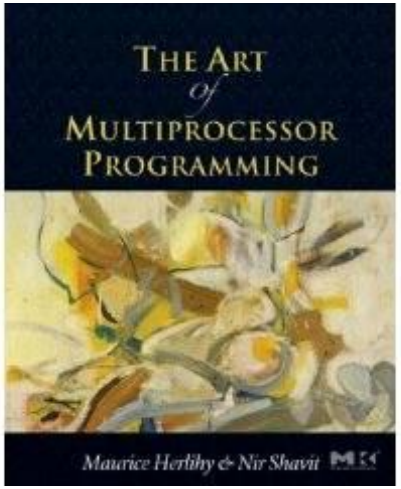
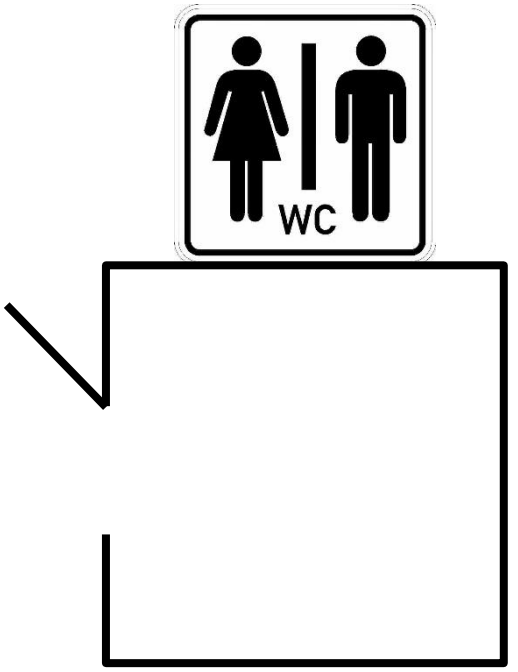
Mutual Exclusion

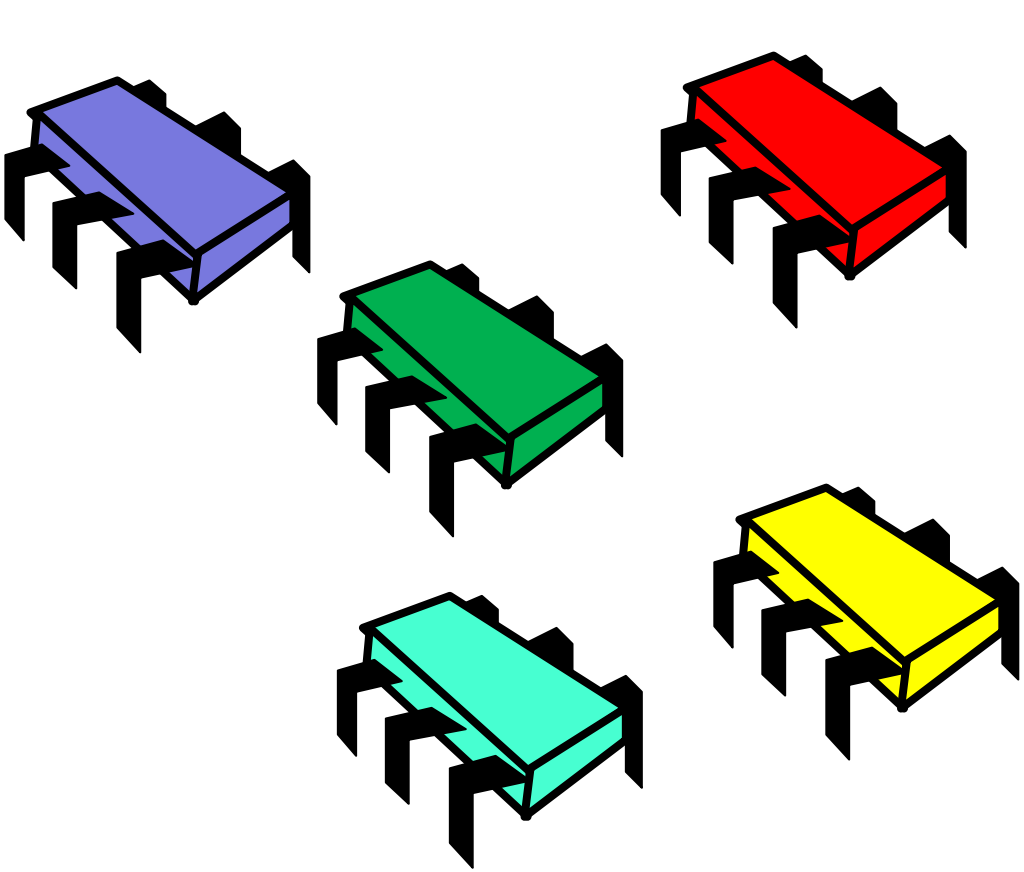


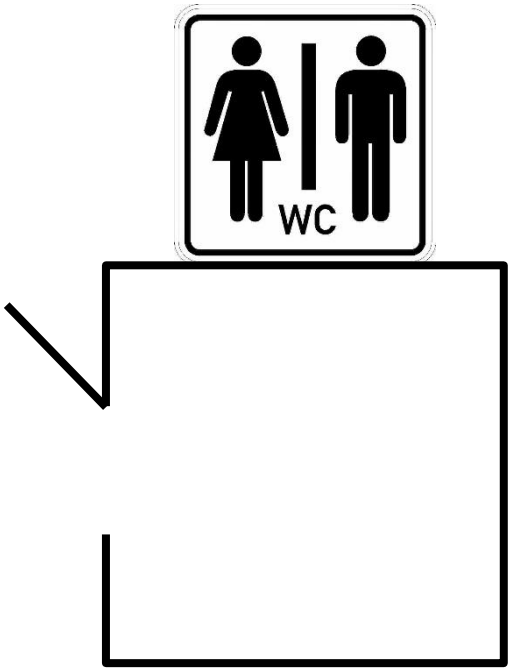
Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

What Is Mutual Exclusion?

