TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Replication and Consistency

02 Mutual Exclusion

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Winter Term 2019

Thank you!

These slides are based on companion material of the following books:

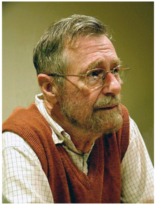
- **The Art of Multiprocessor Programming** by Maurice Herlihy and Nir Shavit
- **Synchronization Algorithms and Concurrent Programming** by Gadi Taubenfeld

Goals of this lecture

- Formalize our understanding of mutual exclusion
- Discuss protocols for 2 threads and extensions for N threads
  - Fairness
  - Inherent costs
- Learn how to argue about and prove various properties in an asynchronous concurrent setting

The History of the Mutual Exclusion Problem

- First solution by Dekker
- Fischer, Knuth, Lynch, Rabin, Rivest, ...
- 1974 Bakery algorithm by Lamport
- 1981 Peterson's algorithm
- Hundreds of published solutions - not all correct!



Quelle: Wikipedia

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.*

Warning!

- You will never use these protocols!
  - Get over it ...
- You are advised to understand them
  - The same issues show up everywhere
  - Except they will be hidden and more complex

Preliminaries

Time

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

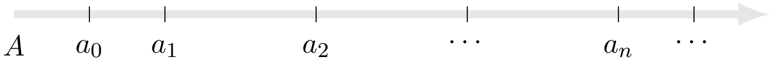
Time is, like, Nature's way of making sure that everything doesn't happen all at once." (Anonymous, circa 1968)

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Formal system model

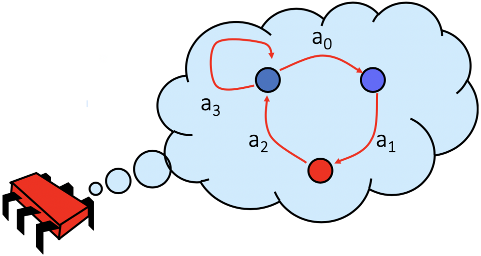
Definition

A **thread** A is a sequence $a_0, a_1, \ldots$ of events.



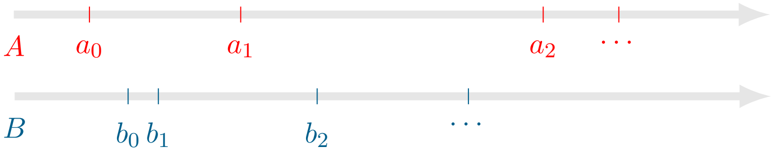
- "Trace" model
- An event a_0 of thread A is
  - instantaneous
  - at a unique point in time (no simultaneous events!)
- *Notation:* $a_0 \rightarrow a_1$ indicates order

Threads are State Machines

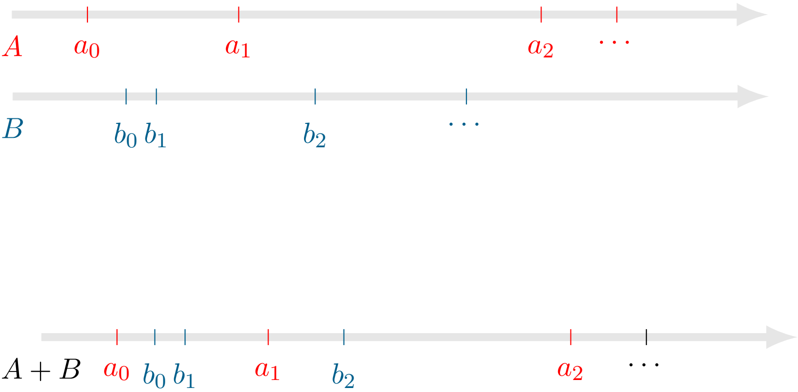


- Thread State: Program counter + local variables
- System state: Thread states + shared variables
- Events are state transitions
  - Assign value to shared variable
  - Assign value to local variable
  - Read value from shared/local variable
  - Invoke method
  - Return from method etc.

Modelling Concurrency via Interleaving



Modelling Concurrency via Interleaving



Intervals

An interval $A_0 = (a_0, a_1)$ is the time between events a_0 and a_1.

Intervals

An interval $A_0 = (a_0, a_1)$ is the time between events a_0 and a_1.

