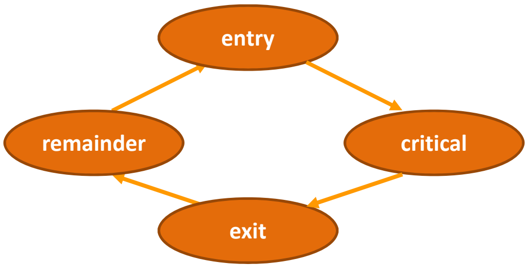
236755
Topic 2: Mutual Exclusion

Winter 2019-20

Prof. Hagit Attiya

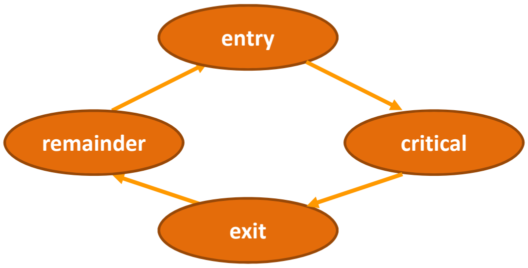
Mutual Exclusion (Mutex) Problem



Each process's code is divided into four sections:

- **remainder**: not interested in using the resource, go to…
- **entry**: synchronize with others to ensure mutually exclusive access to the …
- **critical**: use some resource; when done, enter the…
- **exit**: clean up; when done, go back to the remainder

Mutex Algorithm



Specifies code for entry and exit sections to ensure:

- **safety:** at most one process is in its critical section at any time (**mutual exclusion**), and
- some **liveness** or **progress** condition

Liveness Properties for Mutex Algorithms

no deadlock: if a process is in its entry section at some time, then later **some** process is in its critical section

no starvation: if a process is in its entry section at some time, then later the **same** process is in its critical section

bounded waiting: no deadlock + while a process is in its entry section, other processes enter the critical section no more than a certain number of times

Mutex using Test&Set

test-and-set variable holds two values, 0 or 1, and provides two (atomic) operations

```
test&set(V):
   temp = V
   V = 1
   return temp
```

```
reset(V):
   V = 0
```

Code for entry section:

```
repeat
    t = test&set(V)
until (t == 0)
```

Or

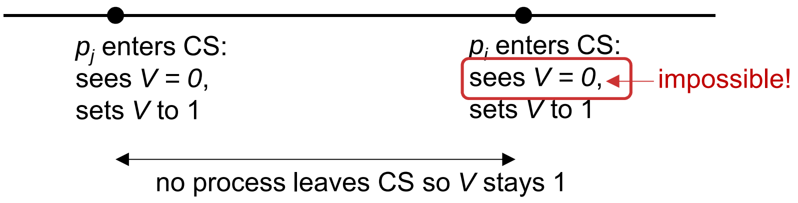
```
wait until test&set(V) == 0
```

Code for exit section:

```
reset(V)
```

T&S Algorithm Ensures Mutual Exclusion

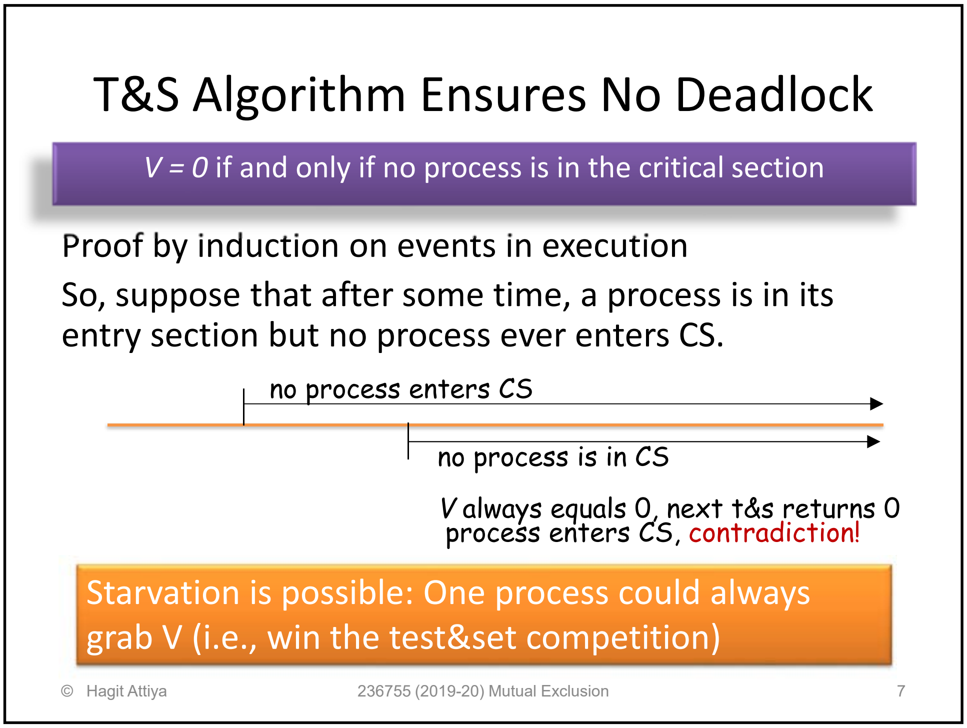
Otherwise, consider first violation, when some p_i enters CS but another p_j is already in CS



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

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

no process leaves CS so V stays 1



T&S Algorithm Ensures No Deadlock

V = 0 if and only if no process is in the critical section

Proof by induction on events in execution

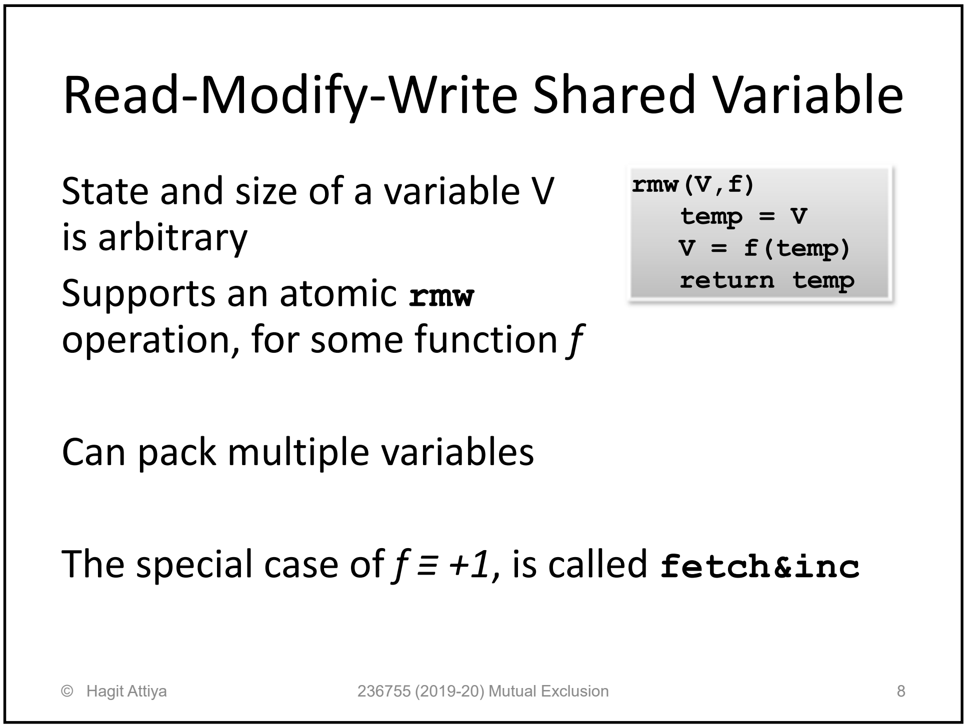
So, suppose that after some time, a process is in its entry section but no process ever enters CS.

no process enters CS

no process is in CS

V always equals 0, next t&s returns 0
process enters *CS*, contradiction!

Starvation is possible: One process could always grab V (i.e., win the test&set competition)



Read-Modify-Write Shared Variable

State and size of a variable V is arbitrary

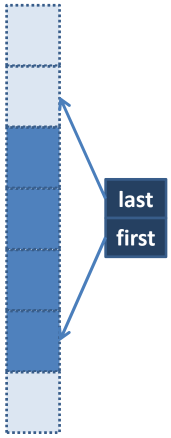
Supports an atomic **`rmw`** operation, for some function *f*

```
rmw(V,f)
    temp = V
    V = f(temp)
    return temp
```

Can pack multiple variables

The special case of *f ≡ +1*, is called **`fetch&inc`**

Overview of Algorithm



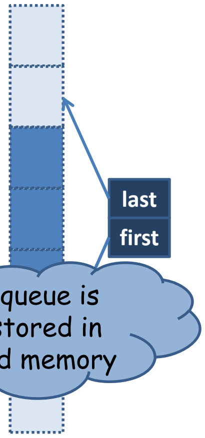
Virtually, processes wait in a circular queue of length *n*

Waiting process locally stores its position in the queue

Shared pointers *first* and *last* track the active part of the queue
  - Indices between 0 and *n-1*
  - Packed into one shared variable V

Space complexity
- V has n^2 states
- size of V is $2\log_2 n$ bits

Mutex Algorithm Using RMW



Code for entry section:

```
// increment last to enqueue self
position = rmw(V,(V.first,V.last+1 mod n)

// wait until first equals this value
repeat
    queue = rmw(V,V)
until (queue.first == position.last)
```

The queue is not stored in shared memory

Code for exit section:

```
// dequeue self
rmw(V,(V.first+1 mod n,V.last))
```

Sketch of Correctness Proof

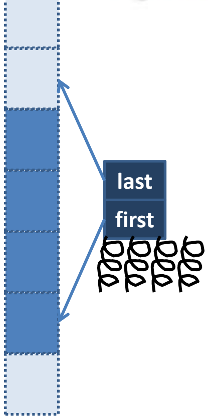
- **Mutual Exclusion:**
  - Only the process at the head of the queue (*V.first*) can enter the CS, and only one process is at the head at any time.
- **FIFO order:**
  - Follows from FIFO order of enqueuing, and since no process stays in CS forever.

Spinning

Processes in entry section repeatedly access V (**spinning**)

Very time-inefficient in certain multiprocessor architectures

Local spinning: each waiting process spins on a different shared variable

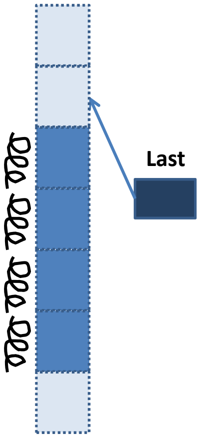


RMW Mutex Algorithm w/ Local Spinning

Shared RMW variables

Last cycles through 0 … n–1

- tracks the index to be given to the next process that starts waiting
- initially 0



RMW Mutex Algorithm w/ Local Spinning

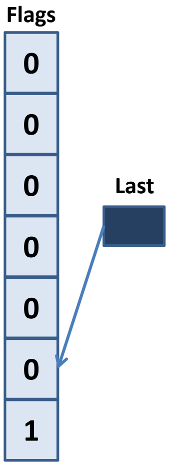
Shared RMW variables

Last cycles through 0 … n–1

- tracks the index to be given to the next process that starts waiting
- initially 0

Flags[0..n-1]: array of binary variables

- processes spin on these variables
- no two processes spin on the same variable at the same time
- initially Flags[0] is 1 ("has lock")
  Flags[i] is 0 ("must wait”) for i > 0



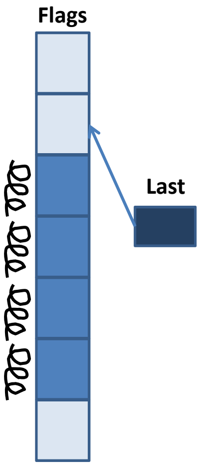
RMW Mutex Algorithm w/ Local Spinning

entry section:

- get next index from Last and store in a local variable *myPlace*
  - increment Last (with wrap-around)
- spin on Flags[myPlace] until = 1 (means process "has lock" and can enter CS)
- set Flags[myPlace] to 0 (“must wait")

exit section:

- set Flags[myPlace+1] to 1 (“has lock”) (i.e., tap next process in line)
  - use modulo to wrap around



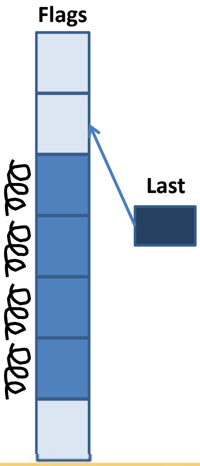
RMW Mutex Algorithm w/ Local Spinning

entry section:

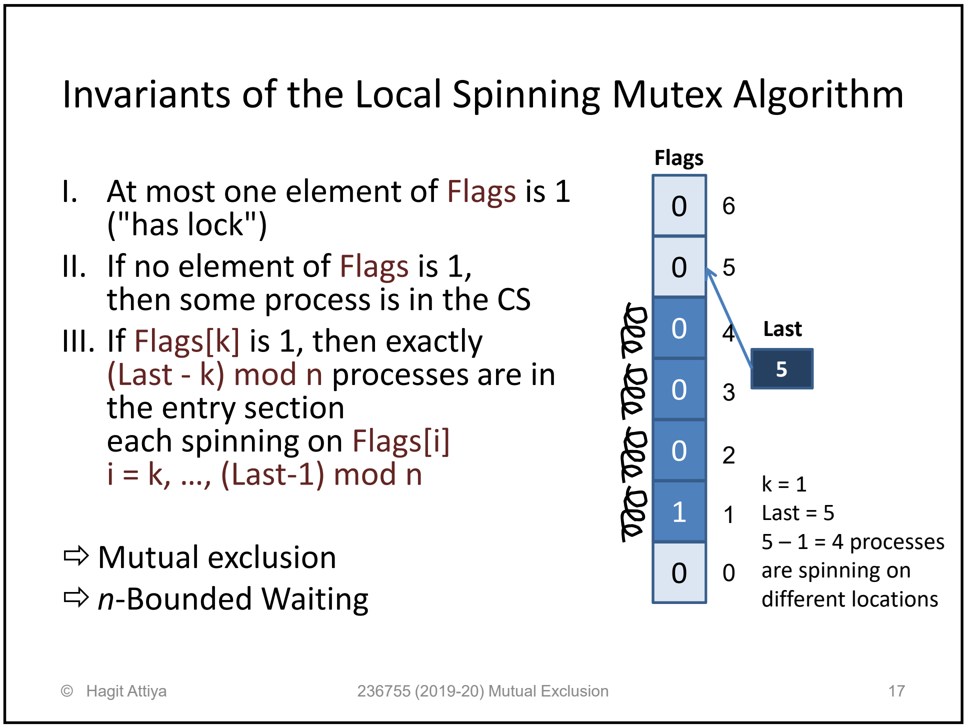
```
myPlace = rmw(Last,Last+1 mod n)
wait until Flags[myPlace] == 1
Flags[myPlace] = 0
```

exit section:

```
Flags[myPlace+1 mod n] = 1
```



Must apply RMW on last to ensure counter is correct



Invariants of the Local Spinning Mutex Algorithm

I. At most one element of Flags is 1 ("has lock")

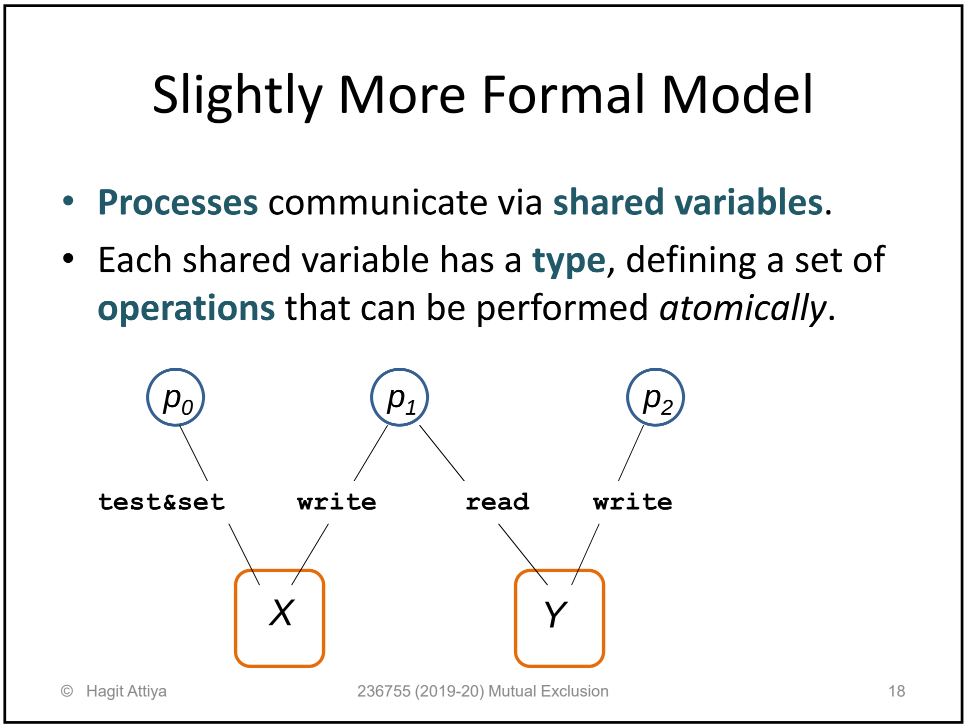
II. If no element of Flags is 1, then some process is in the CS

III. If Flags[k] is 1, then exactly (Last - k) mod n processes are in the entry section each spinning on Flags[i] i = k, …, (Last-1) mod n

⇨ Mutual exclusion

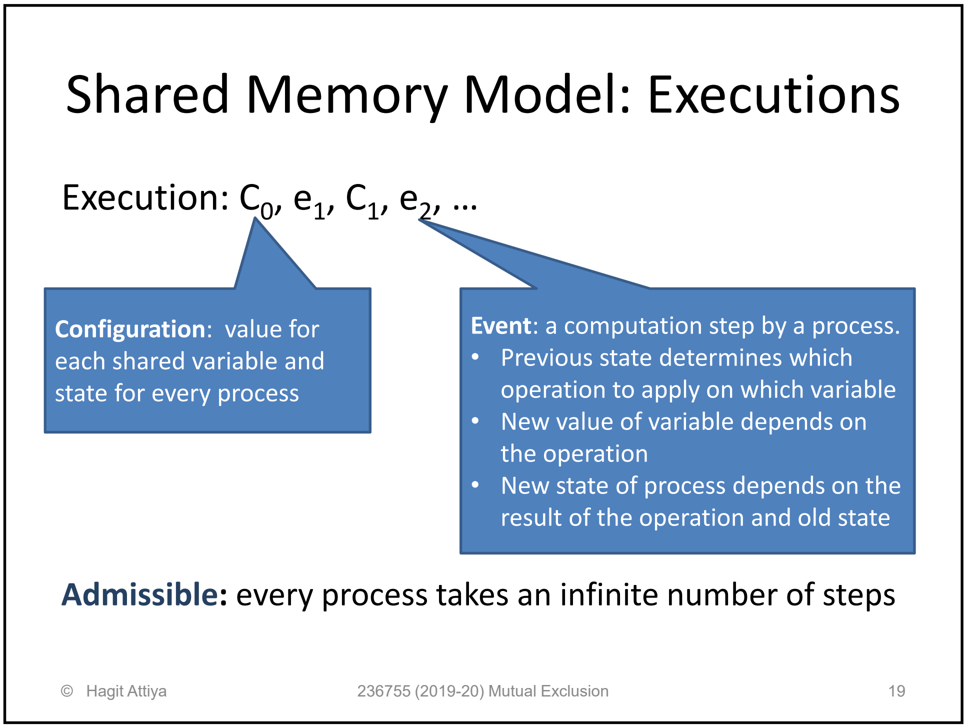
⇨ *n*-Bounded Waiting

k = 1
Last = 5
5 – 1 = 4 processes are spinning on different locations



Slightly More Formal Model

- **Processes** communicate via **shared variables**.
- Each shared variable has a **type**, defining a set of **operations** that can be performed *atomically*.



