Mutual Exclusion

Acknowledgement:
Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming"
by Maurice Herlihy and Nir Shavit
What We'll Cover Today

Chapter 2 of:

Digital copy can be obtained via WUSTL library:
http://catalog.wustl.edu/search/
Mutual Exclusion

• **Critical section:** a block of code that accesses shared modifiable data or resource that should be operated on by only one thread at a time.

• **Mutual exclusion:** a property that ensures that a critical section is only executed by a thread at a time.
Mutual Exclusion

In his 1965 paper "Solution of a problem in concurrent programming control," 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." [1]
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

But first, we need some terminology to describe time and concurrency ...
Concurrent Programming and the Notion of Time

“Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external.”

--- Issac Newton, 1689
Concurrent Programming and the Notion of Time

In a concurrent computation, there is really not a sense of time (i.e., a global clock) that all threads in the computation can read and use to relate to each other.

Instead, we will think in terms of the ordering among partially ordered events, and will use time as a tool for explaining the ordering. That is, the notion of time is local, not global.
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

![Diagram of time and threads](image)
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
- Thread B
Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)

`time`
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"
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 Is a Partial Order

• Irreflexive: never true that $A \rightarrow A$

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

• Transitive: If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

• Also: $A \rightarrow B$ and $B \rightarrow A$ might both be false!
Repeated Events

while (mumble) {
    a₀; a₁;
}

For simplicity, we omit the superscript when it's clear which instance we are referring to.
Implementing a Counter

critical section

Make these steps *indivisible* using locks
Locks (Mutual Exclusion)

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

```java
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;
    }
}
```
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;
    }
}
```
Mutual Exclusion

- Let $CS_i^k$ be thread $i$'s $k$-th critical section execution
- And $CS_j^m$ be thread $j$'s $m$-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
• Then either
  – $CS_i^k \rightarrow CS_j^m$ or $CS_j^m \rightarrow CS_i^k$
Properties of a Good Locking Algorithm

• Mutual exclusion
• Deadlock freedom
• Starvation freedom
Properties of a Good Locking Algorithm

• Mutual exclusion

• **Deadlock freedom**: system as a whole makes progress.
  If some thread calls `lock()` and never returns, then other threads must complete `lock()` and `unlock()` calls infinitely often.

• Starvation freedom
Properties of a Good Locking Algorithm

• Mutual exclusion

• **Deadlock freedom:** system as a whole makes progress.
  If some thread calls `lock()` and never returns, then other threads must complete `lock()` and `unlock()` calls infinitely often.

• **Starvation freedom:** individual thread makes progress. *(This implies deadlock freedom.)*
  If some thread calls `lock()`, it will eventually return.
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;
        ...
    }
}

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]) {}
}
}

Each thread has a 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
• WLOG, assume $CS_A^j$ overlaps $CS_B^k$.
• Consider each thread's last read and write in `lock()` before entering the overlapping critical section.
• Derive a contradiction.
LockOne Satisfies Mutual Exclusion

(1) From each thread's `lock()` execution:
\[
\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow \text{read}_A(\text{flag}[B]==\text{false}) \rightarrow \text{CS}_A \\
\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{read}_B(\text{flag}[A]==\text{false}) \rightarrow \text{CS}_B
\]

(2) Since A and B have not executed `unlock()`, no flags have been set to false. Thus, it must be that:
\[
\text{read}_A(\text{flag}[B]==\text{false}) \rightarrow \text{write}_B(\text{flag}[B]=\text{true})
\]

Combine observations:
\[
\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow \text{read}_A(\text{flag}[B]==\text{false}) \quad (1) \\
\rightarrow \text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{read}_B(\text{flag}[A]==\text{false}) \quad (2)
\]

Must have read flag[A] = true; contradiction!
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 LockTwo 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 [2]

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}
public void unlock() {
    flag[i] = false;
}
```
Peterson's Algorithm [2]

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

Announce interest
Peterson's Algorithm [2]

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {}
}
public void unlock() {
    flag[i] = false;
}
```
Peterson's Algorithm [2]

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

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

Wait while other is interested & I'm the victim
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}
public void unlock() {
    flag[i] = false;
}

Peterson's Algorithm [2]
Peterson's Satisfies 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) {};
}
```
Peterson's Satisfies Mutual Exclusion

