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

```
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:

```
reset(V):    // set value to 0
    V = 0

test&set(V):        // set value to 1 and return old value
    tmp = V
    V = 1
    return tmp
```

Mutex using Test-and-Set

- Entry: repeat t = test&set(V) until (t == 0)
- Exit: 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 → Exit → Successful entry → Exit …
  - V: 0 → 1 → 0 → 1 → 0 …

- Suppose deadlocks, process i in entry but no process enters CS ever after
  - Eventually, process in CS exits → 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

```
tmp = V;
V = f(V);
return V;
```

Mutex using RMW

- V = (head, tail)   // initially equal
- enqueue(V) = (V.head, V.tail+1)
- dequeue(V) = (V.head+1, V.tail)

- Entry: pos = RMW(V, enqueue)
  while (V.head != pos.tail) no-op;

- Exit: RMW(V, 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

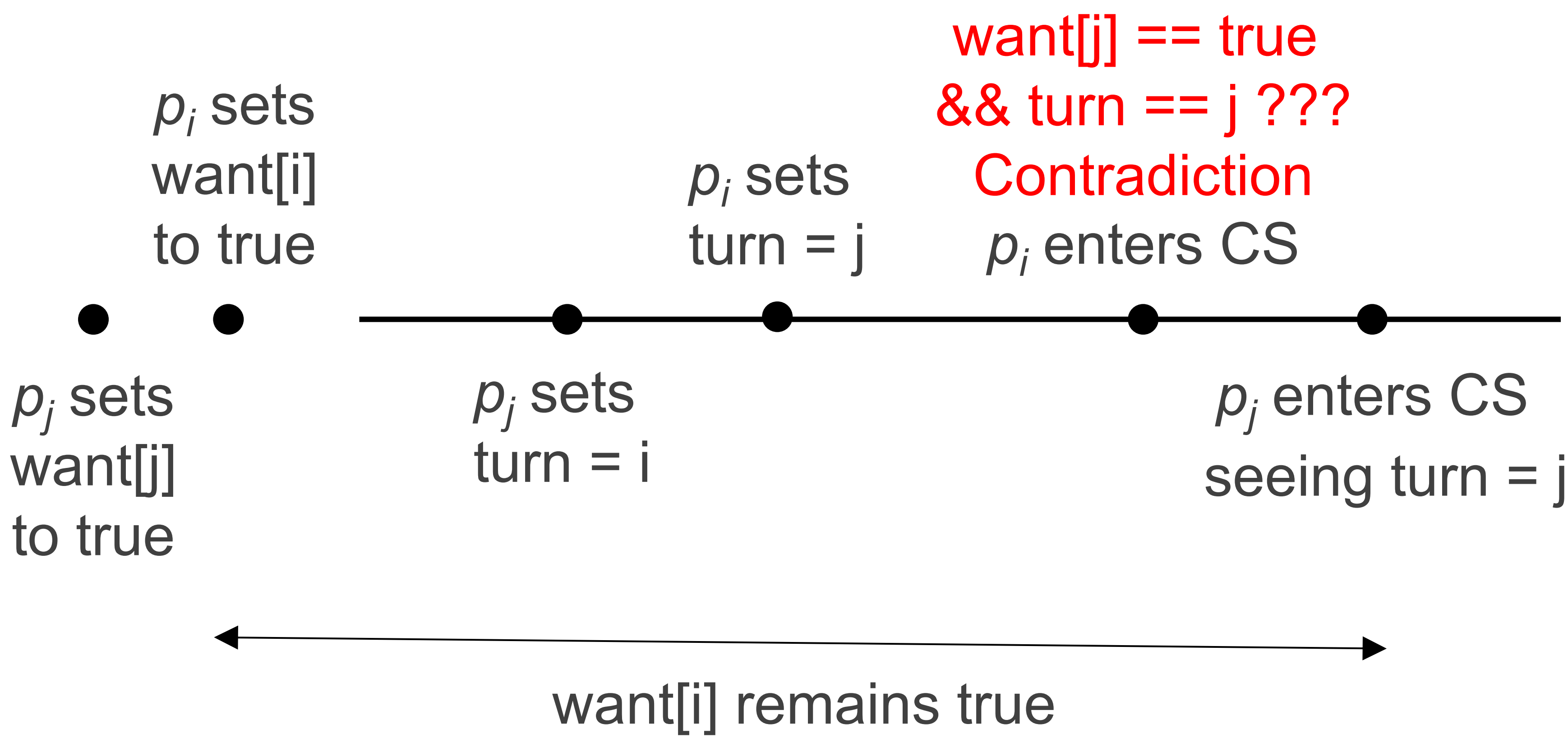
```
// entry
want[1] = true
turn = 0; // you go first
while (turn == 0 &&
      want[0] == true)
      no-op;        // wait

