Topic 2: Mutual Exclusion

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.

**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.

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

Impossible! No process leaves CS so $V$ stays 1
T&S Algorithm Ensures No Deadlock

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.

\[ V = 0 \text{ if and only if no process is in the critical section} \]

\( V \) 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 \( \text{rmw} \) operation, for some function \( f \)

Can pack multiple variables

The special case of \( f \equiv +1 \), is called \( \text{fetch&inc} \)
Overview of Algorithm

Virtually, processes wait in a circular queue of length \( n \)

Waiting process \textit{locally} stores its position in the queue

Shared pointers \textit{first} and \textit{last} track the active part of the queue
- Indices between 0 and \( n-1 \)
- Packed into one shared variable \( V \)

\textbf{Space complexity}
- \( V \) has \( n^2 \) states
- Size of \( V \) is \( 2\log_2 n \) bits

Mutex Algorithm Using RMW

\textbf{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)
```

\textbf{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 ($V\text{.first}$) 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

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

exit section:
```python
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

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

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

\[ \begin{align*}
C & \xrightarrow{p_0 \text{ solo}} C_0 \xrightarrow{p_1} C_1 \xrightarrow{p_2} C_2 \ldots \xrightarrow{p_{n-1}} C_{n-1} \\
& \quad \text{with } p_0 \text{ in CS, } \ldots, p_{n-1} \text{ in entry } \}
\end{align*} \]

Contradiction!

\[ p_h \text{ enters CS } k+1 \text{ times} \]

---

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 \( \rho \)

\( \triangleright \) Indistinguishability
Lower Bound: Afterthoughts

Does the proof work with no starvation?

A more complicated proof shows that number of memory states is $\sqrt{n}$
$\implies \Omega(\log n)$ bits

# Shared Memory States: Summary

<table>
<thead>
<tr>
<th>Progress property</th>
<th>Upper bound</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>no deadlock</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(test&amp;set alg)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no starvation</td>
<td>$n/2 + c$</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>(Burns et al.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bounded waiting</td>
<td>$n^2$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>(FIFO)</td>
<td>(queue)</td>
<td></td>
</tr>
</tbody>
</table>
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 pi
  – 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)

pi’s most recent read of Number[k]  
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)

pi’s most recent write to Number[i]  
pi reads false from Choosing[k]  
pk chooses number after seeing pi’s number, and picks a larger one  
(Number[k],k) > (Number[i],i)
Proof of Key Claim: Case 2

When process $i$ is in the critical section for every process $k \neq i$ not in the remainder ($\text{Number}[k] \neq 0$), $(\text{Number}[i], i) < (\text{Number}[k], k)$

*Proved using arguments similar to Case 1.*

Mutual Exclusion for Bakery Algorithm

**Lemma:** If $p_i$ is in the critical section, then $\text{Number}[i] > 0$.

Proof by straightforward induction.

- If $p_i$ and $p_k$ are simultaneously in CS, both have $\text{Number} > 0$.
- By previous lemma,
  - $(\text{Number}[k], k) > (\text{Number}[i], i)$ and
  - $(\text{Number}[i], i) > (\text{Number}[k], k)$

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

<table>
<thead>
<tr>
<th>Progress property</th>
<th># memory states</th>
<th># read / write variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>no deadlock</td>
<td>2 (test&amp;set alg)</td>
<td>1</td>
</tr>
<tr>
<td>no starvation</td>
<td>n/2 + c (Burns et al.)</td>
<td>3n Booleans (tournament)</td>
</tr>
<tr>
<td>bounded waiting</td>
<td>n² (queue)</td>
<td>2n unbounded (bakery)</td>
</tr>
<tr>
<td>(FIFO)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flag Principle

Bounded 2-Process Mutex w/o Deadlock

Entry section

<table>
<thead>
<tr>
<th>Process (P_0)</th>
<th>Process (P_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Want[0] = 1</td>
<td>Want[1] = 0</td>
</tr>
<tr>
<td>wait until Want[1] == 0</td>
<td>wait until Want[0] == 0</td>
</tr>
<tr>
<td></td>
<td>Want[1] = 1</td>
</tr>
<tr>
<td></td>
<td>if Want[0] == 1 goto Line 1</td>
</tr>
</tbody>
</table>

Exit section:

| Want[0] = 0 | Want[1] = 0 |
Bounded 2-Process Mutex w/o Deadlock

Entry section

Process $P_0$

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.

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

Contradiction!