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

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}
```
Peterson's Satisfies Mutual Exclusion

(3) write_B(victim=B) → write_A(victim=A)

W.L.O.G. assume A is the last thread to write victim
Peterson's Satisfies Mutual Exclusion

(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) $\text{write}_A(\text{victim}=A) \Rightarrow \text{read}_A(\text{flag}[B])$
   $\Rightarrow \text{read}_A(\text{victim})$
Peterson's Satisfies Mutual Exclusion

(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 it could not have entered the CS (QED)
Peterson's Is Deadlock Free*

```java
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

* Assuming a thread doesn't just enter the CS and die.
Peterson's Is Starvation Free*

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

- Thread A blocked only if thread B repeatedly reenters so that
  `flag[j] == true && victim == i`
- When thread B re-enters
  - it sets `victim` to `j`.
  - So thread A gets in

* Assuming a thread doesn't just enter the CS and die.
Lamport's Bakery Algorithm [3]

• Provides First-Come-First-Served for n threads

• Basic idea:
  – Take a “number”
  – Wait until lower numbers have been served

• If two threads get the same number, use thread ID to break tie:
  – \((a,i) > (b,j)\)
  If \(a > b\), or \(a = b\) and \(i > j\) (lexicographic order)
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;
    }
  }

  ...
}

One per thread, indexed by thread ID
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));
    }

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

I'm interested
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));
    }
    public void unlock() {
        flag[i] = false;
    }
}

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

Someone is interested whose (label,tID) in lexicographic order is lower
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));
  }

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

Labels are always increasing.
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));
    }

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

}
Bakery Satisfies Mutual Exclusion

WLOG, assume $CS_A$ and $CS_B$ overlap:

WLOG, also assume that thread A has a lower (label, tID) than B when both are in CS (at t3).

Since a thread's label is always increasing, and B doesn't change its label once `lock()` returns, B must has a higher (label, tID) than A when it enters $CS_B$. 
Bakery Satisfies Mutual Exclusion

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));
}

• B only enters CS\_B if it read
  - flag[A] is false, or
  - (label[A], A) > (label[B], B)

• We just showed that when B enters CS\_B, A has a lower (label, tID).

• B must have seen that flag[A] is false.
Bakery Satisfies Mutual Exclusion

- B must have seen flag[A] == false

(1) \( \text{read}_B(\text{flag}[A]) \Rightarrow \text{write}_A(\text{flag}[A]) \)
   (since flag[A] hasn't changed)

(2) \( \text{write}_B(\text{label}[B]) \Rightarrow \text{read}_B(\text{flag}[A]) \) and
   \( \text{write}_A(\text{flag}[A]) \Rightarrow \text{read}_A(\text{label}[B]) \Rightarrow \text{write}_A(\text{label}[A]) \)
   (from the code)

Combining Observation:
write_B(label[B]) \( \Rightarrow \) read_B(flag[A])
\( \Rightarrow \) write_A(flag[A]) \( \Rightarrow \) read_A(label[B]) \( \Rightarrow \) write_A(label[A])

so it must be that label[A] > label[B],
which contradicts that A has a lower (label[A], tID). (QED)
Bakery is Deadlock-Free

• There is always one thread with the lowest (label, tID)
• Ties are impossible
First-Come-First-Serve

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

• First-come first-serve:
  for threads A and B, if $D_A^k \rightarrow D_B^j$ then $CS_A^k \rightarrow CS_B^j$

• r-Bounded waiting:
  for threads A and B, if $D_A^k \rightarrow D_B^j$ then $CS_A^k \rightarrow CS_B^{j+r}$
Bakery Is First-Come-First-Serve

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

If \( D_A \rightarrow D_B \) then B must see a lower label for A and be locked out while A is in CS_A.
Bakery Y2^{32}K 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));
    }
    public void unlock() {
        flag[i] = false;
    }
}
```

Lock breaks if label[i] overflows

64-bit system ... we will let grand children worry about that.
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[] in Bakery)
  – Multi-Reader-Multi-Writer (victim in Peterson's)
  – Not that interesting: SRMW and SRSW
Theorem [4]

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

N registers such as flag[]...
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 [4]

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

(So multiple writers don't help)
Final Words

• 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 MRSW-Registers.

• N Registers inefficient
  – Because writes “cover” older writes

• Need stronger hardware primitives
References


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.