Mutual Exclusion

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Adapted by Larry Wittie for CSE391 S14
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
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)
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 of events” 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
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?
• 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! (They overlap, as in image above.)
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 (disjoint) A, B,
  - Either A \rightarrow B or B \rightarrow A
Repeated Events (in code loops)

while (mumble) {
    \(a_0; a_1;\)
}

\(k\)-th occurrence of event \(a_0\)

\(k\)-th occurrence of interval \(A_0=(a_0,a_1)\)
Implementing a Counter

```java
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)

```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
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

```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;
    }
}
```
Mutual Exclusion

• Let $CS_i^k \iff$ be thread $i$’s $k$-th critical section execution
Mutual Exclusion

- Let $CS_i^k \leftrightarrow$ be thread $i$’s $k$-th critical section execution
- And $CS_j^m \leftrightarrow$ be thread $j$’s $m$-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 $j$’s $m$-th execution
• Then either
  $\text{-----} \iff \text{-----} \text{ or } \text{-----} \iff \text{-----} \iff \text{-----}$
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 $CS_i^k \leftarrow CS_j^m$
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 $CS_j^m \Rightarrow CS_i^k$. 

Art of Multiprocessor Programming  Ch02 CSE391
Multicore Programming S14  2/4,6/14
Deadlock-Free

• If some thread calls \texttt{lock()} 
  – And never returns (from repeated \texttt{trying}) 
  – Then other threads must complete \texttt{lock()} and \texttt{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]) {}
    }
}
LockOne

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]) {}  
    }

    Set my flag
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
From the Code

• write_A(flag[A]=true) \rightarrow read_A(flag[B]==false) \rightarrow CS_A

• write_B(flag[B]=true) \rightarrow read_B(flag[A]==false) \rightarrow CS_B

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

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

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

Both in CS => Each saw other’s flag as false before other set its flag to 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
  – $\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow \text{read}_A(\text{flag}[B]==\text{false})$
  – $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{read}_B(\text{flag}[A]==\text{false})$

• Assume each reads other’s flag as false before other sets it true but code says each sets its flag true before reading other’s flag.  XXX
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
  - $\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow \text{read}_A(\text{flag}[B]==\text{false})$
  - $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{read}_B(\text{flag}[A]==\text{false})$
• 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:
  - read\(_A\)(flag[B]==false) \rightarrow write\(_B\)(flag[B]=true)
  - read\(_B\)(flag[A]==false) \rightarrow write\(_A\)(flag[A]=true)

• From the code:
  - write\(_A\)(flag[A]=true) \rightarrow read\(_A\)(flag[B]==false)
  - write\(_B\)(flag[B]=true) \rightarrow read\(_B\)(flag[A]==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() {}
}
LockTwo

```java
public class Lock2 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 $\text{victim} == j$
  – Cannot be both 0 and 1

• Not deadlock free
  – Sequential execution deadlocks
  – Concurrent execution does not
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

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**
- **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

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

W.L.O.G. assume A is the last thread to write victim

(Without Loss Of Generality)
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) \( \text{write}_B(\text{flag}[B]=\text{true}) \)\n
(3) \( \text{write}_B(\text{victim}=B) \)\n
(2) \( \text{write}_A(\text{victim}=A) \)\n\hspace{1cm} \Rightarrow \ \text{read}_A(\text{flag}[B]) \)\n\hspace{1cm} \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 $\text{flag}[B] == \text{true}$ and $\text{victim} == A$, so $A$ could not have entered the Critical Section (QED There is Mutual Exclusion)
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 entry: other’s flag is `false`
- Both enter: one or the other not the victim
Starvation Free

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

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

public void unlock() {
    flag[i] = false;
}
```
The Filter Algorithm for $n$ Threads

There are $n-1$ “waiting rooms” called levels

- At each level $L$
  - At least one enters level
  - At least one blocked if many threads try
- Only one thread makes it through to $L_{n-1}$
class Filter implements Lock {
    int[] level;  // level[i] for thread i
    int[] victim; // victim[L] for level L

    public Filter(int n) {
        level = new int[n];
        victim = new int[n];
        for (int i = 1; i < n; i++) {
            level[i] = 0;
        }
    }

    ...
}
Filter

class Filter implements Lock {
  ...

