Distributed algorithms in the shared memory model

Emmanuelle Anceaume
emmanuelle.anceaume@irisa.fr

Slides: http://people.irisa.fr/Emmanuelle.Anceaume/
Shared memory model

• You have already seen different distributed algorithms in the message passing paradigm
• We now turn our attention to the other major communication model for distributed systems: the shared memory model
• In a shared memory system, processors communicate asynchronously via a common memory area
• This memory area contains a set of shared variables
• Several types of variables can be employed
  • Type specifies the operations that can be performed on it and the values that can be returned by the operations
Shared memory model (cont’d)

- The most common type is a read/write register
  - Operations: reads and writes such that a read returns the last written value (i.e. the value written by the last preceding write)
- Other types of shared variables exist, which are more powerful in the operations they perform
  - Test&Set(): allows to atomically read and update a variable
  - Read-Modify-Write(): allows to atomically read the current value of the variable, compute a new value as a function of the current value, write the new value to the variable, and returns the previous value of the variable.
  - Compare&Swap(): allows to atomically performs a conditional write
Shared memory model (cont’d)

• Shared variables (also called registers) can be further characterized according to their access patterns, i.e., how many processors can access the variable

• The type of shared variables determines the possibility of solving a given problem

• In this lesson we will consider the mutual exclusion problem

• Then we will see how from very basic registers we can build registers with stronger properties
  • Building a MWMR atomic register from SWSR safe ones
Modelling a Shared Memory System

• We first describe a formal model of shared memory systems
• As in the case of message-passing systems,
  • we model processors as state machines and
  • We model executions as alternating sequences of configurations and events
• The difference is the nature of configurations and events
Shared Memory System

- There are $n$ processors/cores-processes $p_1, \ldots, p_n$ running concurrently in the system
- Processors communicate by accessing shared registers $R_0, R_1, \ldots R_m$
- Each processor is modelled as a state machine
Each register has a type, which specifies
1. The values that can be taken on by the register
2. The operations that can be performed on the register
3. The values that can be returned by each operation (if any)
4. The new value of the register resulting from each operation (if any)

An initial value can be specified for each register

e.g. an integer-valued read/write register R can be accessed by two operations
1. R.read() which returns the value in R, without modifying R’s content
2. R.write(ν) changes R’s value to ν, does not return any value
Shared Memory System

Configuration

• A configuration in the shared memory model is a vector

\[ C = (q_1, q_2, \ldots, q_n, r_1, r_2, \ldots r_m), \]

where

• \( q_i \) is the state of processor \( p_i \), \( 1 \leq i \leq n \), and
• \( r_j \) is the value of register \( R_j \), \( 1 \leq j \leq m \)

• In the initial configuration \( C_0 \), all the processors are in their initial state and the registers are initialised with their initial values

➢ A configuration describes the distributed system at some point in time
Events

• An event is a computational step by any of the processors

• At each computational step by some processor $p_i$ the following happen atomically:
  1. $p_i$ chooses the shared variable (register) to access with a specific operation based on $p_i$'s current state
  2. The specified operation is performed on the shared variable
  3. $p_i$’s state changes according to $p_i$’s local program based on $p_i$’s current state and the value returned by the shared variable operation
Shared Memory System

Execution fragment

• An execution fragment of an algorithm is a finite or infinite sequence of the form

\[ C_0, \Phi_1, ..., C_{k-1}, \Phi_k, C_k, \Phi_{k+1}, ..., \]

where each \( C_k \) is a configuration and each \( \Phi_k \) is an event

• The application of \( \Phi_k \) to \( C_{k-1} \) results in \( C_k \)
  • Suppose that \( \Phi_k = i \) (computational step made by \( p_i \)) and \( p_i \)’s state in \( C_{k-1} \) indicates that \( R_j \) is the shared register to be accessed by \( p_i \)
  • \( C_k \) is the result of changing \( C_{k-1} \) in accordance with \( p_i \)’s state in \( C_{k-1} \) and the value \( R_j \) in \( C_{k-1} \)
  \[ \text{Thus the only change are to } p_i \text{’s state and the value of register } R_j \]
Shared Memory System

Execution
An execution is an execution fragment starting from an initial configuration $C_0$.

Schedule
A schedule of an execution is the sequence of steps $\Phi_1, \Phi_2, \Phi_3, ..., $ taken in the execution.

Terminated states
As in the message-passing case, each processor’s state set has a set of terminated states. When a processor is in a terminated state it does not access to any shared variable.
An execution must satisfy several properties

**Safety properties**
- A condition that must hold in every finite prefix of the sequence
- It states that nothing bad has happened yet!

**Liveness properties**
- A condition that must hold a certain number of times (probably an infinite number of times)
- It states that eventually something good must happen!
Complexity measures

In shared memory systems, there are no messages to count. The focus is on the space complexity, i.e., the amount of shared memory needed to solve a given problem.

The amount shared memory is measured in two ways:

- the number of distinct shared registers/variables, and
- the amount of shared space (i.e. number of bits, equivalently, how many distinct values) required by the algorithm.
Mutual exclusion problem

The problem concerns a group of processors which occasionally need access to some resource that cannot be used simultaneously by more than a single processor.

- The printer or any other output device
- A record of a shared database or a shared data structure, etc.

