Lecture 17-18: Mutual Exclusion

CS 539 / ECE 526
Distributed Algorithms
Mutual Exclusion

• Process A
  non-critical section
  critical section
  remainder section
  repeat (possibly)

• Process B
  non-critical section
  critical section
  remainder section
  repeat (possibly)

• Examples:
  Delete p in link list
  Balance += 100
  Sell last seat to A
  Delete p’s parent
  Balance += 200
  Sell last seat to B
Outline

• Mutual exclusion problem definition

• Using strong primitives
  – Test-and-Set
  – Atomic queue and Read-Modify-Write

• Using shared registers
  – Using atomic registers: Peterson
  – Using safe registers: Bakery

• Fast Mutex
Mutual Exclusion (Mutex)

- Process A
  entry
  critical section
  exit
  remainder section
  repeat (possibly)

- Process B
  entry
  critical section
  exit
  remainder section
  repeat (possibly)

• Entry: request to enter critical section, coordinate with other threads
• Exit: clean-up work
An Easy Problem?

• Process A
  - Lock.lock()
  - critical section
  - Lock.unlock()
  - remainder section
  - repeat (possibly)

• Process B
  - Lock.lock()
  - critical section
  - Lock.unlock()
  - remainder section
  - repeat (possibly)

• Not a solution: have to solve the mutex problem to build a lock / semaphore
Mutual Exclusion [Dijkstra 1965]

• n processes may request exclusive right to enter **critical section**

• Safety (mutual exclusion): at most one process in critical section

• Liveness: no deadlock (next slide)

• Fairness: several variants (next slide)
Mutual Exclusion Fairness

- **Deadlock free**: if a process is in entry, eventually *some* process is in critical section
  - No fairness guarantee

- **Starvation free**: if a process is in entry, eventually *that* process is in critical section

- **Bounded waiting**: if a process is in entry, it is in critical section before a bounded number of times that other processes in critical section
Problem Definition Remark

• An implied requirement: the mutex algorithm is entirely implemented in entry & exit
  – Remainder (non-critical) section is unchanged app code

• Token ring and certain other practical algorithms disqualified
  – Cannot expect a process to participate in mutex if it is uninterested
Token Ring Algorithm

```plaintext
var token[n];          // initialized to {1, 0, 0, ... , 0}

// code for process i
while ( token[i] == 0 ) {no-op;}  // not my turn, wait

critical section;

token[i] = 0;
token[i+1] = 1;

remainder section

repeat (possibly)
```
Efficiency Metrics

• A mutex algorithm often infinitely spins on a register, so we will not focus on cost of computation or memory access

• Instead, we will focus on space complexity (e.g., number of registers used)
Outline

- Mutual exclusion problem definition
- Using strong primitives
  - Test-and-Set
  - Atomic queue and Read-Modify-Write
- Using shared registers
  - Using atomic registers: Peterson
  - Using safe registers: Bakery
- Fast Mutex
Test-and-Set

• A **test-and-set** variable $V$ stores a binary value (0 or 1) and supports two (atomic) operations:

  \[
  \text{reset}(V): \quad \text{// set value to 0} \\
  V = 0
  \]

  \[
  \text{test\&set}(V): \quad \text{// set value to 1 and return old value} \\
  \text{tmp} = V \\
  V = 1 \\
  \text{return \ tmp}
  \]
Mutex using Test-and-Set

- Entry: repeat \( t = \text{test\&set}(V) \) until \( t == 0 \)

- Exit: \( \text{reset}(V) \)

- Intuition: when multiple processes compete, only one process wins (sees \( V=0 \))
Mutual Exclusion (Safety)

• Proof: Consider the first time mutual exclusion is violated: proc $p_j$ enters Critical Section (CS) when proc $p_i$ is already in CS

$p_i$ enters CS:
sees $V = 0$,
sets $V$ to 1

$p_j$ enters CS:
sees $V = 0$, impossible!
sets $V$ to 1

no process leaves CS
(because of first violation), so $V$ stays 1
Deadlock Free (Liveness)

- Lemma: \( V = 0 \) iff no process in critical section
  - Successful entry \( \rightarrow \) Exit \( \rightarrow \) Successful entry \( \rightarrow \) Exit …
  - \( V: 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \) …

- Suppose deadlocks, process \( i \) in entry but no process enters CS ever after
  - Eventually, process in CS exits \( \rightarrow V = 0 \) (by Lemma)
  - Process \( i \) enters, contradiction

- How about starvation freedom? No.
Mutex using Atomic Queue

- **Entry:** enqueue(Q, i)  // code for process I
  while ( head(Q) != i ) no-op;

- **Exit:** dequeue(Q)

- First-come-first-serve, best fairness possible
  - Satisfy starvation free and bounded waiting