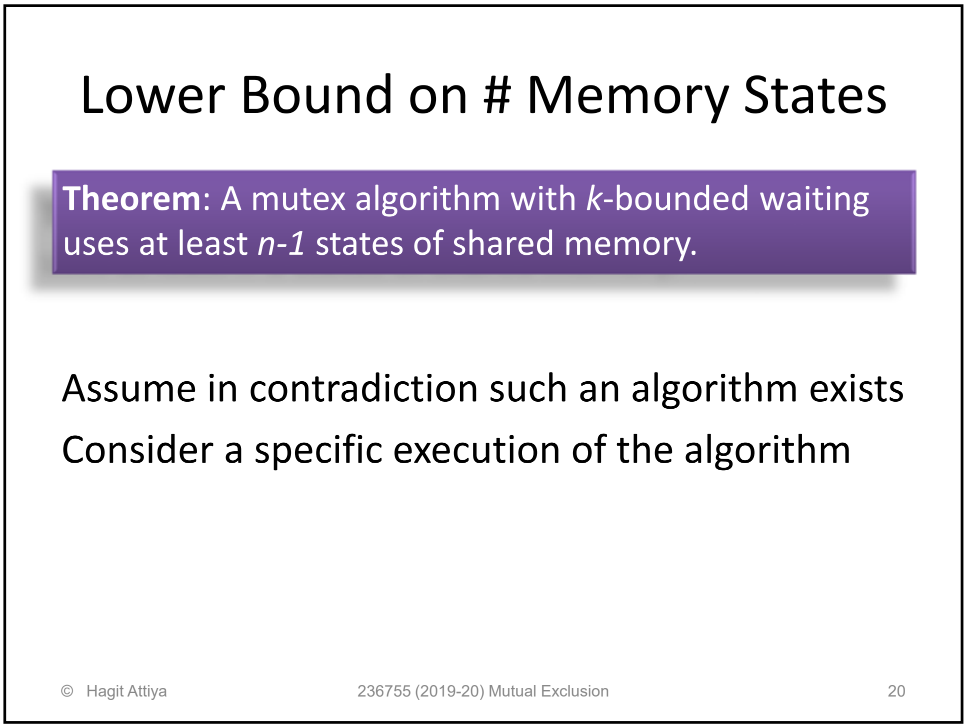
Shared Memory Model: Executions

Execution: C_0, e_1, C_1, e_2, …

Configuration: value for each shared variable and state for every process

Event: a computation step by a process.
- Previous state determines which operation to apply on which variable
- New value of variable depends on the operation
- New state of process depends on the result of the operation and old state

Admissible: every process takes an infinite number of steps



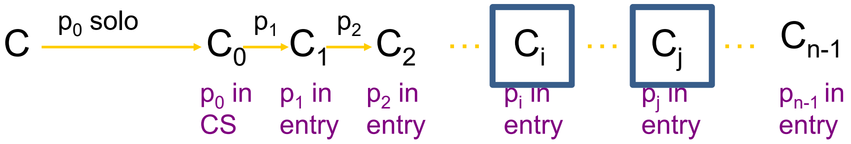
Lower Bound on # Memory States

Theorem: A mutex algorithm with *k*-bounded waiting uses at least *n-1* states of shared memory.

Assume in contradiction such an algorithm exists

Consider a specific execution of the algorithm

Lower Bound on # Memory States



$C \xrightarrow{p_0 \text{ solo}} C_0 \xrightarrow{p_1} C_1 \xrightarrow{p_2} C_2 \cdots \boxed{C_i} \cdots \boxed{C_j} \cdots C_{n-1}$

p_0 in CS

p_1 in entry

p_2 in entry

p_i in entry

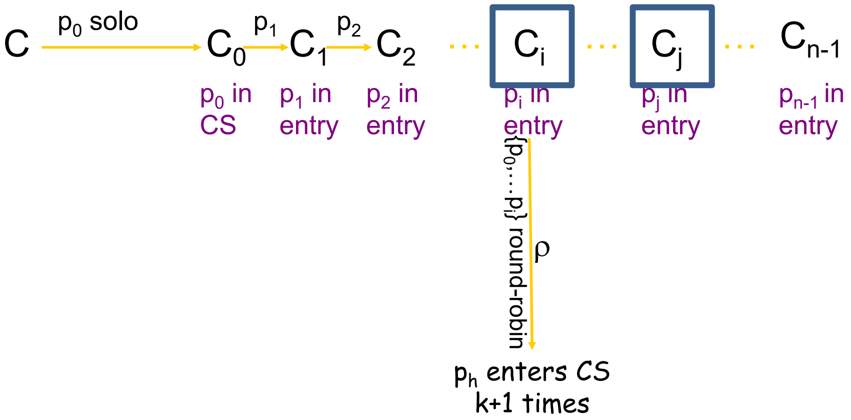
p_j in entry

p_{n-1} in entry

If # memory states < n-1

For some *i* < *j* the shared memory is in the same state in C_i and C_j

Lower Bound on # Memory States



$C \xrightarrow{p_0 \text{ solo}} C_0 \xrightarrow{p_1} C_1 \xrightarrow{p_2} C_2 \cdots \boxed{C_i} \cdots \boxed{C_j} \cdots C_{n-1}$

p_0 in CS

p_1 in entry

p_2 in entry

p_i in entry

p_j in entry

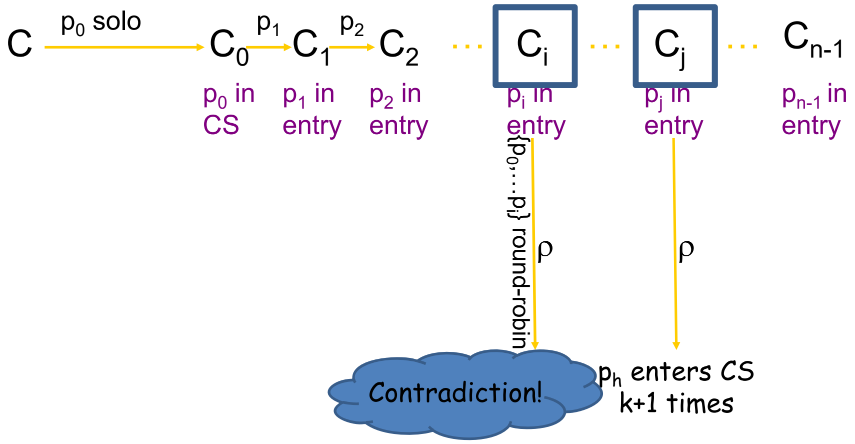
p_{n-1} in entry

$\{p_0 \ldots p_i\}$ round-robin

ρ

p_h enters CS k+1 times

Lower Bound on # Memory States



Lower Bound: Afterthoughts

Why $p_0,\ldots,p_i$ (and especially p_h) do the same thing when executing from C_j as when executing from C_i?

- they are in the same states in C_j and C_i
- the shared memory is the same in C_j and C_i
- only differences between C_i and C_j are (perhaps) the states of $p_{i+1},\ldots,p_j$
  and they don't take any steps in ρ

☝ **Indistinguishability**

Lower Bound: Afterthoughts

Does the proof work with no starvation?

A more complicated proof shows that number of memory states is $\sqrt{n}$

⇨ Ω(log n) bits

Shared Memory States: Summary

Progress property	Upper bound	Lower bound
no deadlock	2 (test&set alg)	2
no starvation	n/2 + c (Burns et al.)	$\sqrt{n}$
bounded waiting (FIFO)	n^2 (queue)	n-1

Randomization “Beats” the Lower Bound

Reducing the liveness in *every* execution

Probabilistic no-starvation: every process has non-zero probability of getting into the critical section each time it is in its entry section

☞There is a randomized mutex algorithm using *O(1)* states of shared memory

---

Mutex with Read/Write Variables

In an atomic step, a process can

read a variable or
write a variable
but not both!

The Bakery algorithm ensures

no starvation
mutual exclusion

Using *2n* shared read/write variables

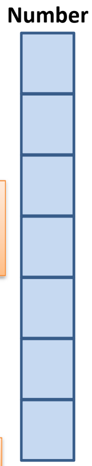
Bakery Algorithm: Take 1

Number[i], integer, initially 0

- written by p_i
- read by others

Code for entry section:

```
Number[i] = 1+max{Number[1],...,Number[n]}
for j = 1 to n do
        wait until Number[j] > Number[i]
```



Code for exit section:

```
Number[i] = 0
```

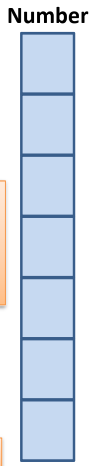
Bakery Algorithm: Take 2

Number[i], integer, initially 0

- written by p_i
- read by others

Code for entry section:

```
Number[i] = 1+max{Number[1],...,Number[n]}
for j = 1 to n do
        wait until (Number[j] == 0)
          or (Number[j],j) > (Number[i],i))
```



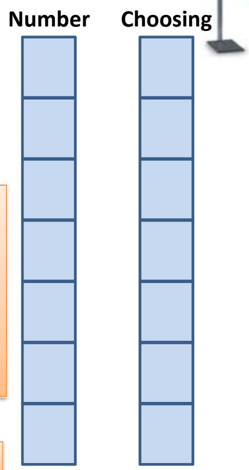
Code for exit section:

```
Number[i] = 0
```

Bakery Algorithm: Take 3

Number[i], integer, initially 0
Choosing[i], Boolean, initially false

- written by p_i
- read by others



Code for entry section:

```
Choosing[i] = true
Number[i] = 1+max{Number[1],...,Number[n]}
Choosing[i] = false
for j = 1 to n do
       wait until Choosing[j] == false
       wait until (Number[j] == 0)
         or (Number[j],j) > (Number[i],i))
```

Code for exit section:

```
Number[i] = 0
```

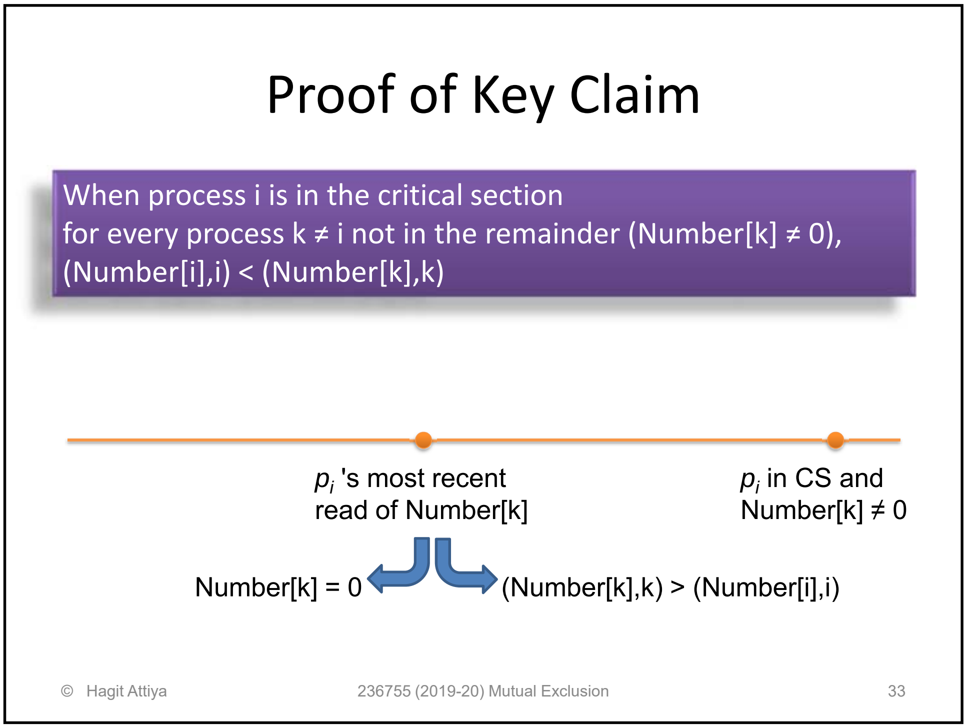
Correctness of Bakery Mutex: Key Claim

When process i is in the critical section
for every process k ≠ i not in the remainder (Number[k] ≠ 0),
(Number[i],i) < (Number[k],k)

Seems intuitive from the code, but is not trivial

This is not exactly the original Bakery algorithm

Everything I need to know about
concurrent programming,
I learned from the Bakery algorithm



Proof of Key Claim

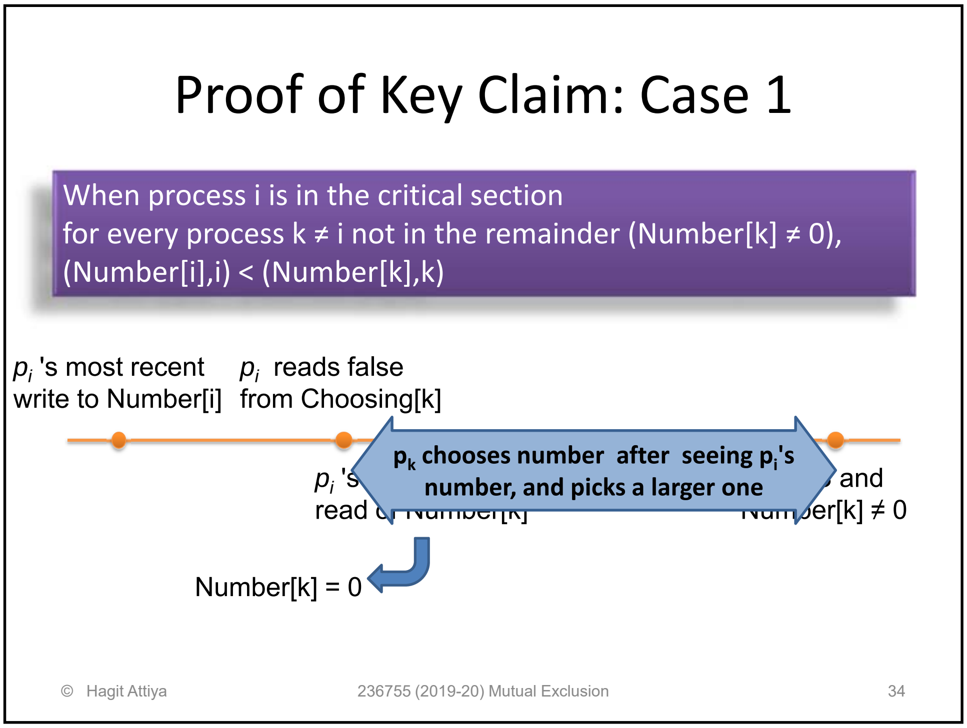
When process i is in the critical section
for every process k ≠ i not in the remainder (Number[k] ≠ 0),
(Number[i],i) < (Number[k],k)

p_i 's most recent
read of Number[k]

p_i in CS and
Number[k] ≠ 0

Number[k] = 0

(Number[k],k) > (Number[i],i)



Proof of Key Claim: Case 1

When process i is in the critical section
for every process k ≠ i not in the remainder (Number[k] ≠ 0),
(Number[i],i) < (Number[k],k)

p_i 's most recent
write to Number[i]

p_i reads false
from Choosing[k]

p_k chooses number after seeing p_i's number, and picks a larger one

p_i 's
read

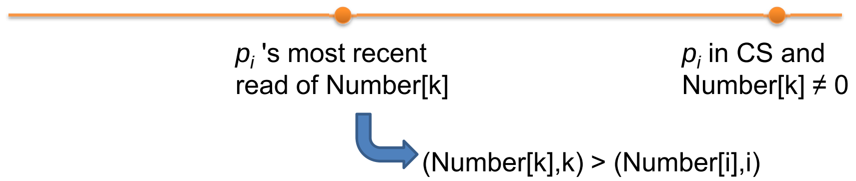
and
er[k] ≠ 0

Number[k] = 0

Proof of Key Claim: Case 2

When process i is in the critical section for every process k ≠ i not in the remainder (Number[k] ≠ 0), (Number[i],i) < (Number[k],k)

Proved using arguments similar to Case 1.



p_i 's most recent read of Number[k]

p_i in CS and Number[k] ≠ 0

(Number[k],k) > (Number[i],i)

Mutual Exclusion for Bakery Algorithm

Lemma: If p_i is in the critical section, then Number[i] > 0.