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

  want[j] == true && turn = j

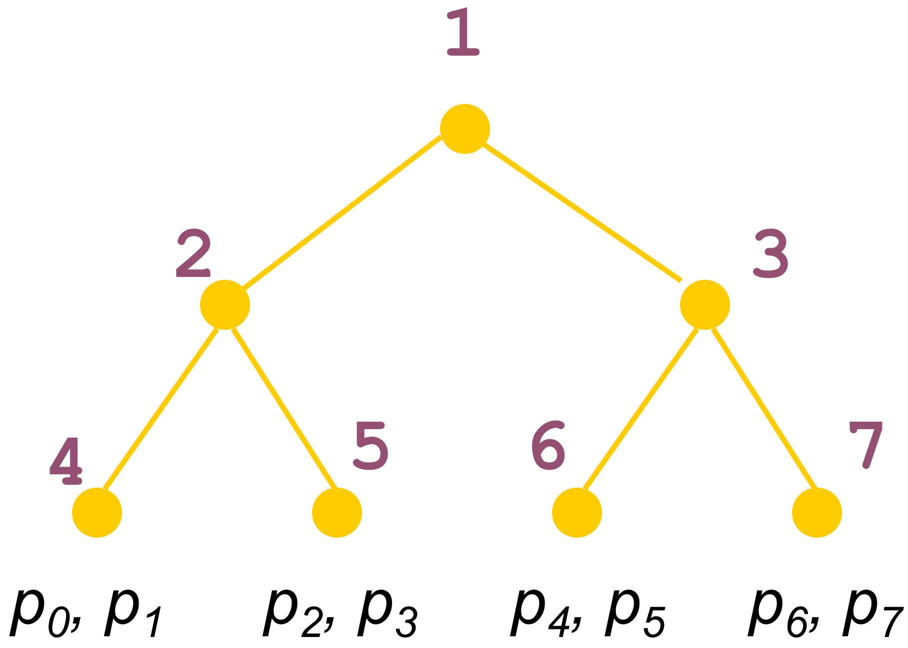
- p_j enters or is already in CS, eventually exits