Task

Give definitions and examples of

- Overlapping intervals
- Disjoint intervals

Precedence

Definition

Interval A_i **precedes** (**happens before**) interval B_j ($A_i \rightarrow B_j$) if end event of A_i is before start event of B_j.

Question

Precedence defines a *partial order* on intervals

- Irreflexive: Never true that $A_i \rightarrow A_i$
- Antisymmetric: If $A_i \rightarrow B_j$, then not true that $B_j \rightarrow A_i$
- Transitive: If $A_i \rightarrow B_j$ and $B_j \rightarrow C_k$, then $A_i \rightarrow C_k$

Why is precedence not a total order?

Repeated Events

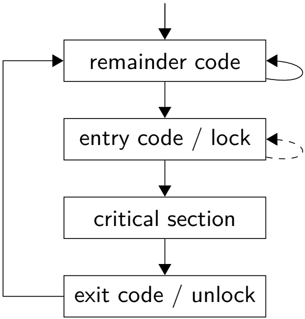
```
while (...) {
  a0; a1;
}
```

- a_0^k denotes k-th occurrence of event a_0, etc.
- A_0^k denotes k-th occurrence of interval A_0

Mutual exclusion

The problem

- Want to guarantee mutually exclusive access to some shared resource for several competing processes
- Avoid **race conditions**, i.e. flaws that occur when the timing or ordering of events affects a program's correctness



Formal properties

Mutual Exclusion

Critical sections of different threads do not overlap.
For threads A and B and integers j and k, either $CS_A^k \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^k$.

Deadlock Freedom

If some thread is trying to enter its critical section, then **some** thread (not necessarily the same one!) eventually enters its critical section.

Starvation Freedom

If a thread is trying to enter its critical section, then **this** thread must eventually enter its critical section.

Question

Which statement is correct?

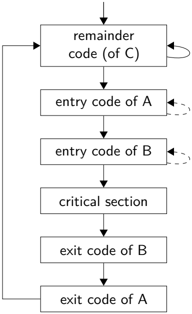
- Deadlock freedom implies starvation freedom.
- Starvation freedom implies deadlock freedom.

Assumptions

- The remainder code does not influence the behavior of other threads.
- Shared objects used in entry/exit code may not be referred to in remainder code or critical section.
- Process cannot fail (i.e. stop) while in entry code, critical section or exit code.
- Process executes critical section and exit code in a finite number of steps.

Question

The following control flow graph sketches some algorithm C that employs algorithms A and B.



1. If both A and B are deadlock-free, then C is deadlock-free.
2. If both A and B are starvation-free, then C is starvation-free.
3. If either A or B satisfy mutual exclusion, then C satisfy mutual exclusion.
4. If A is deadlock-free and B is starvation-free, then C is starvation-free.

Protocols for Mutual Exclusion

Two-Thread vs n-Thread Solutions

- First: Two-thread solutions
  - Illustrate most basic ideas
  - Algorithms fit on one slide
- Then: n-Thread solutions

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Protocol LockOne

Initially: `turn = 0`

Thread 0:

```
while (true) {
   remainder code
   turn = 0
   while (turn == 1)
      {skip;}
   critical section
}
```

Thread 1:

```
while (true) {
   remainder code
   turn = 1
   while (turn == 0)
      {skip;}
   critical section
}
```

Question

Does it solve the mutual exclusion problem?

Protocol LockOne

Initially: `turn = 0`

Thread 0:

```
while (true) {
   remainder code
   turn = 0
   while (turn == 1)
      {skip;}
   critical section
}
```

Thread 1:

```
while (true) {
   remainder code
   turn = 1
   while (turn == 0)
      {skip;}
   critical section
}
```

Question

Does it solve the mutual exclusion problem?

Under sequential execution, threads cannot proceed.

$\Rightarrow$ Mutual exclusion, but not deadlock-freedom

Convention

Initially: ...

Thread 0:

```
while (true) {
   remainder code
   entry code
   critical section
   exit code
}
```

Thread 1:

```
while (true) {
   remainder code
   entry code
   critical section
   exit code
}
```

Convention

Initially: ...

