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

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

\[ V = 0 \] 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

\[ V \text{ always equals } 0, \text{ next } t&s \text{ returns } 0 \]
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 \( \text{fetch}\&\text{inc} \)
Overview of Algorithm

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

Waiting process locally stores its position in the queue

Shared pointers \( \text{first} \) and \( \text{last} \) 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 mod n))

// 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 mod n, 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 ($V_{first}$) 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

\[
\text{myPlace} = \text{rmw}(Last, Last + 1 \mod n) \\
\text{wait until Flags[myPlace] } == 1 \\
\text{Flags[myPlace] } = 0
\]

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

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

\[ C \xrightarrow{p_0 \text{ solo}} C_0 \xrightarrow{p_1} C_1 \xrightarrow{p_2} \cdots \xrightarrow{p_i} C_i \xrightarrow{\rho} \cdots \xrightarrow{p_j} C_j \xrightarrow{\rho} \cdots \xrightarrow{p_{n-1}} C_{n-1} \]

\[ \rho \]

Contradiction!

\[ p_n \text{ enters CS } \]

\[ k+1 \text{ times} \]

Lower Bound: Afterthoughts

Why \( p_0, \ldots, p_i \) (and especially \( p_n \)) 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>n/2 + c (Burns et al.)</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>bounded waiting</td>
<td>n^2 (queue)</td>
<td>n-1</td>
</tr>
<tr>
<td>(FIFO)</td>
<td></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