Mutual Exclusion

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
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.” (Isaac Newton, 1689)

• “Time is what keeps everything from happening at once.” (Ray Cummings, 1922)
Events

- An \textit{event} $a_0$ of thread A is
  - Instantaneous
  - No simultaneous events (break ties)
Threads

- A *thread* $A$ is (formally) a sequence $a_0, a_1, \ldots$ 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
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

\[ a_0 \quad b_0 \quad B_0 \quad b_1 \quad A_0 \quad a_1 \]
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?
Precendence Ordering

- Never true that $A \rightarrow A$
- If $A \rightarrow B$ then not true that $B \rightarrow A$
- If $A \rightarrow B$ & $B \rightarrow C$ then $A \rightarrow C$
- Funny thing: $A \rightarrow B$ & $B \rightarrow A$ might both be false!
Partial Orders
(review)

• Irreflexive:
  – Never true that $A \rightarrow A$

• Antisymmetric:
  – If $A \rightarrow B$ then not true that $B \rightarrow A$

• Transitive:
  – If $A \rightarrow B$ & $B \rightarrow C$ then $A \rightarrow C$
Total Orders
(review)

• Also
  – Irreflexive
  – Antisymmetric
  – Transitive

• Except that for every distinct A, B,
  – Either A → B or B → A
Repeated Events

while (mumble) {
    a₀; a₁;
}

$k$-th occurrence of event $a₀$

$k$-th occurrence of interval $A₀ = (a₀, a₁)$
Locks (Mutual Exclusion)

```java
public interface Lock {
    public void lock();
    public void unlock();
}
```
Locks (Mutual Exclusion)

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

*acquire lock*
Locks (Mutual Exclusion)

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

acquire lock
release lock
Using Locks

```java
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

```java
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

```java
global 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)
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;
    }
}
Mutual Exclusion

• Let $CS_{i^k}$ be thread i's k-th critical section execution
Mutual Exclusion

- Let $CS_i^k \iff$ be thread i's k-th critical section execution
- And $CS_j^m \iff$ 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 $CS_i^k \iff$ be thread i's k-th critical section execution
- And $CS_j^m \iff$ be j's m-th execution
- Then either
  - $CS_i^k \rightarrow CS_j^m$
Mutual Exclusion

- Let $CS_i^k$ be thread i's k-th critical section execution
- And $CS_j^m$ be j's m-th execution
- Then either
  - $CS_i^k \rightarrow CS_j^m$
  - or $CS_j^m \rightarrow CS_i^k$
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
class LockOne implements Lock {
  private boolean[] flag = new boolean[2];
  public void lock() {
    flag[i] = true;
    while (flag[j]) {}  
  }
}
class LockOne implements Lock {

private boolean[] flag = new boolean[2];

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

Each thread has flag
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];
    public void lock() {
        flag[i] = true;
        while (flag[j]) {} 
    }
}
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** $\text{CS}_A^j$ overlaps $\text{CS}_B^k$

• **Consider each thread's last**
  – $(j^{th}$ and $k^{th})$ read and write …
  – in $\text{lock}()$ before entering

• **Derive a contradiction**
From the Code

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

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

```java
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:
  – read_A(flag[B]==false) → write_B(flag[B]=true)
  – read_B(flag[A]==false) → write_A(flag[A]=true)

• From the code
  – write_A(flag[A]=true) → read_A(flag[B]==false)
  – write_B(flag[B]=true) → read_B(flag[A]==false)
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!