Thread 0:

```
entry code
critical section
exit code
```

Thread 1:

```
entry code
critical section
exit code
```

Protocol LockTwo

Initially: `flag[0] = flag[1] = false`

Thread 0:

```
while (true) {
   remainder code
   flag[0] = true
   while (flag[1]) {skip;}
   critical section
   flag[0] = false
}
```

Thread 1:

```
while (true) {
   remainder code
   flag[1] = true
   while (flag[0]) {skip;}
   critical section
   flag[1] = false
}
```

Question

Does it solve the mutual exclusion problem?

Protocol LockTwo

Initially: `flag[0] = flag[1] = false`

Thread 0:

```
while (true) {
   remainder code
   flag[0] = true
   while (flag[1]) {skip;}
   critical section
   flag[0] = false
}
```

Thread 1:

```
while (true) {
   remainder code
   flag[1] = true
   while (flag[0]) {skip;}
   critical section
   flag[1] = false
}
```

Question

Does it solve the mutual exclusion problem?

If each thread sets its flag to `true` and waits for the other, they will wait forever.

$\Rightarrow$ Mutual exclusion, but not deadlock-freedom

Protocol LockThree

Initially: `flag[0] = flag[1] = false`

Thread 0:

```
while (flag[1]) {skip;}
flag[0] = true
critical section
flag[0] = false
```

Thread 1:

```
while (flag[0]) {skip;}
flag[1] = true
critical section
flag[1] = false
```

Question

Does it solve the mutual exclusion problem?

Protocol LockThree

Initially: `flag[0] = flag[1] = false`

Thread 0:

```
while (flag[1]) {skip;}
flag[0] = true
critical section
flag[0] = false
```

Thread 1:

```
while (flag[0]) {skip;}
flag[1] = true
critical section
flag[1] = false
```

Question

Does it solve the mutual exclusion problem?
If each thread pass the while-loop at the same time and set their flag to `true`, they both enter the critical section.
$\Rightarrow$ Deadlock-freedom, but no mutual-exclusion

Peterson's Algorithm

Initially: `flag[0] = flag[1] = false, turn = 0`

Thread 0:

```
flag[0] = true
turn = 1

while (flag[1] && turn == 1) {
   skip;
}
critical section

flag[0] = false
```

Thread 1:

```
flag[1] = true
turn = 0

while (flag[0] && turn == 0) {
   skip;
}
critical section
flag[1] = false
```

In detail

```
// Announce interest
flag[i] = true

// Defer to the other
turn = j

// Wait while other is interested and not my turn
while (flag[j] && turn == j) {skip;}

critical section

// no longer interested
flag[i] = false
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

In detail

```
// Announce interest
flag[i] = true

// Defer to the other
turn = j

// Wait while other is interested and not my turn
while (flag[j] && turn == j) {skip;}

critical section

// no longer interested
flag[i] = false
```

Does it matter if we replace the order of line 1 and 2?

In detail

```
// Announce interest
flag[i] = true

// Defer to the other
turn = j

// Wait while other is interested and not my turn
while (flag[j] && turn == j) {skip;}

critical section

// no longer interested
flag[i] = false
```

Does it matter if we replace the order of line 1 and 2?

Does not satisfy mutual exclusion anymore!

Proof Idea: Mutual Exclusion

- If thread 0 in critical section: `flag[0] = true, turn = 0`
- If thread 1 in critical section: `flag[1] = true, turn = 1`

Proof Idea: Mutual Exclusion

- If thread 0 in critical section: `flag[0] = true, turn = 0`
- If thread 1 in critical section: `flag[1] = true, turn = 1`

$\Rightarrow$ Cannot both be true

Proof Idea: Deadlock Freedom

In entry code for thread `j`:

```
while (flag[i] && turn == i) {};
```

Thread blocked

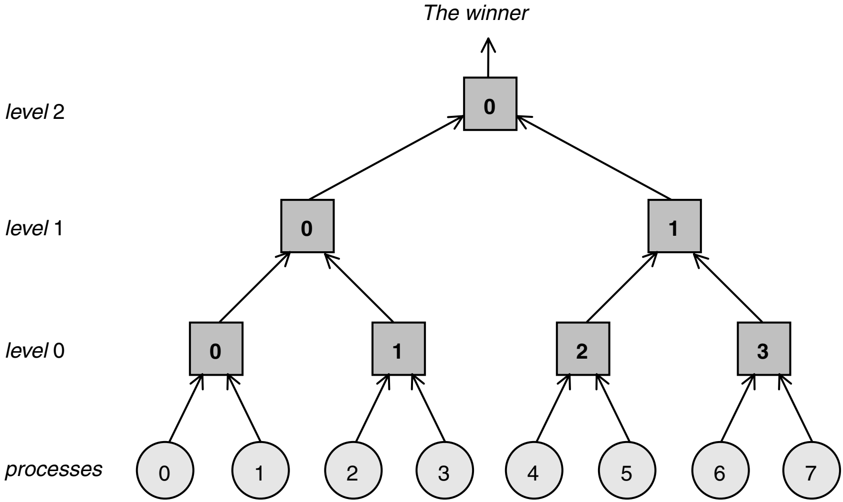
- only at while loop
- only if it is not its turn

$\Rightarrow$ Only one thread will have its value in turn!

Proof Idea: Starvation Freedom

- Thread `i` blocked only if `j` repeatedly re-enters so that `flag[j] && turn == j`
- When thread `j` re-enters (i.e. calls again the entry code), it sets `turn` to `i`.
- Therefore, `i` eventually gets in.

Extension for N-Threads: Tournament Algorithms



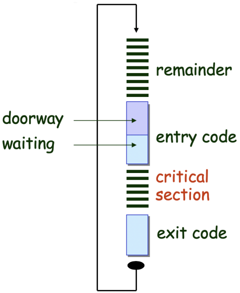
Properties

For Tournament Algorithm based on Peterson's Algorithm:

- Satisfies mutual exclusion and starvation freedom
- Contention-free time complexity is $4 \log n$ accesses to shared memory
- Uses $3(n-1)$ shared registers, three for each node ($=$ lock)
- One process can enter its critical section arbitrarily many times ahead of another slower process from a different subtree

$\Rightarrow$ Want stronger fairness guarantee!

Bounded Waiting



Divide entry code into two parts:

- Doorway interval D_A
    - Always finished in finite steps
- Waiting interval W_A
    - May take unbounded number of steps

r-Bounded Waiting

For threads A and B:

- If $D_A^k \rightarrow D_B^j$, then $CS_A^k \rightarrow CS_B^{j+r}$
- B cannot overtake A by more than r times
- First-come-first-served (FIFO) means $r = 0$

r-Bounded Waiting

For threads A and B:

- If $D_A^k \to D_B^j$, then $CS_A^k \to CS_B^{j+r}$
- B cannot overtake A by more than r times
- First-come-first-served (FIFO) means $r = 0$

For Tournament Algorithm from before:

- No one starves
- But very weak fairness: Not r-bounded for any r!
- That is pretty lame…

Bakery Algorithm

- Provides FIFO
- Idea:
  - Take a number
  - Wait until lower numbers have been served
- For symmetry breaking, we use lexicographic order on tuples:

$$(a, i) < (b, j) \text{ if } a < b \text{ or } a = b \text{ and } i < j$$

Bakery Algorithm

Initially: For all i = 1,...,n: `number[i] = 0, choosing[i] = false`

```
choosing[i] = true
number[i] = 1 + max {number[j] | (1 ≤ j ≤ n)}
choosing[i] = false
for j = 1 to n {
      await (choosing[j] = false)
      await (number[j] = 0) || (number[j], j) ≥ (number[i],i))
}
critical section
number[i] = 0
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Computing the Maximum

```
local1 = 0
for local2 = 1 to n do
      local3 = number[local2]
      if local1 < local3 then local1 = local3
number[i] = 1 + local1
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Computing the Maximum

```
local1 = 0
for local2 = 1 to n do
      local3 = number[local2]
      if local1 < local3 then local1 = local3
number[i] = 1 + local1
```

Question

Is this version also correct?