Each processor may need to execute a code segment called critical section, such that at any time:

- at most one processor is in the critical section (mutual exclusion)
- If one or more processors try to enter the critical section, then one of them eventually succeeds as long as no processor stays in the critical section forever (deadlock prevention)
Mutual exclusion problem

• Original solutions to solve the mutual exclusion problem relies on special synchronization support such as semaphores
• Here we focus on distributed software solution based on basic shared variables
• One can partition the code of a processor as follows
  1. Entry: the code executed in preparation for entering the critical section
  2. Critical: the code to be protected from concurrent execution
  3. Exit: the code executed on leaving the critical section
  4. Remainder: the rest of the code

The problem is to design the entry and exit code in a way that guarantees that the mutual exclusion and deadlock-freedom properties are satisfied.
Mutual exclusion problem

• We assume that the variables (both local and shared) accessed in the Entry and Exit sections are not accessed in the Critical and Remainder sections
• We also assume that processors do not stay forever in the critical and exit sections
• Thus we make the following assumptions:
  • If a processor takes a step while in the Remainder section then it immediately enters the Entry section
  • If a processor takes a step while in the Critical Section then it immediately enters the Exit section
• An execution is admissible if for every processor $p_i$
  • Either $p_i$ takes an infinite number of steps of computation
  • Or $p_i$ ends in the Remainder section
Mutual exclusion problem

• An algorithm for a shared memory system solves the mutual exclusion problem with no deadlock or no lockout if the following holds:

1. **Mutual exclusion**: In every configuration of every execution, at most one processor is in the critical section

2. **Deadlock-freedom**: In every admissible execution, if some processor is in the entry section in a configuration, then there is a later configuration in which some processor is in the critical section

3. **Starvation-freedom**: In every admissible execution, if some processor is in the entry section in a configuration, then there is a later configuration in which that same processor is in the critical section

Note that mutual exclusion is a safety property (invariants), and deadlock-freedom and starvation-freedom are liveness properties.
There are several ways to design mutual exclusion algorithms. They depend on the operations and properties provided to the processors/processes/threads by the underlying shared memory communication system. We distinguish 3 different families of mutual exclusion algorithms.
**Mutual exclusion problem**

1. **Specialized hardware primitives**
   - hardware primitives are offered by the multiprocessors architectures (e.g. binary test&set registers, read/modify/write registers).
   - Those primitives are more sophisticated (i.e. stronger) than simple atomic r/w registers

2. **Atomic read/write registers**
   - the only way processors communicate is through those objects

3. **Mutex without atomicity**
   - Is atomicity at a lower level required to solve atomicity at a higher level?
   - The response is no!
   - Mutual exclusion algorithms can be design with weaker registers than atomic ones
Mutual exclusion based on specialized hardware primitives

• Of course, nearly all shared memory multiprocessors propose built-in primitives, i.e., atomic operations implemented in hardware, specially designed to address synchronization issues

• E.g. Test&Set, Compare&Swap,...

• A binary Test&Set shared variable V is a binary variable that sets V to 1 and returns its previous value

• A Compare&Swap(old,new) shared variable V is a variable that compares whether V is equal to old, and if so sets it to new and return true. Otherwise it returns false
Binary Test & Set variable

- We will see that with a Test&Set variable, one bit is sufficient to guarantee mutual exclusion with no deadlock
Binary Test & Set register

- A Binary Test&Set register/variable V is a binary register that supports 2 operations
  1. test&set
  2. reset

  \[
  \text{test&set}(V) : \{
  \text{temp} := V \\
  V := 1 \\
  \text{return } (\text{temp})
  \}
  \]

  \[
  \text{reset}(V) : \{
  V := 0 \\
  \text{return}
  \}
  \]

- The test&set\((V)\) operation **atomically reads and updates the variable**: The shared variable V is “set” to 1 and its previous value is returned.
- The reset\((V)\) operation is merely a write.
Binary Test & Set register

- There is a simple Mutual Exclusion algorithm that uses a test&set register

```
Algorithm: { // Mutual_Exclusion using T&S
Initially V = 0 // Shared variable
<Entry>: 
1: wait until (test&set(V) = 0)
<Critical Section>
<Exit>: 
2: Reset(V)
<Remainder>
}
```

Assume that initially V=0

- In the entry section, pᵢ repeatedly test V until it returns 0
- The last test by pᵢ assigns 1 to V, which causes the next test to return 1, prohibiting any pⱼ, i ≠ j, from entering the CS
There is a simple Mutual Exclusion algorithm that uses a test&set register

**Algorithm:** // Mutual_Exclusion using T&S
Initially V = 0 // Shared variable
<Entry>:
1: wait until (test&set(V) = 0)
<Critical Section>
<Exit>:
2: Reset(V)
<Remainder>

**Th:** Mutual exclusion is guaranteed

Proof: by contradiction

- Suppose both p_i and p_j are in the CS together
- Let t be the earliest time at which both p_i and p_j are in the CS
- Assume that p_i is already in the CS when p_j enters it
- When p_i enters the CS, it tests V, sees that V=0, and sets V to 1
- V remains equal to 1 until some proc leaves the CS
- By assumption of t, no processor other than p_i is in the CS when
  p_j enters the CS
- Thus V is always equal to 1 up to the time p_j enters the CS
- Thus when p_j tests V it should return 1 and not 0
- Thus p_j cannot enter the CS
- A contradiction
Binary Test & Set register

• There is a simple Mutual Exclusion algorithm that uses a test&set register

Algorithm: {  // Mutual_Exclusion using T&S
Initially V = 0  // Shared variable
<Entry>:  
1: wait until (test&set(V) = 0)
<Critical Section>
<Exit>:  
2: Reset(V)
<Remainder>
}

Th: No Deadlock is guaranteed

Proof: by contradiction
- Suppose that there is an execution in which, after some point at least p_i is in the Entry section but no processor ever enters the CS
- Since no proc remains forever in the CS then there is some point from which at least p_i is in the Entry section but no processor is in the CS
- The key is to note that V=0 iff no proc is in the CS
- Thus any proc that executes L1 discovers that V=0 and enters the CS
- A contradiction

There is a simple Mutual Exclusion algorithm that uses a test&set register.