- Atomic queue feels like a very strong primitive
Read-Modify-Write (RMW)

- Supports regular read
- Supports RMW(V, f): in one atomic step
  - Read current value
  - Compute certain function(s) of current value
  - Update value

\[
\text{tmp} = V; \\
V = f(V); \\
\text{return } V;
\]
Mutex using RMW

• \( V = (\text{head}, \text{tail}) \)  // initially equal

• \( \text{enqueue}(V) = (V.\text{head}, V.\text{tail}+1) \)

• \( \text{dequeue}(V) = (V.\text{head}+1, V.\text{tail}) \)

• Entry: \( \text{pos} = \text{RMW}(V, \text{enqueue}) \)
  while \( (V.\text{head} \neq \text{pos}.\text{tail}) \) no-op;

• Exit: \( \text{RMW}(V, \text{dequeue}) \)
Mutex using RMW Proof & Remark

• Mutual exclusion (safety) proof:
  – Each process has a unique pos.tail
  – Only the proc whose pos.tail == V.head can be in CS

• Liveness/fairness proof:
  – Bounded waiting: pos.tail – V.head

• Remark: did not actually implement a queue, since no data is stored; weaker primitive than atomic queue, available in real processors
Outline

• Mutual exclusion problem definition

• Using strong primitives
  – Test-and-Set
  – Atomic queue and Read-Modify-Write

• Using shared registers
  – Using atomic registers: Peterson
  – Using safe registers: Bakery

• Fast Mutex
Mutex using Atomic Registers

- Simplest mutex algorithm by Peterson in 1981
- For 2 procs only, can be extended to $n$ procs
- Uses *three* atomic registers
  - Two single-writer two-reader: `want[]`
  - One two-writer two-reader: `turn`
Peterson Algorithm

- Process 0
  // entry
  want[0] = true
  turn = 1; // you go first
  while (turn == 1 && want[1] == true)
    no-op; // wait

  critical section

  // exit
  want[0] = false

- Process 1
  // entry
  want[1] = true
  turn = 0; // you go first
  while (turn == 0 && want[0] == true)
    no-op; // wait

  critical section

  // exit
  want[1] = false
Peterson Safety Proof

- Consider the first time mutual exclusion is violated: proc \( p_j \) enters Critical Section (CS) when proc \( p_i \) is already in CS
Peterson Fairness Proof

• Peterson lock achieves bounded waiting

• Proof: $p_i$ stuck in entry only if it sees
  \[\text{want}[j] == \text{true} \land \land \text{turn} = j\]

• $p_j$ enters or is already in CS, eventually exits

• $p_j$ in entry again, sets turn = i

• $p_i$ enters CS
Tournament Tree

- From 2-process mutex to $n$-process mutex
- Space complexity: $3(n-1)$ Boolean atomic registers
Bakery Algorithm

• Lamport, 1974
• Solves $n$-process mutex
• Uses $2n$ single-writer *safe* registers

• Intuition: each customer gets ticket in entry, smallest ticket gets served first
  – DMV algorithm may relate better for U.S.
Bakery Algorithm

var choosing[n], number[n];  // one per process, initialized to 0
// entry code for process i
choosing[i] = true;
number[i] = 1 + max(number[1], number[2], … number[n]);
choosing[i] = false;
for j = 1:n  // wait for everyone who may come before me
    while ( choosing[j] ) no-op;
    while ( number[j] != 0 && ( number[j], j ) < ( number[i], i ) ) no-op;
end for
critical section;
number[i] = 0;  // exit
Bakery Safety Proof

• Lemma 1: If $p_i$ in CS, then $\text{number}[i] > 0$
  – Straightforward, no other process writes $\text{number}[i]$

• Lemma 2: If $p_i$ in CS, then for all $j \neq i$, either $\text{number}[j] == 0$ or $(\text{number}[j], j) > (\text{number}[i], i)$
  – $p_i$ saw the condition held
  – If $p_i$ saw the latter was true, it will remain true until
    • $p_j$ resets $\text{number}[j]$ to 0
    • Next time $p_j$ chooses $\text{number}[j] > \text{number}[i]$
  – Can focus on the other case (next slide)
Bakery Safety Proof

• Lemma 2: If \( p_i \) in CS, then for all \( j \neq i \), either \( \text{number}[j] == 0 \) or \( (\text{number}[j], j) > (\text{number}[i], i) \)

- \( p_i \) finishes choosing \( \text{number}[i] \)
- \( p_i \) sees \( \text{choosing}[j] = \text{false} \)
- \( p_i \) sees \( \text{number}[j] = 0 \)
- \( p_i \) enters CS

\( p_j \) sets \( \text{number}[j] > 0 \)

a stable 0
or
a transient 0
(overlapping write to safe register)
Bakery Safety Proof