Impossible in a partial order
Deadlock Freedom

- LockOne Fails deadlock-freedom
  - Concurrent execution can deadlock

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

- Sequential executions OK
public class LockTwo implements Lock {
    private int victim;
    public void lock() {
        victim = i;
        while (victim == i) {};
    }

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

    public void unlock() {}
}
public class Lock2 implements Lock {
    private int victim;
    public void lock() {
        victim = i;
        while (victim == i) {};
    }
    public void unlock() {}
}
LockTwo Claims

• Satisfies mutual exclusion
  – If thread \( i \) in CS
  – Then \( \text{victim} == j \)
  – Cannot be both 0 and 1
• Not deadlock free
  – Sequential execution deadlocks
  – Concurrent execution does not

```java
public void LockTwo() {
    victim = i;
    while (victim == i) {}
}
```
Peterson's Algorithm

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

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

Announce I'm interested
Peterson's Algorithm

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

Announce I'm interested
Defer to other
Peterson's Algorithm

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

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

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim
Peterson's Algorithm

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

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

- **Announce I'm interested**
- **Defer to other**
- **Wait while other interested & I'm the victim**
- **No longer interested**
Mutual Exclusion

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

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

From the Code
Also from the Code

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

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

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

W.L.O.G. assume $A$ is the last thread to write $\text{victim}$
Combining Observations

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

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

(2) \(\text{write}_A(\text{victim}=A) \Rightarrow \text{read}_A(\text{flag}[B]) \Rightarrow \text{read}_A(\text{victim})\)
Combining Observations

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

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

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

$\Rightarrow \text{read}_A(\text{victim})$
Combining Observations

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

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

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

A read flag[B] == true and victim == A, so it could not have entered the CS \( \text{ (QED) } \)
Deadlock Free

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

• Thread blocked
  – only at while loop
  – only if other's flag is true
  – only if it is the victim
• Solo: other's flag is false
• Both: one or the other not the victim
Starvation Free

- Thread i blocked only if j repeatedly re-enters so that
  \[ \text{flag}[j] == \text{true} \text{ and } \text{victim} == i \]
- When j re-enters
  - it sets \text{victim} to j.
  - So i gets in

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

public void unlock() {
    \text{flag}[i] = \text{false};
}
```
Bounded Waiting

• Want stronger fairness guarantees
• Thread not “overtaken” too much
• If A starts before B, then A enters before B?
• But what does “start” mean?
• Need to adjust definitions ….
Bounded Waiting

• Divide `lock()` method into 2 parts:
  – Doorway interval:
    • Written $D_A$
    • always finishes in finite steps
  – Waiting interval:
    • Written $W_A$
    • may take unbounded steps
r-Bounded Waiting

• For threads A and B:
  – If $D_A^k \rightarrow D_B^j$
    • A's $k$-th doorway precedes B's $j$-th doorway
  – Then $CS_A^k \rightarrow CS_B^{j+r}$
    • A's $k$-th critical section precedes B's $j+r$-th critical section
    • B cannot overtake A more than $r$ times

• First-come-first-served $\Rightarrow r = 0$
What is “r” for Peterson's Algorithm?

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

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

Answer: r = 0
First-Come-First-Served

• For threads A and B:
  – If $D_A^k \rightarrow D_B^j$
    • A's $k$-th doorway precedes B's $j$-th doorway
  – Then $CS_A^k \rightarrow CS_B^j$
    • 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 $i > j$
class Bakery implements Lock {
    boolean[] flag;
    Label[] label;

    public Bakery (int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false; label[i] = 0;
        }
    }

    ...
}
class Bakery implements Lock {
    boolean[] flag;
    Label[] label;

    public Bakery(int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false; label[i] = 0;
        }
    }
    ...
}

Bakery Algorithm
class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k]
            && (label[i], i) > (label[k], k));
    }
}
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k] && (label[i],i) > (label[k],k));
    }
}
class Bakery implements Lock {
  ...
  public void lock() {
    // I'm interested
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (∃k flag[k]
      && (label[i], i) > (label[k], k));
  }
}
Bakery Algorithm