**Algorithm:**

```plaintext
{  // Mutual_Exclusion using T&S
    Initially V = 0  // Shared variable
    <Entry>:
    1: wait until (test&set(V) = 0)
    <Critical Section>
    <Exit>:
    2: Reset(V)
    <Remainder>
}
```

This algorithm guarantees mutual exclusion without deadlock with one binary Test&Set register.
Mutual exclusion based on Read/Write Register

• We now concentrate on systems in which processors access the shared variables only by read and write operations
• We will present two algorithms that provide mutual exclusion and no starvation for n processors
• Both algorithms use O(n) distinct shared registers/variables
Read/Write Register

1 – Which kind of information is contained in a register?
   - binary (boolean) / multivalued

2 – Who is allowed to access a register?
   - SWSR (Single Writter / Single Reader)
   - SWMR (Single Writter / Multiple Readers)
   - MWMR (Multiple Writter / Multiple Readers)

3 – How behave the register when concurrently accessed?
   - safe – regular - atomic
Safe read/write register

Properties:
• a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
• a read() that overlaps a write() returns any possible values of the register
• This is the simplest kind of register we can think of to enable any 2 procs to communicate

[1,..6]-valued safe register X
Safe read/write register

Properties:
• a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
• a read() that overlaps a write() returns any possible values of the register

[1,..6]-valued safe register X
Regular read/write register

Properties:

• a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one

• a read() that overlaps a write() returns either the old or the new written value
Regular read/write register

Properties:
• a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
• a read() that overlaps a write() returns either the old or the new written value

[1,..6]-valued regular register X
Atomic read/write register

Properties:
• a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one

[1,..6]-valued regular register X
Atomic read/write register

Properties:
• A read() not concurrent with any write() obtains the correct value, i.e., the most recently written one.
• Concurrent reads() and writes() behave as if they occur in some definite order, i.e., as if reads() and writes() occur sequentially.

[1,..6]-valued atomic register X
Atomic read/write register

More precisely, in presence of concurrency

- (a) Each operation invocation appears as if it had been executed instantaneously at a point of the time line between its start event and its end event, and
- (b) the resulting sequence of invocations is such that any read invocation returns the value written by the closest preceding write invocation.
Atomic read/write register

More precisely, in presence of concurrency

• (a) Each operation invocation appears as if it had been executed instantaneously at a point of the time line between its start event and its end event, and

• (b) the resulting sequence of invocations is such that any read invocation returns the value written by the closest preceding write invocation.
Atomic read/write register

More precisely, in presence of concurrency

• (a) Each operation invocation appears as if it had been executed instantaneously at a point of the time line between its start event and its end event, and

• (b) the resulting sequence of invocations is such that any read invocation returns the value written by the closest preceding write invocation.
Atomic Read/Write Register

Properties of an atomic shared register

1. Each invocation of a read or write operation
   1. Appears as if it was executed at a single point $\ell(\text{op})$ of the time line
   2. $\ell(\text{op})$ is such that $s(\text{op}) \leq \ell(\text{op}) \leq e(\text{op})$
   3. For any two operations $\text{op}_1$ and $\text{op}_2$: $(\text{op}_1 \neq \text{op}_2) \implies \ell(\text{op}_1) \neq \ell(\text{op}_2)$.

2. Each read invocation returns the value written by the closest preceding write operation in the sequence defined by the $\ell()$ instants associated with the invocations.

$\Rightarrow$ This means that an atomic register is such that all its operations invocations *appear* as if they have been executed sequentially
Atomic Read/Write Register

Properties of an atomic shared register

3. An atomic r/w register can be
   • single-writer/single-reader (SWSR) : a single process can write the register and a single one can read it, both processes being different
   • single-writer/multiple-reader (SWMR) : a single process can write the register but it can be read by multiple processes
   • multiple-writer/multiple-reader (MWMR) : the register can be read and written by multiple processes
Example

Space-time diagram depicting an execution of a MWMR register
Atomic Read/Write Register

Questions What are the correct execution(s) ?

Space-time diagram depicting an execution of a MWMR register
Atomic Read/Write Register

Questions What are the correct execution(s) ?

X.write(1), X.read() → 1,
Atomic Read/Write Register

Questions What are the correct execution(s)?

$p_2 \xrightarrow{X.\text{write}(1)} X.\text{read()} \rightarrow 1$

$p_3 \xrightarrow{X.\text{write}(3)} X.\text{write}(2) \xrightarrow{X.\text{write}(2)} X.\text{read()} \rightarrow ?$

X.\text{write}(1), X.\text{read()} \rightarrow 1,
Questions What are the correct execution(s)?

Atomic Read/Write Register

X.write(1), X.read() → 1, X.write(2), X.write(3),
Questions: What are the correct execution(s)?

Atomic Read/Write Register

X.write(1), X.read() → 1, X.write(2), X.write(3), X.read() → 3, X.read() → 3
Atomic Read/Write Register

Questions What are the correct execution(s) ?