Proof by straightforward induction.

⇨If p_i and p_k are simultaneously in CS, both have Number > 0.

By previous lemma,

- (Number[k],k) > (Number[i],i) and
- (Number[i],i) > (Number[k],k)

Contradiction!

☞The algorithm ensures mutex

No Starvation for the Bakery Algorithm

Must be waiting on Choosing[] or Number[]

- Let p_i be starved process with smallest (Number[i],i).
- Any process entering entry section after p_i has chosen its number chooses a larger number.
- Every process with a smaller number eventually enters CS (not starved) and exits.
- Thus p_i cannot be stuck on Choosing[] or Number[].

Summary of Mutex Algorithms

Progress property	# memory states	# read / write variables
no deadlock	2 (test&set alg)	1
no starvation	n/2 + c (Burns et al.)	3n Booleans (tournament)
bounded waiting (FIFO)	n^2 (queue)	2n unbounded (bakery)

Flag Principle

Bounded 2-Process Mutex w/o Deadlock

Want[0] 0

Want[1] 0

Entry section

Process P_0

```
Want[0] = 1
wait until Want[1] == 0
```

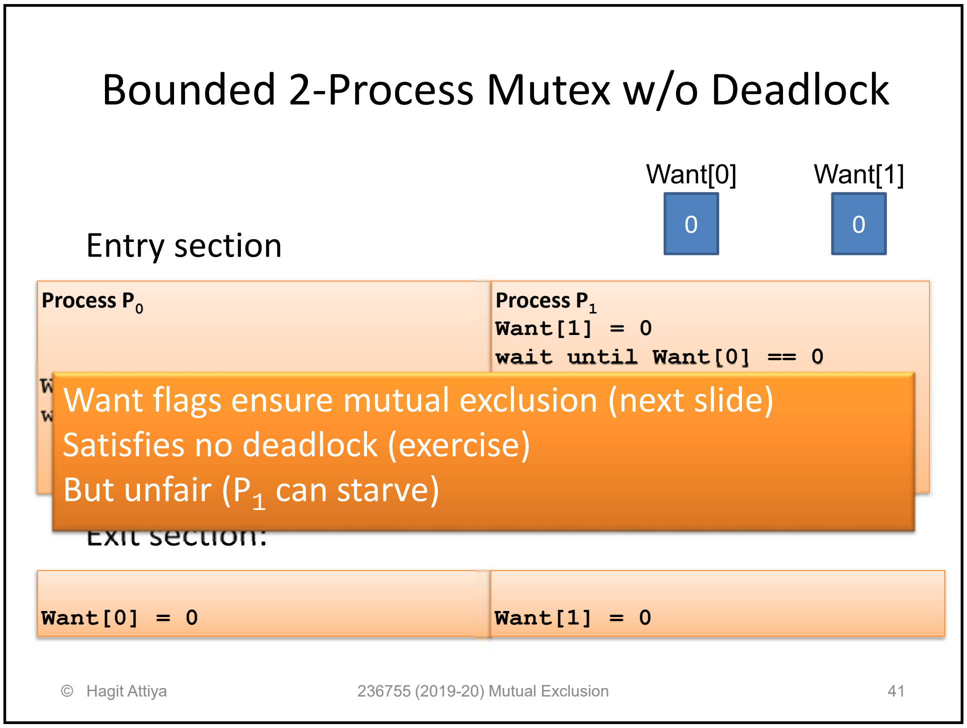
Process P_1

```
Want[1] = 0
wait until Want[0] == 0
Want[1] = 1

if Want[0] == 1 goto Line 1
```

Exit section:

```
Want[0] = 0
```

```
Want[1] = 0
```



Bounded 2-Process Mutex w/o Deadlock

Entry section

Process P_0

```
W
w
```

Process P_1

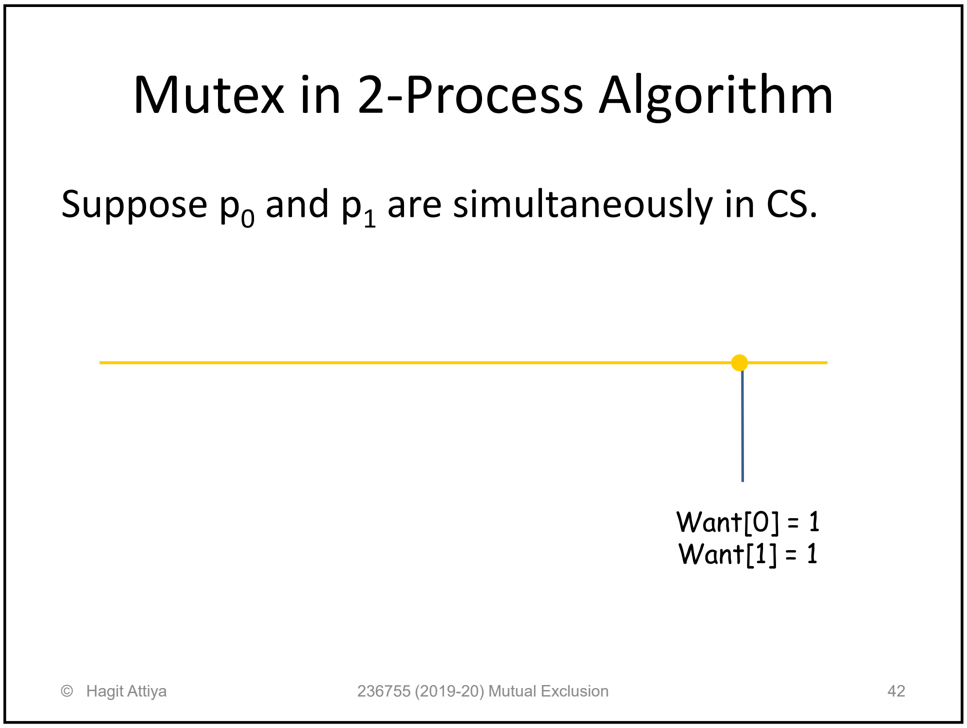
```
Want[1] = 0
wait until Want[0] == 0
```

Want flags ensure mutual exclusion (next slide)
Satisfies no deadlock (exercise)
But unfair (P_1 can starve)

Exit section:

```
Want[0] = 0
```

```
Want[1] = 0
```



Mutex in 2-Process Algorithm

Suppose p_0 and p_1 are simultaneously in CS.

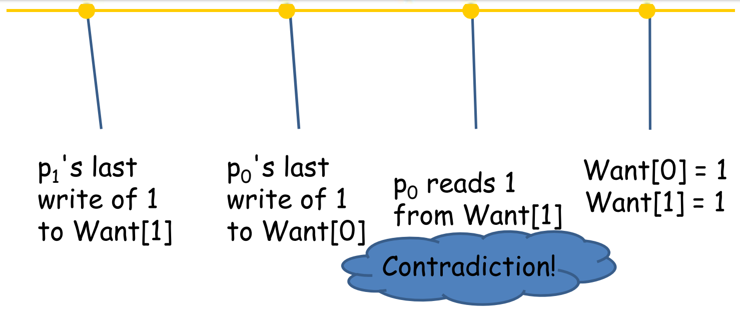
Mutex in 2-Process Algorithm

Process P_0

```
Want[0] = 1
wait until Want[1] == 0
```

Process P_1

```
Want[1] = 0
wait until Want[0] == 0
Want[1] = 1

if Want[0] == 1 goto Line 1
```



p_1's last write of 1 to Want[1]

p_0's last write of 1 to Want[0]

p_0 reads 1 from Want[1]

Want[0] = 1
Want[1] = 1

Contradiction!

Bounded 2-Process Mutex w/o Starvation

Priority	Want[0]	Want[1]
0	0	0

Entry section

```
Want[i] = 0
wait until Want[1-i] == 0 or Priority == i
Want[i] = 1
if (Priority == 1-i) then
     if (Want[1-i] == 1) then goto Line 1
else wait until (Want[1-i] == 0)
```

Exit section:

```
Priority = 1-i
Want[i] = 0
```

No-Deadlock for 2-Process Mutex

- Useful for showing no-starvation.
- If one process stays in remainder forever, other one cannot be starved
  - E.g., if p_1 stays in remainder forever, then p_0 keeps reading Want[1] = 0.
- So any deadlock starves both processes

No-Deadlock for 2-Process Mutex

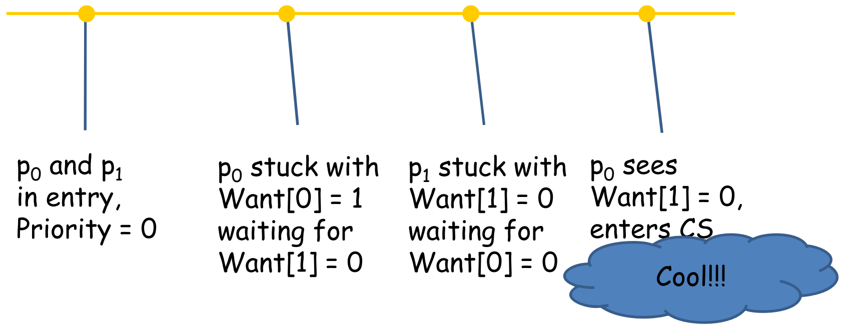
Both processes are in their entry section
Priority remains fixed, e.g. at 0

p_0 and p_1
in entry,
Priority = 0

No-Deadlock for 2-Process Mutex

```
Code for p0
Want[i] = 0
wait until Want[1-i] == 0 or Pri
Want[i] = 1
if (Priority == 1-i) then
     if (Want[1-i] == 1) then go
else wait until (Want[1-i] == 0)
```

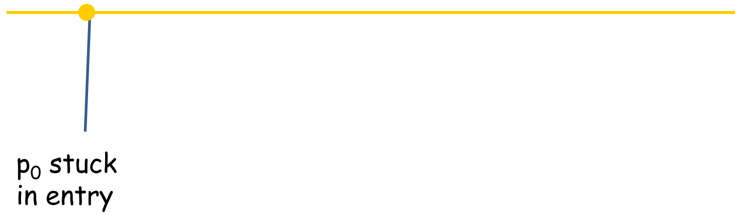
```
Code for p1
Want[i] = 0
wait until Want[1-i] == 0 or Priority == i
Want[i] = 1
if (Priority == 1-i) then
     if (Want[1-i] == 1) then goto Line 1
else wait until (Want[1-i] == 0)
```

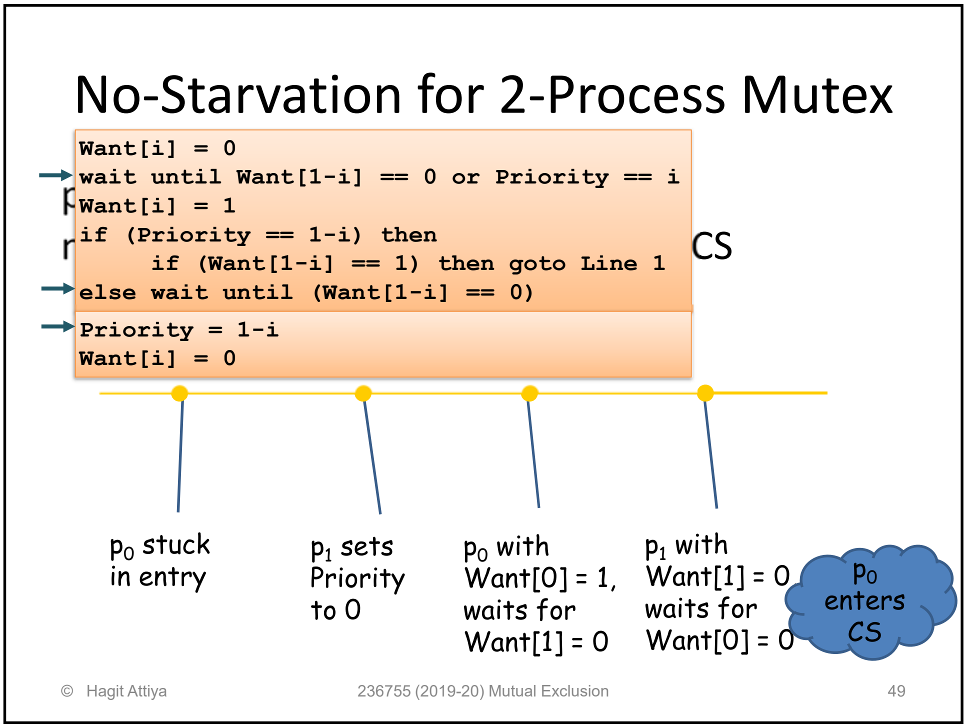


No-Starvation for 2-Process Mutex

p_0 is starved

no deadlock ⇨ p_1 repeatedly enters CS





No-Starvation for 2-Process Mutex

```
Want[i] = 0
wait until Want[1-i] == 0 or Priority == i
Want[i] = 1
if (Priority == 1-i) then
     if (Want[1-i] == 1) then goto Line 1
else wait until (Want[1-i] == 0)
Priority = 1-i
Want[i] = 0
```

p_0 stuck in entry

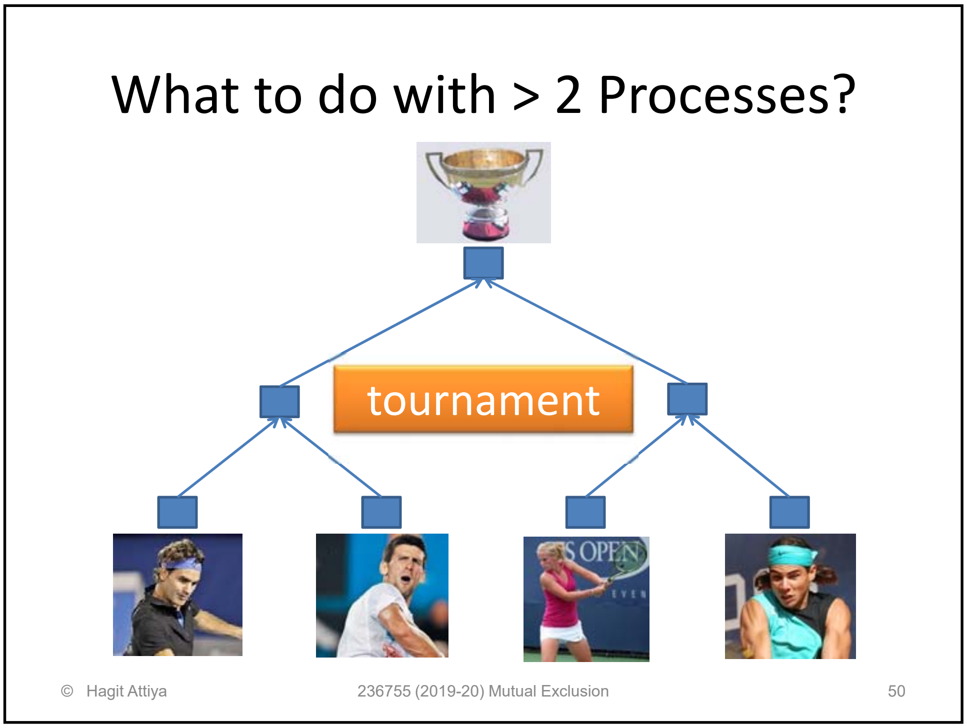
p_1 sets Priority to 0

p_0 with Want[0] = 1, waits for Want[1] = 0

p_1 with Want[1] = 0 waits for Want[0] = 0

p_0 enters CS

© Hagit Attiya 236755 (2019-20) Mutual Exclusion 49



What to do with > 2 Processes?

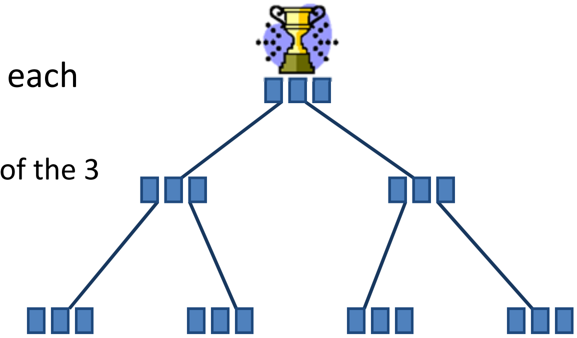
© Hagit Attiya 236755 (2019-20) Mutual Exclusion 50

Tournament Tree Mutex

Tournament tree: complete binary tree with *n-1* nodes

2-process mutex in each inner node

- separate copies of the 3 shared variables



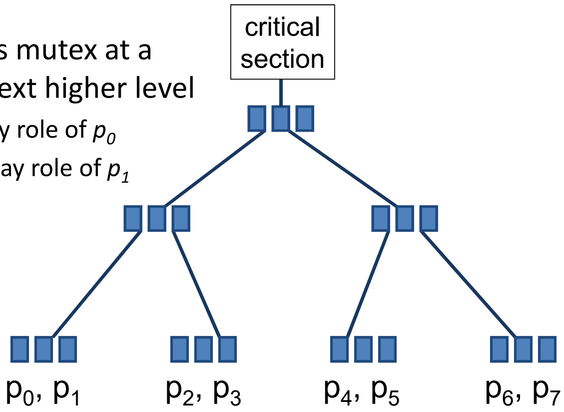
Tournament Tree Mutex

Two (fixed) processes start at each leaf

Winner of the 2-process mutex at a node proceeds to the next higher level

- coming from left, play role of p_0
- coming from right, play role of p_1

Winner at the root enters CS



Tournament Tree Mutex Algorithm

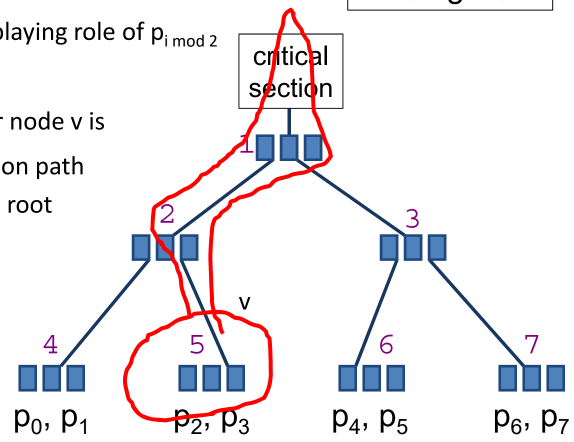
$k = \lceil \log n \rceil - 1$

Tree nodes numbered in preorder

p_i begins at node $2^k + \lfloor i/2 \rfloor$, playing role of $p_{i \bmod 2}$

After winning node v, CS for node v is

- entry code for all nodes on path from v's parent $\lfloor v/2 \rfloor$ to root
- real critical section
- exit code for all nodes on path from root to v's parent $\lfloor v/2 \rfloor$



Analysis of Tournament Tree Mutex

Correctness: based on correctness of 2-process algorithm and tournament structure:

- projection of an admissible execution of tournament algorithm onto a particular node is an admissible execution of 2-process algorithm
- mutex for tournament algorithm follows from mutex for 2-process algorithm at the root
- no starvation for tournament algorithm follows from no starvation for the 2-process algorithms at all nodes

Space Complexity: *3n* Boolean shared variables.

