Mutual Exclusion

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

• Today we will try to formalize our understanding of mutual exclusion
• We will also use the opportunity to show you how to argue about and prove various properties in an asynchronous concurrent setting
Mutual Exclusion

In his 1965 paper E. W. Dijkstra wrote:

"Given in this paper is a solution to a problem which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. [...] Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved."
Mutual Exclusion

- Formal problem definitions
- Solutions for 2 threads
- Solutions for $n$ threads
- Fair solutions
- Inherent costs
Warning

• You will never use these protocols
  – Get over it
• You are advised to understand them
  – The same issues show up everywhere
  – Except hidden and more complex
Why is Concurrent Programming so Hard?

• Try preparing a seven-course banquet
  – By yourself
  – With one friend
  – With twenty-seven friends ...

• Before we can talk about programs
  – Need a language
  – Describing time and concurrency
Time

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

- “Time is, like, Nature’s way of making sure that everything doesn’t happen all at once.” (Anonymous, circa 1968)
Events

• An event $a_0$ of thread A is
  - Instantaneous
  - No simultaneous events (break ties)
Threads

- A thread $A$ is (formally) a sequence $a_0, a_1, \ldots$ 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
Intervals may be Disjoint
Precedence

Interval $A_0$ precedes interval $B_0$
• **Notation:** \( A_0 \rightarrow B_0 \)

• **Formally,**
  - End event of \( A_0 \) before start event of \( B_0 \)
  - Also called “happens before” or “precedes”
Precedence Ordering

• Remark: $A_0 \Rightarrow B_0$ is just like saying
  - 1066 AD $\Rightarrow$ 1492 AD,
  - Middle Ages $\Rightarrow$ Renaissance,

• Oh wait,
  - what about this week vs this month?
Precedence Ordering

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

• **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

• Also
  – Irreflexive
  – Antisymmetric
  – Transitive

• Except that for every distinct A, B,
  – Either $A \Rightarrow B$ or $B \Rightarrow A$
Repeated Events

while (mumble) {
    a₀; a₁;
}

$k$-th occurrence of event $a₀$

$k$-th occurrence of interval $A₀ = (a₀, a₁)$
Implementing a Counter

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        long 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();
}
```

Locks (Mutual Exclusion)

```java
class 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 = temp + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
```
Using Locks

```java
public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock(); // acquire Lock
        try {
            int temp = value;
            value = temp + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
```
public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = temp + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
Using Locks

public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = temp + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
Mutual Exclusion

- Let $\text{CS}_{i}^{k}$ 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 $CS^k_i$ be thread $i$’s $k$-th critical section execution
• And $CS^m_j$ be $j$’s $m$-th execution
• Then either
  – or
Mutual Exclusion

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

- Let $\text{CS}_i^k \leftrightarrow$ be thread $i$’s $k$-th critical section execution
- And $\text{CS}_j^m \leftrightarrow$ be $j$’s $m$-th execution
- Then either
  - $\text{CS}_i^k \Rightarrow \text{CS}_j^m$
  - or
  - $\text{CS}_j^m \Rightarrow \text{CS}_i^k$
Deadlock-Free

• If some thread calls \texttt{lock()} 
  - And never returns 
  - 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

• Two-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

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

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]) {} 
    }
}

LockOne
Wait for other flag to go false
Set my flag
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 the lock() method before entering
- Derive a contradiction
From the Code

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

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

```java
class LockOne implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        while (flag[j]) {} // This should be modified to handle multiple conditions
    }
}
```
From the Assumption

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

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

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

- **From the code**
  - \( \text{write}_A(\text{flag}[A]==true) \rightarrow \text{read}_A(\text{flag}[B]==false) \)
  - \( \text{write}_B(\text{flag}[B]==true) \rightarrow \text{read}_B(\text{flag}[A]==false) \)
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:
  - read\textsubscript{A}(flag[B] == false) $\Rightarrow$ write\textsubscript{B}(flag[B] = true)
  - read\textsubscript{B}(flag[A] == false) $\Rightarrow$ write\textsubscript{A}(flag[A] = true)

• From the code:
  - write\textsubscript{A}(flag[A] = true) $\Rightarrow$ read\textsubscript{A}(flag[B] == false)
  - write\textsubscript{B}(flag[B] = true) $\Rightarrow$ read\textsubscript{B}(flag[A] == false)
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)
Cycle!

Impossible in partial order
Deadlock Freedom

• LockOne Fails deadlock-freedom
  – Concurrent execution can deadlock