X.write(1), X.read() →1, X.write(2), X.read()→2, X.write(3), X.read()→3
Concurrency is intimately related to non-determinism

• It is not possible to predict which execution will be produced
• It is only possible to enumerate the set of possible correct executions that could be produced

• Why Atomicity is important?

• Atomicity allows the composition of shared objects for free
• i.e. if you consider two registers X1 and X2, the composite object [X1,X2] made of X1 and X2, and which offers to processes 4 operations is also atomic
• Everything appears as if at most one operation at a time was executed
• So we can reason on sequences not only for each register taken separately but also on the whole set of registers as if they were a single atomic register
2-Mutual exclusion

- The algorithm ensures mutual exclusion without deadlock for two processors $p_0$ and $p_1$
- It is due to Peterson (1981)
- This algorithm gives priority to one processor $p_0$
- We will convert this algorithm to one that guarantees no starvation

- Each processor $p_i$ uses 1 binary SWMR atomic register $\text{Want}[i]$
  - $\text{Want}[i]=1$ if processor $p_i$ is interested in entering the critical section and
  - $\text{Want}[i]=0$ otherwise
“2-Mutual exclusion” using SWMR atomic registers

Algorithm: Mutual exclusion for two processes $p_0$ and $p_1$

- Initially $Want[0]=0$; $Want[1]=0$;

<table>
<thead>
<tr>
<th>Code executed by $p_0$</th>
<th>Code executed by $p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;Entry&gt;</strong>:</td>
<td><strong>&lt;Entry&gt;</strong>:</td>
</tr>
<tr>
<td>1:</td>
<td>1: $Want[1] := 0$</td>
</tr>
<tr>
<td>2:</td>
<td>2: wait until ($Want[0]=0$)</td>
</tr>
<tr>
<td>4:</td>
<td>4:</td>
</tr>
<tr>
<td>5: wait until $Want[1]=0$</td>
<td>5: if ($Want[0]=1$) then goto Line1</td>
</tr>
<tr>
<td><strong>&lt;Critical Section&gt;</strong></td>
<td><strong>&lt;Critical Section&gt;</strong></td>
</tr>
<tr>
<td><strong>&lt;Exit&gt;</strong>:</td>
<td><strong>&lt;Exit&gt;</strong>:</td>
</tr>
<tr>
<td>7:</td>
<td>7:</td>
</tr>
<tr>
<td>8: $Want[0] = 0$</td>
<td>8: $Want[1] = 0$</td>
</tr>
<tr>
<td><strong>&lt;Remainder&gt;</strong></td>
<td><strong>&lt;Remainder&gt;</strong></td>
</tr>
</tbody>
</table>

The algorithm is asymmetric: $p_0$ enters the CS whenever CS is empty, without considering that $p_1$’s wants to do so $p_1$ enters the CS only if $p_0$ is not interested in it at all

$p_i$ raises its flag to notify that it wishes to enter the CS

$p_i$ observes $p_{1-i}$’s flag, and if up waits

**If $p_i$ executes this part of the code then $Want[i]=1$**
Algorithm: Mutual exclusion for two processes $p_0$ and $p_1$

Initially $\text{Want}[0] = 0$; $\text{Want}[1] = 0$;

<table>
<thead>
<tr>
<th>Code executed by $p_0$</th>
<th>Code executed by $p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;Entry&gt;</strong></td>
<td><strong>&lt;Entry&gt;</strong></td>
</tr>
<tr>
<td>1:</td>
<td>1: ( \text{Want}[1] := 0 )</td>
</tr>
<tr>
<td>2:</td>
<td>2: \text{wait until (Want}[0] = 0)</td>
</tr>
<tr>
<td>3: \text{Want}[0] := 1</td>
<td>3: \text{Want}[1] := 1</td>
</tr>
<tr>
<td>4:</td>
<td>4:</td>
</tr>
<tr>
<td>5: \text{if (Want}[0] = 1 \text{ then goto Line1}</td>
<td>5:</td>
</tr>
<tr>
<td>6: \text{wait until Want}[1] = 0</td>
<td>6:</td>
</tr>
<tr>
<td><strong>&lt;Critical Section&gt;</strong></td>
<td><strong>&lt;Critical Section&gt;</strong></td>
</tr>
<tr>
<td><strong>&lt;Exit&gt;</strong></td>
<td><strong>&lt;Exit&gt;</strong></td>
</tr>
<tr>
<td>7:</td>
<td>7:</td>
</tr>
<tr>
<td>8: \text{Want}[0] = 0</td>
<td>8: \text{Want}[1] = 0</td>
</tr>
<tr>
<td><strong>&lt;Remainder&gt;</strong></td>
<td><strong>&lt;Remainder&gt;</strong></td>
</tr>
</tbody>
</table>

Th: The algorithm provides Mutual Exclusion

Proof: by contradiction

- Assume both $p_0$ and $p_1$ are in the CS
- By construction both $\text{Want}[i] = 1$
- Assume that wlog that $p_0$ ’s last write to \text{Want}[0] before entering the CS follows $p_1$ ’s last write to \text{Want}[1] before entering the CS
- We have “\text{Want}[0].\text{write}(1)” -> “\text{Want}[1].\text{read}()” and “\text{Want}[1].\text{write}(1)” -> “\text{Want}[0].\text{write}(1)”,
- thus $p_0$ ’s read of $\text{Want}[1]$ should return 1 ($\text{Want}$ registers are atomic, thus return the last written value).