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i] = L;
      victim[L] = i;
      while ((\exists k \neq i level[k] >= L) &&
        victim[L] == i ) {};
    }
  }

  public void unlock() {
    level[i] = 0;
  }
}
class Filter implements Lock {
    ...

    public void lock() {
        for (int L = 1; L < n; L++) {
            level[i] = L;
            victim[L] = i;
            while ((∃ k != i) level[k] >= L) &&
                victim[L] == i) {};
        }

        public void unlock(int i) {
            level[i] = 0;
        }
    }
}
Filter

class Filter implements Lock {
    
    public void lock() {
        for (int L = 1; L < n; L++) {
            level[i] = L;
            victim[L] = i;
            while ((∃ k != i) level[k] >= L) &&
                victim[L] == i) {}; // busy wait
        }
    }

    public void unlock(int i) {
        level[i] = 0;
    }
}

Announce intention to enter level L
Filter

class Filter implements Lock {
    int level[n];
    int victim[n];
    public void lock() {
        for (int L = 1; L < n; L++) {
            level[i] = L;
            victim[L] = i;
            while ((\exists k \neq i) level[k] >= L) &&
                  victim[L] == i) {};
        }
    }
    public void unlock(int i) {
        level[i] = 0;
    }
}

Give priority to anyone but me
public void lock() {
    for (int L = 1; L < n; L++) {
        level[i] = L;
        victim[L] = i;
        while (((exists k != i) level[k] >= L) && (
            victim[L] == i)) {
        }
    }
}
public void unlock(int i) {
    level[i] = 0;
}
class Filter implements Lock {
    int level[n];
    int victim[n];
    public void lock() {
        for (int L = 1; L < n; L++) {
            level[i] = L;
            victim[L] = i;
            while ((∃ k != i) level[k] >= L) &&
            victim[L] == i) {};
        }
    }
    public void release(int i) {
        level[i] = 0;
    }
}

Thread i enters level L when it completes the do_nothing {} wait_loop
Claim

- If at most $n$ threads start at level $L=0$,
- At most $n-L$ threads enter level $L$
- Mutual exclusion occurs at level $L=n-1$
Induction Hypothesis

- No more than \( n-(L-1) \) at level \( L-1 \)
- Induction step: by contradiction
- Assume all at level \( L-1 \) enter level \( L \)
- A last to write victim[\( L \)]
- \( B \) is any other thread at level \( L \)
- Induction step: by contradiction

\( ncs \)

\( \text{assume} \)

\( L-1 \) has \( n-(L-1) \)

\( L \) has \( n-L \)

\( \text{must prove} \)

\( \# \leq n \)

\( \# \leq n-1 \)

\( \# \leq n-2 \)

\( \# \leq 2 \)

\( \# \leq 1 \)
Proof Structure

Assume A is last to write victim[L]

Show that A must have seen B in level[L] and, since victim[L] == A, A could not have entered level L

Assumed to enter L-1

Level: L-1, n-L+1 = 4

Level: L, n-L = 4

By way of contradiction XXX, assume all 4 enter L
Just Like Peterson

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

```java
public void lock() {
    for (int L = 1; L < n; L++) {
        level[i] = L;
        victim[L] = i;
        while ((\exists k \neq i) level[k] \geq L)
            && victim[L] == i) {;}
    }
}
```

From the Code
From the Code

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

```java
public void lock() {
    for (int L = 1; L < n; L++) {
        level[i] = L;
        victim[L] = i;
        while ((\exists k \neq i) \text{ level}[k] \geq L)
            && victim[L] == i) {}
    }
}
```
By Assumption

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

By assumption, $A$ is the last thread to write $\text{victim}[L]$
Combining Observations

(1) \( \text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B) \)

(3) \( \text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A) \)

(2) \( \text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B]) \rightarrow \text{read}_A(\text{victim}[L]) \)
Combining Observations

(1) \( \text{write}_B(\text{level}[B]=L) \Rightarrow \)

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

(2) \( \Rightarrow \text{read}_A(\text{level}[B]) \)

\( \Rightarrow \text{read}_A(\text{victim}[L]) \)
Combining Observations

(1) \( \text{write}_B(\text{level}[B]=L) \rightarrow \)

(3) \( \text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A) \)

(2) \( \rightarrow \text{read}_A(\text{level}[B]) \rightarrow \text{read}_A(\text{victim}[L]) \)

A read level[B] ≥ L and victim[L] == A, so A could not have entered level L! XXX, so if level L-1 has n-(L-1) then level L has n-L. Since only n-0 threads can start at L=0, level L=n-1 has at most 1 thread => mutex.
No Starvation

• Filter Lock satisfies properties:
  – Just like Peterson Algorithm at any level
  – So no one starves

• But what about fairness?
  – Threads can be overtaken by others
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
  - {Filter `lock()` lacks distinct $D_A$ and $W_A$ code}
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
Fairness Again

• Filter Lock satisfies properties:
  – No one starves
  – But very weak fairness
    • Can be overtaken arbitrary # of times
  – That’s pretty lame…
Bakery Algorithm

• Provides First-Come-First-Served
• 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

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 {
  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)) {
    }
}
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)) {};
    }

I’m interested
Bakery Algorithm

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

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 (∃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 (∃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;
    }
}

labels are always increasing

No longer interested
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] = \text{max}(label[0], ..., label[n-1])+1;
        \text{while} (\exists k \text{ flag}[k] \&\& (\text{label}[i], i) > (\text{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]