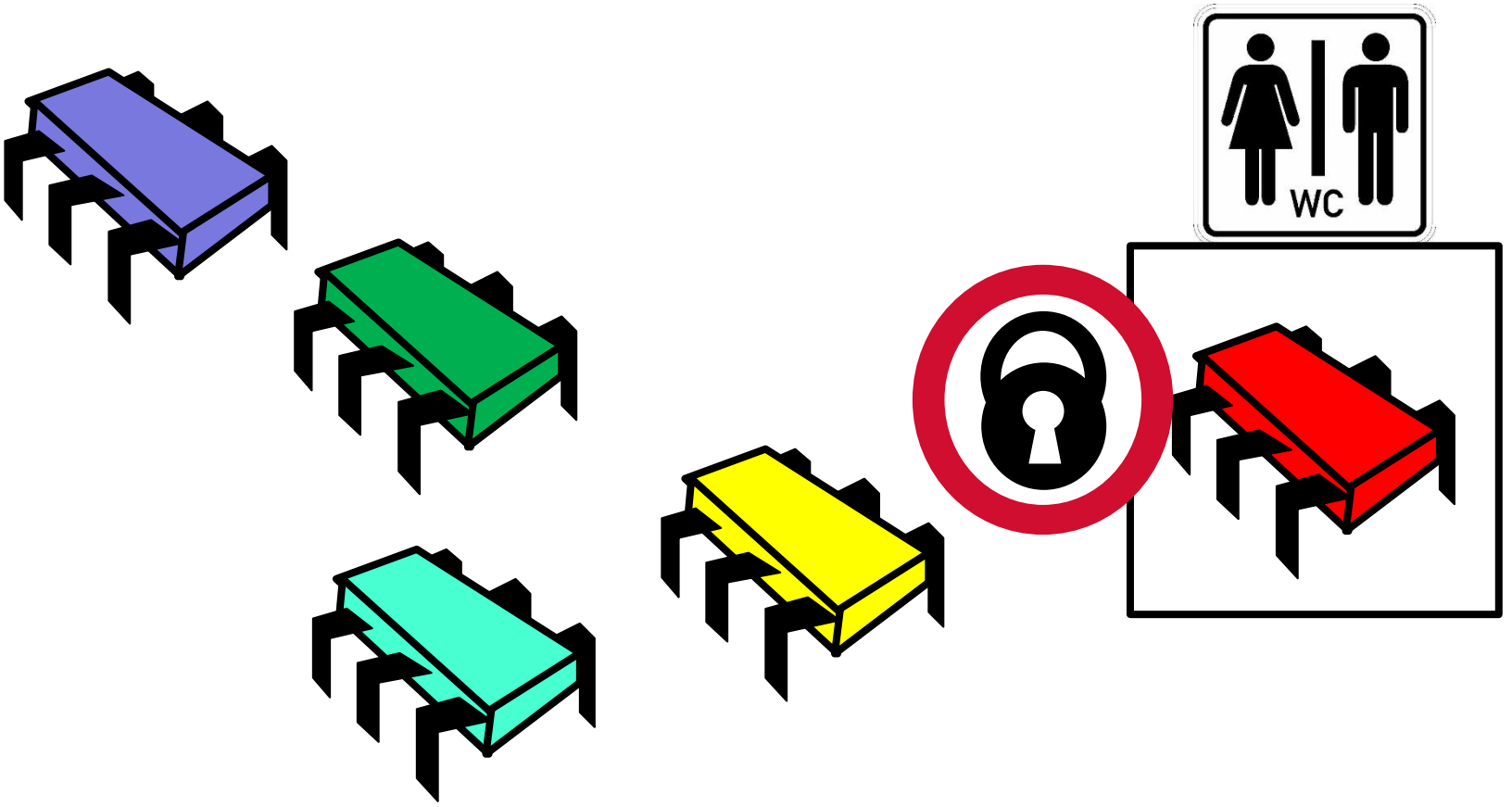


What Is Mutual Exclusion?

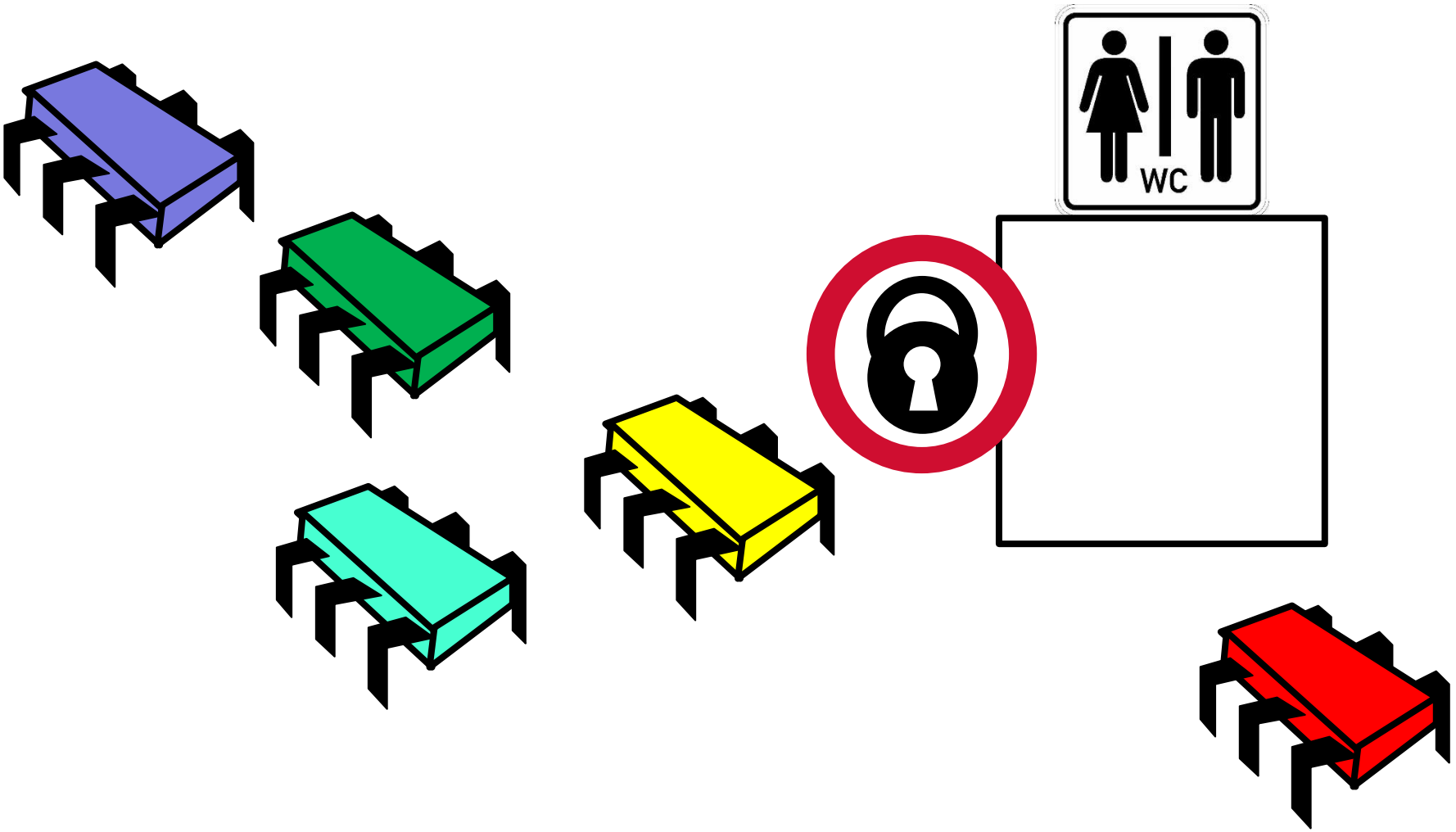




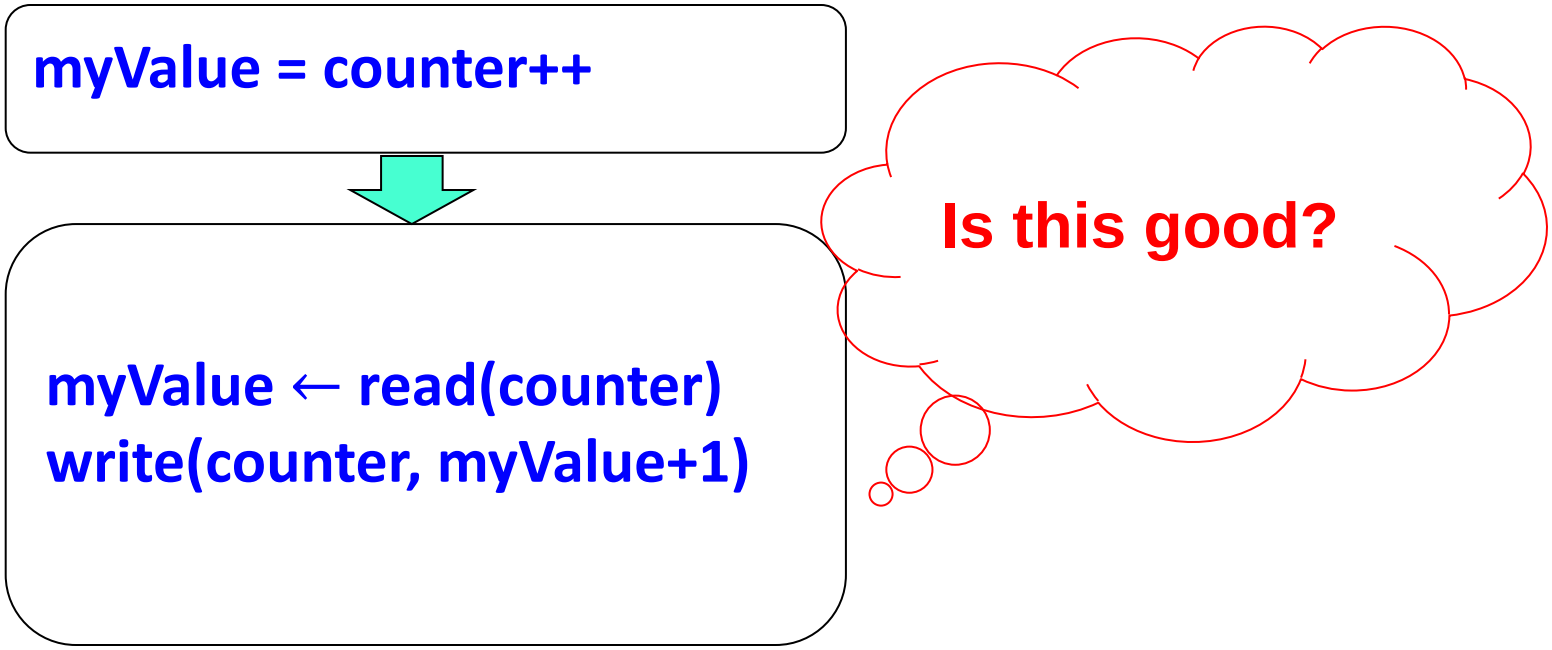
What Is Mutual Exclusion?