A contradiction
“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic registers

• The algorithm is deadlock-free (left as exercise)
• However it is not starvation-free: if \( p_0 \) is continuously interested in entering the CS then \( p_1 \) will never enter it because it gives up when \( p_0 \) is interested.
• To achieve no lockout we modify the algorithm so that instead of always giving priority to \( p_0 \), each processor gives priority to the other processor on leaving the CS

• Each processor \( p_i \) uses 1 binary SWMR register \( \text{Want}[i] \) and 1 MWMR register \( \text{Priority} \)
  • \( \text{Want}[i]=1 \) if processor \( p_i \) is interested in entering the critical section and
  • \( \text{Want}[i]=0 \) otherwise
  • \( \text{Priority} = i \) if processor \( p_i \) has priority at the moment. It is initialized to 0
"2-Mutual exclusion" using 2 SWMR and 1 MWMR atomic registers

Algorithm: 2-Mutual exclusion for two processes

Code executed by p_i

---

Initially **Want[i]=0; Priority:=0**

<Entry>:
1: **Want[i]:=0**
2: wait until (**Want[1-i]=0** or **Priority = i**)  
3: **Want[i]:=1**
4: if (**Priority =1-i**) then
5: if (**Want[1-i] = 1**) then goto Line 1
6: else wait until **Want[1-i]=0**

<Critical Section>

<Exit>:
7: **Priority =1-i**
8: **Want[i]=0**

<Remainder>

**Th: The algorithm provides Mutual Exclusion**

**Proof:** by contradiction

Assume both p_0 and p_1 are in the CS

- By construction both **Want[i]=1**
- Assume that wlog that p_0 ‘s last write to **Want[0] before entering** the CS follows p_1 ‘s last write to **Want[1] before entering** the CS
- Note that p_0 can enter the CS either through Line 5 or Line 6.
- In both cases p_0 reads **Want[1]=0**.
- However by assumption we have
  - "**Want[0].write(1)" -> "Want[1].read()** “ and
  - "**Want[1].write(1)" -> "Want[0].write(1)”, and thus

p_0 reads **Want[1]** should return 1

A contradiction

![Diagram showing the states of the processes](diagram.png)
The algorithm is without deadlock

Proof: by contradiction
Suppose that there is a point after which some processor is forever in the <Entry> section, and no processor enters the CS

Case 1: both $p_0$ and $p_1$ are in the <Entry> section.

- Thus $Priority$ never changes. Wlog $Priority=0$.
- Thus $p_0$ passes the test in Line 2 and loops forever in Line 6 with $Want[0]:=1$.
- Since $Priority=0$, $p_1$ does not reach Line 6 and wait in Line 2, with $Want[1]:=0$.
- Thus $p_0$ will pass the test in Line 6 and enters the CS. A contradiction.

Case 2: A single processor is forever in the <Entry> section. Say $p_0$.

- Since $p_1$ does not stay forever in the CS or in the <Exit> section, it will set $Priority=1$ and $Want[1]=0$ forever.
- Thus $p_0$ does not loop forever in the <Entry> section (lines 2, 5, 6) and enters the CS. A contradiction.
Th: The algorithm provides No starvation

Proof: by contradiction
Suppose that there is a configuration after which some processor is starved and thus is forever in the <Entry> section. Wlog this is \( p_0 \).

Case 1: Suppose \( p_1 \) executes line 7 at some later point.
- \( Priority = 0 \) forever after.
- Thus \( p_0 \) passes the test L2 and skips Line 4
- Thus \( p_0 \) is stuck executing line 6 waiting for \( want[1]=0 \) which never occurs
- But since \( p_1 \) does not stay forever in the CS, this would mean that
- it is stuck in the Entry section, violating the deadlock free property

Case 2: \( p_1 \) never executes line 7 at any later point.
- Since no-deadlock holds, \( p_1 \) is forever in the remainder section.
- Thus, \( want[1] == 0 \) henceforth.
- Then \( p_0 \) cannot be stuck in the entry section!
A contradiction!!!
“n-Mutual exclusion” using SWMR and MWMR atomic registers

• To solve the mutual exclusion problem with \( n \) processors we rely on the 2-Mutual exclusion solution.

• Processes compete pairwise, using a two process algorithm.

• The pairwise competitions are arranged in a complete binary tree.

• The tree is called the tournament tree.
“n-Mutual exclusion” using SWMR and MWMR atomic registers

• Each process begins at a specific leaf of the tree
• At each level, the winner moves up to the next higher level, and competes with the winner of the competition on the other side.
• The process on the left side plays the role of \( p_0 \), while the process on the right side plays the role of \( p_1 \).
• The process that wins at the root enters the critical section
“n-Mutual exclusion” using SWMR and MWMR atomic registers

• Let $k = \lceil \log n \rceil - 1$
• The root is numbered 1
• The left child of node $v$ is numbered $2v$
• The right child of node $v$ is numbered $2v + 1$
• Note that the leaves of the tree are numbered:
  \[ 2^k, 2^{k+1}, ..., 2^{k+1} - 1 \]
• With each node $v$ of the binary tree, we associate
  • $\text{Want}^v[0] = 0$, $\text{want}^v[1] = 0$, $\text{Priority}^v = 0$
  which are the variables for the instance of the “2-Mutex” executed at this node $v$
• The algorithm is recursive
“n-Mutual exclusion” using SWMR and MWMR atomic registers