```
local1 = i
for local2 = 1 to n do
   if number[local1] < number[local2] then local1 = local2
number[i] = 1 + number[local1]
```

Properties of the Bakery Algorithm

- Satisfies mutual exclusion and FIFO
- Works with **safe registers**: Reads which are concurrent with writes may return arbitrary value

Proof idea: FIFO

- If $D_A \rightarrow D_B$, then A's number is smaller than B's
- writeA(number[A]) $\rightarrow$ readB(number[A]) $\rightarrow$ writeB(number[B]) $\rightarrow$ readB(choosing[A])
- So B is locked out while `choosing[A]` is true

Question

The Bakery Algorithm is succinct, elegant, and fair.

So why isn't it practical?

Question

The Bakery Algorithm is succinct, elegant, and fair.

So why isn't it practical?

- The size of `number[i]` is unbounded
  - But variants with bounded space exist where numbers are re-used
- Well, you have to read N distinct variables

Classification of registers

- Shared read/write memory locations called **registers** (historical reasons)
- Different flavors
  - SRSW = Single-Reader-Single-Writer
  - MRSW = Multi-Reader-Single-Writer (like `flag[]`)
  - MRMW = Multi-Reader-Multi-Writer (like `number[]` or `turn`)
  - [Not that interesting: SRMW]

Observation

Any deadlock-free mutual exclusion algorithm for N threads using only SWMR registers must use at least N such registers.

Observation

Any deadlock-free mutual exclusion algorithm for N threads using only SWMR registers must use at least N such registers.

Proof: Before entering its critical section a thread must write at least once. . . .

Can we do better using MWMR registers ?

Theorem (Lower Bound)

Any deadlock-free mutual exclusion algorithm for N threads must use at least N shared (MWMR) registers.

Proof: Tricky!(Burns and Lynch 1993)

$\Rightarrow$ Let's have a look at the case for two threads!

Proving Algorithmic Impossibility

To show no algorithm exists:

- Assume by way of contradiction one does exist
- Show a bad execution that violates assumed properties

In our case, assume an algorithm for deadlock-free mutual exclusion using $< N$ registers and show how several threads can reach the CS at the same time.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Theorem (Lower Bound) for Two Threads

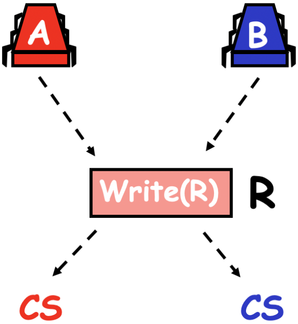
Any deadlock-free mutual exclusion algorithm for 2 threads must use at least 2 shared MWMR registers.

Theorem (Lower Bound) for Two Threads

Any deadlock-free mutual exclusion algorithm for 2 threads must use at least 2 shared MWMR registers.

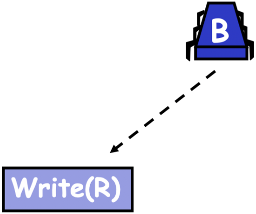
Proof: Assume one register suffices and derive a contradiction

Proof (1): Two-thread executions



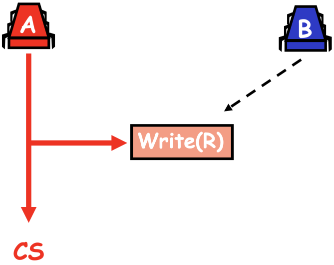
- Threads run, reading and writing register R
- Deadlock-freedom $\Rightarrow$ at least one thread gets in

