Shared Memory Algorithms for Mutual Exclusion

The complexity of the solution depends on the grain of atomicity.

Atomic reads and writes
Using read-modify-writes
DEC’s LL and SC primitives

Solutions using atomic reads and writes

Intellectually challenging, and has a long history. The real world does not care for it very much. To realize the challenge, try to write your own algorithms first, and prove that it satisfies all properties. It is not easy!
Warm Up

{process 0}                      {process 1}

**define f**: shared Boolean (*initially false*)

**do** true †
   **do** f = true † skip **od**;
   f:=true;
   CS;
   f:= false;
**od**

Why doesn’t it work?

Historic Dekker’s solution
**Peterson’s two-process solution**

```plaintext
program petersen;
define flag[0], flag[1]: shared boolean;
turn: shared integer
initially flag[0] = false, flag[1] = false, turn = 0 or 1

{Program for process 0}  {Program for process 1}

<table>
<thead>
<tr>
<th>do</th>
<th>true []</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>flag[0] = true;</td>
</tr>
<tr>
<td>2:</td>
<td>turn = 0;</td>
</tr>
<tr>
<td>3:</td>
<td>do (flag[1] &amp;&amp; turn = 0) [] skip od</td>
</tr>
<tr>
<td>4:</td>
<td>critical section;</td>
</tr>
<tr>
<td>5:</td>
<td>flag[0] = false;</td>
</tr>
<tr>
<td>6:</td>
<td>non-critical section codes;</td>
</tr>
<tr>
<td>od</td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td>flag[1] = true;</td>
</tr>
<tr>
<td>8:</td>
<td>turn = 1;</td>
</tr>
<tr>
<td>9:</td>
<td>do (flag[0] &amp;&amp; turn = 1) [] skip od</td>
</tr>
<tr>
<td>10:</td>
<td>critical section;</td>
</tr>
<tr>
<td>11:</td>
<td>flag[1] = false;</td>
</tr>
<tr>
<td>12:</td>
<td>non-critical section codes;</td>
</tr>
<tr>
<td>od</td>
<td></td>
</tr>
</tbody>
</table>
```
PROOFS

1. **At most one process enters its CS.**

Let 0 be in CS. Can 1 enter its CS?

0 in CS \[\square\] flag[1] = false OR turn = 1 OR both.

To enter CS, 1 must see flag[0] = false OR turn = 0 OR both.

But 0 in CS \[\square\] flag[0] = true! So turn = 0 should hold.

Case 1.

process 0 reads \textbf{flag[1] = false} in step 3

\[\square\] process 1 has not executed step 7

\[\square\] process 1 eventually sets \textbf{turn} to 1 (step 8)

\[\square\] process 1 checks \textbf{turn} (step 9) and finds \textbf{turn =1}

\[\square\] process 1 waits in step 9 and cannot enter its CS

Case 2.

process 0 reads \textbf{turn = 1} in step 3

\[\square\] process 1 executed step 8 after 0 executed step 2

\[\square\] in step 9 process 1 reads \textbf{flag[0] = true} and \textbf{turn = 1}

\[\square\] process 1 waits in step 9 and cannot enter its CS
2. *Deadlock is impossible.*

Hint: \( (\text{flag}[1] \rightarrow \text{turn} = 0) \land (\text{flag}[0] \rightarrow \text{turn} = 1) = \text{false} \)

3. *Progress (eventual entry into CS)*

Study this yourself.
Distributed Snapshot

The one-dollar bank

Depending on how you count the money, even if exactly $1 is in circulation, and there are a million branches, you can fake it as if $1 million is in circulation.

How to collect the global state correctly?

Important applications include (i) deadlock detection (ii) termination detection (iii) rollback recovery etc.
Cuts and Consistent Cuts. A cut is a set of events – it contains at least one event per process. A cut is called consistent, if the following property holds: Let \(a, b\) be two events in a distributed system. Then

\[(a \in \text{consistent cut } C) \land (b \prec a) \land b \in C\]

Cut 1 = \(\{a, b, c, m, k\}\) is consistent

Cut 2 = \(\{a, b, c, d, g, m, e, k, i\}\) is not consistent. Why?
Properties of Consistent Cuts

The set of local states immediately before the recording events of a correct snapshot form a consistent cut.

A snapshot that is of practical interest is the most recent one. Let C1 and C2 be two consistent cuts and C1 \( \sqsubset \) C2. Then C2 is more recent than C1.

Analyze why certain cuts in the one-dollar bank are inconsistent.

Note that recording events are non-invasive, and do not alter the local or global states. How to record the global state of a distributed system?
Chandy Lamport Algorithm

It works on a (1) strongly connected graph where (2) each channel is a FIFO channel. An initiator initiates the algorithm by sending out a marker ★

White and red processes

Initially every process is white. When a process receives a marker, it turns red if it has not already done so. Furthermore, assume that every action executed by a process, and every message sent by a process gets the color of that process.
The steps of the algorithm

Step 1.
In one atomic action, the initiator does the following:

- Turns red
- Records its own state
- Sends a marker along all of its outgoing channels

Step 2
Every other process, upon receiving a marker for the first time (and before doing anything else) does the following in one atomic action:

- Turns red
- Records its own state
- Sends markers along all of its outgoing channels

The algorithm terminates when (1) every process turns red, and (2) Every process has received a marker through each incoming channel.
**Why does it work?**

Lemma 1. No red message is received via a white action.

\[
\begin{array}{cccccccc}
  w(i) & w(j) & w(k) & w(l) & r(j) & r(l) & r(i) & r(k) \\
  \hline
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

(SSS is the ideal snapshot state)

**Theorem.** The state recorded by Chandy-Lamport algorithm is equivalent to the ideal snapshot state SSS.

**Hint.** A pair of actions \((a, b)\) can be scheduled in any order, if there is no causal order between them, so \((a; b)\) is equivalent to \((b; a)\)