Algorithm: n-Mutual exclusion for two processes
---------------------------------------------------------------
Procedure Node(v:integer, side:0/1)
1: \textbf{Want}^v[\textbf{side}]:=0
2: wait until (\textbf{Want}^v[1-\textbf{side}]=0 or \textbf{Priority}^v = \textbf{side})
3: \textbf{Want}^v[\textbf{side}]:=1
4: if (\textbf{Priority}^v =1-\textbf{side}) then
5: \hspace{1cm} if (\textbf{Want}^v[1-\textbf{side}]=1) then goto Line 1
6: else wait until \textbf{Want}^v[1-\textbf{side}]=0
7: if (v=1) then
8: \hspace{1cm} \textbf{<Critical Section>}
9: else Node(\lfloor v/2 \rfloor,v \mod 2)
10: \textbf{Priority}^v =1-\textbf{side}
11: \textbf{Want}^v[\textbf{side}]=0
End procedure

A Critical Section at each node v

The CS at each node v (L7-9) includes
- the Entry section (L1-6) executed by the nodes on the path from that node’s parent to the root,
- the real CS, and
- the exit section (L10-11) executed by the nodes from the root to that node’s parent.
“n-Mutual exclusion” using SWMR and MWMR atomic registers

Algorithm: n-Mutual exclusion for two processes

Procedure Node(v:integer, side:0/1)
1:  \textit{Want}^v[\textit{side}] := 0
2:  wait until (\textit{Want}^v[1-\textit{side}] = 0 \text{ or } \textit{Priority}^v = \textit{side})
3:  \textit{Want}^v[\textit{side}] := 1
4:  if (\textit{Priority}^v = 1 - \textit{side}) then
5:      if (\textit{Want}^v[1-\textit{side}] = 1) then goto Line 1
6:      else wait until \textit{Want}^v[1-\textit{side}] = 0
7:  if (v=1) then
8:      \textit{<Critical Section>}
9:  else Node(\lfloor v/2 \rfloor, v \mod 2)
10: \textit{Priority}^v = 1 - \textit{side}
11: \textit{Want}^v[\textit{side}] = 0
End procedure

To begin the competition for the CS, process \( p_i \) begins by calling \( \text{Node}(2^k + \lfloor i/2 \rfloor, i \mod 2) \), where \( k = \lceil \log n \rceil - 1 \)

Ex. \( n = 8, k=2 \)

\( p_0 \rightarrow \text{Node}(4,0); p_1 \rightarrow \text{Node}(4,1); p_2 \rightarrow \text{Node}(5,0); p_3 \rightarrow \text{Node}(5,1); \)
\( \ldots p_6 \rightarrow \text{Node}(7,0), p_7 \rightarrow \text{Node}(7,1) \)
“n-Mutual exclusion” using SWMR and MWMR atomic registers

Algorithm: n-Mutual exclusion for two processes
-----------------------------------------------------------------
Procedure Node(v:integer, side:0/1)
1:  Want^v[side]: =0
2:  wait until (Want^v[1-side]=0 or Priority^v = side)
3:  Want^v[side]: =1
4:  if (Priority^v =1-side) then
5:   if (Want^v[1-side] = 1) then goto Line 1
6:  else wait until Want^v[1-side] =0
7:  if (v=1) then
8:   <Critical Section>
9:  else Node(⌊v/2⌋,v mod 2)
10:  Priority^v =1-side
11:  Want^v[side]=0
End procedure

Correctness proof

We need to show that for every node v of the tournament algorithm, only one process performs instruction Node(v,0) and Node(v,1) at a time.

By induction on the level of v, starting at the leaves

1. Basis: v is a leaf. By construction only one processor ever performs the instructions of Node(v,i), i=0,1
2. Induction: v is not a leaf.
   - By the code, if a processor executes Node(v,0), then it is in the critical section for v’s left child.
   - By the inductive hypothesis and the fact that the 2-mutex algorithm guarantees mutual exclusion, only one processor at a time is in the critical section for v’s left child.
   - Thus only one processor at a time executes instructions of Node(v,0).
   - The same argument applies to show that a single processor executes the instructions of Node(v,1).
“n-Mutual exclusion” using SWMR and MWMR atomic registers

Algorithm: n-Mutual exclusion for two processes

```
Procedure Node(v:integer, side:0/1)
1:   Want^v[side]:=0
2:   wait until (Want^v[1-side]=0 or Priority^v = side)
3:   Want^v [side] :=1
4:   if (Priority^v =1-side) then
5:      if (Want^v[1-side] = 1) then goto Line 1
6:   else wait until Want^v[1-side] =0
7:   if (v=1) then
8:      <Critical Section>
9:   else Node(⌊v/2⌋,v mod 2)
10:  Priority^v =1-side
11:  Want^v[side]=0
End procedure
```

Complexity

This algorithm requires 3 shared read/write registers at each node of the binary tree. Thus O(n) shared read/write variables are needed.
Mutual exclusion without relying on atomic primitives

• So far, we have seen that one can solve mutual exclusion either by using high level hardware primitives or by using atomic objects, that is objects which, when concurrently accessed by different processes, behave as if they were accessed sequentially.

• We will now see that we can solve mutual exclusion with weaker types of objects.

