assumptions:

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

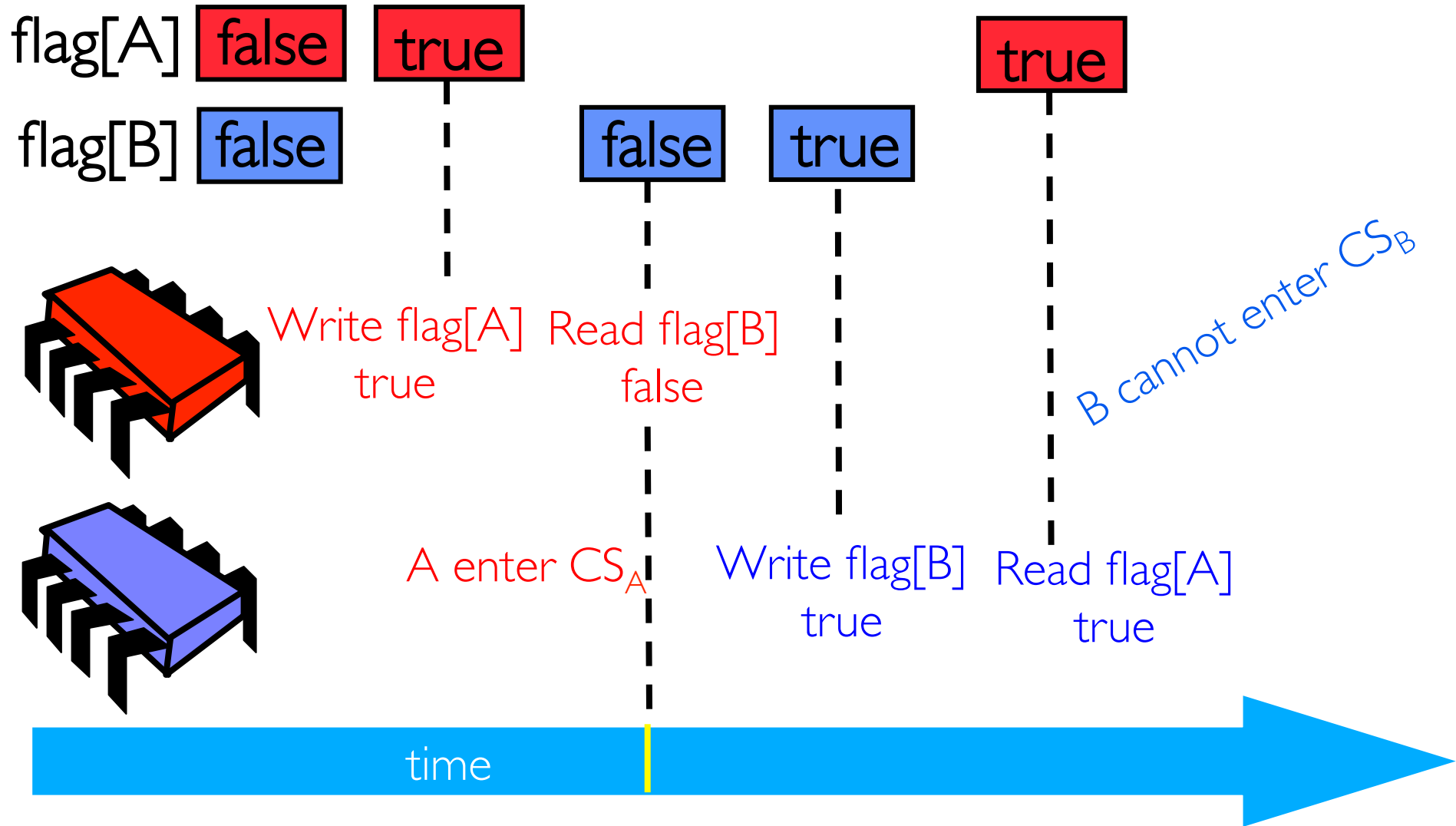