- 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
```

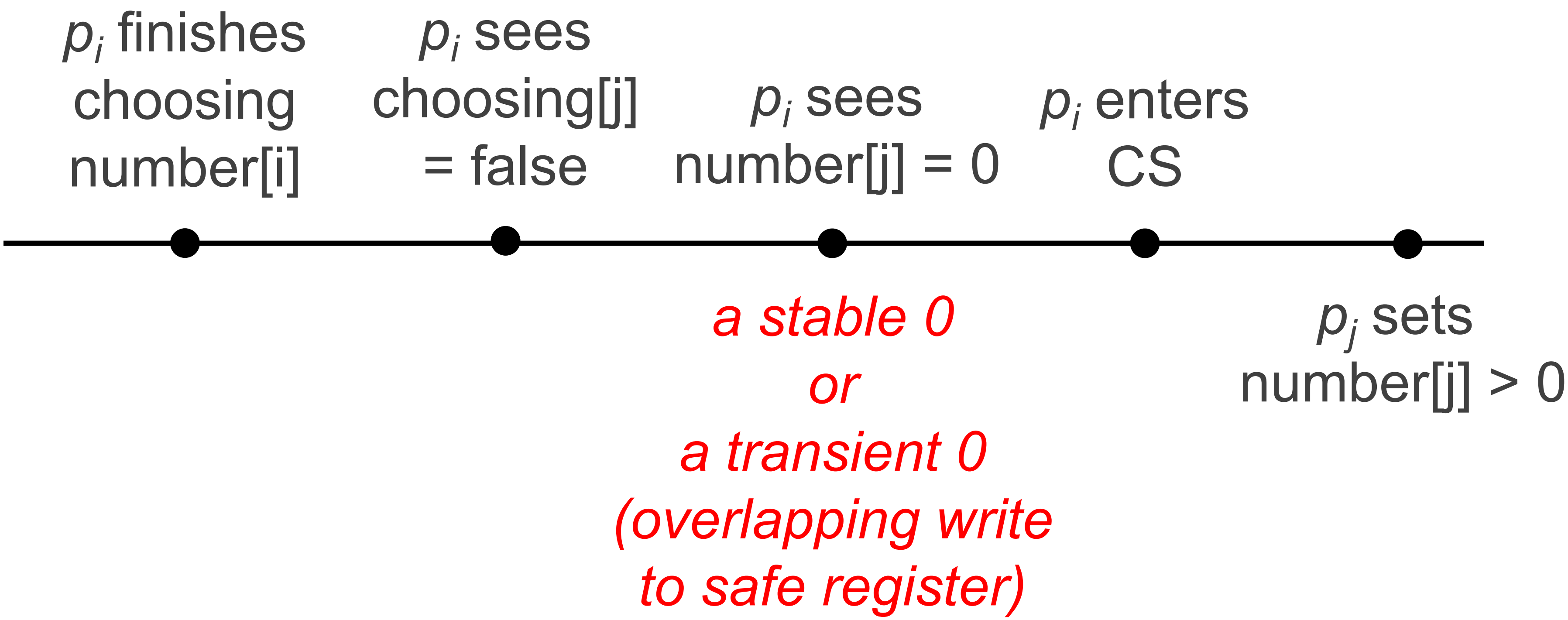
Bakery Safety Proof

- Lemma 1: If p_i in CS, then number[i] > 0
  - Straightforward, no other process writes number[i]

- Lemma 2: If p_i in CS, then for all j ≠ i, either number[j] == 0 or (number[j], j) > (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 number[j] to 0
    - Next time p_j chooses number[j] > number[i]
  - Can focus on the other case (next slide)

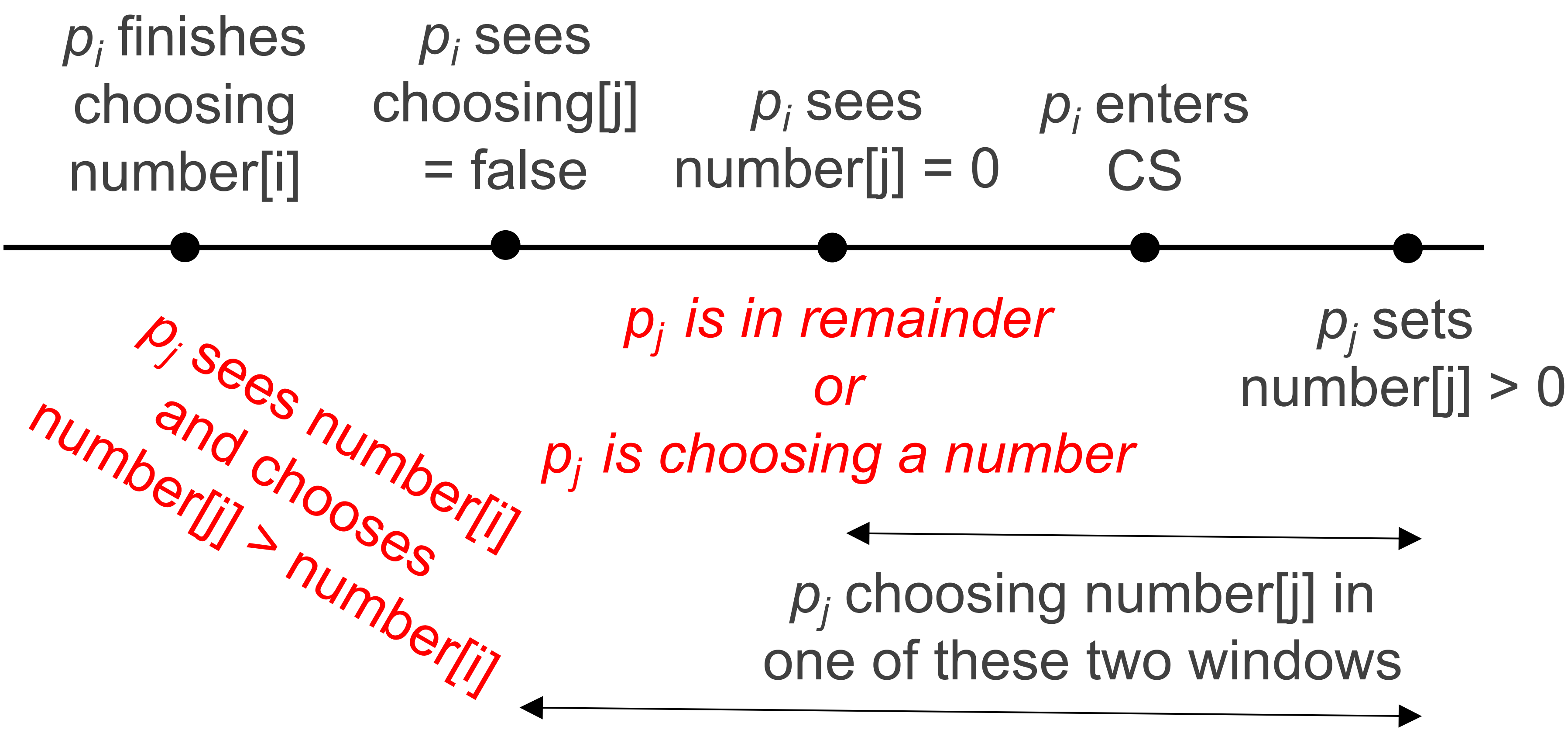
Bakery Safety Proof

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



Bakery Safety Proof

- Lemma 2: If p_i in CS, then for all j ≠ i, either number[j] == 0 or (number[j], j) > (number[i], i)



Bakery Safety Proof

- Lemma 1: If p_i in CS, then number[i] > 0
  - Straightforward, no other process writes number[i]

- Lemma 2: If p_i in CS, then for all j ≠ i, either number[j] == 0 or (number[j], j) > (number[i], i)