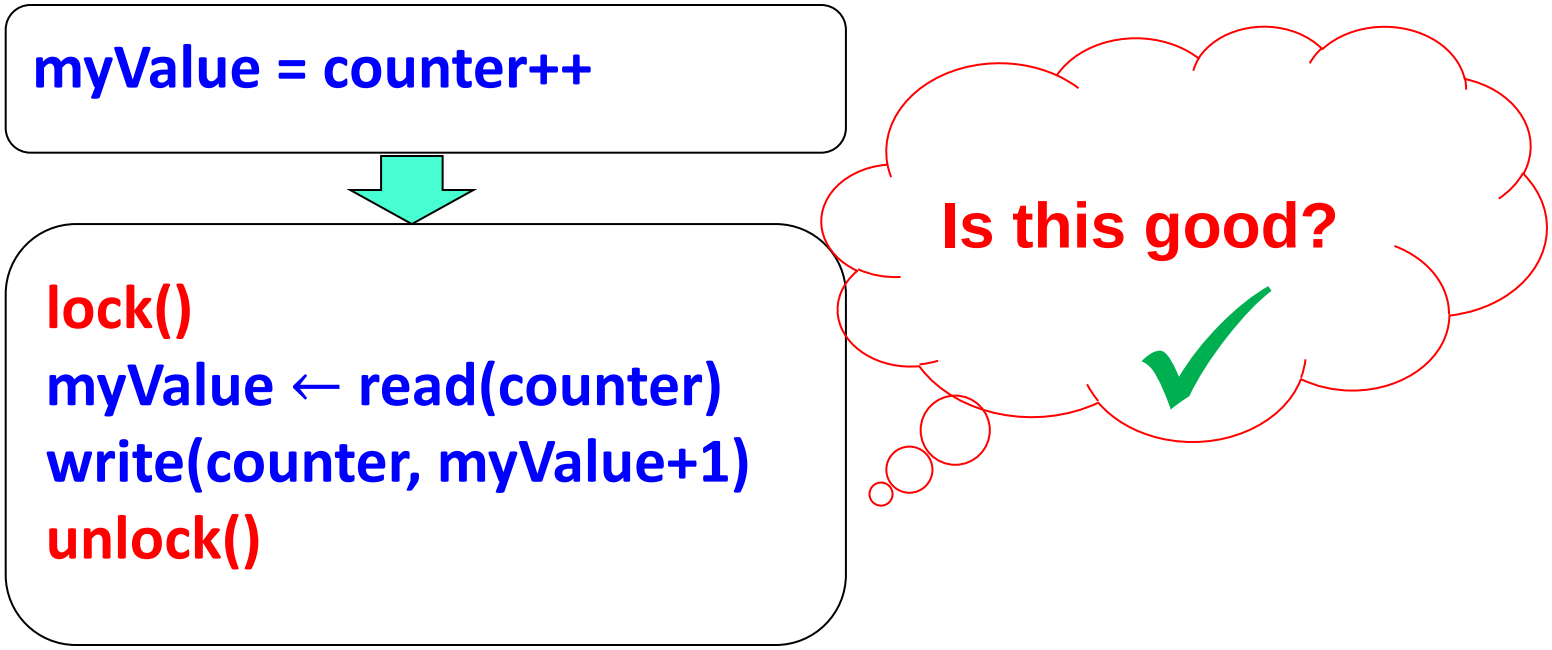
What Is Mutual Exclusion?