► From the code

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

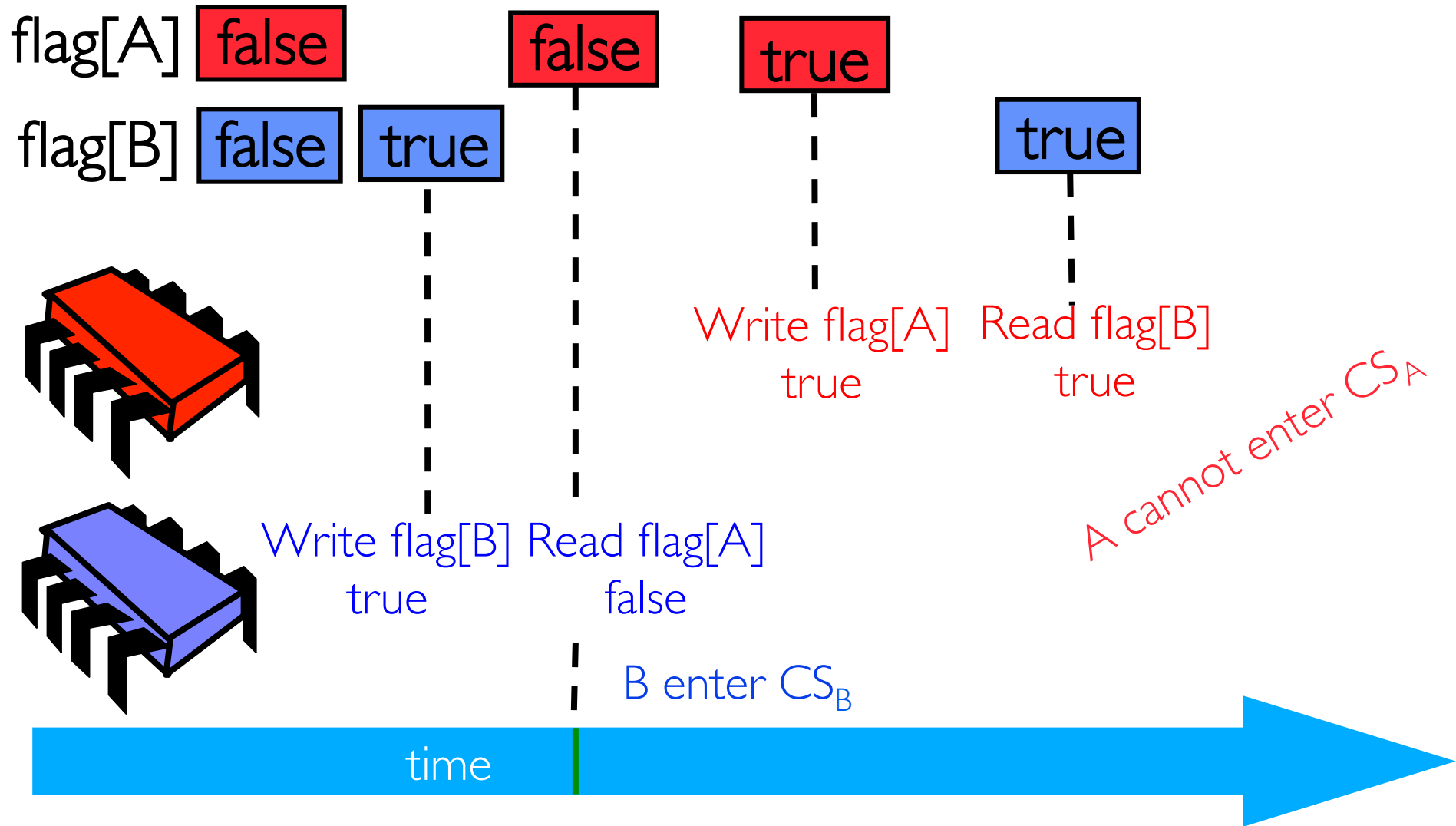
Cycle!