```plaintext
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) {java
    }

    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

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

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

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

• If thread 0 in critical section,
  – flag[0]=true,
  – !flag[1] ||
    victim = 1

• If thread 1 in critical section,
  – flag[1]=true,
  – !flag[0] ||
    victim = 0

Cannot both be true
Mutual Exclusion Proved

Thread A

write_A(flag[A]=true)
write_A(victim=A)
read_A(flag[B])
read_A(victim)
CS_A

Thread B

write_B(flag[B]=true)
write_B(victim=B)
read_B(flag[A])
read_B(victim)
CS_B
Deadlock Free

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

• Thread blocked
  - only at while loop
  - only if it is the victim

• One or the other must not be 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() {
    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
  - At least one enters level
  - At least one blocked if many try

- Only one thread makes it through
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;
        }
    }

    ...
}

Thread 2 at level 4
class Filter implements Lock {
    ...

    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() {
        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 release(int i) {
        level[i] = 0;
    }
}

One level at a time
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 release(int i) {
        level[i] = 0;
    }
    }

Announce intention to enter level L
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;
        }
    }
}

Filter
Give priority to anyone but me
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;
    }
}

Filter

Wait as long as someone else is at the same or higher level, and I’m designated victim.
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 enters level L when it completes the loop.
Claim

• Start at level $L=0$
• At most $n-L$ threads enter level $L$
• Mutual exclusion 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 \( L \)
- \( L-1 \) has \( n-L+1 \)
- \( L \) has \( n-L \)
- Prove
Proof Structure

Assumed to enter L-1

n-L+1 = 4

By way of contradiction all enter L

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

(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) \text{ level}[k] \geq L)
            && victim[L] == i) {};
    }
}
```
From the Code

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

```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) \( \text{write}_B(\text{level}[B]=L) \implies \text{write}_B(\text{victim}[L]=B) \)

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

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

(1) write\(_B\) (level[B] = L) \(\Rightarrow\) \\
(3) write\(_B\) (victim[L] = B) \(\Rightarrow\) write\(_A\) (victim[L] = A) \\
(2) \forall i \quad \Rightarrow\) read\(_A\) (level[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] >= L) && victim[L] == i) {};
    }
}
```
Combining Observations

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

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

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

Thus, A reads $\text{level}[B] \geq L$, A was last to write $\text{victim}[L]$, so it could not have entered level $L$!
No Starvation

• Filter Lock satisfies properties:
  – 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
• Need to adjust definitions ....
Bounded Waiting

• Divide \texttt{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 $A$ by more than $r$ times

- First-come-first-served means $r = 0$. 
Fairness Again

• Filter Lock satisfies properties:
  - No one starves
  - But very weak fairness
    • Not $r$-bounded for any $r$!
  - That’s pretty lame…
Lamport's 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;
        }
    }

    ...
}

Art of Multiprocessor Programming
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() {
        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));
    }

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));
    }

Someone is interested
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

With lower (label,k) in lexicographic order
Bakery Algorithm

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

```java
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 \( D_A \to D_B \) then A’s label is smaller
- And:
  - \( \text{write}_A(\text{label}[A]) \to \text{read}_B(\text{label}[A]) \to \text{write}_B(\text{label}[B]) \to \text{read}_B(\text{flag}[A]) \)
- So B is 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 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 \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 $Y_2^{32}K$ 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 Really Matter?

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

- **No**
  - 64-bit counters

- **Maybe**
  - 32-bit counters
Summary of Lecture

• In the 1960’s many 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/](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.