Summary of R / W Mutex Algorithms

Progress property	**# read / write variables**
no deadlock	
no starvation (tournament)	3n Booleans
FIFO (bakery)	2n (Booleans + unbounded)

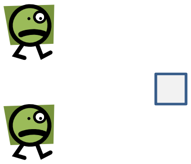
Can we do better?

Lower Bound on Number of Variables

Theorem: A mutex algorithm ensuring no deadlock uses at least *n* shared variables

For every *n*, reach a configuration in which *n* variables are **covered**

Covering



Several processes write to the same location
Write of early process is lost, if no read in between

☞Must write to distinct locations

Process p **covers a register R in a configuration C** if its next step from C is a write to R

Quiescence and Appearing Quiescent

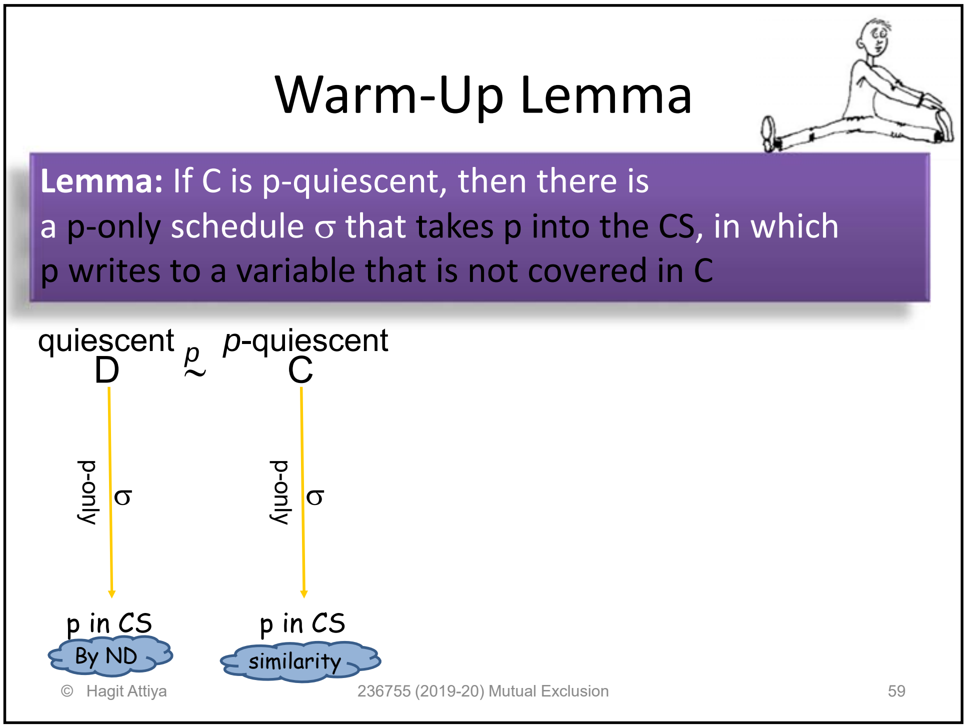
A configuration is **quiescent** if all processes are in the remainder

P is a set of processes, C and D configurations

$C \overset{P}{\sim} D$ if each process in P has same state in C and D and all shared variables have same value in C and D

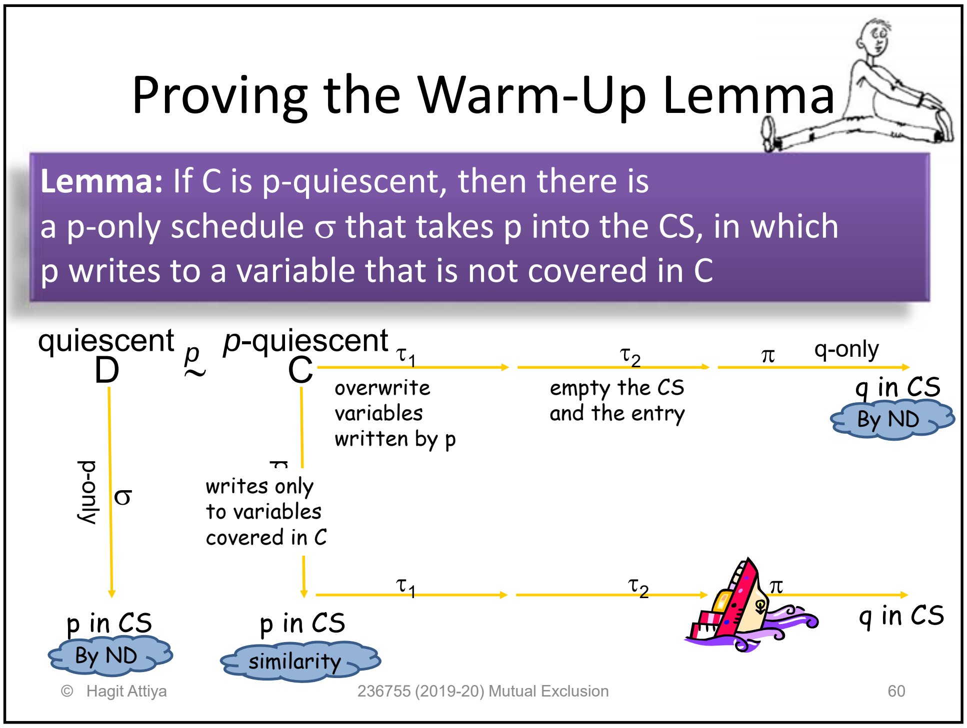
C is **P-quiescent** if it is indistinguishable to processes in P from a quiescent configuration

- I.e., $C \overset{P}{\sim} D$ for some quiescent configuration D



Warm-Up Lemma

Lemma: If C is p-quiescent, then there is a p-only schedule σ that takes p into the CS, in which p writes to a variable that is not covered in C



Proving the Warm-Up Lemma

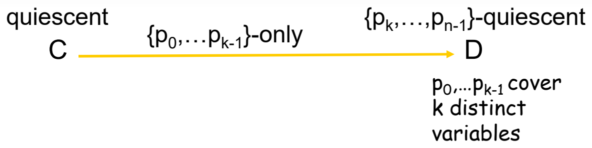
Lemma: If C is p-quiescent, then there is a p-only schedule σ that takes p into the CS, in which p writes to a variable that is not covered in C

Inductive Claim

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\dots,p_{k-1}$ only, s.t.

(a) $p_0,\dots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\dots,p_{n-1}\}$-quiescent



Proof is by induction on k

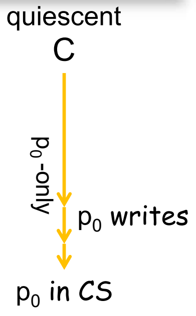
Taking k = n implies the lower bound

Base Case: k = 1

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\dots,p_{k-1}$ only, s.t.

(a) $p_0,\dots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\dots,p_{n-1}\}$-quiescent



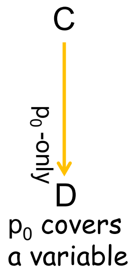
By warm-up lemma, there is a p_0-only schedule that takes p_0 into the CS, in which p_0 writes

Base Case: k = 1

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\ldots,p_{k-1}$ only, s.t.

(a) $p_0,\ldots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\ldots,p_{n-1}\}$-quiescent



By warm-up lemma, there is a p_0-only schedule that takes p_0 into the CS, in which p_0 writes

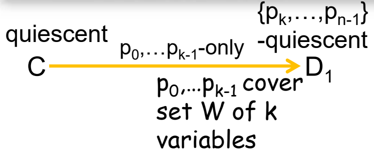
☝ Desired *D* is just before *p_0*'s first write.

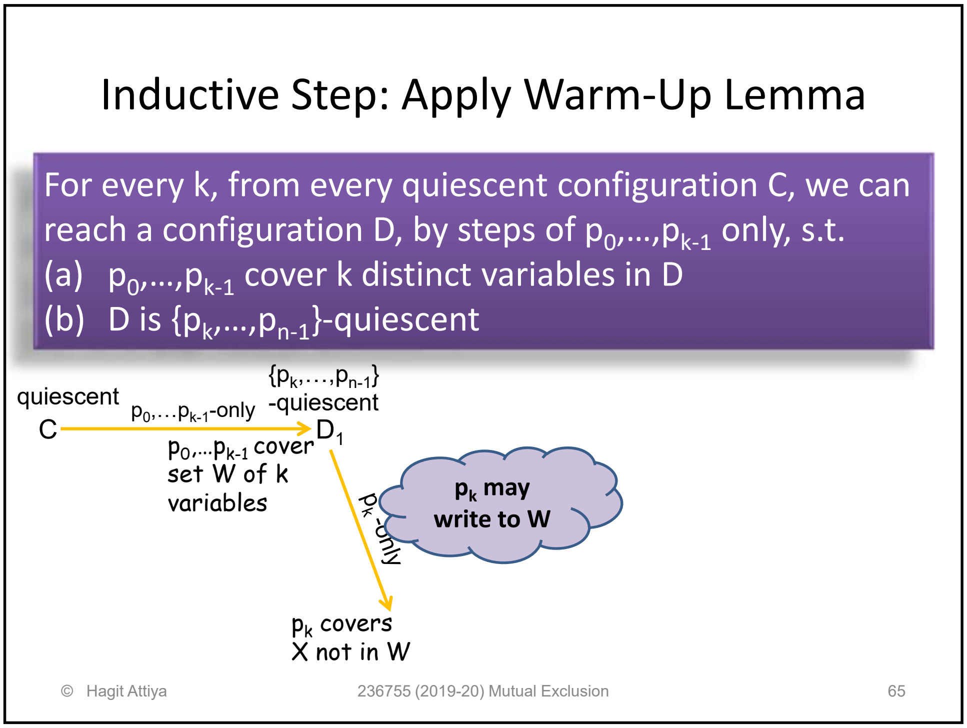
Inductive Step: Assume for *k*

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\ldots,p_{k-1}$ only, s.t.

(a) $p_0,\ldots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\ldots,p_{n-1}\}$-quiescent





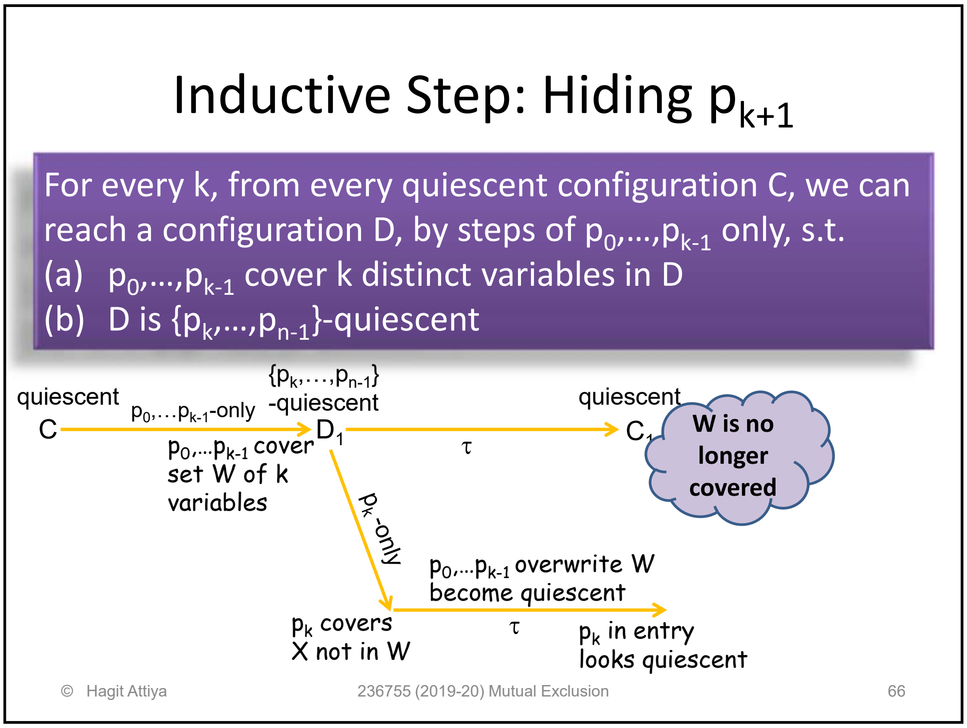
Inductive Step: Apply Warm-Up Lemma

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\ldots,p_{k-1}$ only, s.t.

(a) $p_0,\ldots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\ldots,p_{n-1}\}$-quiescent

quiescent
C
$p_0,\ldots p_{k-1}$-only
$\{p_k,\ldots,p_{n-1}\}$
-quiescent
D_1
$p_0,\ldots p_{k-1}$ cover set W of k variables
p_k-only
p_k may write to W
p_k covers X not in W



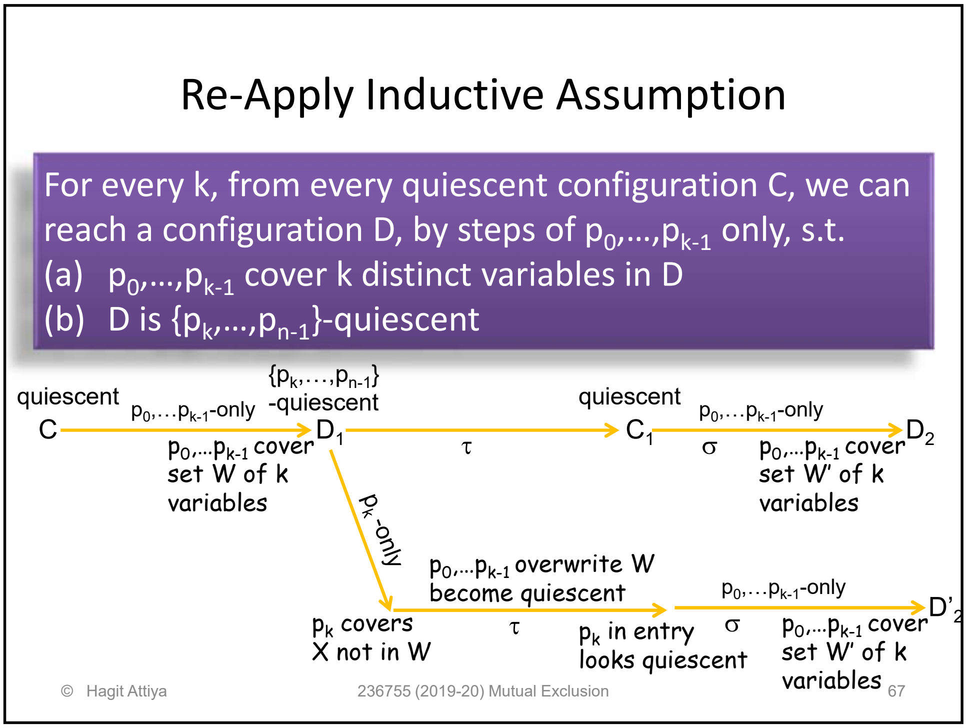
Inductive Step: Hiding p_{k+1}

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\ldots,p_{k-1}$ only, s.t.

(a) $p_0,\ldots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\ldots,p_{n-1}\}$-quiescent

quiescent
C
$p_0,\ldots p_{k-1}$-only
$\{p_k,\ldots,p_{n-1}\}$
-quiescent
D_1
τ
quiescent
C_1
W is no longer covered
$p_0,\ldots p_{k-1}$ cover set W of k variables
p_k-only
$p_0,\ldots p_{k-1}$ overwrite W become quiescent
τ
p_k covers X not in W
p_k in entry looks quiescent



Re-Apply Inductive Assumption

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\ldots,p_{k-1}$ only, s.t.

