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

\[ \text{impossible!} \]

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

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

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.

\[ \text{no process enters CS} \]

\[ \text{no process is in CS} \]

\[ V \text{ always equals 0, next t&s returns 0} \]

\[ \text{process enters CS, contradiction!} \]

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 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:

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

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

Code for exit section:

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

The queue is not stored in shared memory
Sketch of Correctness Proof

• **Mutual Exclusion:**
  – Only the process at the head of the queue ($V\text{.}_\text{first}$) 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

Flags

Last

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

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: $C_0, e_1, C_1, e_2, \ldots$

**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

For some \( i < j \) the shared memory is in the same state in \( C_i \) and \( C_j \)

If # memory states < \( n-1 \)
Lower Bound on # Memory States

C \xrightarrow{p_0\text{ solo}} C_0 \xrightarrow{p_1\text{ in }CS} C_1 \xrightarrow{p_2\text{ in entry}} C_2 \ldots \xrightarrow{p_i\text{ in entry}} C_i \xrightarrow{p_j\text{ in entry}} C_j \ldots \xrightarrow{p_{n-1}\text{ in entry}} C_{n-1}

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 \(\rho\)

\(\text{Indistinguishability}\)
Lower Bound: Afterthoughts

Does the proof work with no starvation?

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

$\Rightarrow \Omega(\log n)$ 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>$2$ (test&amp;set alg)</td>
<td>$2$</td>
</tr>
<tr>
<td>no starvation</td>
<td>$\frac{n}{2} + c$ (Burns et al.)</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>bounded waiting (FIFO)</td>
<td>$n^2$ (queue)</td>
<td>$n-1$</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:

\[
\text{Number}[i] = 1 + \max\{\text{Number}[1], \ldots, \text{Number}[n]\}
\]
for \( j = 1 \) to \( n \) do
wait until \( \text{Number}[j] > \text{Number}[i] \)

Code for exit section:

\[
\text{Number}[i] = 0
\]

Bakery Algorithm: Take 2

Number[i], integer, initially 0
- written by \( p_i \)
- read by others

Code for entry section:

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

Code for exit section:

\[
\text{Number}[i] = 0
\]
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 \( k \neq i \) not in the remainder \( (Number[k] \neq 0) \),
\( (Number[i],i) < (Number[k],k) \)

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)

\[ \pi_i \text{ in CS and } Number[k] \neq 0 \]

\[ \pi_i \text{'s most recent read of } Number[k] \]

\[ Number[k] = 0 \]

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

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)

\[ \pi_i \text{ in CS and } Number[k] \neq 0 \]

\[ \pi_i \text{'s most recent read of } Number[k] \]

\[ Number[k] = 0 \]

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

\[ p_k \text{ chooses number after seeing } \pi_i \text{'s number, and picks a larger one} \]
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.

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
- \( (\text{Number}[i],i) > (\text{Number}[k],k) \)

¬ 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 (FIFO)</td>
<td>n^2 (queue)</td>
<td>2n unbounded (bakery)</td>
</tr>
</tbody>
</table>
Flag Principle

Bounded 2-Process Mutex w/o Deadlock

Entry section

<table>
<thead>
<tr>
<th>Process P₀</th>
<th>Process P₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>Want[0] = 1&lt;br&gt;wait until Want[1] == 0</td>
<td>Want[1] = 0&lt;br&gt;wait until Want[0] == 0&lt;br&gt;Want[1] = 1&lt;br&gt;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 P₀

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

Process P₁

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

Want flags ensure mutual exclusion (next slide)
Satisfies no deadlock (exercise)
But unfair (P₁ can starve)

Exit section

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

Mutex in 2-Process Algorithm

Suppose p₀ and p₁ 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

$P_0$'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

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

<table>
<thead>
<tr>
<th>Code for p₀</th>
<th>Code for p₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>Want[i] = 0</td>
<td>Want[i] = 0</td>
</tr>
<tr>
<td>wait until Want[1-i] == 0 or Priority == i</td>
<td>wait until Want[1-i] == 0 or Priority == i</td>
</tr>
<tr>
<td>Want[i] = 1</td>
<td>Want[i] = 1</td>
</tr>
<tr>
<td>if (Priority == 1-i) then</td>
<td>if (Priority == 1-i) then</td>
</tr>
<tr>
<td></td>
<td>if (Want[1-i] == 1) then goto Line 1</td>
</tr>
<tr>
<td></td>
<td>else wait until Want[1-i] == 0</td>
</tr>
</tbody>
</table>

No-Starvation for 2-Process Mutex

p₀ is starved
no deadlock ⇒ p₁ repeatedly enters CS

p₀ stuck in entry
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

p0 is starved, no deadlock → p1 repeatedly enters CS
p0 stuck in entry
p1 sets Priority to 0
p0 with Want[0] = 1, waits for Want[1] = 0
p1 with Want[1] = 0, waits for Want[0] = 0
p0 enters CS

What to do with > 2 Processes?

tournament
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 left, 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 $2^k \lfloor i/2 \rfloor$, 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 $\lfloor v/2 \rfloor$

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

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

C is \textit{P-quiescent} if it is indistinguishable to processes in P from a quiescent configuration

— i.e., \( C \sim D \) 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.

![Diagram](image1)

230755 (2014) Mutual exclusion
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

 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

Desired $D$ is just before $p_0$'s first write.
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

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
**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

---

**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