Bounded 2-Process Mutex w/o Starvation

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 $\text{Want}[1] = 0$.
• 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
Both processes are in their entry section.
Priority remains fixed, e.g. at 0

$p_0$ stuck with $\text{Want}[0] = 1$
waiting for $\text{Want}[1] = 0$
$p_1$ stuck with $\text{Want}[1] = 0$
waiting for $\text{Want}[0] = 0$

No-Deadlock for 2-Process Mutex

Code for $p_0$
\[
\text{Want}[i] = 0 \\
\text{wait until Want}[1-i] == 0 \text{ or Priority} == i \\
\text{Want}[i] = 1 \\
\text{if (Priority} == 1-i) \text{ then} \\
\quad \text{if (Want}[1-i] == 1) \text{ then goto Line 1} \\
\quad \text{else wait until (Want}[1-i] == 0) \\
\]

Code for $p_1$
\[
\text{Want}[i] = 0 \\
\text{wait until Want}[1-i] == 0 \text{ or Priority} == i \\
\text{Want}[i] = 1 \\
\text{if (Priority} == 1-i) \text{ then} \\
\quad \text{if (Want}[1-i] == 1) \text{ then goto Line 1} \\
\quad \text{else wait until (Want}[1-i] == 0) \\
\]

No-Starvation for 2-Process Mutex

$p_0$ is starved
no deadlock $\Rightarrow p_1$ repeatedly enters CS

$p_0$ stuck in entry
No-Starvation for 2-Process Mutex

- \( \text{Want}[i] = 0 \)
  - wait until \( \text{Want}[1-i] = 0 \) or \( \text{Priority} = i \)
  - \( \text{Want}[i] = 1 \)
  - if (Priority == 1-i) then
    - if (Want[1-i] == 1) then goto Line 1
  - else wait until (Want[1-i] == 0)
  - Priority = 1-i
  - \( \text{Want}[i] = 0 \)

\( p_0 \) 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

What to do with > 2 Processes?

- tournament

- \( p_0 \), \( p_1 \), \( p_2 \), \( p_3 \)
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

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

Tree nodes numbered in preorder
$p_i$ begins at node $2^k\lceil i/2 \rceil$, playing role of $p_i \mod 2$

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 $\lfloor v/2 \rfloor$

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

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

<table>
<thead>
<tr>
<th>Progress property</th>
<th># read / write variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>no deadlock</td>
<td></td>
</tr>
<tr>
<td>no starvation (tournament)</td>
<td>3n Booleans</td>
</tr>
<tr>
<td>FIFO (bakery)</td>
<td>2n (Booleans + unbounded)</td>
</tr>
</tbody>
</table>

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 \sim P 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 \sim P D for some quiescent configuration D
**Warm-Up Lemma**

**Lemma:** If C is p-quiescent, then there is a p-only schedule $\sigma$ 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 $\sigma$ 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 \( p_0, \ldots, p_{k-1} \) 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

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 \( p_0, \ldots, p_{k-1} \) 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
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

$\triangleright$ 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

\[ \text{Inductive Step: Hiding } p_{k+1} \]

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
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: 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
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)
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:

\[
\text{myPlace} = \text{rmw(}\text{Last, Last+1 mod n})
\]

wait until Flags[myPlace] == 1

Flags[myPlace] = 0

Flags[myPlace+1 mod n] = 1

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

Local-Spin Mutex w/ Swap

Atomic register-to-memory swap operations, also called fetch-and-store
More common than \textit{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