(a) $p_0,\ldots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\ldots,p_{n-1}\}$-quiescent

quiescent C $\xrightarrow{p_0,\ldots p_{k-1}\text{-only}}$ D_1 $\xrightarrow{\tau}$ quiescent C_1 $\xrightarrow{\sigma}$ D_2

$p_0,\ldots,p_{k-1}$-only

$\{p_k,\ldots,p_{n-1}\}$-quiescent

$p_0,\ldots p_{k-1}$ cover set W of k variables

$p_0,\ldots p_{k-1}$-only

$p_0,\ldots p_{k-1}$ cover set W' of k variables

p_k-only

$p_0,\ldots p_{k-1}$ overwrite W become quiescent

p_k covers X not in W

τ

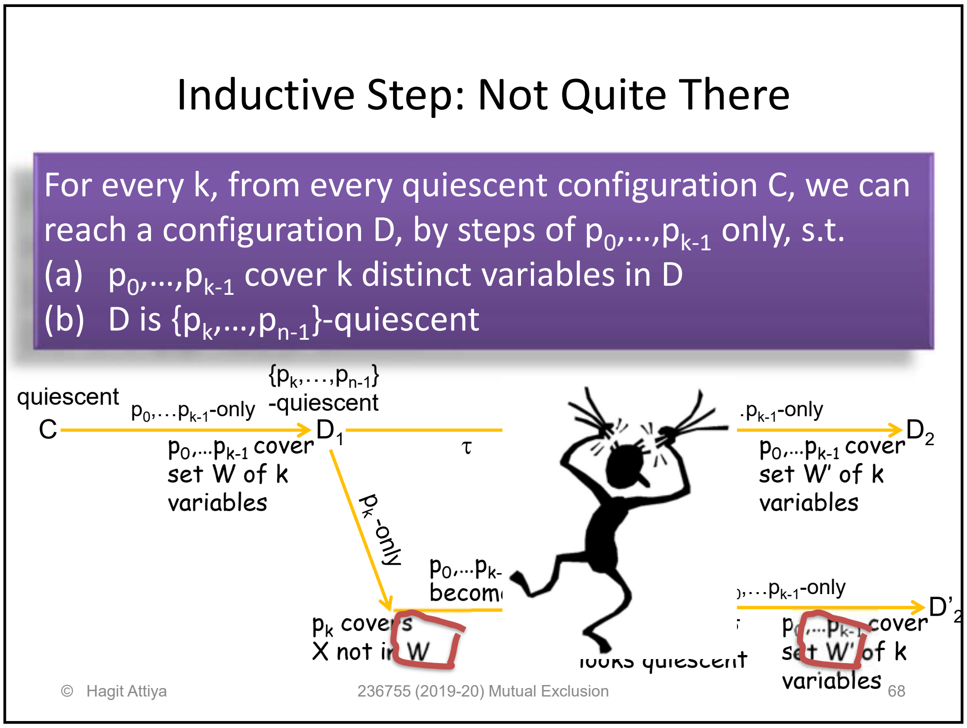
p_k in entry looks quiescent

σ

$p_0,\ldots p_{k-1}$-only

D'_2

$p_0,\ldots p_{k-1}$ cover set W' of k variables



Inductive Step: Not Quite There

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\ldots,p_{k-1}$ only, s.t.

(a) $p_0,\ldots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\ldots,p_{n-1}\}$-quiescent

quiescent C $\xrightarrow{p_0,\ldots p_{k-1}\text{-only}}$ D_1 τ

$\{p_k,\ldots,p_{n-1}\}$-quiescent

$p_0,\ldots p_{k-1}$ cover set W of k variables

$.p_{k-1}$-only D_2

$p_0,\ldots p_{k-1}$ cover set W' of k variables

p_k-only

$p_0,\ldots p_{k-}$ becom

p_k covers X not in W

looks quiescent

$,\ldots p_{k-1}$-only D'_2

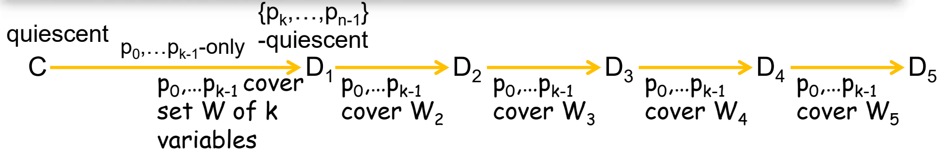
$p_0,\ldots p_{k-}$ cover set W' of k variables

Completing the Inductive Step

For every k, from every quiescent configuration C, we can reach a configuration D, by steps of $p_0,\ldots,p_{k-1}$ only, s.t.

(a) $p_0,\ldots,p_{k-1}$ cover k distinct variables in D

(b) D is $\{p_k,\ldots,p_{n-1}\}$-quiescent

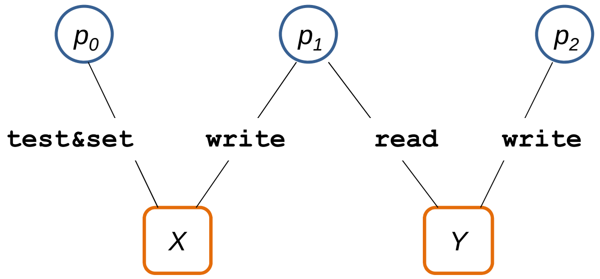


Repeat until the sets are repeated ($W_i = W_j$), and then apply the previous argument

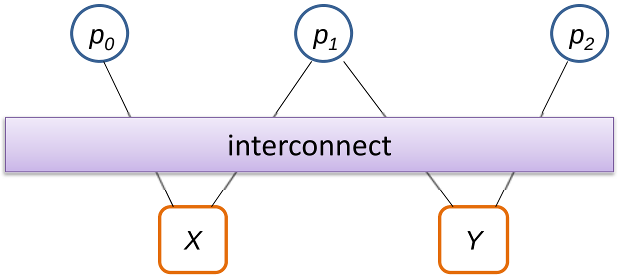
---

Optimizing Memory Locality

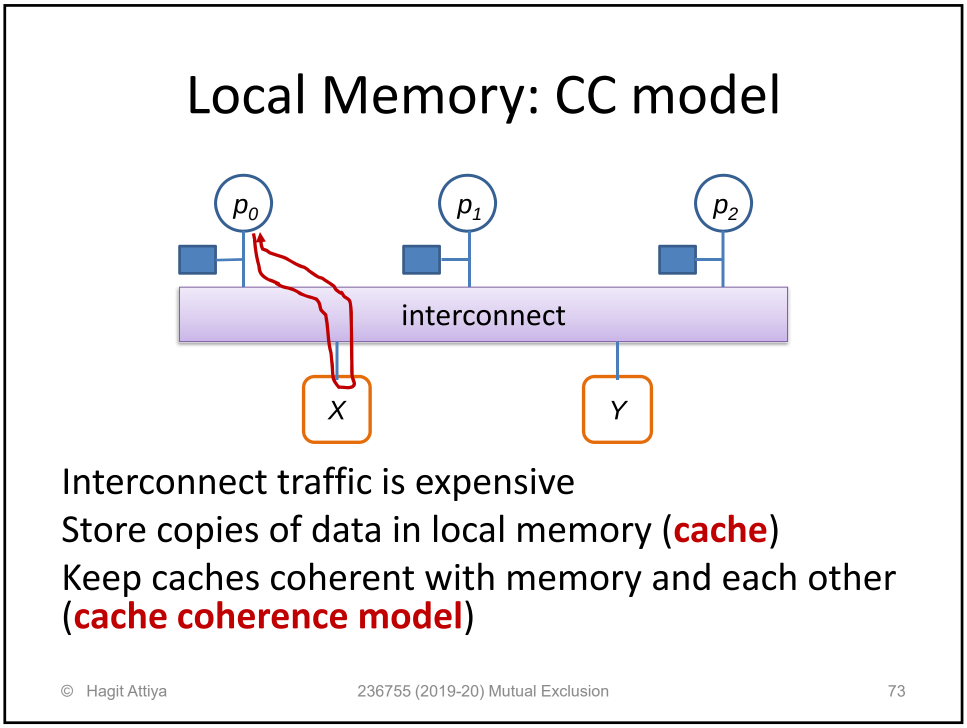
Memory Access, in Formal Model



In Reality: Memory Interconnect



Memory is accessed through an interconnection network (e.g., a bus)

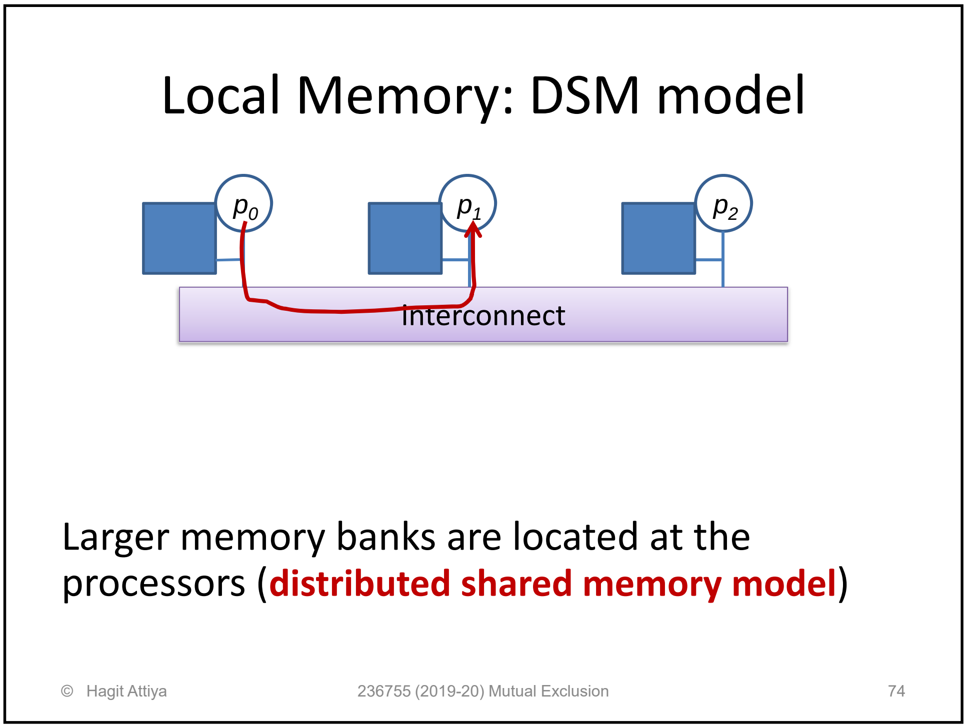


Local Memory: CC model

Interconnect traffic is expensive

Store copies of data in local memory (**cache**)

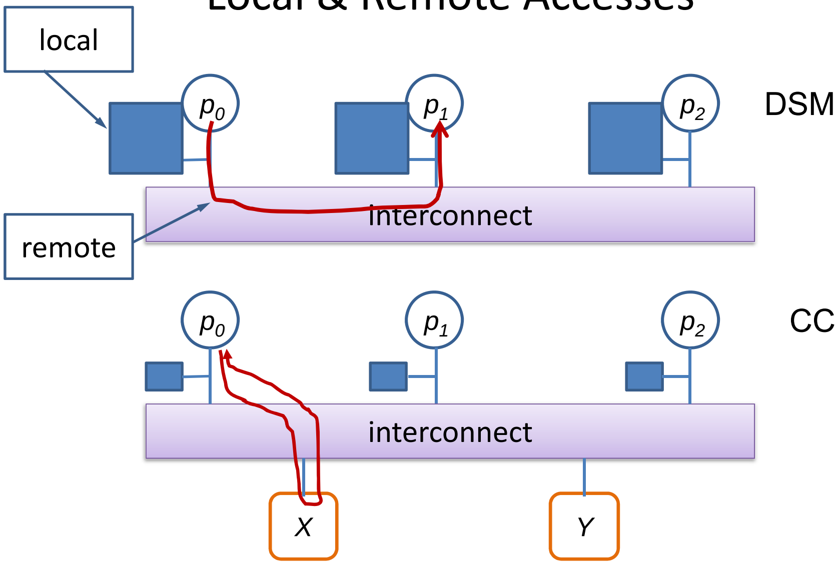
Keep caches coherent with memory and each other (**cache coherence model**)



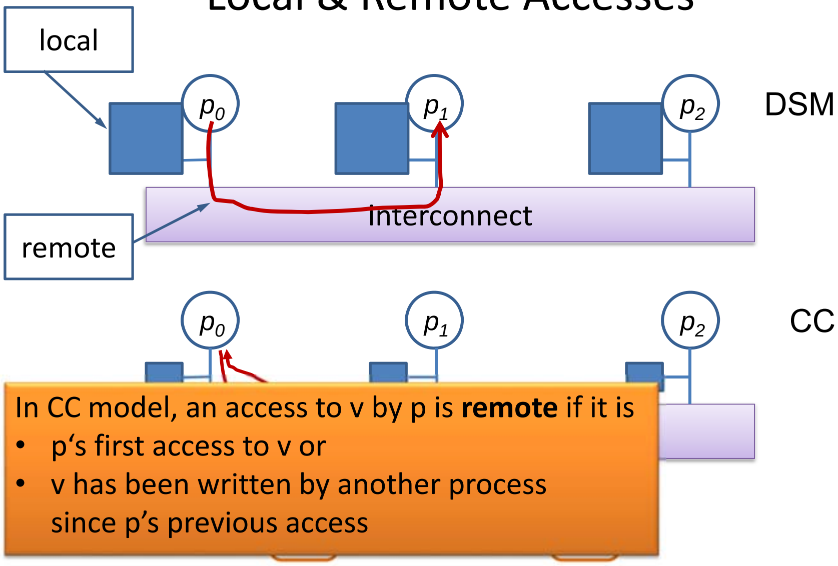
Local Memory: DSM model

Larger memory banks are located at the processors (**distributed shared memory model**)

Local & Remote Accesses



Local & Remote Accesses



In CC model, an access to v by p is **remote** if it is

- p's first access to v or
- v has been written by another process since p's previous access

Local Spinning

- An algorithm is **local-spin** if all busy waiting is in read-only loops of local-accesses, which do not cause interconnect traffic

☝ An algorithm may be local-spin on one model (DSM or CC) and not local-spin on the other!

☝ The **remote memory references** (**RMR**) complexity of an algorithm is the number of remote accesses

R / W 2-Process Mutex

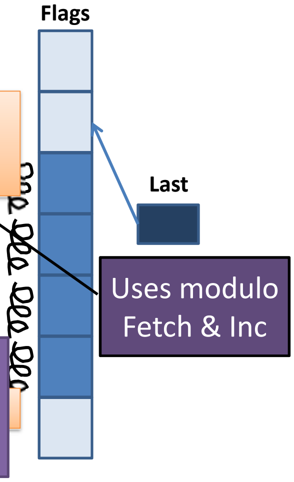
```
Want[i] = 0
wait until Want[1-i] == 0 or Priority == i
Want[i] = 1
if (Priority == 1-i) then
     if (Want[1-i] == 1) then goto Line 1
else wait until (Want[1-i] == 0)
```

- Is this algorithm local-spin?
  - In the DSM model? No
  - In the CC model? Yes
- What is its RMR complexity?
  - In the DSM model? Unbounded
  - In the CC model? Constant

Recall Anderson's Algorithm

entry section:

```
myPlace = rmw(Last,Last+1 mod n)
wait until Flags[myPlace] == 1
Flags[myPlace] = 0
```



Uses modulo Fetch & Inc

Is this algorithm local-spin?
In the CC model? Yes
In the DSM model? No

Local-Spin Mutex w/ Swap

```
swap(W, new)
   prev = W
   W = new
   return prev
```

Atomic register-to-memory swap operations, also called fetch-and-store

More common than **fetch&inc mod n**

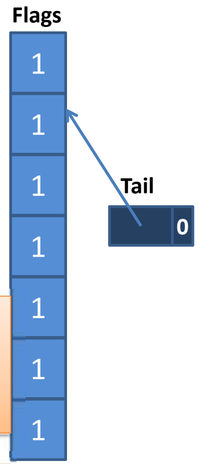
Each process spins on its own location in array

Array contains the queue of waiting processes

Each entry in the array holds a pointer to the next process in line.