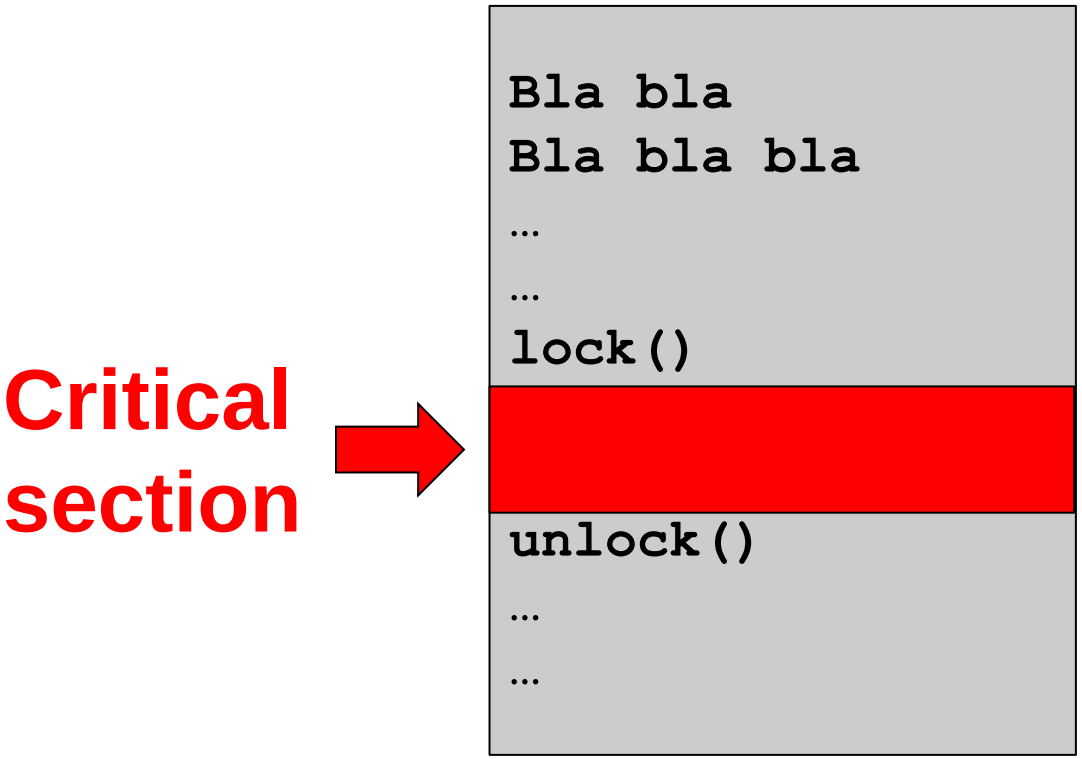
A More Realistic Example



A More Realistic Example

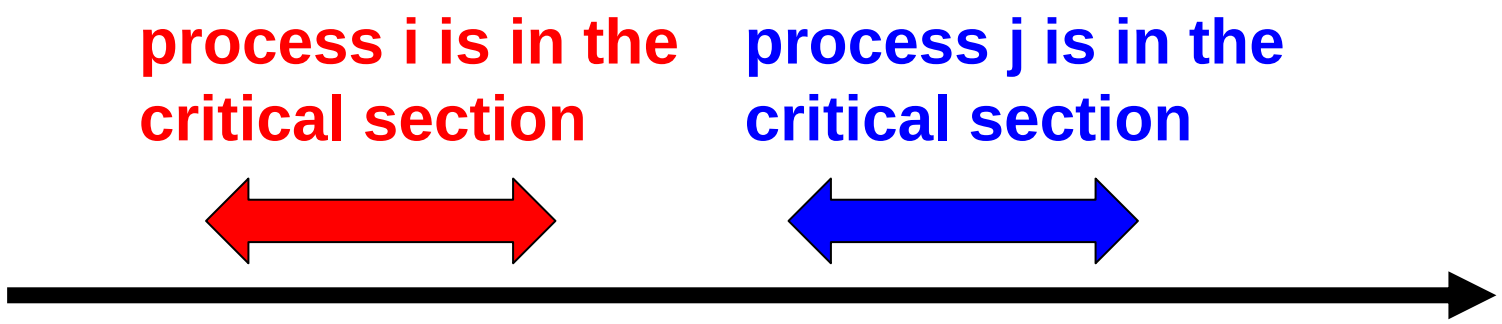


The Critical Section



Requirements

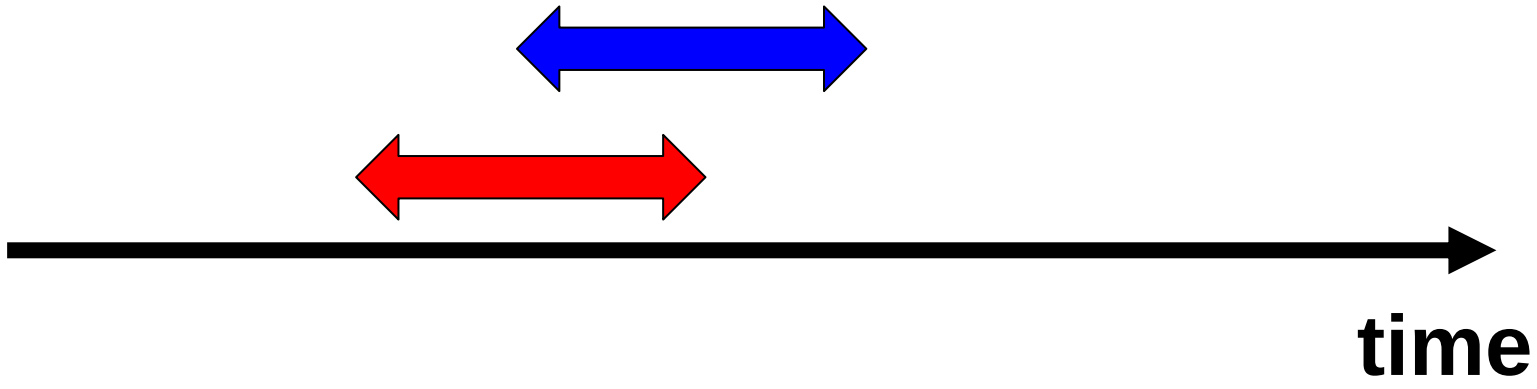
- Mutual exclusion:



- Critical sections may not overlap

Requirements

- Violation of mutual exclusion:



Liveness

- The trivial solution for mutual exclusion:

Deadlock-Free

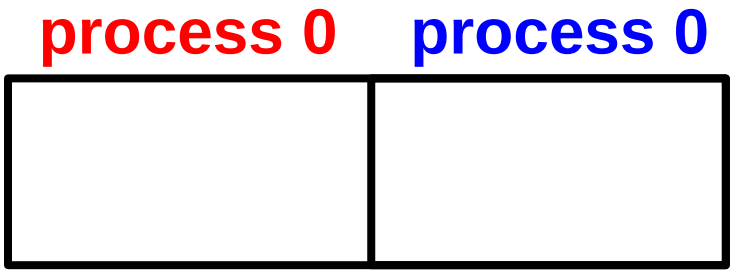
- If some thread calls **`lock()`**
  - And never returns
  - Then other threads must complete **`lock()`** and **`unlock()`** calls infinitely often
- System as a whole makes progress
  - Even if individuals starve

Starvation-Free

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

2-PROCESS MUTUAL EXCLUSION

Attempt 1

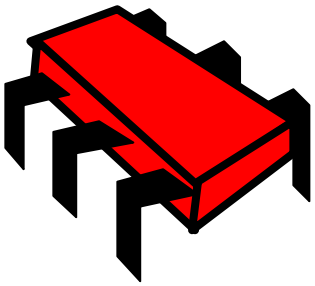


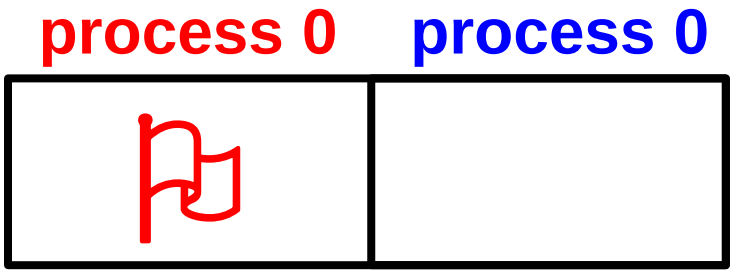
flag[2]

Lock() code for process i:

```
flag[i] = 1;
while(flag[1-i] == 1) ; // wait
```

Attempt 1





flag[2]

```
Lock() code for process i:
flag[i] = 1;
while(flag[1-i] == 1) ; // wait
```

Attempt 1


```
Lock() code for process i:
flag[i] = 1;
while(flag[1-i] == 1) ; // wait
```

Attempt 1


```
Lock() code for process i:
flag[i] = 1;
while(flag[1-i] == 1) ; // wait
Unlock() code for process i:
flag[i] = 0;
```

Attempt 1: Mutual Exclusion?

```
Lock() code for process i:
flag[i] = 1;
while(flag[1-i] == 1) ; // wait
Unlock() code for process i:
flag[i] = 0;
```

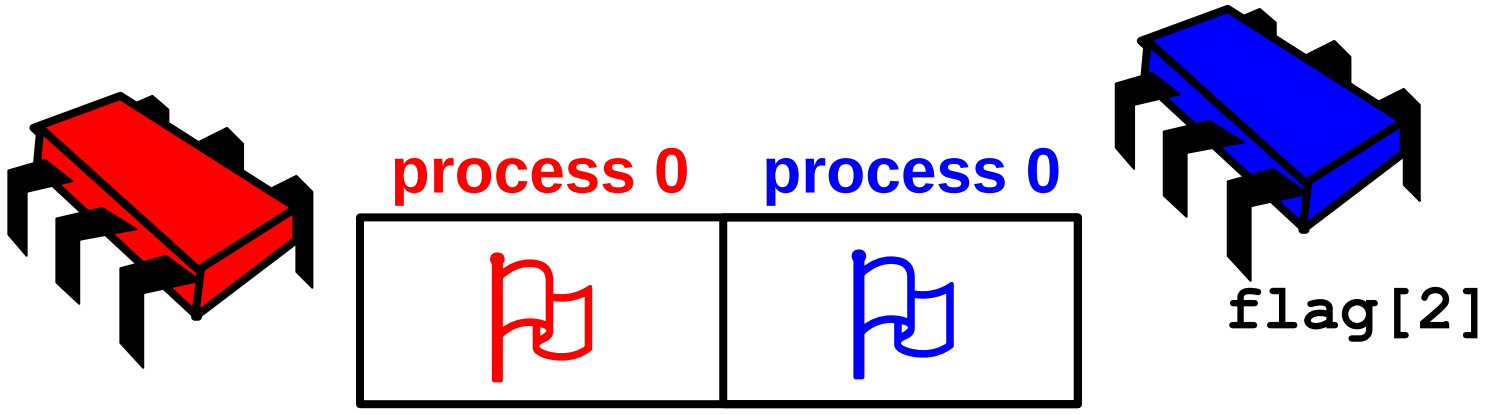
- Yes
- Suppose both are in the critical section…

Attempt 1: Deadlock Freedom?

```
Lock() code for process i:
flag[i] = 1;
while(flag[1-i] == 1) ; // wait
Unlock() code for process i:
flag[i] = 0;
```

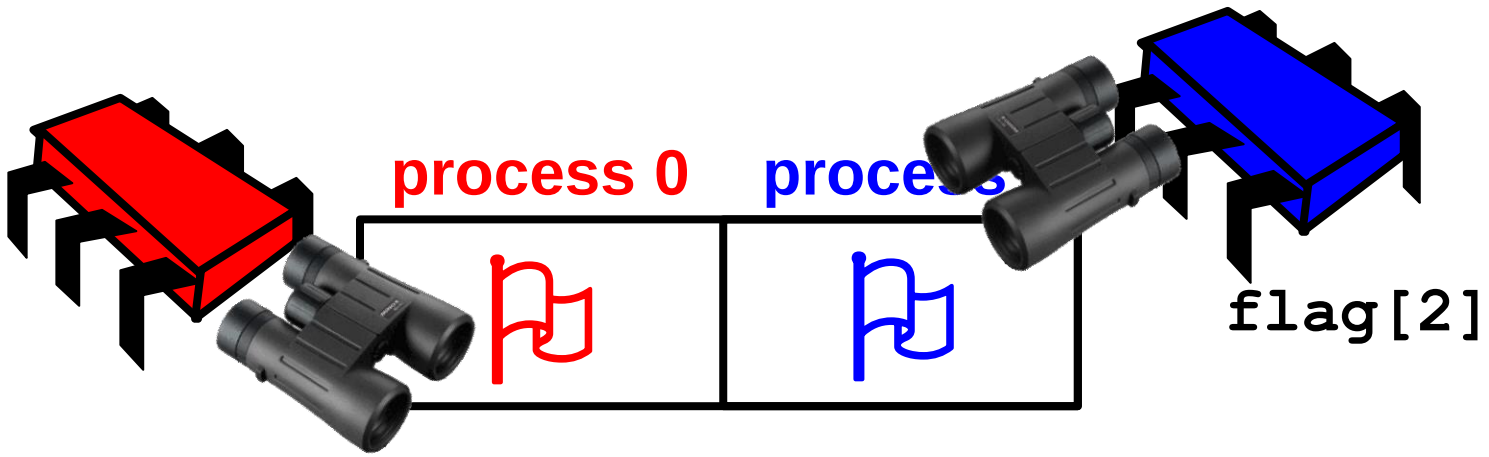
- Nope

Deadlock:




```
Lock() code for process i:
flag[i] = 1;
while(flag[1-i] == 1) ; // wait
```

Deadlock:




```
Lock() code for proces
flag[i] = 1;
while(flag[1-i] == 1)
```

But sequential executions are OK!