- If p_i and p_j are both in CS, then number[i] and number[j] are both positive, and

  (number[j], j) >< (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]

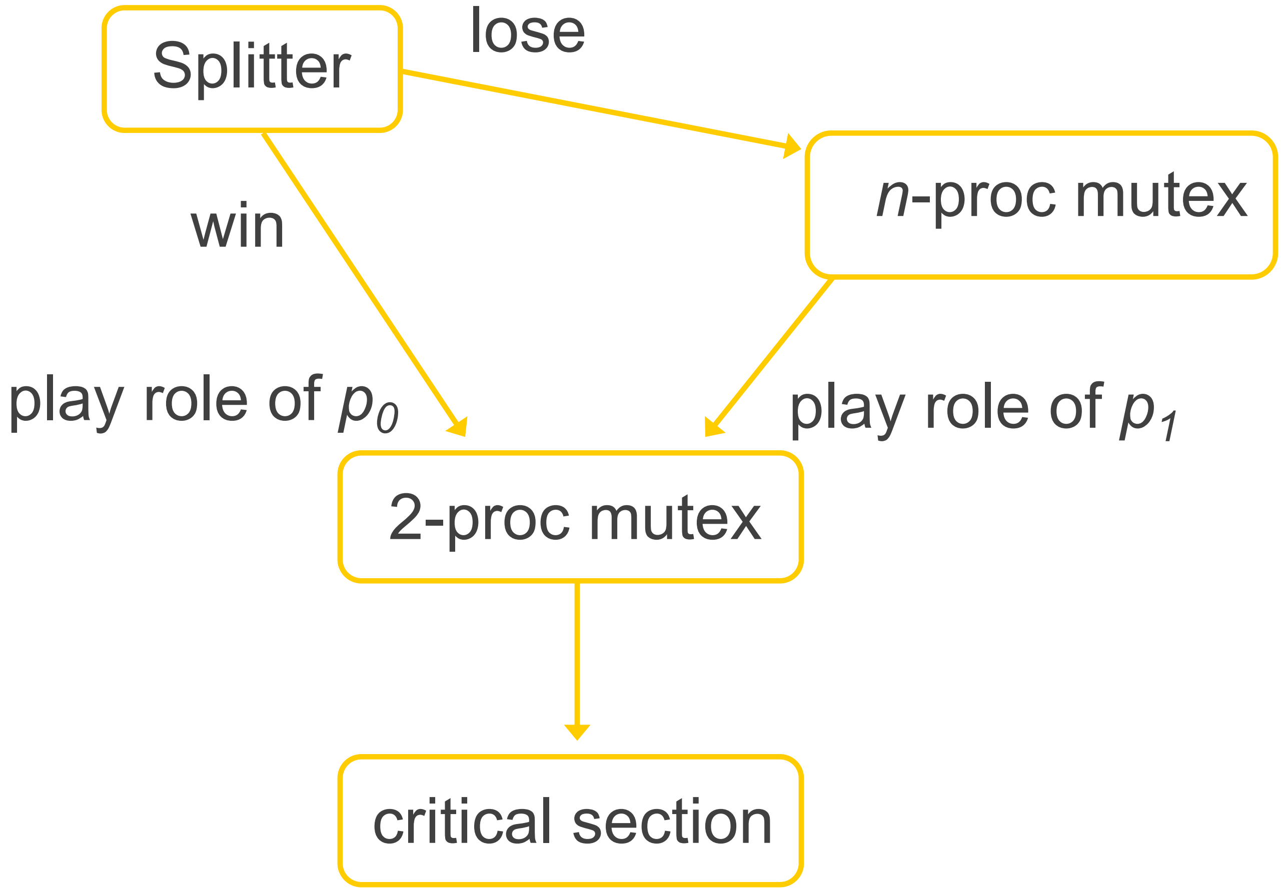
- Fast mutex: O(1) time if no contention
- Must use multi-writer registers
  - Each proc must leave some trace of entering CS
  - If each register has a single writer, must read n registers to make sure no process already in CS

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 [Moir-Anderson, 1995]

```
// 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

p_1	p_2	p_3
`winner = 1`		
	`winner = 2`	
`door == open`		
	`door == open`	
`close door`		
	`door = closed`	
`winner == 2` `& lose`	`winner == 2` `& win`	
		`winner = 3`
		`door == closed` `& lose`

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 → lose
    - p_j sees door open → p_j write winner before p_i → p_j sees a different winner once in the door → 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