- Lemma 2: If $p_i$ in CS, then for all $j \neq i$, either $\text{number}[j] == 0$ or $(\text{number}[j], j) > (\text{number}[i], i)$

$p_i$ finishes choosing number[i]  
$p_i$ sees choosing[j] = false  
$p_i$ sees number[j] = 0  
$p_i$ enters CS

$p_j$ is in remainder  
or  
$p_j$ is choosing a number

$p_j$ sees number[i] and chooses number[j] > number[i]

$p_j$ sets number[j] > 0

$p_j$ choosing number[j] in one of these two windows
Bakery Safety Proof

• Lemma 1: If $p_i$ in CS, then $\text{number}[i] > 0$
  – Straightforward, no other process writes $\text{number}[i]$

• Lemma 2: If $p_i$ in CS, then for all $j \neq i$, either $\text{number}[j] == 0$ or $(\text{number}[j], j) > (\text{number}[i], i)$

• If $p_i$ and $p_j$ are both in CS, then $\text{number}[i]$ and $\text{number}[j]$ are both positive, and
  $(\text{number}[j], j) \succ (\text{number}[i], i)$
Bakery Fairness Proof

- Starvation freedom: eventually, every $p_j$ with a smaller (number[j], j) enters and exits CS

- Bounded waiting: n
Bakery Algorithm Pros and Cons

• Use weak (single-writer, safe) registers
  – Historic significance: first mutex solution without assuming lower-level atomicity
    • Atomic ≈ mutex
    • Atomic register ≈ mutex for read/write
    • Exercise: where did Peterson rely on atomicity?
      – Modern view: atomic register expensive to build

• Infinite-sized variables number[]
  – Possible (but very hard) to avoid
  – Not an issue in practice
Outline

• Mutual exclusion problem definition

• Using strong primitives
  – Test-and-Set
  – Atomic queue and Read-Modify-Write

• Using shared registers
  – Using atomic registers: Peterson
  – Using safe registers: Bakery

• Fast Mutex
Fast Mutex [Lamport, 1987]

- In the two n-process mutex algorithms we’ve seen so far (tournament tree & bakery), a proc spends $O(\log n)$ or $O(n)$ time before entering CS even when there is no contention
Fast Mutex [Lamport, 1987]

• Fast mutex: $O(1)$ time if no contention

• Must use multi-writer registers
  – Each proc must leave some trace of entering CS
  – If each register has a single writer, must read $n$ registers to make sure no process already in CS
Fast Mutex using Splitter

- Idea: fast-forward at most one process (to CS), other procs (if any) run $n$-proc mutex

- A splitter should guarantee
  - At most one winner
  - If a process runs alone, it wins
    - If there is contention, possibly no winner
Fast Mutex using Splitter

Splitter

lose

win

2-proc mutex

play role of $p_0$

play role of $p_1$

$n$-proc mutex

critical section

Borrowed from Jennifer Welch's slides of CSCE 668 at Texas A&M
```javascript
// two MRMW atomic register, re-initialize in exit
var door = "open", winner = -1;

// entry code for process i
winner = i
if (door == "closed") return "lose"
else
    door = "closed"
    if (winner == id) return "win"
    else return "lose"
```
# Splitter Sample Execution

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>winner = 1</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>winner = 2</code></td>
<td></td>
</tr>
<tr>
<td><code>door == open</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>door == open</code></td>
<td></td>
</tr>
<tr>
<td><code>close door</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>door = closed</code></td>
<td></td>
</tr>
<tr>
<td><code>winner == 2</code> &amp; <code>lose</code></td>
<td><code>winner == 2</code> &amp; <code>win</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>winner = 3</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>door == closed</code> &amp; <code>lose</code></td>
</tr>
</tbody>
</table>

Borrowed from Jennifer Welch's slides of CSCE 668 at Texas A&M
Splitter Proofs

• Liveness: if $p_i$ executes alone, $p_i$ wins
  – Can easily verify

• Safety: at most one process wins
  – Proof: let $p_i$ be the last process to update `winner` before `door` is set to “closed”; no other $p_j$ can win
    • $p_j$ sees door closed $\implies$ lose
    • $p_j$ sees door open $\implies$ $p_j$ write winner before $p_i$ $\implies$ $p_j$ sees a different winner once in the door $\implies$ lose
Remarks

• Exit section must reset splitter

• Modular algorithm, can plug in any 2-proc and n-proc mutex algorithms
  – But if applied to Bakery, lose the advantage of using single-writer safe registers only

• Not adaptive: even if two processes contend, may have to run the expensive n-proc mutex
Mutual Exclusion Summary

• Basic problem in distributed computing

• Practical solutions: test-and-set, RMW

• Theoretically better solutions: Peterson, Tournament tree, Bakery, fast mutex