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 flag[A] == 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 flag[A] == false
• Labeling\(_B\) ⟹ read\(_B\)(flag[A]) ⟹
  write\(_A\)(flag[A]) ⟹ Labeling\(_A\)
• Which contradicts the assumption that A
  has an earlier label
Bakery Y2\textsuperscript{32}K Bug

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 Y2³²K Bug

```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)){};
    }
```

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
Timestamps

• Label variable is really a timestamp
• Need ability to
  – Read others’ timestamps
  – Compare them
  – Generate a later timestamp
• Can we do this without overflow?
The Good News

• One can construct a
  – Wait-free (no mutual exclusion)
  – Concurrent
  – Timestamping system
  – That never overflows
The Good News

• One can construct a
  – Wait-free (no mutual exclusion)
  – Concurrent
  – Timestamping system
  – That never overflows

This part is hard
Instead …

• We construct a **Sequential timestamping system**
  – Same basic idea
  – But simpler
• As if we use mutex to read & write atomically
• No good for building locks
  – But useful anyway
Precedence Graphs

- Timestamps form directed graph
- Edge $x$ to $y$
  - Means $x$ is later timestamp
  - We say $x$ dominates $y$
Unbounded Counter Precedence Graph

- Timestamping = move tokens on graph
- Atomically
  - read others’ tokens
  - move mine
- Ignore tie-breaking for now
Unbounded Counter Precedence Graph

0 1 2 3

......
Unbounded Counter Precedence Graph

takes 0  takes 1  takes 2
Two-Thread Bounded Precedence Graph
Two-Thread Bounded Precedence Graph
Two-Thread Bounded Precedence Graph
Two-Thread Bounded Precedence Graph $T^2$

and so on …
Three-Thread Bounded Precedence Graph?
Three-Thread Bounded Precedence Graph?

What to do if one thread gets stuck?
Graph Composition

$T^3 = T^2 \times T^2$

Replace each vertex with a copy of the graph.
Three-Thread Bounded Precedence Graph $T^3$

![Three-Thread Bounded Precedence Graph](image)
Three-Thread Bounded Precedence Graph $T^3$
Three-Thread Bounded Precedence Graph $T^3$

and ...
Three-Thread Bounded Precedence Graph $T^3$

and …
Three-Thread Bounded Precedence Graph $T^3$

and so on…
In General

\[ T^k = T^2 \times T^{k-1} \]

K threads need \(3^k\) nodes

label bit size = \(\log_2(3^k) = 2k\)
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 \( N \) MRSW (multi-reader/single-writer) registers are needed to solve deadlock-free mutual exclusion.

\( N \) registers like Flag[]...
Proving Algorithmic Impossibility

• To show no algorithm exists:
  • assume by way of contradiction one does,
  • show a **bad execution** that violates properties:
  • in our case assume an alg for deadlock free mutual exclusion using < N registers
Proof: Need N-MRSW Registers

Each thread must write to some register

A \rightarrow \text{write} \rightarrow \text{CS}
B \rightarrow \text{write} \rightarrow \text{CS}
C \rightarrow \text{write} \rightarrow \text{CS}

...can’t tell whether A is in critical section
Upper Bound

- Bakery algorithm
  - Uses $2N$ MRSW registers
- So the bound is (pretty) tight
- But what if we use MRMW registers?
  - Like `victim[]`?
Bad News Theorem

At least $N$ MRMW multi-reader/multi-writer registers are needed to solve deadlock-free mutual exclusion.

(So multiple writers don’t help)
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
Two Thread Execution

- Threads run, reading and writing R
- Deadlock free so at least one gets in
Covering State for One Register Always Exists

In any protocol, B has to write to the register before entering CS, so stop it just before
Proof: Assume Cover of 1

A runs, possibly writes to the register, enters CS
Proof: Assume Cover of 1

B Runs, first obliterating any trace of A, then also enters the critical section.
Theorem

Deadlock-free mutual exclusion for 3 threads requires at least 3 multi-reader multi-writer registers
Proof: Assume Cover of 2

Only 2 registers
Run A Solo

A \rightarrow B \rightarrow C

Write(R_B) \rightarrow Write(R_C)

Writes to one or both registers, enters CS

CS
Obliterate Traces of A

Other threads obliterate evidence that A entered CS
Mutual Exclusion Fails

CS looks empty, so another thread gets in.
Proof Strategy

- Proved: a contradiction starting from a covering state for 2 registers
- Claim: a covering state for 2 registers is reachable from any state where CS is empty
• If we run B through CS 3 times, B must return twice to cover 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?

Write($R_A$)  Write($R_B$)
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 $R_B$
- Run B again until it is about to write to $R_B$
- Now we are done

Write($R_A$)  Write($R_B$)
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 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.