```java
class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k] && (label[i], i) > (label[k], k));
    }
}
```

Take increasing label (read labels in some arbitrary order)
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (\exists k flag[k] && (label[i], i) > (label[k], k));
    }
}
Bakery Algorithm

class Bakery implements Lock {
    boolean flag[n];
    int label[n];

    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (exists k flag[k] && (label[i], i) > (label[k], k));
    }

Someone is interested …

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

class Bakery implements Lock {
    ...
    public void unlock() {
        flag[i] = false;
    }
}
Bakery Algorithm

class Bakery implements Lock {

    ...

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

}

No longer interested

labels are always increasing
No Deadlock

- There is always one thread with earliest label
- Ties are impossible (why?)
First-Come-First-Served

• If $D_A \Rightarrow D_B$ then
  – A's label is smaller

• And:
  – $\text{write}_A(\text{label}[A]) \Rightarrow$
  – $\text{read}_B(\text{label}[A]) \Rightarrow$
  – $\text{write}_B(\text{label}[B]) \Rightarrow \text{read}_B(\text{flag}[A])$

• So B sees
  – smaller label for A
  – locked out while $\text{flag}[A]$ is true

```java
class Bakery implements Lock {
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], …, label[n-1])+1;
        while (\exists k \text{ flag}[k]
            && (label[i],i) >
            (label[k],k));
    }
```
Mutual Exclusion

• Suppose A and B in CS together
• Suppose A has earlier label
• When B entered, it must have seen
  – flag[A] is false, or
  – label[A] > label[B]

```java
class Bakery implements Lock {
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ...
                         label[n-1]) + 1;
        while (∃k flag[k]
                   && (label[i],i) >
                   (label[k],k));
    }
```
Mutual Exclusion

- Labels are strictly increasing so
- B must have seen $\text{flag}[A] == \text{false}$
Mutual Exclusion

- Labels are strictly increasing so
- B must have seen flag[A] == false
- Labeling_B \(\Rightarrow\) read_B(flag[A]) \(\Rightarrow\) write_A(flag[A]) \(\Rightarrow\) Labeling_A
Mutual Exclusion

- Labels are strictly increasing so
- B must have seen $\text{flag}[A] == \text{false}$
- $\text{Labeling}_B \Rightarrow \text{read}_B(\text{flag}[A]) \Rightarrow \text{write}_A(\text{flag}[A]) \Rightarrow \text{Labeling}_A$
- Which contradicts the assumption that A has an earlier label
Bakery Y2^{32}K Bug

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k]
            && (label[i],i) > (label[k],k));
    }
}
Bakery Y2³²K Bug

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1])+1;
        while (∃k flag[k]
               && (label[i],i) > (label[k],k));
    }

Mutex breaks if label[i] overflows
Does Overflow Actually Matter?

• Yes
  – Y2K
  – 18 January 2038 (Unix `time_t` rollover)
  – 16-bit counters

• No
  – 64-bit counters

• Maybe
  – 32-bit counters
Deep Philosophical Question

• The Bakery Algorithm is
  – Succinct,
  – Elegant, and
  – Fair.

• Q: So why isn't it practical?

• A: Well, you have to read $N$ distinct variables
Shared Memory

• Shared read/write memory locations called *Registers* (historical reasons)
• Come in different flavors
  – Multi-Reader-Single-Writer (*flag*[])
  – Multi-Reader-Multi-Writer (*victim*[])
  – Not that interesting: SRMW and SRSW
Theorem

At least $\text{N}$ MRSW (multi-reader/single-writer) registers are needed to solve deadlock-free mutual exclusion.

$\text{N}$ registers such as `flag[]`...
Theorem

Deadlock-free mutual exclusion for 3 threads requires at least 3 multi-reader multi-writer registers
Theorem

Deadlock-free mutual exclusion for $n$ threads requires at least $n$ multi-reader multi-writer registers
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
  – Because writes “cover” older writes

• Need stronger hardware operations
  – that do not have the “covering problem”

• In next lectures - understand what these operations are…
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- You are free:
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
- Under the following conditions:
  - Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.