  Safe read/write registers

• By doing so, we will show that we can implement atomic operations without relying on underlying atomic objects.
Read/Write safe register

Properties:

• a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one

• a read() that overlaps a write() returns any possible values of the register

[1,..6]-valued safe register X
Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Bakery algorithm (Lamport)
• Algorithm that guarantees mutual exclusion among n processors
• Very simple algorithm that guarantees first-in-first-served property
• safe MWMR registers

Main idea
• Considering processors that wish to enter the critical section as customers in a bakery
• Each customer arriving at the bakery gets a ticket and the next with the smallest ticket is the next to be served
• A customer which is not waiting in the line is ticket “0”
In practice

**Shared registers (SWMR):**

- **Number[n]** = array of n non-negative integers
  - Number[i] = ticket number of processor \( p_i \) - writable solely by \( p_i \)
- **Flag[n]** = array of n Boolean values
  - Flag[i] = true while \( p_i \) is in the process of obtaining its ticket - writable solely by \( p_i \)

**Each processor** \( p_i \) wishing to enter the CS

- chooses a number which is greater than the numbers of all the other processors and writes it to its ticket = \((\text{Number}[i], i)\)
- after getting its ticket \( p_i \) waits until its ticket is the smallest one, and then enters the CS
Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Algorithm: Bakery (code executed by processor pi, 1 ≤ i ≤ n)
Initially Flag[i]=false and Number[i]=0 for 1 ≤ i ≤ n

<Entry>
1: Flag[i] = true
2: Number[i] = max(Number[1],.., Number[n])+1
3: Flag[i] = false
4: for j=1 to n (j ≠ i) do
5: wait until Flag[j]=false
6: wait until Number[j]=0 or (Number[j],j) > (Number[i],i)
<Critical Section>
<Exit>
7: Number[i] = 0
<Remainder>

1. Operations on registers are not atomic, i.e., they cannot be abstracted as having been executed “instantaneously”
   ➢ Thus we need to consider their beginning and ending times
2. Terminology:
   p_i is ”in the doorway” when it executes L2
   p_i is “in the bakery” when it executes lines L3-7
Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Algorithm: Bakery (code executed by processor pi, 1 ≤ i ≤ n)
Initially Flag[i]=false and Number[i]=0 for 1 ≤ i ≤ n

<Entry>:
1: Flag[i] = true
2: Number[i] = max(Number[1],.., Number[n])+1
3: Flag[i] = false
4: for j=1 to n (j ≠ i) do
5: wait until Flag[j]=false
6: wait until Number[j]=0 or (Number[j],j) > (Number[i],i))

<Critical Section>

<Exit>:
7: Number[i] = 0

<Remainder>:

Lemma 1: Let p_i and p_j be both in the bakery, and such that p_i entered the bakery (L3-7) before p_j entered the doorway (L2). Then Number[i] < Number[j]

Proof
➢ Since p_i enters the bakery before p_j entered the doorway, Number[i] was written before p_j reads it.
➢ Thus there is no concurrent access to Number[i]
➢ And thus p_j reads the correct value of Number[i]
➢ Thus Number[j] ≥ Number[i]+1
Algorithm: Bakery (code executed by processor $p_i$, $1 \leq i \leq n$)
Initially $Flag[i]$=false and $Number[i]$=0 for $1 \leq i \leq n$

<Entry>:
1: $Flag[i] = \text{true}$
2: $Number[i] = \max(\text{Number}[1], ..., \text{Number}[n]) + 1$
3: $Flag[i] = \text{false}$
4: for $j = 1$ to $n$ ($j \neq i$) do
5: \hspace{1em} \text{wait until } \text{Flag}[j] = \text{false}$
6: \hspace{1em} \text{wait until } \text{Number}[j] = 0 \text{ or } (\text{Number}[j], j) > (\text{Number}[i], i))

<Critical Section>

<Exit>:
7: $Number[i] = 0$

Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Lemma 2: Let $p_i$ and $p_j$ be such that $p_i$ is inside the CS while $p_j$ is in the bakery (L3-7). Then
$(\text{Number}[i], i) < (\text{Number}[j], j)$

Proof

- Notice that since $p_j$ is in the bakery it can be in the CS
- As $p_i$ is in the CS it read $Flag[j] = \text{false}$ (L5)
- According to the time of that read and the time at which $p_i$ wrote $Flag[j]$ (L1 or L3) we have
  - Either $t_1 < t_2$
  - Or $t_3 < t_4$

Case 1: $t_1 < t_2$

Case 2: $t_3 < t_4$
Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Algorithm: Bakery (code executed by processor $p_i$, $1 \leq i \leq n$
Initially $\text{Flag}[i]=\text{false}$ and $\text{Number}[i]=0$ for $1 \leq i \leq n$

<Entry>:
1: $\text{Flag}[i] = \text{true}$
2: $\text{Number}[i] = \max(\text{Number}[1],...,\text{Number}[n])+1$
3: $\text{Flag}[i] = \text{false}$
4: for $j=1$ to $n$ ($j \neq i$) do
5: wait until $\text{Flag}[j]=\text{false}$
6: wait until $\text{Number}[j]=0$ or $(\text{Number}[j],j) > (\text{Number}[i],i))$

<Critical Section>

<Exit>:
7: $\text{Number}[i] = 0$

Lemma 2: Let $p_i$ and $p_j$ be such that $p_i$ is inside the CS while $p_j$ is in the bakery. Then
$(\text{Number}[i],i) < (\text{Number}[j],j)$

Proof (continue)
- Case $t_1 < t_2$:
  - $p_i$ entered the bakery before $p_j$ enters the doorway
  - Thus from Lemma 1, $\text{Number}[i] < \text{Number}[j]$
  - Which ends this case

---

Case 1: $t_1 < t_2$

$p_i$ $\text{Flag}[j].\text{read}()$ $\rightarrow$ false (Line 5)

$p_j$ $\text{Flag}[j].\text{write}(true)$ (Line 1)

$t_1$ $t_2$
Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Algorithm: Bakery (code executed by processor pi, 1 ≤ i ≤ n)
Initially Flag[i]=false and Number[i]=0 for 1 ≤ i ≤ n