Attempt 2

```
code for process i:
      int victim;
      lock()
      {
            victim = i;
            while (victim == i) {} // wait
      }

      unlock() {}
```

Attempt 2

```
code for process i:
      int victim;
      lock()
      {
            victim = i;
            while (victim == i) {} // wait
      }

      unlock() {}
```

Attempt 2

- 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

```
lock() {
  victim = i;
  while (victim == i) {};
 }
```

Peterson's Algorithm

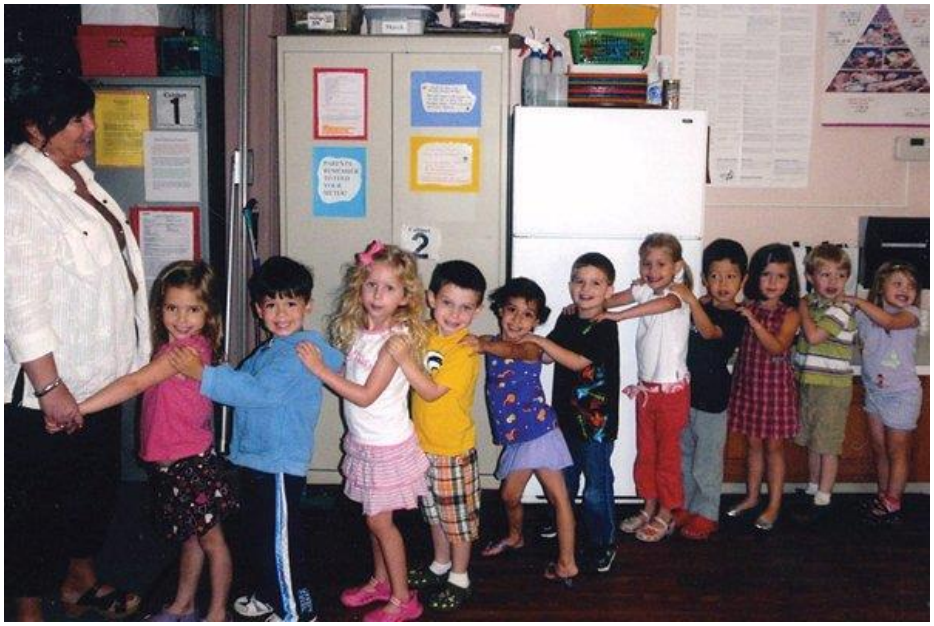
```
lock()
{
      flag[i] = true;
      victim  = i;
      while (flag[j] && victim == i) {};
}
unlock()

      flag[i] = false;
}
```

N-PROCESS MUTEX: LAMPORT'S BAKERY ALGORITHM

Lamport’s Bakery Algorithm

A queue in Israel…

Lamport's Bakery Algorithm



A queue in California

Bakery Algorithm

- Lock():
  1. “Take a number”
  2. Wait until lower numbers are done
- Lexicographic order
  - (a,i) > (b,j)
    - If a > b, or a = b and i > j

Bakery Algorithm

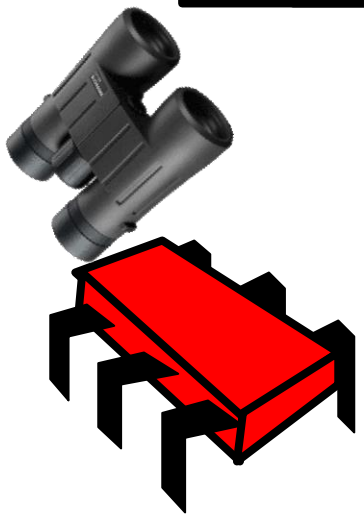
`flag[N]`

`number[N]`

Bakery Algorithm

flag[N]

number[N]



```
max_num = max { number[1],…,number[N] };
number[i] = max_num;
```

Bakery Algorithm

flag[N]

number[N]

```
wait while ∃k:
       flag[k] == 1
        and
       (number[k],k) < (number[i],i)
```

Deep Philosophical Question

- The Bakery Algorithm is
  - Succinct,
  - Elegant, and
  - Fair.
- Q: So why isn't it practical?
- A: N distinct variables!

Theorem

At least N registers are needed to solve deadlock-free mutual exclusion.

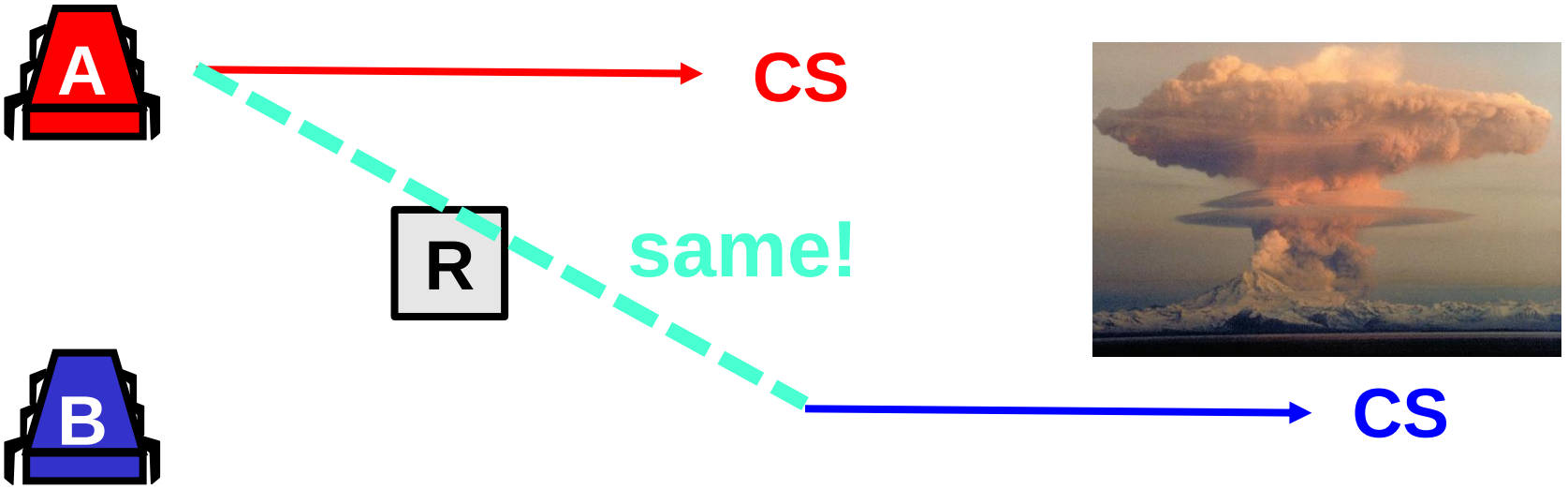
(No matter how big they are!)