Proof (2): Covering State for One Register Always Exists



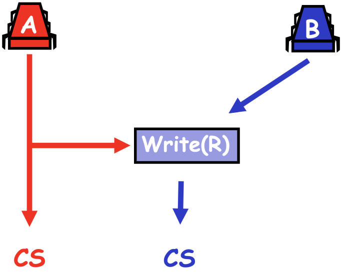
- In any protocol, B has to write to R before entering CS
- Stop it just before

Proof (3): While B is covering R



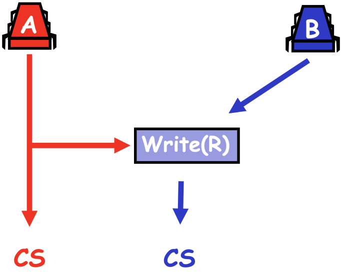
- A runs, possibly writes to R and enters CS

Proof (4): Now B (over)writes



- B Runs, first obliterating any trace of A, then also enters CS

Proof (4): Now B (over)writes



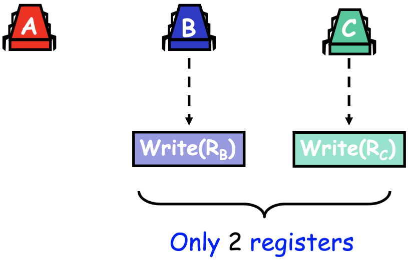
- B Runs, first obliterating any trace of A, then also enters CS

$\Rightarrow$ Mutual exclusion violated!

Theorem (Lower Bound) for Three Threads

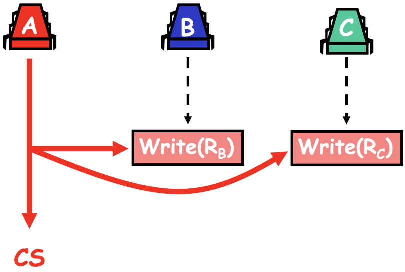
Any deadlock-free mutual exclusion algorithm for 3 threads must use at least 3 shared MWMR registers.

Proof (1)



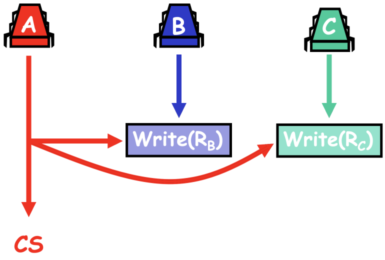
- Assume covering state for 2 threads

Proof (2)



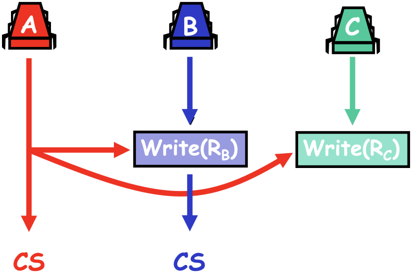
- Now A runs, write to one or both registers, enters CS

Proof (3)



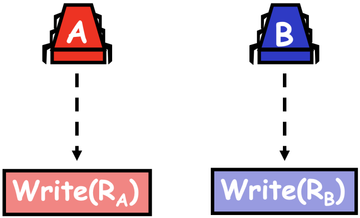
- Other threads obliterate evidence that A entered CS

Proof (4)



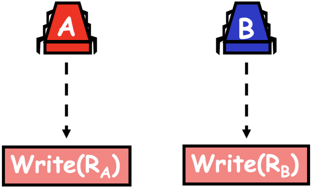
- Other thread gets in because situation cannot be distinguished

Proof (5): How do we reach this covering state?

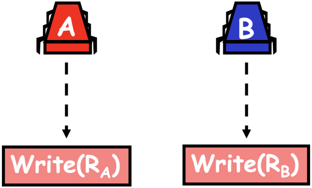


- Start in covering state of B for register R_B
  - If we run B through CS 3 times, B must return twice to cover some register $\rightarrow R_B$
  - Pigeon-hole principle
- Run system until A is about to write to (uncovered) R_A
  - Must exist!

Proof (6): Are we done?

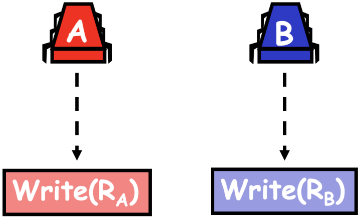


Proof (6): Are we done?



- No! A could have written to R_B
- So, CS no longer looks empty
- Observable by thread C

Proof (7): One more round



- Run B to obliterate traces of A in R_B
- Run B again till it is about to write to R_B
- Now we are done

From 3 to N threads

- Proof by induction
- There is a covering state where k threads not in CS cover k distinct registers
- Proof follows when $k = N - 1$

Summary

- In the 1960's many incorrect solutions to starvation-free mutual exclusion using RW-registers were published ...
- Today, we know how to solve FIFO with N-thread mutual exclusion using $2N$ RW-Registers

Summary

- In the 1960's many incorrect solutions to starvation-free mutual exclusion using RW-registers were published ...
- Today, we know how to solve FIFO with N-thread mutual exclusion using $2N$ RW-Registers
- N RW-Registers inefficient
  - Reason: Writes "cover" older writes
- Need stronger hardware operations that do not have the "covering problem"
- Following lectures: Understand what these operations are!

Copyright

Burns, James E., and Nancy A. Lynch. 1993. "Bounds on Shared Memory for Mutual Exclusion." *Inf. Comput.* 107 (2): 171–84. https://doi.org/10.1006/inco.1993.1065.