<Entry>:
1: Flag[i] = true
2: Number[i] = max(Number[1],.., Number[n])+1
3: Flag[i] = false
4: for j=1 to n (j ≠ i) do
5:     wait until Flag[j]=false
6:     wait until Number[j]=0 or (Number[j],j) > (Number[i],i))
<Critical Section>
<Exit>:
7: Number[i] = 0

Lemma 2: Let p_i and p_j be such that p_i is inside the CS while p_j is in the bakery. Then
(Number[i],i) < (Number[j],j)

Proof (continue)
- Case t3 < t4 (i.e., t_{i,s}(3) < t_{i,e}(5)): (H1)
- p_j is sequential thus p_j ended L2 < t3 (i.e t_{j,e}(2) < t_{i,s}(3)) (P1)
- p_i is sequential thus t_{i,s}(5) < t_{i,s}(6) (P2)
(3)
- p_i is sequential thus t_{i,s}(5) < t_{i,s}(6)
- t_{i,e}(2) < t_{i,s}(6) (H1)
- t_{j,e}(2) < t_{i,s}(6) (P3)
- Thus the last read by p_i of Number[j] occurred after Number[j] was written by p_j
- As p_j is in the CS it exited the 2^{nd} wait statement, i.e., Number[j]=0 or (Number[j],j) > (Number[i],i))
- As t_{i,e}(2) < t_{i,s}(6) (P3) we have Number[j]≠0
- Thus Number[j],j) > (Number[i],i))
- Which ends this case
Algorithm: Bakery (code executed by processor $pi$, $1 \leq i \leq n$)
Initially $Flag[i]=false$ and $Number[i]=0$ for $1 \leq i \leq n$

<Entry>:
1: $Flag[i] = true$
2: $Number[i] = max(Number[1], ..., Number[n])+1$
3: $Flag[i] = false$
4: for $j=1$ to $n$ ($j \neq i$) do
5: wait until $Flag[j]=false$
6: wait until $Number[j]=0$ or $(Number[j],j) > (Number[i],i))$

<Critical Section>

<Exit>:
7: $Number[i] = 0$

<Remainder>:

**Th**: The Bakery algorithm satisfies mutual exclusion

Proof

By contradiction

- Suppose that both $p_i$ and $p_j$ are both in the CS
- As $p_i$ is in the CS and $p_j$ is in the bakery, by Lemma 2, we have $(Number[i],i) < (Number[j],j)$
- Similarly, as $p_j$ is in the CS and $p_i$ is in the bakery, by Lemma 2, we have $(Number[j],j) < (Number[i],i)$
- As ($j \neq i$) both pairs are totally ordered. It follows that each pair contradicts the other, from which the mutual exclusion property holds
Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Algorithm: Bakery (code executed by processor pi, 1 ≤ i ≤ n)
Initially Flag[i]=false and Number[i]=0 for 1 ≤ i ≤ n

<Entry>:
1: Flag[i] = true
2: Number[i] = max(Number[1],.., Number[n])+1
3: Flag[i] = false
4: for j=1 to n (j ≠ i) do
5: wait until Flag[j]=false
6: wait until Number[j]=0 or (Number[j],j) > (Number[i],i))
<Critical Section>
<Exit>:
7: Number[i] = 0
<Remainder>:

Th: The Bakery algorithm is deadlock free
Proof
By contradiction
- Suppose that processes never exit the doorway (L4-6).
  Let D be this set of procs
- Thus there is a time after which ∀j = 1,..n, we have Flag[j] = false
- Thus no process of D can be blocked forever at L5
- Let (Number[i],i) be the smallest pair of a proc in D
- Thus eventually when pi evaluates L6, this predicate is satisfied for all pj
- Thus pi enters the CS, which contradicts the assumption, and proves the Theorem
Bakery Algorithm

Mutual exclusion problem using safe read/write registers

Algorithm: Bakery (code executed by processor pi, 1 ≤ i ≤ n)
Initially \( Flag[i] \) = false and \( Number[i] \) = 0 for \( 1 \leq i \leq n \)

<Entry>:
1: \( Flag[i] \) = true
2: \( Number[i] \) = max(\( Number[1] \),.., \( Number[n] \)) + 1
3: \( Flag[i] \) = false
4: for \( j = 1 \) to \( n \) \( (j \neq i) \) do
5: \hspace{1cm} wait until \( Flag[j] \) = false
6: \hspace{1cm} wait until \( Number[j] \) = 0 or \( (Number[j],j) > (Number[i],i) \)

<Critical Section>

<Exit>:
7: \( Number[i] \) = 0
<Remainder>:

Th: The Bakery algorithm preserves FIFO order

- Suppose that both \( p_i \) and \( p_j \) are competing for the CS
- Suppose that \( p_j \) wins, execute the Remainder section, and again, enters the doorway (L2), and enters the bakery.
- \( p_i \) is still waiting to enter the CS
- Thus Lemma 1 applies: \( Number[i] < Number[j] \)
- Thus \( p_j \) cannot bypass \( p_i \)

Note that any algorithm that is both deadlock-free and first-come-first-served is starvation-free

Lemma 1: Let \( p_i \) and \( p_j \) be both in the bakery, and such that \( p_i \) entered the bakery before \( p_j \) entered the doorway.
Then \( Number[i] < Number[j] \)
• Mutual exclusion part comes from

• Constructions part comes from
  • On interprocess communication, Part I and II. Distributed computing. Vol 1, Number 2, pages 77 – 101.