Theorem (For 2 Threads)

Theorem: Deadlock-free mutual exclusion for 2 threads requires at least 2 multi-reader multi-writer registers

Proof: assume one register suffices and derive a contradiction

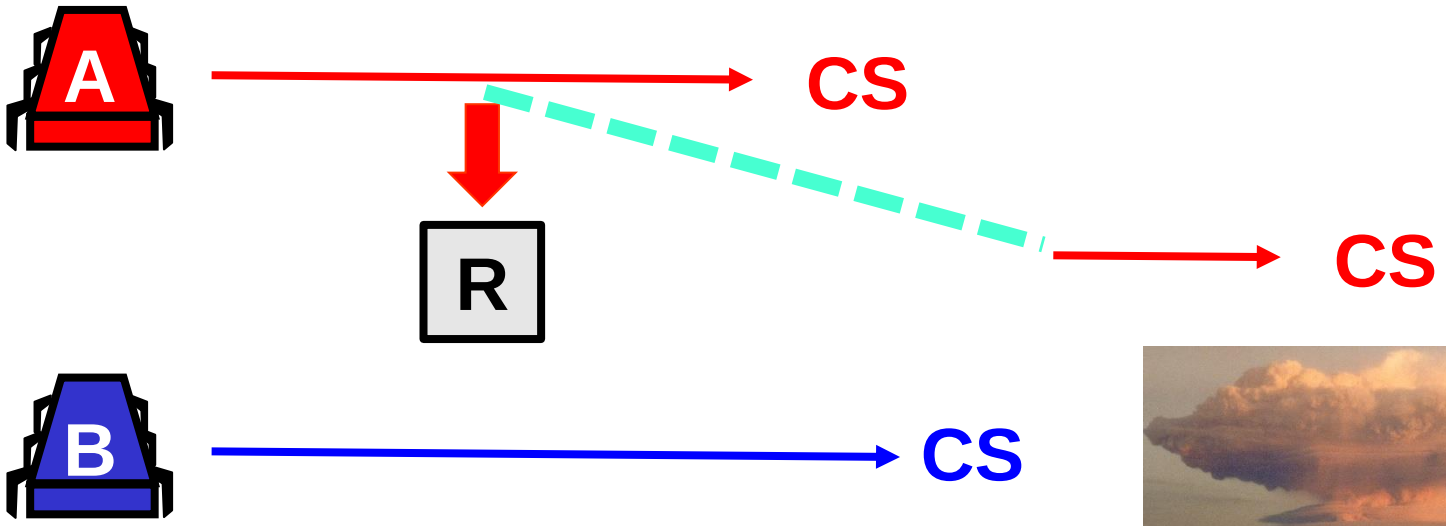
Proof (2 Threads)



Claim: A must write to R

Suppose not…

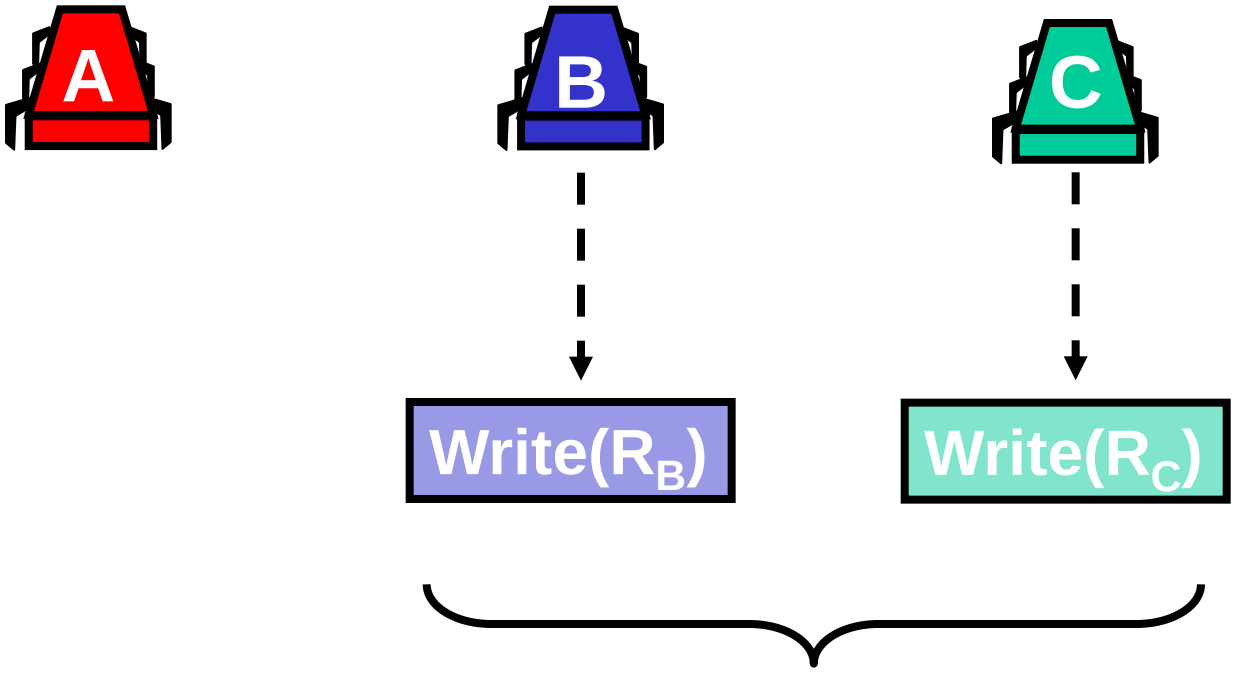
Proof (2 Threads)