Local-Spin Mutex w/ Swap

[Graunke & Thakkar, 1989]



Shared variables:
Flags[0..n-1], binary; all initially 1
Tail {binary, {0,..,n-1}}, initially {0,0}

Local variables:
myRecord, prev {binary, {0,..,n-1}},
temp binary

entry section:

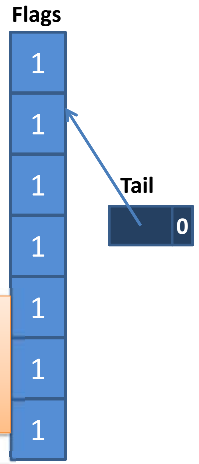
```
myRecord.value = Flags[i]
myRecord.slot = i
prev = swap(Tail, myRecord)
wait until(Flags[prev.slot] ≠ prev.value)
```

exit section:

```
Flags[i]= 1 - Flags[i]
```

Local-Spin Mutex w/ Swap

[Graunke & Thakkar, 1989]



Shared variables:
Flags[0..n-1], binary; all initially 1
Tail {binary, {0,..,n-1}}, initially {0,0}

Local variables:
myRecord, prev {binary, {0,..,n-1}},
temp binary

entry section:

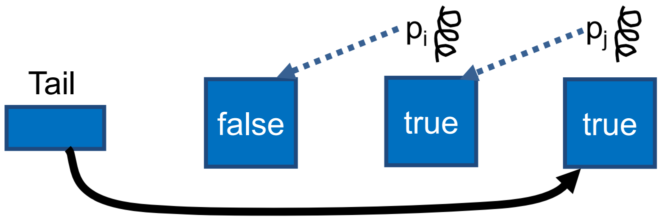
```
myRecord.value = Flags[i]
.value)
Flags[i]= 1 - Flags[i]
```

Is this algorithm local-spin?
  In the CC model? Yes
  In the DSM model? No

CLH Lock

[Craig 1993] and [Landin & Hagers, 1994]

- Also a queue, but does not allocate space for all processes
- Instead, "thread" records in a (virtual) linked list



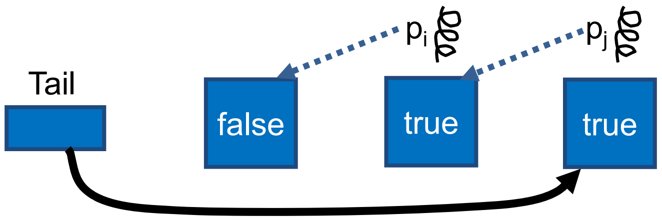
CLH Lock

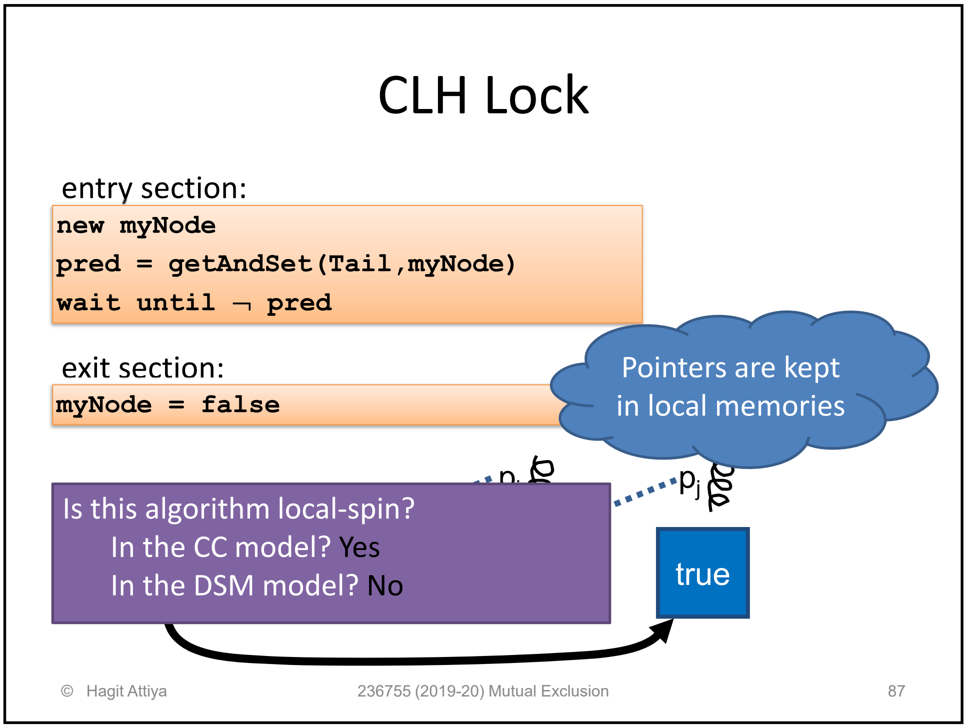
entry section:

```
new myNode
pred = getAndSet(Tail,myNode)
wait until ¬ pred
```

exit section:

```
myNode = false
```





CLH Lock

entry section:

```
new myNode
pred = getAndSet(Tail,myNode)
wait until ¬ pred
```

exit section:

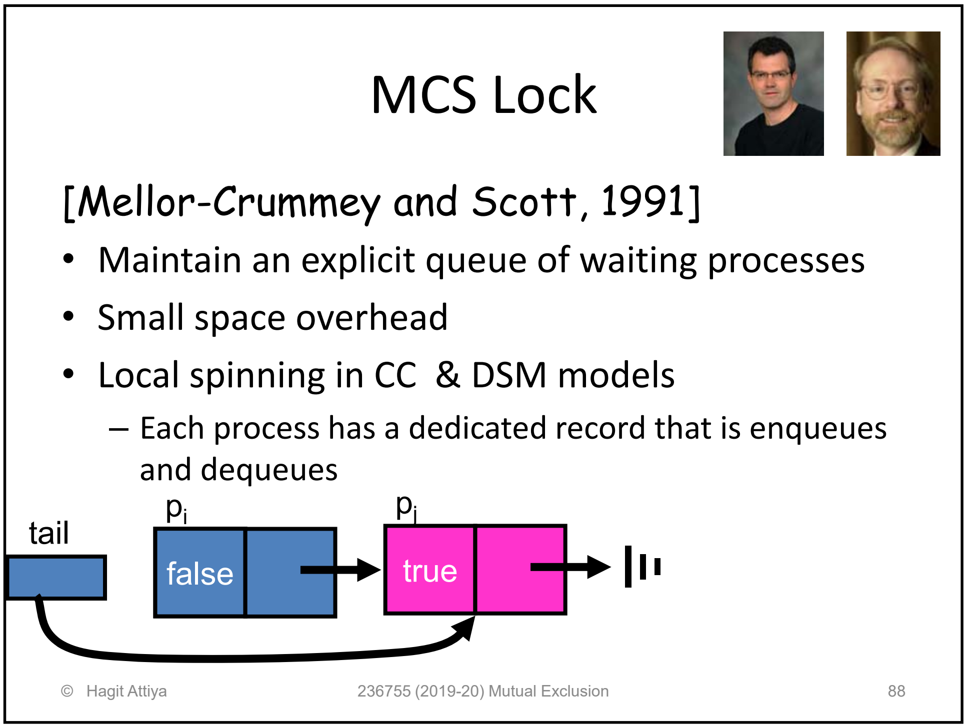
```
myNode = false
```

Pointers are kept in local memories

Is this algorithm local-spin?

In the CC model? Yes

In the DSM model? No



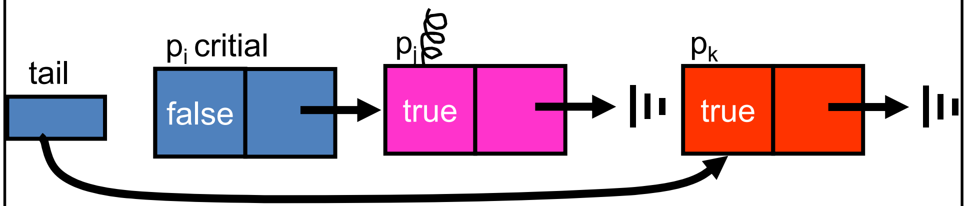
MCS Lock

[Mellor-Crummey and Scott, 1991]

- Maintain an explicit queue of waiting processes
- Small space overhead
- Local spinning in CC & DSM models
  - Each process has a dedicated record that is enqueues and dequeues

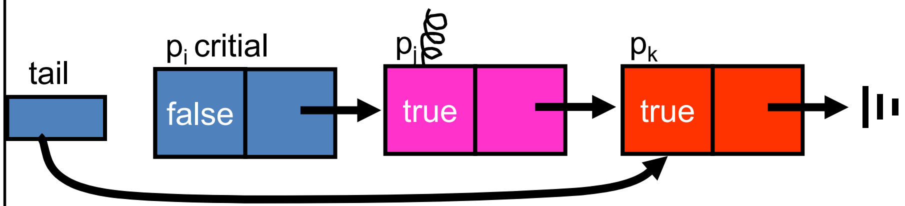
MCS Lock: Enqueing for the lock

- Set tail to point to your record (with compare&set)



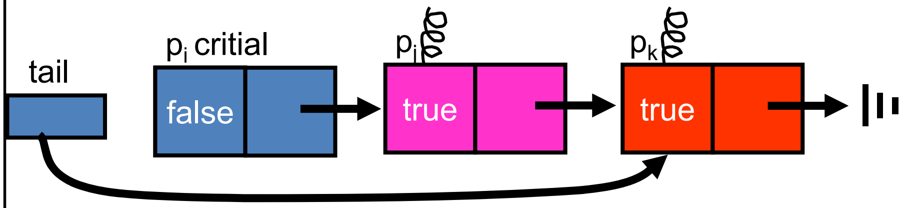
MCS Lock: Enqueing for the lock

- Set tail to point to your record (with CAS)
- Make last element point to your record



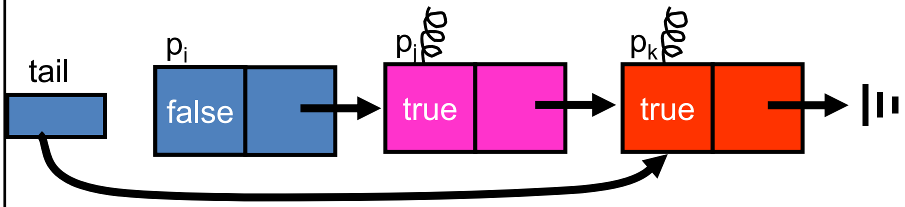
MCS Lock: Enqueing for the lock

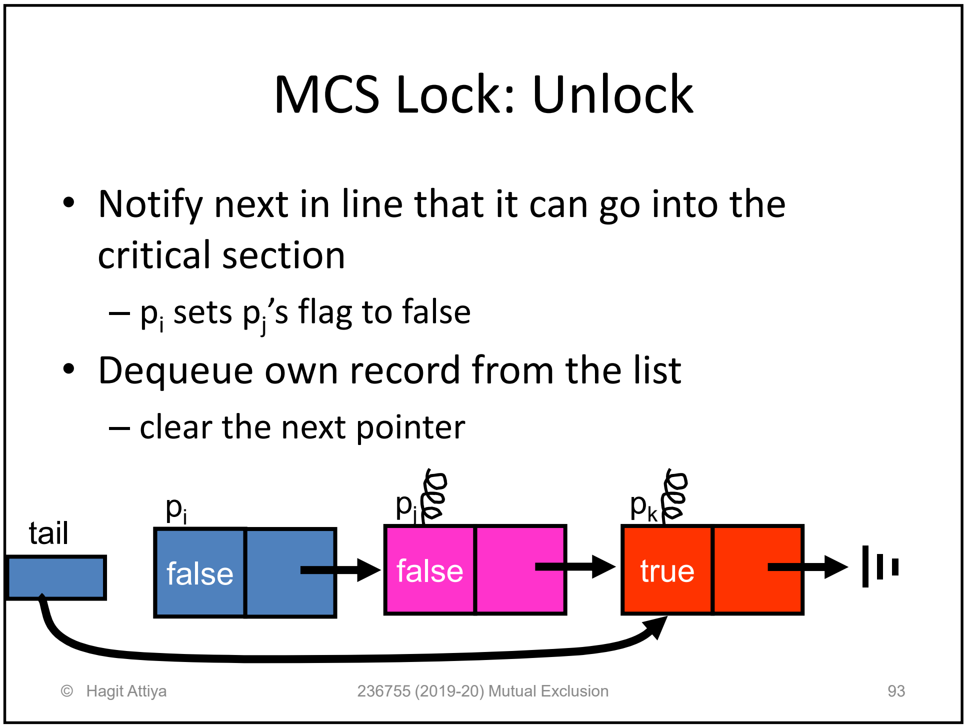
- Set tail to point to your record (with CAS)
- Make last element point to your record
- Spin on your own record



MCS Lock: Unlock

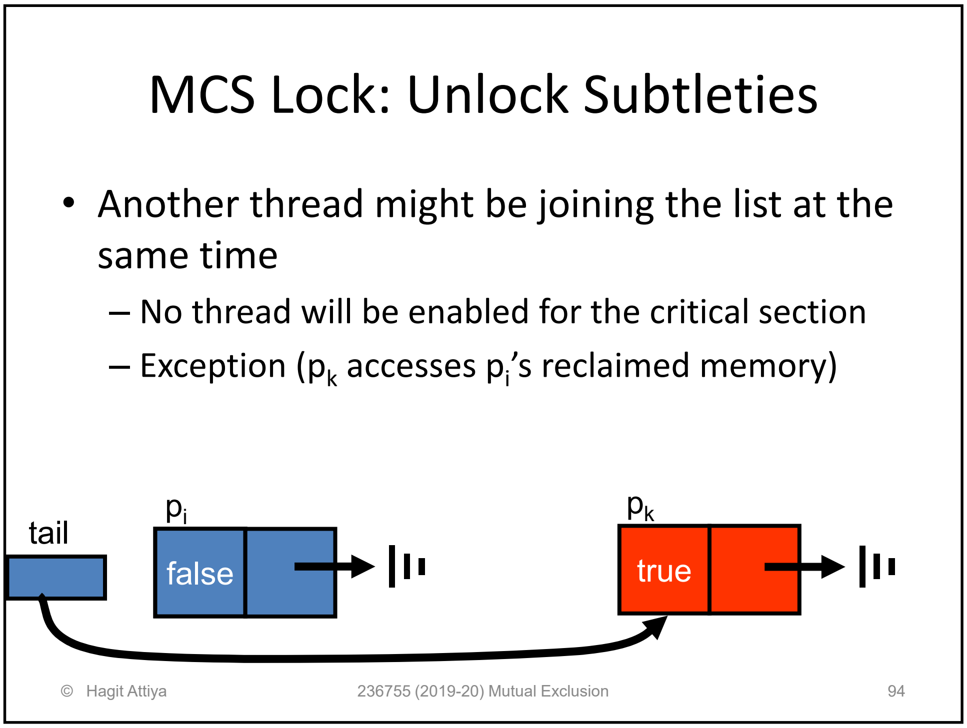
- Notify next in line that it can go into the critical section





MCS Lock: Unlock

- Notify next in line that it can go into the critical section
  - p_i sets p_j's flag to false
- Dequeue own record from the list
  - clear the next pointer

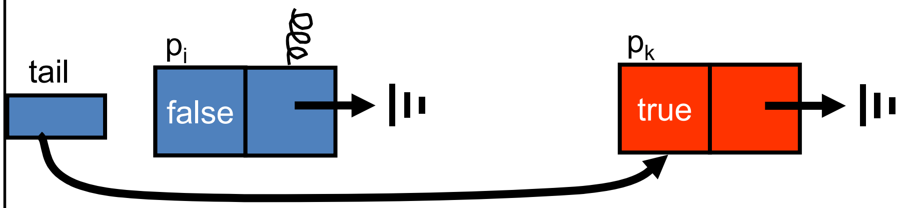


MCS Lock: Unlock Subtleties

- Another thread might be joining the list at the same time
  - No thread will be enabled for the critical section
  - Exception (p_k accesses p_i's reclaimed memory)

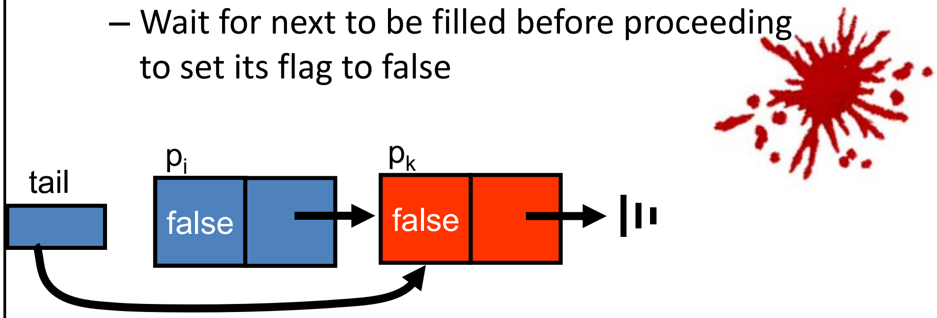
MCS Lock: Unlock Subtleties

- Another thread might be joining the list at the same time
- Can be detected since tail is not null
  - Wait for next to be filled before proceeding



MCS Lock: Unlock Subtleties

- Another thread might be joining the list at the same time
- Can be detected since tail is not null
  - Wait for next to be filled before proceeding to set its flag to false



MCS Queue-Based Algorithm

```
Shared Qnode nodes[0..n-1]
Shared Qnode *tail initially null
Local Qnode *myNode, initially &nodes[i]
Local Qnode *successor

acquire-lock
myNode->next = null          // prepare to be last in queue
pred = swap(&tail, myNode )  // tail now points to myNode
if (pred ≠ null)             // should wait for a predecessor
   myNode->locked = true     // prepare to wait
   pred->next = myNode       // let predecessor know to unlock me
   wait until ( myNode.locked == false )

release-lock
if (myNode.next == null)     // not sure there is successor
   if (compare-and-swap(&tail, myNode, null) == false)
      wait until (myNode->next ≠ null) // wait for successor id
      successor = myNode->next      // get pointer to successor
      successor->locked = false     // unlock successor
else                         // for sure, there is successor
      successor = myNode->next      // get pointer to successor
      successor->locked = false     // unlock successor
```

MCS Queue-Based Algorithm

```
Shared Qnode nodes[0..n-1]
Shared Qnode *tail initially null
Local Qnode *myNode, initially &nodes[i]
Local Qnode *successor

acquire-lock
myNode->next = null          // prepare to be last in queue
pred = swap(&tail, myNode )  // tail now points to myNode
if (pred ≠ null)             // should wait for a predecessor
   myNode->locked = true     // prepare to wait
   pred->next = myNode       // let predecessor know to unlock me
   wait until ( myNode.locked == false )
```

Uses swap and CAS

Is this algorithm local-spin?

