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 2

Number[i], integer, initially 0
   - written by pi,
   - read by others

Code for entry section:

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

Code for exit section:

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

Bakery Algorithm: Take 3

Number[i], integer, initially 0
Choosing[i], Boolean, initially false
   - written by pi,
   - read by others

Code for entry section:

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

Code for exit section:

\[
\text{Number[i]} = 0
\]
Correctness of Bakery Mutex: Key Claim

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

Seems intuitive from the code, but is not trivial

This is not exactly the original Bakery algorithm

Proof of Key Claim

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

\( \text{Number}[k] = 0 \) \( \rightarrow \) \( (\text{Number}[k],k) > (\text{Number}[i],i) \)

\( \text{p}_i \) 's most recent read of \( \text{Number}[k] \)

\( \text{p}_i \) in CS and \( \text{Number}[k] \neq 0 \)
Proof of Key Claim: Case 1

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

$$(\text{Number}[i], i) < (\text{Number}[k], k)$$

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.

$(\text{Number}[k], k) > (\text{Number}[i], i)$
Mutual Exclusion for Bakery Algorithm

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

Proof by straightforward induction.

\[\Rightarrow \text{If } p_i \text{ and } p_k \text{ are simultaneously in CS, both have Number } > 0.\]

By previous lemma,

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

\[\text{Contradiction!}\]

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 \((\text{Number}[i],i)\).

- 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>n² (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 $P_0$</th>
<th>Process $P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Want[0] = 1, wait until Want[1] == 0</td>
<td>Want[1] = 0, wait until $W[0] == 0$</td>
</tr>
<tr>
<td></td>
<td>Want[1] = 1, if Want[0] == 1 goto Line 1</td>
</tr>
</tbody>
</table>

**Exit section:**

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

---

Want flags ensure mutual exclusion (next slide)
Satisfies no deadlock (exercise)
But unfair ($P_1$ can starve)
Mutex in 2-Process Algorithm

Suppose $p_0$ and $p_1$ are simultaneously in CS.

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

Suppose $p_0$ and $p_1$ are simultaneously in CS.

Contradiction!

Process $P_0$
Want[0] = 1
wait until Want[1] == 0

Process $P_1$
Want[1] = 0
wait until W[0] == 0
Want[1] = 1
if Want[0] == 1 goto Line 1

$p_0$'s last write of 1 to Want[0]
$p_0$'s last write of 1 to Want[1]
$p_0$ reads 1 from Want[1]
Want[0] = 1
Want[1] = 1

Contradiction!
Bounded 2-Process Mutex w/o Starvation

Entry section

\[
\begin{align*}
\text{Want}[i] &= 0 \\
\text{wait until } \text{Want}[1-i] &= 0 \text{ or } \text{Priority} = i \\
\text{Want}[i] &= 1 \\
\text{if } (\text{Priority} = 1-i) \text{ then} \\
&\quad \text{if } (\text{Want}[1-i] = 1) \text{ then goto Line 1} \\
&\quad \text{else wait until } (\text{Want}[1-i] = 0)
\end{align*}
\]

Exit section:

\[
\begin{align*}
\text{Priority} &= 1-i \\
\text{Want}[1] &= 0
\end{align*}
\]

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

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

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

Cool!!!
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

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
What to do with > 2 Processes?

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 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>Can we do better?</td>
</tr>
<tr>
<td></td>
<td>2n (Booleans + unbounded)</td>
</tr>
</tbody>
</table>
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 \textbf{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.

![Diagram showing the proof of the Warm-Up Lemma](image)

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

 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
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 $p_0,...,p_{k-1}$ only, s.t.
(a) $p_0,...,p_{k-1}$ cover $k$ distinct variables in $D$
(b) $D$ is $\{p_k,...,p_{n-1}\}$-quiescent