Theorem

Deadlock-free mutual exclusion for 3 threads requires at least 3 registers

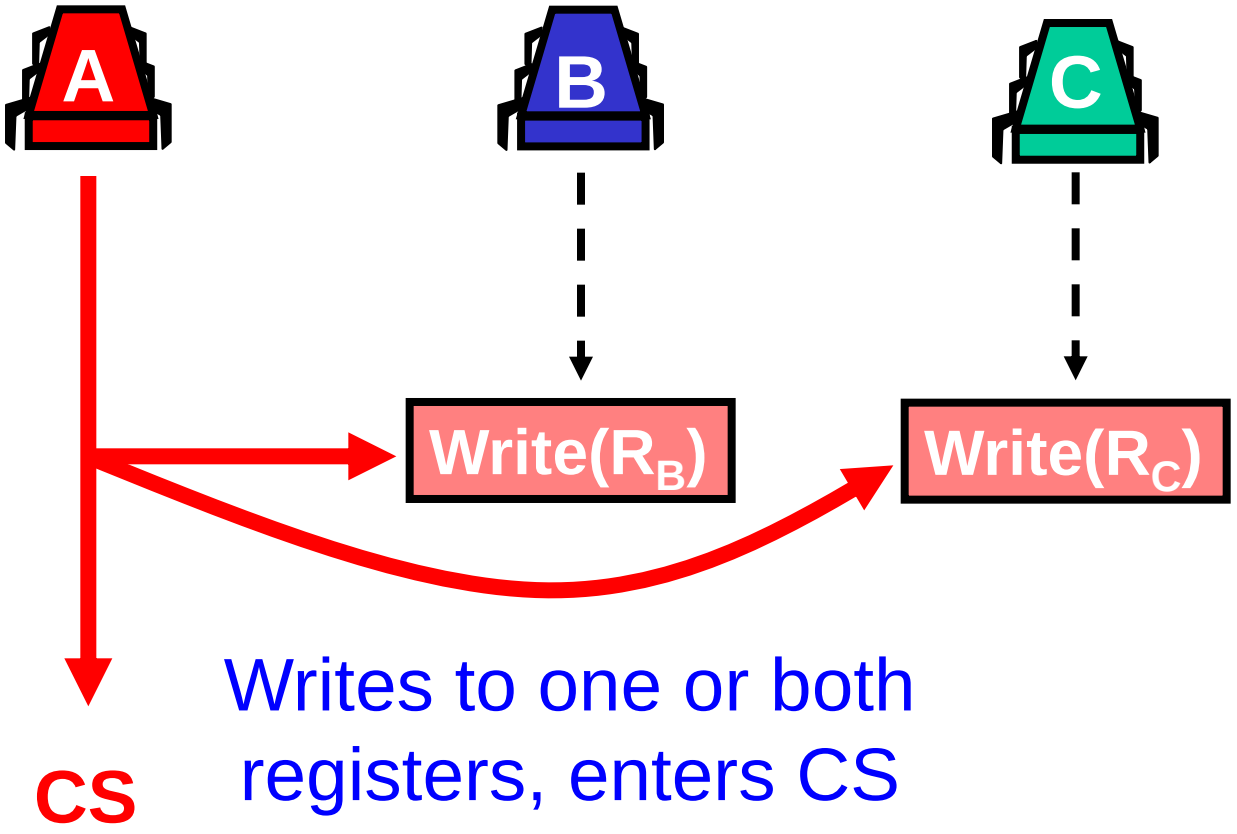
Proof: Assume Cover of 2



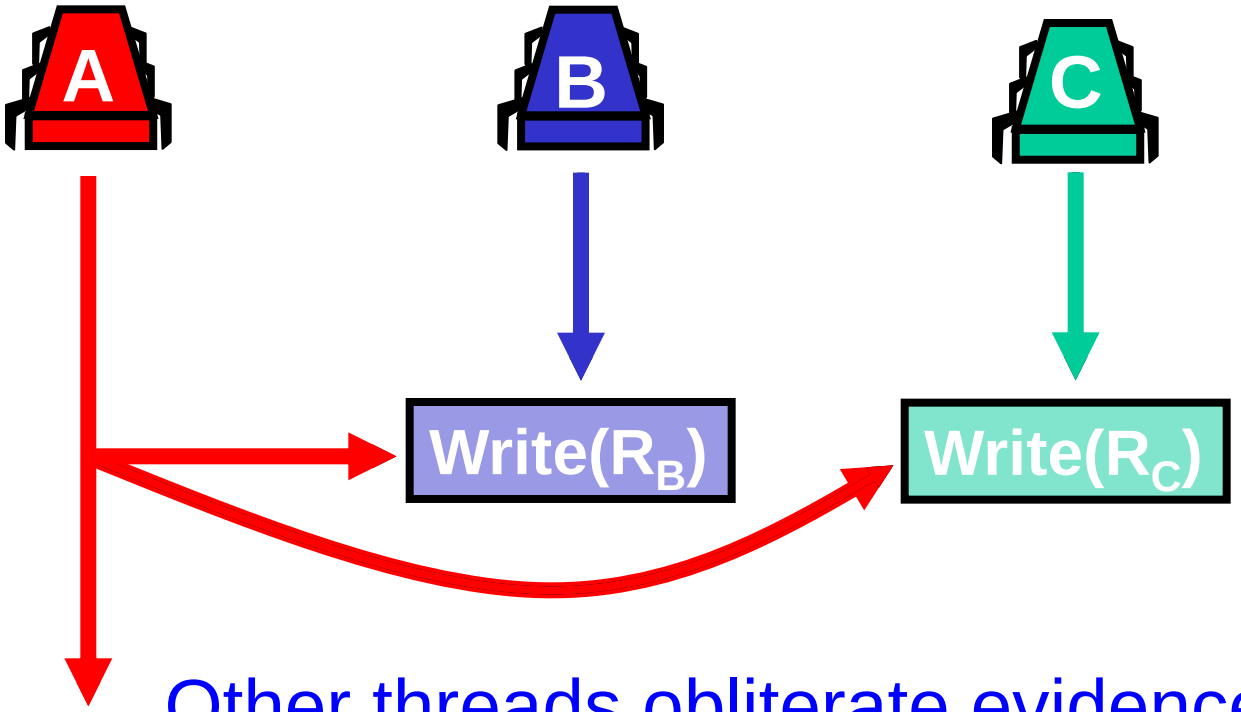
Only 2 registers

Registers look as if CS empty

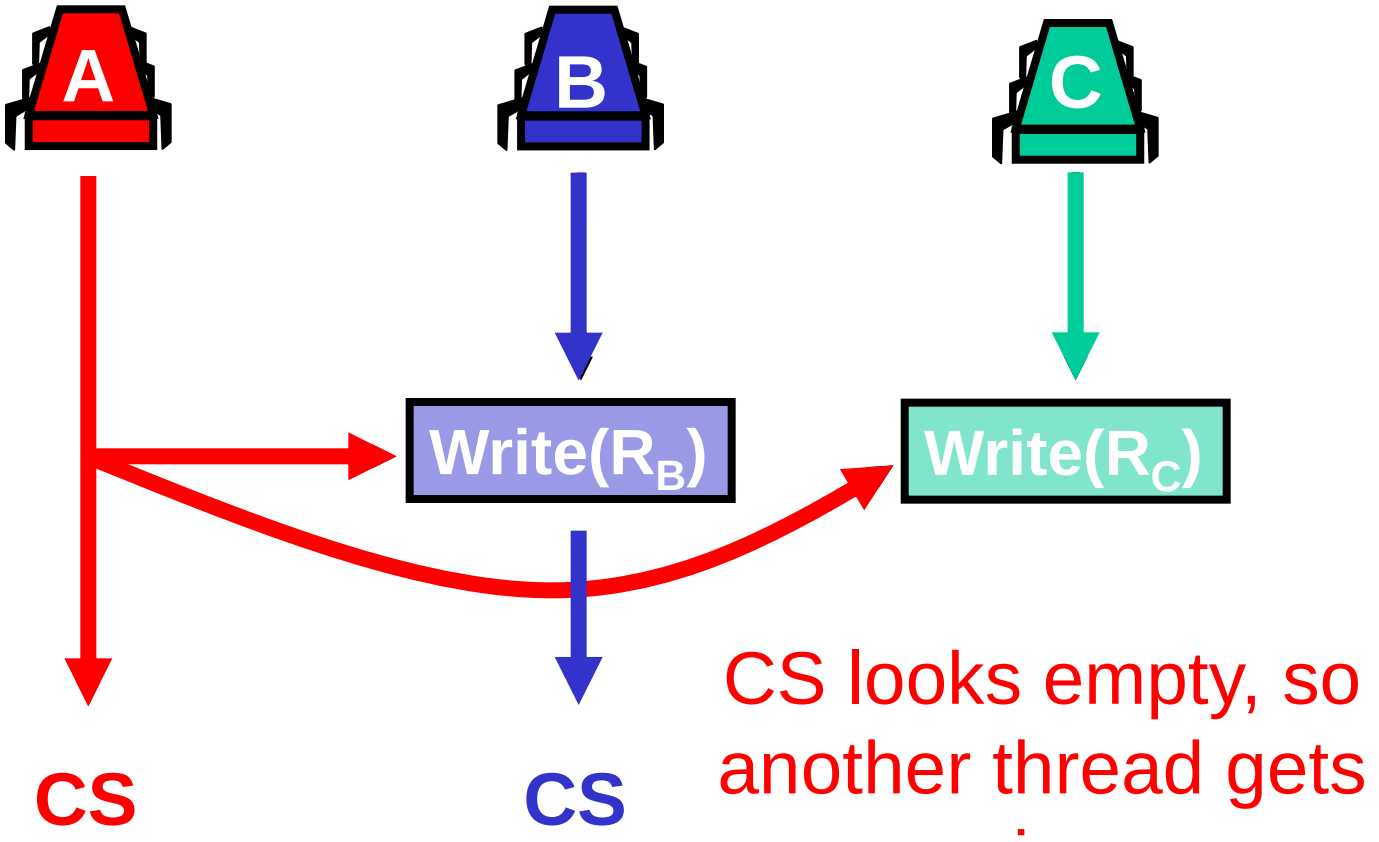
Run A Solo


A
B
C
Write(R_B)
Write(R_C)
CS
Writes to one or both registers, enters CS

Obliterate Traces of A



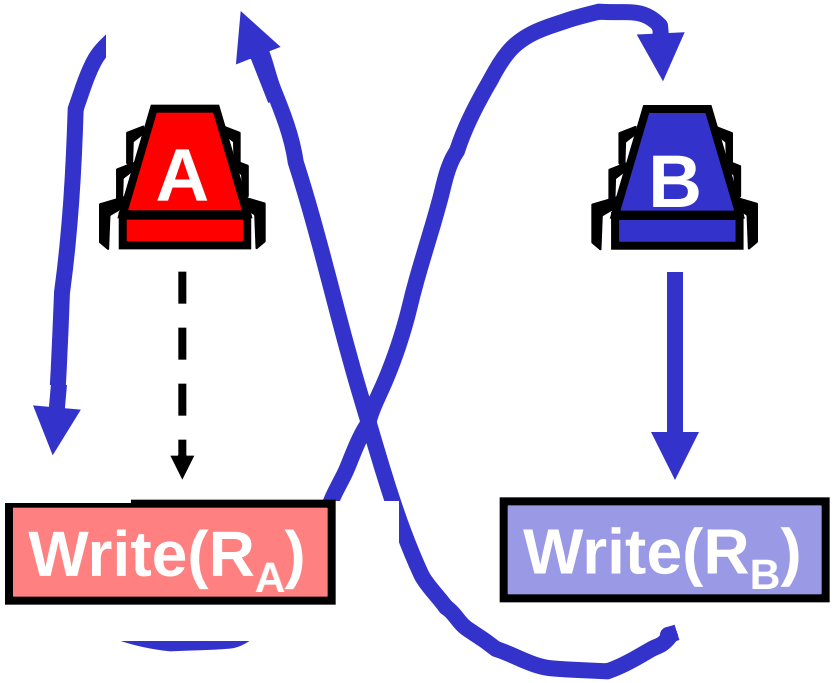
Mutual Exclusion Fails



Proof Strategy

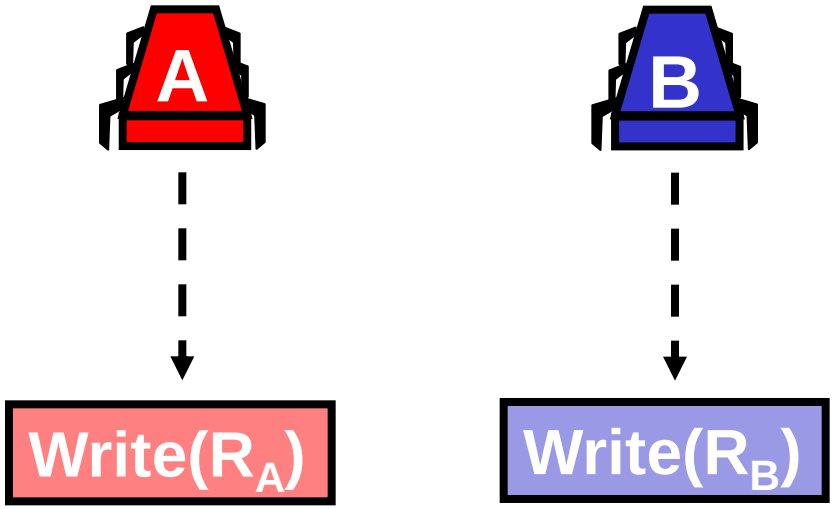
- Proved: a contradiction starting from a covering state for 2 registers
- Claim: can reach a covering state

Covering State for Two



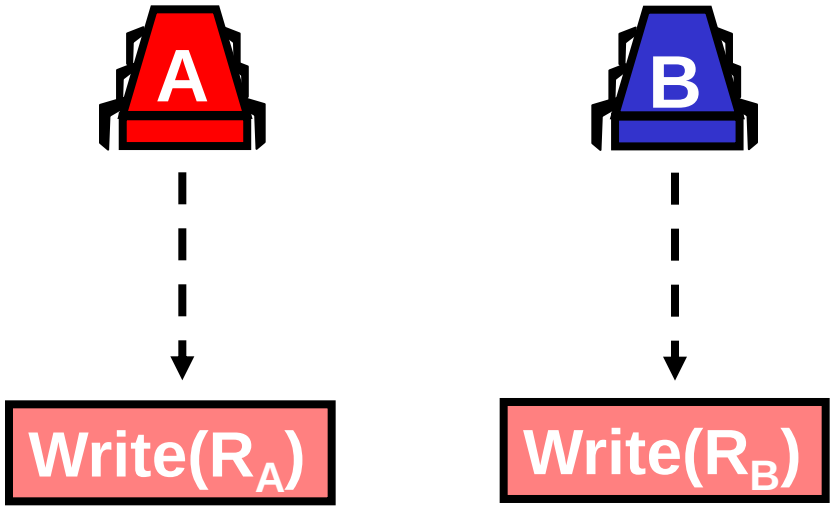
- If we run B through CS 3 times, first write of B must twice be to some register, say R_B

Covering State for Two



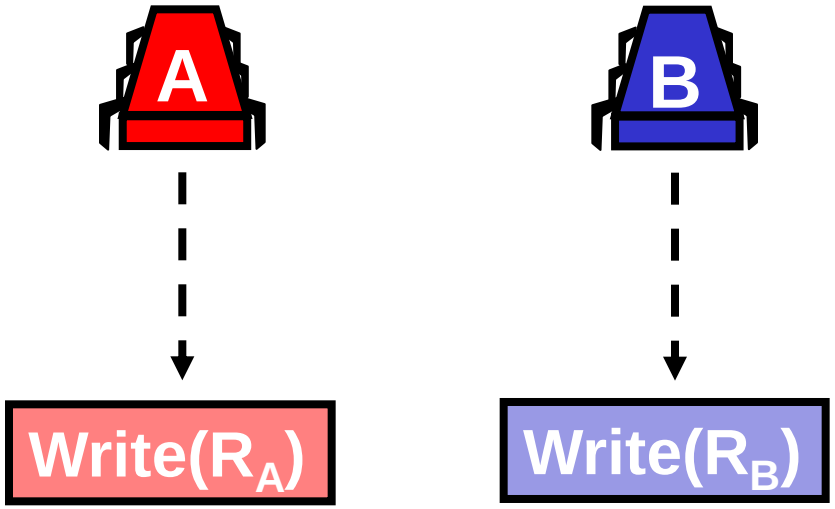