- In the CC model? Yes
- In the DSM model? Yes

```
                                   here is successor
                                   ) == false)
                                   ait for successor id
                                   pointer to successor
                                   ck successor
                                   there is successor
      successor = myNode->next      // get pointer to successor
      successor->locked = false     // unlock successor
```

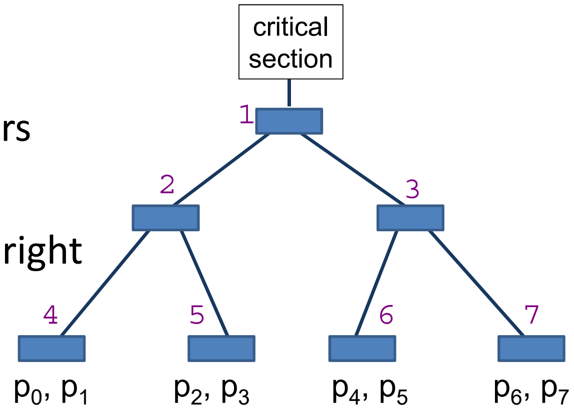
Local-Spin Mutex
without Strong Primitives

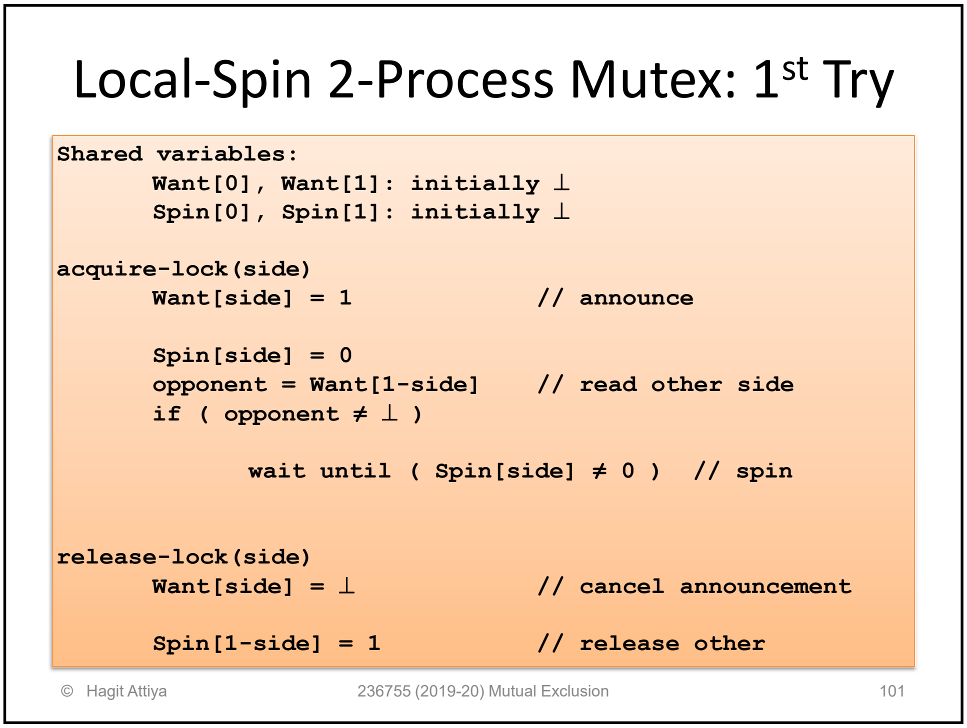
Local-Spin Tournament-Tree Mutex

O(log n) RMR complexity
for CC model
(this is optimal)

O(n log n) registers

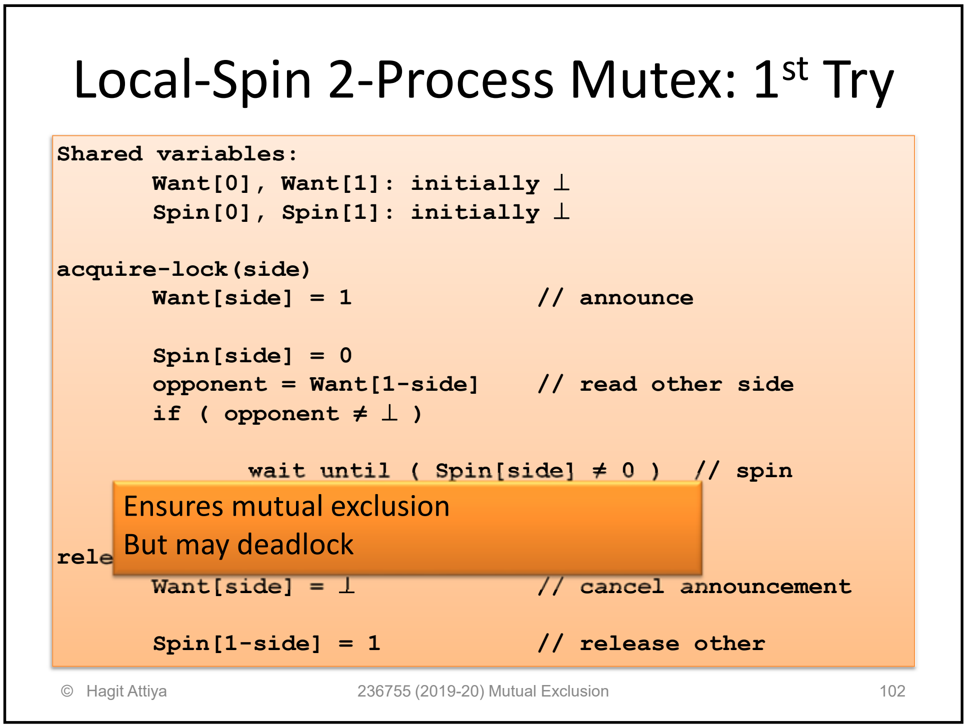
Key is to find the right
2-process mutex





Local-Spin 2-Process Mutex: 1^{st} Try

```
Shared variables:
       Want[0], Want[1]: initially ⊥
       Spin[0], Spin[1]: initially ⊥

acquire-lock(side)
       Want[side] = 1                // announce

       Spin[side] = 0
       opponent = Want[1-side]       // read other side
       if ( opponent ≠ ⊥ )

              wait until ( Spin[side] ≠ 0 )  // spin


release-lock(side)
       Want[side] = ⊥                // cancel announcement

       Spin[1-side] = 1              // release other
```



Local-Spin 2-Process Mutex: 1^{st} Try

```
Shared variables:
       Want[0], Want[1]: initially ⊥
       Spin[0], Spin[1]: initially ⊥

acquire-lock(side)
       Want[side] = 1                // announce

       Spin[side] = 0
       opponent = Want[1-side]       // read other side
       if ( opponent ≠ ⊥ )

              wait until ( Spin[side] ≠ 0 )  // spin


rele
       Want[side] = ⊥                // cancel announcement

       Spin[1-side] = 1              // release other
```

Ensures mutual exclusion
But may deadlock

Local-Spin 2-Process Mutex: Avoid Deadlock

```
Shared variables:
       Tie, Want[0], Want[1]: initially ⊥
       Spin[0], Spin[1]: initially ⊥

acquire-lock(side)
       Want[side] = 1              // announce
       Tie = i                     // tie breaker
       Spin[side] = 0
       opponent = Want[1-side]     // read other side
       if ( opponent ≠ ⊥ ) and ( Tie == i )
              if ( Spin[1-side] == 0 ) Spin[1-side] = 1
              wait until ( Spin[side] ≠ 0 )   // spin
              if ( Tie == i ) wait until ( Spin[side] > 1 )

release-lock(side)
       Want[side] = ⊥              // cancel announcement

       if ( Tie ≠ i ) Spin[1-side] = 2 // release other
```

Local-Spin 2-Process Mutex: Avoid Deadlock

```
Shared variables:
       Tie, Want[0], Want[1]: initially ⊥
       Spin[0], Spin[1]: initially ⊥

acquire-lock(side)
       Want[side] = 1              // announce
       Tie = i                     // tie breaker
       Spin[side] = 0
       opponent = Want[1-side]     // read other side
       if ( opponent ≠ ⊥ ) and ( Tie == i )
              if ( Spin[1-side] == 0 ) Spin[1-side] = 1
              wait until ( Spin[side] ≠ 0 )   // spin
              if ( Tie == i ) wait until ( Spin[side] > 1 )

rel
       Want[side] = ⊥              // cancel announcement

       if ( Tie ≠ i ) Spin[1-side] = 2 // release other
```

Is this local spinning in DSM?

Local-Spin 2-Process Mutex

```
Shared variables:
       Tie, Want[0], Want[1]: initially ⊥
       Spin[0,…,n-1]: initially ⊥

acquire-lock(side)
       Want[side] = i              // announce your identity
       Tie = i                     // tie breaker
       Spin[i] = 0
       opponent = Want[1-side]     // who's competing
       if ( opponent ≠ ⊥ ) and ( Tie == i )
              if ( Spin[opponent] == 0 ) Spin[opponent] = 1
              wait until ( Spin[i] ≠ 0 )
              if ( Tie == i ) wait until ( Spin[i] > 1 )

release-lock(side)
       Want[side] = nil
       opponent = Tie              // who's competing
       if ( opponent ≠ i ) Spin[opponent] = 2
```

Example (for processes 3 and 7)

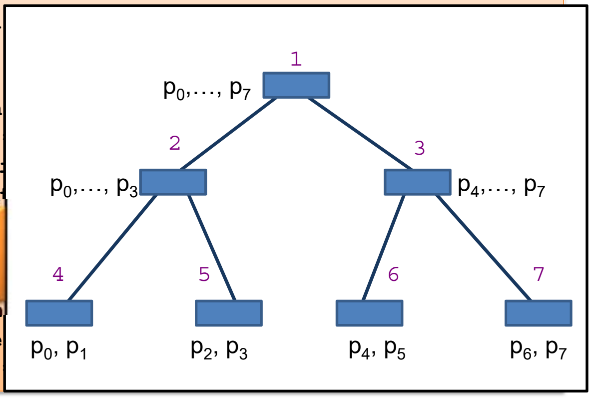
Want[0] = 3	**Want[1] = 7**
Tie = 3	
Spin[3] = 0	**Tie = 7**
opponent = 7	**Spin[7] = 0**
opponent <> ⊥ and Tie <> 3	**opponent = 3**
CRITICAL	**opponent <> ⊥ and Tie == 7**
CRITICAL	**Spin[3] == 0, so Spin[3] = 1**
CRITICAL	**WAIT until Spin[7] <> 0**
CRITICAL	**WAIT**
CRITICAL	**WAIT**
Spin[7] = 1	
Spin[7] = 2	**Tie == 7, so wait until Spin[7] > 1**
	CRITICAL

Local-Spin 2-Process Mutex

```
Shared variables:
       Tie, Want[0], Want[1]; initially ⊥
       Spin[0,…,n-1]; initially ⊥

acquire-lock(side)
       Want[side] = i
       Tie = i
       Spin[i] = 0
       opponent = Wan
       if ( opponent
              if ( Sp
              wait unt
       Want[side] = n
       opponent = Tie
       if ( opponent
```

n registers per node?

An array for each level
O(n log n) total



Optimizing for No Contention

In a well-designed system, most of the time only a single process wants the critical section…

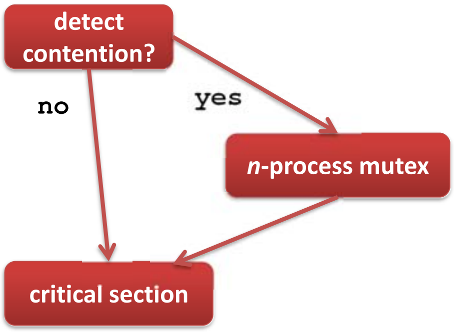
In the algorithms so far, requires O(f(n)) steps:

O(n) for the Bakery algorithm

O(log(n)) for the tournament tree algorithm

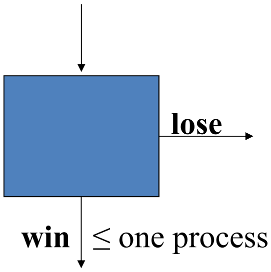
Fast Mutex

Algorithm is **fast** if a process enters CS in O(1) steps, when there is no competition



Must use multi-writer shared variables

Detecting Contention: Splitter



A process wins if it is alone in the splitter

O(1) step complexity

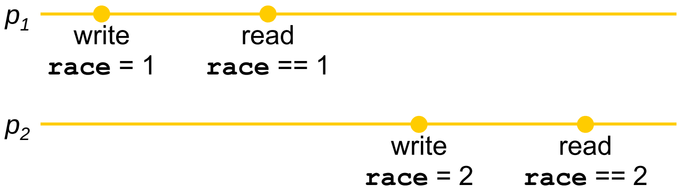
Splitter Implementation: Race Variable

Shared variable: race, initially -1

```
race = $id_i$
if race == $id_i$ then win
else lose
```

If a process is alone, clearly wins

But it is possible that two processes win



p_1

write
`race` = 1

read
`race` == 1

p_2

write
`race` = 2

read
`race` == 2

Doorway Mechanism

- Wrap a doorway mechanism around **`race`**
- Only a process in the first set of processes to concurrently access **`race`** may win
- After writing to **`race`**, check the doorway and if open, close it
- **`race`** chooses a unique one of the captured processes to "win"

Splitter Implementation

Shared variables

door, initially false

race, initially -1

```
race = $id_i$              // write your identifier
if door then return( lose )
door = true
if (race == $id_i$ )     // check race variable
        then return( win )
else return( lose )
```

Requires ≤ 5 read / write operations, and two shared registers.

Splitter Implementation: Race Variable

Shared variables

door, initially false

race, initially -1

```
race = $id_i$              // write your identifier
if door then return( lose )
door = true
if (race == $id_i$ )     // check race variable
        then return( win )
else return( lose )
```

Splitter Implementation: Doorway

Shared variables

door, initially false

race, initially -1

```
race = $id_i$                  // write your identifier
if door then return( lose )
door = true
if (race == $id_i$ )       // check race variable
            then return( win )
else return( lose )
```

Correctness of the Splitter

A process wins when executing the splitter by itself

Follows from the code when there is no concurrency

```
race = $id_i$                  // write your identifier
if door then return( lose )
door = true
if (race == $id_i$ )       // check race variable
            then return( win )
else return( lose )
```

Correctness of the Splitter

At most one process wins the splitter

P: processes that read false from **door** (Line 2)

p_j: last process to write to **race** before **door** is set to true

No process $p_i \neq p_j$ can win:

- $p_i \notin P$ loses in Line 2.
- $p_i \in P$ writes to **race** before p_j but checks again (Line 5) after p_j 's write and loses

```
1. race = id_i              // write your identifier
2. if door then return( lose )
3. door = true
4. if (race == id_i )      // check race variable
            then return( win )
5. else return( lose )
```

Correctness of the Splitter

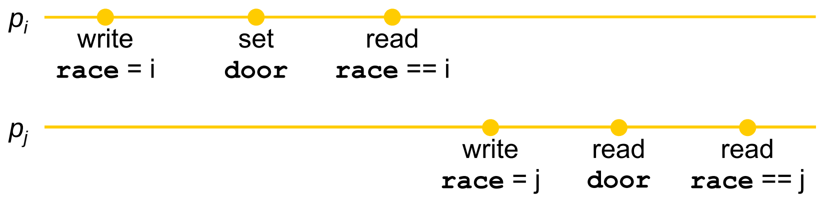
At most one process wins the splitter

P: processes that read false from **door** (Line 2)

p_j: last process to write to **race** before **door** is set to true

No process $p_i \neq p_j$ can win:

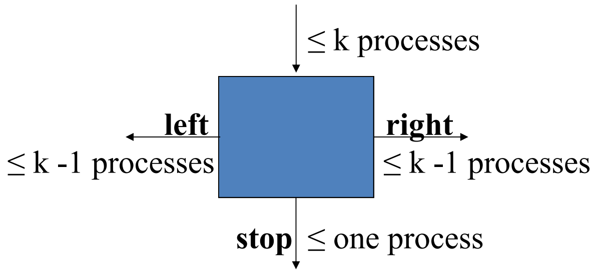
- $p_i \notin P$ loses in Line 2.
- $p_i \in P$ writes to **race** before p_j but checks again (Line 5) after p_j 's write and loses



Detour: Splitting the Losers

```
race = $id_i$              // write your identifier
if door then return( lose )
door = true
if (race == $id_i$ )       // check race variable
        then return( win )
else return( lose )
```

Detour: Splitting the Losers



≤ k processes

left

≤ k -1 processes

right

≤ k -1 processes

stop ≤ one process

```
race = $id_i$              // write your identifier
if door then return( right )
door = true
if (race == $id_i$ )       // check race variable
        then return( stop )
else return( left )
```

Proof of Splitting Property

Not all processes go left, not all processes go right

At least one process (the first) reads false from **`door`**

- ☞ Not all processes return right

If some process reads true from **`door`**

- ☞ Not all processes return left

Otherwise, last process to write to **`race`** returns stop

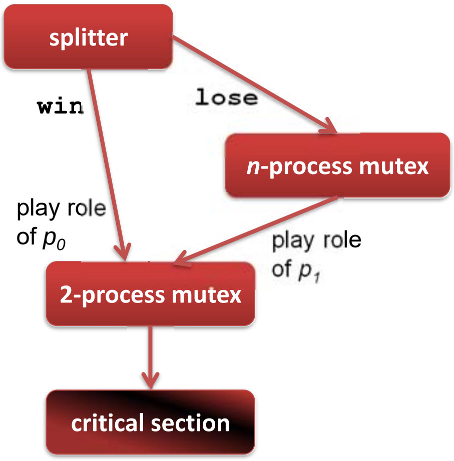
- ☞ not all processes return left

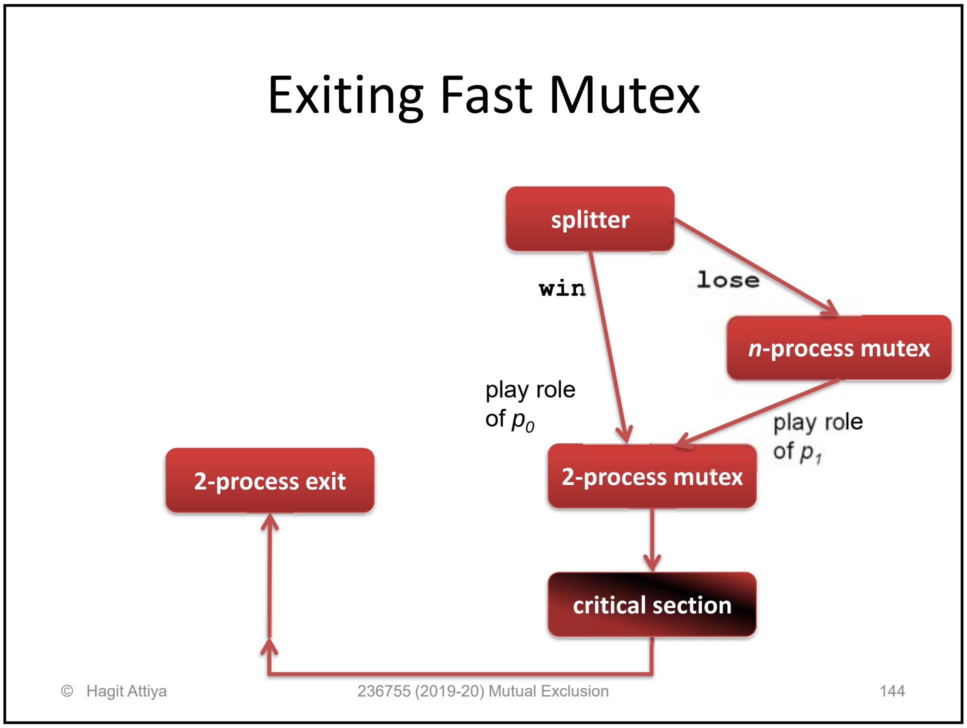
```
race = id_i                // write your identifier
if door then return( right )
door = true
if (race == id_i )         // check race variable
          then return( stop )
else return( left )
```