[Grunke & 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:**

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

**exit section:**

```plaintext
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

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

![Diagram of MCS Lock: Unlock Subtleties](image)
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

Uses swap and CAS
Is this algorithm local-spin?
In the CC model? Yes
In the DSM model? Yes
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: 1st 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

Ensures mutual exclusion
But may deadlock

© Hagit Attiya
Local-Spin 2-Process Mutex: Avoid Deadlock

Shared variables:
- Tie, Want[0], Want[1]: initially $\bot$
- Spin[0], Spin[1]: initially $\bot$

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

release-lock(side)
- Want[side] = $\bot$ // cancel announcement
  if (Tie $\neq i$) Spin[1-side] = 2 // release other

Is this local spinning in DSM?

© Hagit Attiya
Local-Spin 2-Process Mutex

Shared variables:
- Tie, Want[0], Want[1]: initially \( \perp \)
- Spin[0, ..., n-1]: initially \( \perp \)

acquire-lock(side)
- Want[side] = i \hspace{1em} // announce your identity
- Tie = i \hspace{1em} // tie breaker
- Spin[i] = 0
- opponent = Want[1-side] \hspace{1em} // who’s competing
  - if ( opponent \( \neq \) \( \perp \)) and ( Tie == i )
    - if ( Spin[opponent] == 0 ) Spin[opponent] = 1
    - wait until ( Spin[i] \( \neq \) 0 )
    - if ( Tie == i ) wait until ( Spin[i] > 1 )

release-lock(side)
- Want[side] = nil
- opponent = Tie \hspace{1em} // who’s competing
  - if ( opponent \( \neq \) i ) Spin[opponent] = 2

Example (for processes 3 and 7)

Want[0] = 3 \hspace{1em} Want[1] = 7
Tie = 3
Spin[3] = 0 \hspace{1em} Tie = 7
opponent = 7 \hspace{1em} Spin[7] = 0
opponent <> \( \perp \) and Tie <> 3
CRITICAL
opponent ↔ \( \perp \) and Tie ↔ 7
CRITICAL
opponent <> \( \perp \) and Tie == 7
CRITICAL
CRITICAL
WAIT until Spin[7] <> 0
CRITICAL
WAIT
Spin[7] = 1
Spin[7] = 2 \hspace{1em} 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 = Want[1-side]
  - if ( opponent ≠ nil ) 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
  - if ( opponent ≠ i ) Spin[opponent] = 2

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 with $O(1)$ step complexity.
Splitter Implementation: Race Variable

Shared variable: race, initially -1
1. race = id_i
2. if race == id_i then win
3. else lose

If a process is alone, clearly wins

But it is possible that two processes win

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

1. race = id₁ // write your identifier
2. if door then return( lose )
3. door = true
4. if (race == id₁ ) // check race variable
   then return( win )
5. else return( lose )

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

Splitter Implementation: Race Variable

Shared variables
  door, initially false
  race, initially -1

1. race = id₁ // write your identifier
2. if door then return( lose )
3. door = true
4. if (race == id₁ ) // check race variable
   then return( win )
5. else return( lose )
Splitter Implementation: Doorway

Shared variables
   door, initially false
   race, initially -1

1. race = id_1 // write your identifier
2. if door then return( lose )
3. door = true
4. if (race == id_1 ) // check race variable
    then return( win )
5. 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
Correctness of the Splitter

At most one process wins the splitter

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

pj: last process to write to race before door is set to true

No process pi ≠ pj can win:
- pi ∉ P loses in Line 2.
- pi ∈ P writes to race before pj but checks again (Line 5) after pj’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)

pj: last process to write to race before door is set to true

No process pi ≠ pj can win:
- pi ∉ P loses in Line 2.
- pi ∈ P writes to race before pj but checks again (Line 5) after pj’s write and loses

P_i:
write race = i
set door
read race == i

P_j:
write race = j
read door
read race == j
Detour: Splitting the Losers

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 )

Detour: Splitting the Losers

1. race = id_i // write your identifier
2. if door then return( right )
3. door = true
4. if (race == id_i ) // check race variable
   then return( stop )
5. 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

1. race = id₁ // write your identifier
2. if door then return( right )
3. door = true
4. if (race == id₁) // check race variable
   then return( stop )
5. 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
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

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

procedure slow-path

<n-process entry code>

<2-process entry code (0)>

<critical section>

<2-process exit code (0)>

<n-process exit code> & exit

<n-process entry code>

<2-process entry code (1)>

<critical section>

<2-process exit code (1)>

<n-process exit code> & exit

<n-process entry code>

<2-process entry code (1)>

<critical section>

<2-process exit code (1)>

<n-process exit code> & exit

<n-process entry code>

<2-process entry code (1)>

<critical section>

<2-process exit code (1)>

<n-process exit code> & exit

<n-process entry code>

<2-process entry code (1)>

<critical section>

<2-process exit code (1)>

<n-process exit code> & exit

<n-process entry code>

<2-process entry code (1)>

<critical section>

<2-process exit code (1)>

<n-process exit code> & exit

<n-process entry code>

<2-process entry code (1)>

<critical section>

<2-process exit code (1)>

<n-process exit code> & exit

<n-process entry code>

<2-process entry code (1)>

<critical section>

<2-process exit code (1)>

<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

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()

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

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

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

**Shared variables:**
- race: initially \( \perp \)
- 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: Complexity**

A process running solo executes O(1) steps

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