- Start with B covering register R_B for the 1st time
- Run A until it is about to write to uncovered R_A
- Are we done?

Covering State for Two



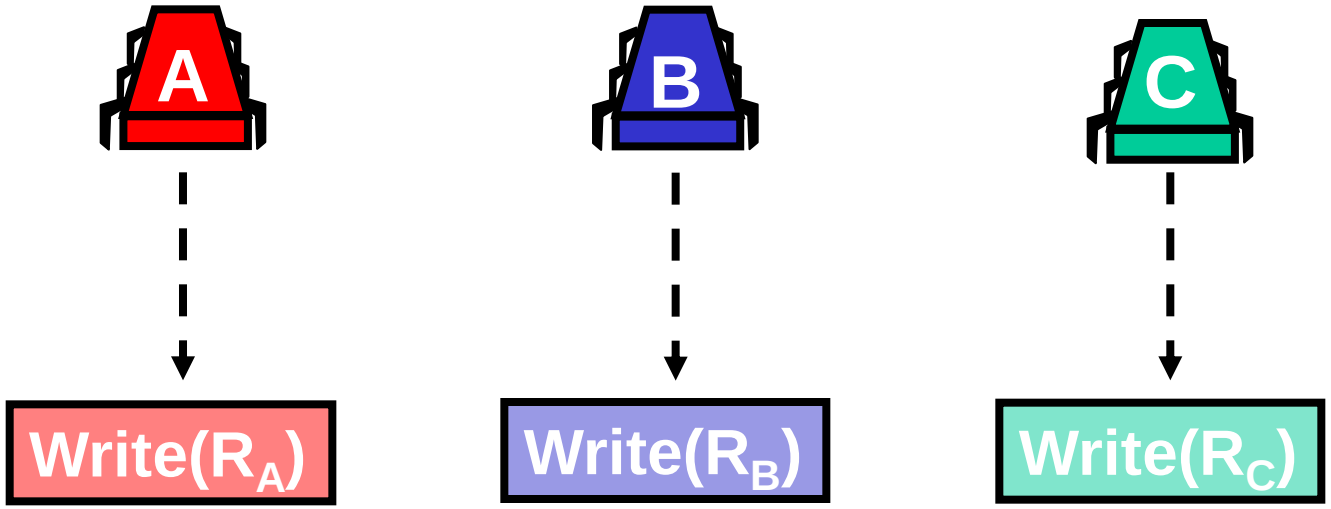
- NO! A could have written to R_B
- So CS no longer looks empty to thread C

Covering State for Two



- Run **B** obliterating traces of **A** in $\mathbf{R_B}$
- Run **B** again until it is about to write to $\mathbf{R_B}$
- Now we are done

Inductively We Can Show



- There is a covering state
  - Where **k** threads not in CS cover **k** distinct registers
  - Proof follows when **k = N-1**

Summary of Lecture

- In the 1960's several incorrect solutions to mutual exclusion using registers were published…
- We saw how to solve N thread mutual exclusion using 2N R/W Registers
- … but N R/W registers are needed

“Cheating” Using Test&Set…

- test&set(x):

  if x == 1 return 0

  if x == 0 set x = 1 and return 1