Ensuring No Deadlock

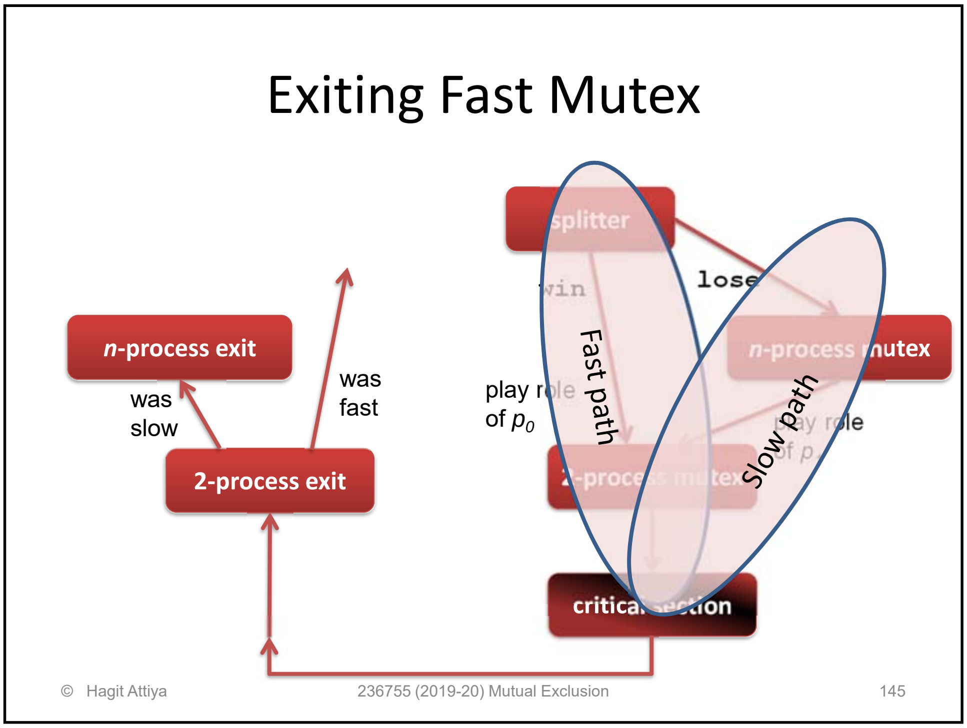
In case of concurrency, it is possible that no process wins the splitter

- ☞ Nodes losing the splitter enter *n*-process mutex
- ☞ Winner of *n*-process mutex competes with winner of splitter using 2-process mutex
- ☞ Winner enters CS



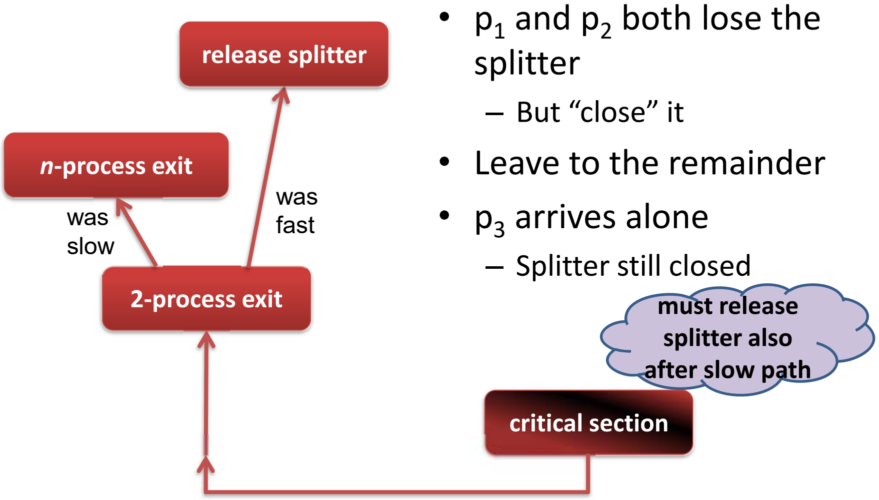


Exiting Fast Mutex



Exiting Fast Mutex

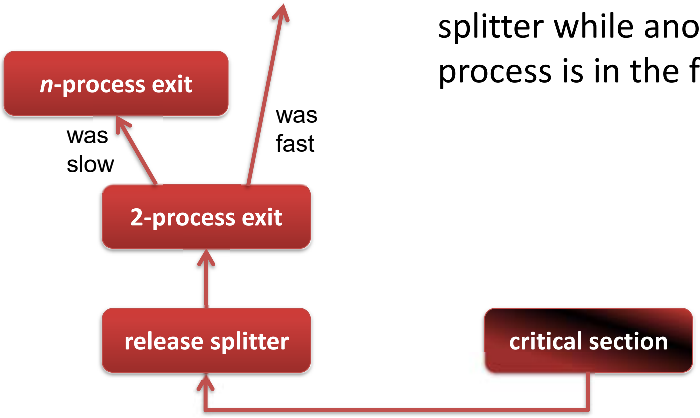
Releasing the Splitter: Take 1



- p_1 and p_2 both lose the splitter
  - But "close" it
- Leave to the remainder
- p_3 arrives alone
  - Splitter still closed

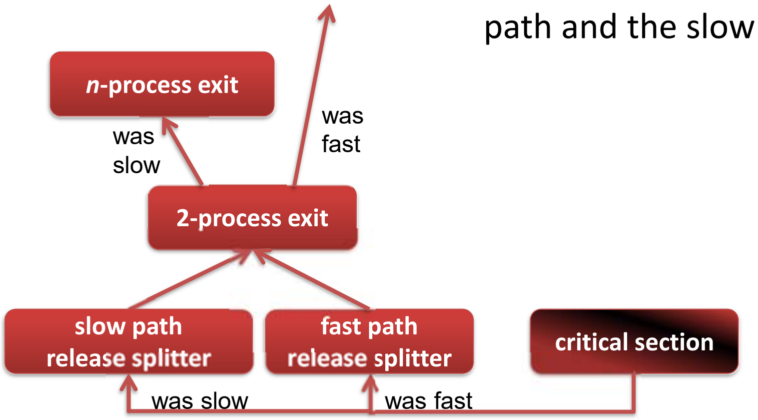
Releasing the Splitter: Take 2

A process from the slow path may release the splitter while another process is in the fast path

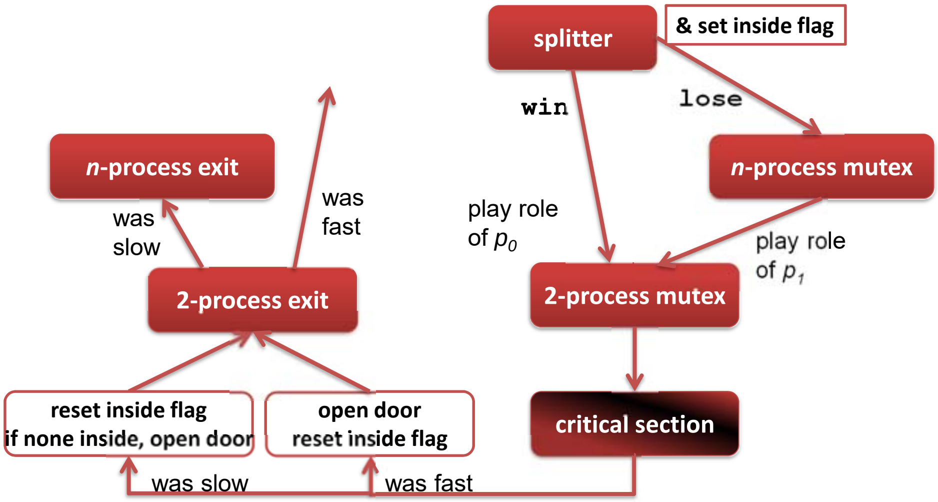


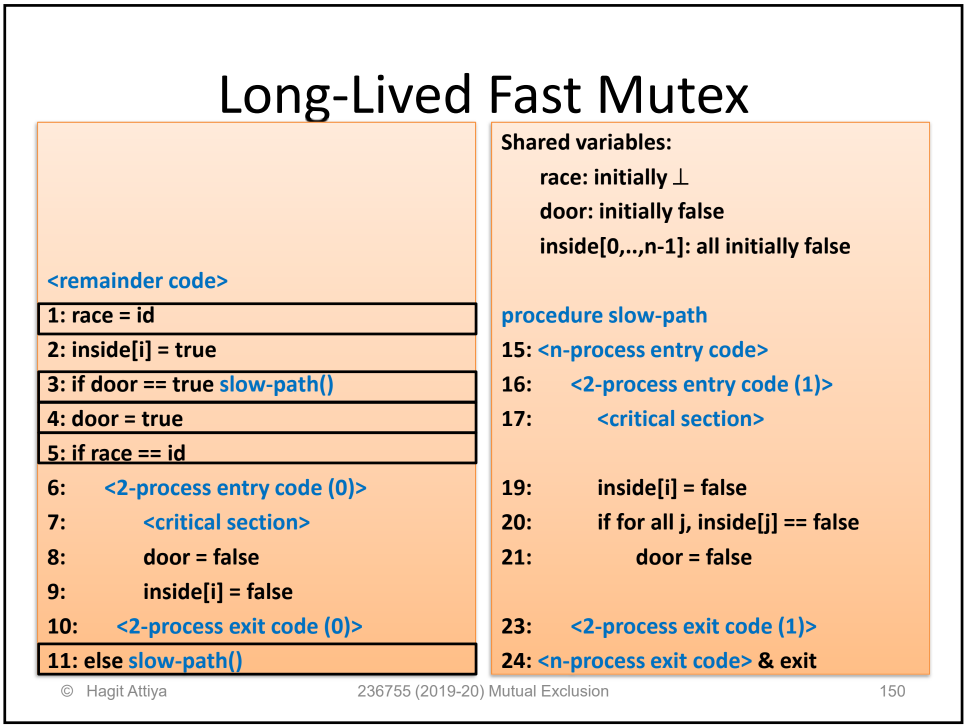
Releasing the Splitter: Take 3

Different release code for processes from the fast path and the slow path



Fast Mutex: Overall Structure

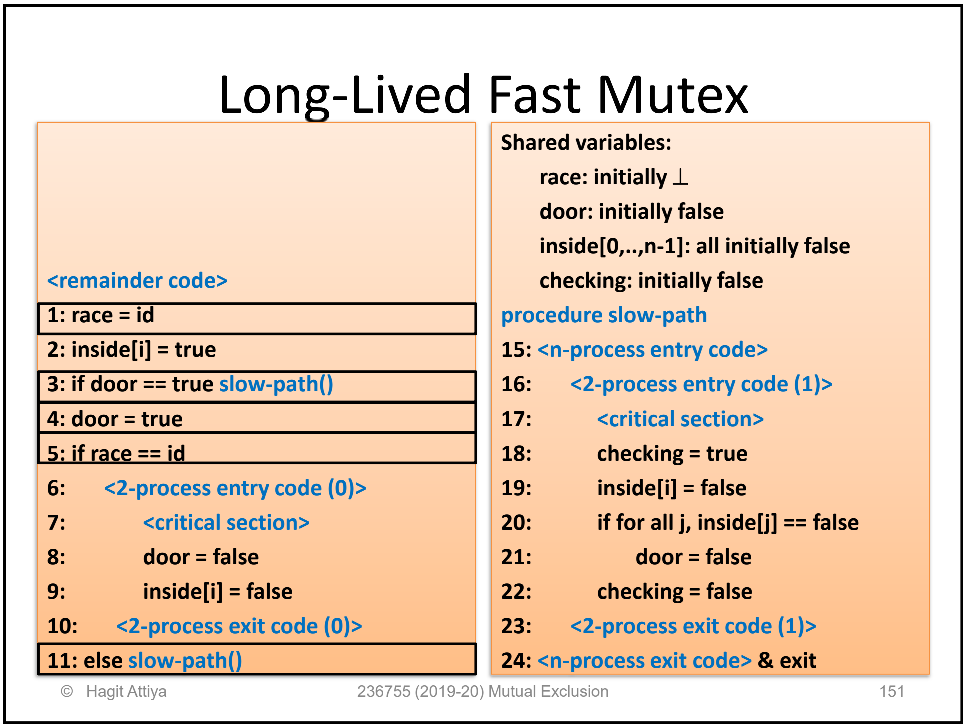




Long-Lived Fast Mutex

```
<remainder code>
1: race = id
2: inside[i] = true
3: if door == true slow-path()
4: door = true
5: if race == id
6:      <2-process entry code (0)>
7:          <critical section>
8:          door = false
9:          inside[i] = false
10:     <2-process exit code (0)>
11: else slow-path()
```

```
Shared variables:
    race: initially ⊥
    door: initially false
    inside[0,..,n-1]: all initially false

procedure slow-path
15: <n-process entry code>
16:     <2-process entry code (1)>
17:         <critical section>

19:         inside[i] = false
20:         if for all j, inside[j] == false
21:             door = false

23:     <2-process exit code (1)>
24: <n-process exit code> & exit
```



Long-Lived Fast Mutex

```
<remainder code>
1: race = id
2: inside[i] = true
3: if door == true slow-path()
4: door = true
5: if race == id
6:      <2-process entry code (0)>
7:          <critical section>
8:          door = false
9:          inside[i] = false
10:     <2-process exit code (0)>
11: else slow-path()
```

```
Shared variables:
    race: initially ⊥
    door: initially false
    inside[0,..,n-1]: all initially false
    checking: initially false
procedure slow-path
15: <n-process entry code>
16:     <2-process entry code (1)>
17:         <critical section>
18:         checking = true
19:         inside[i] = false
20:         if for all j, inside[j] == false
21:             door = false
22:         checking = false
23:     <2-process exit code (1)>
24: <n-process exit code> & exit
```

Long-Lived Fast Mutex

```
<remainder code>
1: race = id
2: inside[i] = true
3: if door or checking == true slow-path()
4: door = true
5: if race == id
6:      <2-process entry code (0)>
7:           <critical section>
8:           door = false
9:           inside[i] = false
10:     <2-process exit code (0)>
11: else slow-path()
```

```
Shared variables:
     race: initially ⊥
     door: initially false
     inside[0,..,n-1]: all initially false
     checking: initially false
procedure slow-path
15: <n-process entry code>
16:      <2-process entry code (1)>
17:           <critical section>
18:           checking = true
19:           inside[i] = false
20:           if for all j, inside[j] == false
21:                door = false
22:           checking = false
23:      <2-process exit code (1)>
24: <n-process exit code> & exit
```

Long-Lived Fast Mutex

```
<remainder code>
1: race = id
2: inside[i] = true
3: if door or checking == true slow-path()
4: door = true
5: if race == id
6:      <2-process entry code (0)>
7:           <critical section>
10:     <2-process exit code (0)>
8:           door = false
9:           inside[i] = false
11: else slow-path()
```

```
Shared variables:
     race: initially ⊥
     door: initially false
     inside[0,..,n-1]: all initially false
     checking: initially false
procedure slow-path
15: <n-process entry code>
16:      <2-process entry code (1)>
17:           <critical section>
18:           checking = true
19:           inside[i] = false
20:           if for all j, inside[j] == false
21:                door = false
22:           checking = false
23:      <2-process exit code (1)>
24: <n-process exit code> & exit
```

Long-Lived Fast Mutex

At each time, at most one process is in Lines 6-10

⇨ Mutual exclusion and no starvation

At each time, at most one process is in Lines 16-23

⇨checking == true when a process is in Lines 19-21

```
1: race = id
2: inside[i] = true
3: if door or checking == true slow-path()
4: door = true
5: if race == id
6:      <2-process entry code (0)>
7:          <critical section>
10:     <2-process exit code (0)>
8:          door = false
9:          inside[i] = false
11: else slow-path()
```

```
procedure slow-path
15: <n-process entry code>
16:     <2-process entry code (1)>
17:         <critical section>
18:         checking = true
19:         inside[i] = false
20:         if for all j, inside[j] == false
21:             door = false
22:         checking = false
23:     <2-process exit code (1)>
24: <n-process exit code> & exit
```

Long-Lived Fast Mutex: Complexity

A process running solo executes O(1) steps

Otherwise, execute O(n) steps (regardless of n-process mutex algorithm)

```
1: race = id
2: inside[i] = true
3: if door or checking == true slow-path()
4: door = true
5: if race == id
6:      <2-process entry code (0)>
7:          <critical section>
10:     <2-process exit code (0)>
8:          door = false
9:          inside[i] = false
11: else slow-path()
```

```
procedure slow-path
15: <n-process entry code>
16:     <2-process entry code (1)>
17:         <critical section>
18:         checking = true
19:         inside[i] = false
20:         if for all j, inside[j] == false
21:             door = false
22:         checking = false
23:     <2-process exit code (1)>
24: <n-process exit code> & exit
```