LockOne Satisfies Mutual Exclusion



LockOne Satisfies Mutual Exclusion



Deadlock Freedom

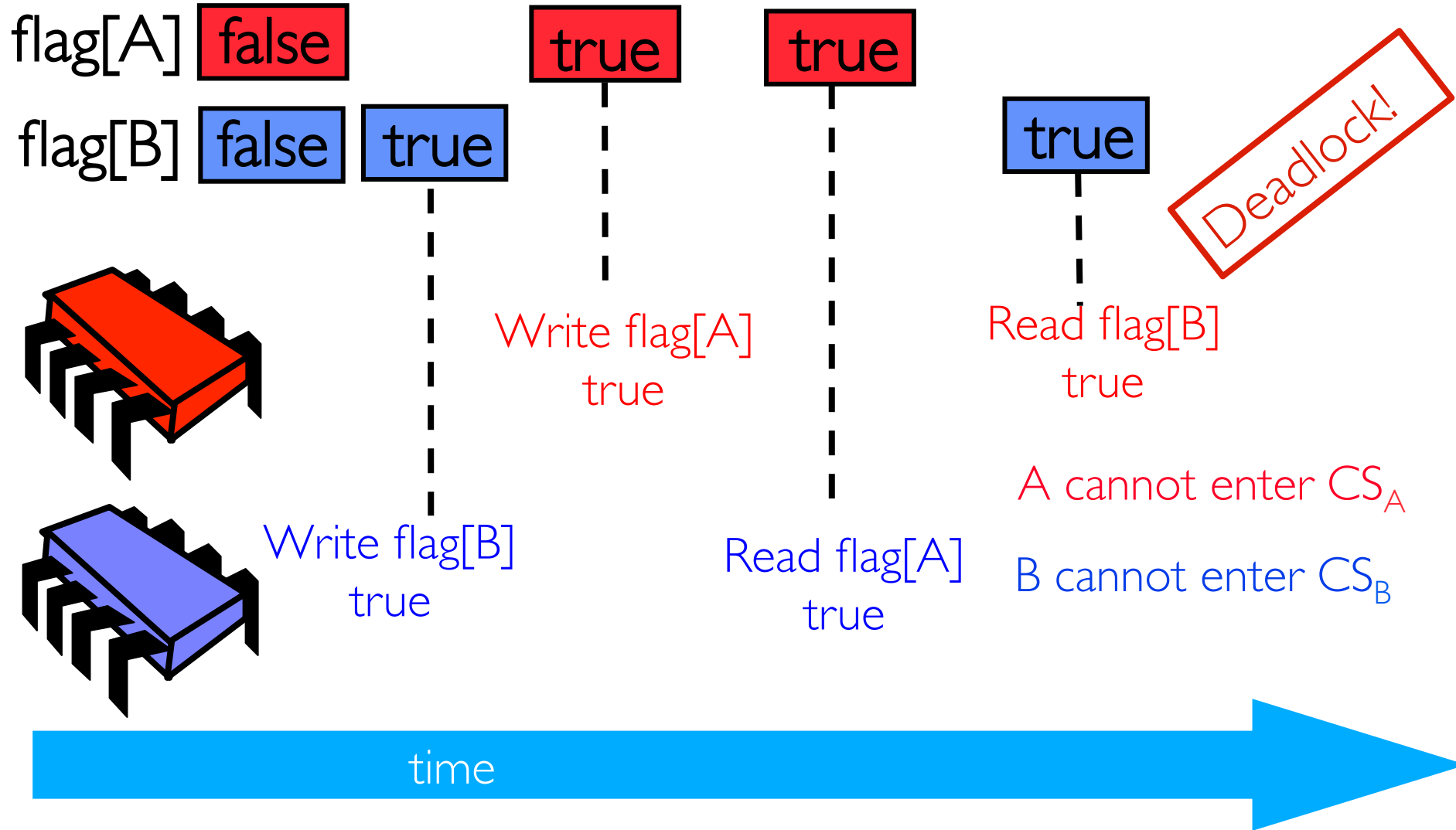
- ▶ LockOne Fails deadlock-freedom

- ▷ Concurrent execution can deadlock

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

- ▷ Sequential executions OK

LockOne May Lead to Deadlock!



LockTwo

```
public class LockTwo implements Lock {
    private int victim;
    public void lock() {
        int i = ThreadID.get();
        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 for
permission



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**
 - ▷ Cannot be both 0 and 1
- ▶ Not deadlock free
 - ▷ Sequential execution deadlocks
 - ▷ Concurrent execution does not

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

Caveats about our model (I):

All variables to spin on must be “volatile”

- ▶ On your laptop, cell phone, server, the “while (flag[j]) {}” would loop non-stop
- ▶ Independent of what threads do
- ▶ Why?

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

Non “volatile” variables are register allocated!

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

Optimizing compilers register allocate flag[j] outside the loop!

```
sw      $t0, 0($t1)    ; flag[i] = true
        lw      $t3, 0($t2) ; read flag[j] into $t3
wait:
        ; while ($t3) {}
        bne     $t3, $zero, wait
```

Making flag[] volatile

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

sw \$t0, 0(\$t1) ; flag[i] = true

lw \$t3, 0(\$t2) ; read flag[j] into \$t3

wait: ; while (\$t3) {}

lw \$t3, 0(\$t2) ; read flag[j] into \$t3

bne \$t3, \$zero, wait

Caveats about our model (2): Events are not instantaneous

- ▶ For now assume events are instantaneous
- ▶ In modern processors
 - ▷ Variable assignments are not!
 - ▷ Loads/stores to memory are overlapped
 - ▷ Are not atomic
 - ▷ There are ISA extensions to help make them atomic
 - ▷ Will learn more about this later + in CS370

Summary

▶ Need

- ▷ Mutual exclusion (make sure you are not in there together)
- ▷ Deadlock freedom (avoid freezing)
- ▷ Starvation freedom (not all protocols satisfy this)

▶ LockOne

- ▷ Implements mutual exclusion
- ▷ Concurrent execution can deadlock

▶ LockTwo

- ▷ Also implements mutual exclusion
- ▷ Concurrent execution cannot deadlock
- ▷ Sequential execution deadlocks