Distributed algorithms: lesson on the shared memory model

Emmanuelle Anceaume
emmanuelle.anceaume@irisa.fr
Shared memory

• You have already seen different distributed algorithms in the message passing paradigm
• We now turn our attention to the other major communication paradigm for distributed systems: the shared memory paradigm
• In a shared memory system, processors communicate asynchronously via a common memory area
• This memory area contains a set of shared variables/objects
Shared memory (cont’d)

• Difference with message passing
  • processors cannot communicate directly with each other, but instead communicate through a pool of shared objects

• It is usually assumed that shared objects do not experience faults
Shared memory (cont’d)

• Several types of shared objects can be employed
  • Type specifies the operations that can be performed on it and the values that
can be returned by the operations

• An object type consists of
  • A set V of values
  • An initial value \( v_0 \) in V
  • A set of operations
  • A set of responses
  • A function \( f: \text{operations} \times V \rightarrow \text{responses} \times V \)
    • function \( f \) describes the response that is returned and the new value of the object
Shared memory (cont’d)

• The most common type is a read/write variable
• 2 Operations: read() and write()
  • f(read()) = (v)
  • f(write(w)) = (ack)

• We will see later in the course that there exist different types R/W variables: safe, regular and atomic ones
Shared objects (also called registers or variables) can be further characterized according to their access patterns, i.e., how many processors can access the variable.

We now formalize the notion processes, shared objects, history (execution).
Shared Memory System

• There are $n$ processors/cores/processes $p_1, \ldots, p_n$ running concurrently in the system
• Besides accessing local variables, processes may execute operations on shared objects
• These objects $X_1, X_2, \ldots$ allow processes to synchronize their computations
• Each process is modelled as a state machine
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
A configuration describes the distributed system at some point in time.

A configuration is a vector

\[ C = (q_1, q_2, ... , q_n, x_1, x_2, ... x_m), \]

where

- \( q_i \) is the state of processor \( p_i \), \( 1 \leq i \leq n \), and
- \( x_j \) is the value of object \( X_j \), \( 1 \leq j \leq m \)

In the initial configuration \( C_0 \), all the processors are in their initial state and the variables are initialised with their initial values.
Event

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

• At each computational step by some processor $p_i$ the following happens atomically:

  1. $p_i$ chooses the shared object to access with a specific operation based on $p_i$‘s current state
  2. The specified operation is performed on the shared object
  3. $p_i$‘s state changes according to $p_i$‘s current state and the value returned by the shared object operation
Execution

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

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

where configuration \( C_k \) is the result of applying event \( \Phi_k \) to \( C_{k-1} \).

• For each triple \( C_{k-1}, \Phi_k, C_k \):
  
  • Suppose that \( \Phi_k = i \) (computational step made by \( p_i \)) and \( p_i \)'s state in \( C_{k-1} \) indicates that \( x_j \) is the shared object to be accessed by \( p_i \).
  
  • Then \( C_k \) is the result of changing \( C_{k-1} \) in accordance with \( p_i \)'s state in \( C_{k-1} \) and the value \( x_j \) in \( C_{k-1} \).
  
  ➢ Thus the only change are to \( p_i \)'s state and the value of object \( x_j \).

\[
C_{k-1} = (q_1, \ldots, q_i, \ldots, q_n, x_1, \ldots, x_j, \ldots, x_m) \quad ----> \quad C_k = (q_1, \ldots, q_i, \ldots, q_n, x_1, \ldots, x_j, \ldots, x_m)
\]
Single-writer, multiple writers

• Typical assumptions
  • Single-writer multi-reader
    • Do not have to arbitrate which of two near-simultaneous writes gets in last
  • Multi-writer multi-reader
    • Usually built from single-writer multi-reader objects

• Less common assumptions
  • single-writer single-reader
    • Acts like message-passing channels
Fairness and crashes

• Fairness condition
  • Every processor gets to perform an operation infinitely often until it crashes

• Wait-free execution
  • Assume that up to n-1 processors can crash
  • No processor can be blocked because of processor crash

• Crash failures rather than Byzantine ones
  • In multi-writer situations, a Byzantine process can do much more damage than in message-passing
Sequential specification

- An object is defined by a **sequential specification** describing for each operation its effect when executed (alone) on the shared object
  - By effect we mean the response that the object returns and the new state of the object after the operation executed

- Note that we assume that each operation can be applied on each state of the variable (we say that the variable is complete). This may require some care: for instance if a dequeue() operation is invoked on an empty queue then a specific response *nil* must be returned
Concurrent execution

- Operations on shared objects are often implemented using lower level operations.
- To model the possibility of concurrency between operations, operations are split into:
  - an invocation and a response.
- The user is responsible for invoking the object.
- The object is responsible for responding.
- The time between the invocation and the response is called the interval of the operation.
Concurrent execution

- A concurrent execution is a sequence of invocations and responses, where after any prefix of the execution, every response corresponds to some invocation.
- There is at most one invocation for each processor (the last) that does not have a corresponding response.
  - Processors are sequential.
- Complete.
  - A concurrent execution is complete if every invocation has a matching response.
- Sequential.
  - A concurrent execution is sequential if the operations do not overlap.
  - i.e., there is at most one invocation without a corresponding response in any prefix of the execution.
History $S$ is a sequence of events in which $e_1$ is an invocation, $e_2$ is the matching response, ..., and $e_7$ is the matching event of $e_6$. Thus $S$ is a sequential history, and is complete.
H: history made of the operations executed on a FIFO queue

$H|_{p_1} = \{e_2, e_7, e_8, e_{13}\}$
$H|_{p_2} = \{e_1, e_3, e_4, e_5, e_6, e_9, e_{10}, e_{11}, e_{12}, e_{14}\}$

H is a complete history

$H'|_{p_1} = \{e_2, e_7, e_8\}$
$H'|_{p_2} = \{e_1, e_3, e_4, e_5, e_6, e_9, e_{10}, e_{11}, e_{12}\}$

H’ is an uncomplete history
Consistency properties

• Consistency properties describe how views of an object by different processes relate to each other

• Strongest consistency property: Linearizability
  • The implementation of an object is linearizable if, for any complete concurrent executions of the object, there is a sequential execution of the object with the same operations and returned values
  • We say that the (total) order of operations in the sequential execution is a linearization of the (partial) order of the operations in the concurrent execution

• For any two operations a and b, we say that a is ordered before b
  \[ a <_H b \]
  • If the response event of operation a in the history H precedes the invoke event for operation
Linearizability

• Finding the sequential execution that corresponds to the concurrent execution

• Linearization point
  • Assign each operation a linearization point between its invocation and its response
  • All operations execute atomically at their linearization point
  • We obtain a sequential execution (must be consistent with the specification of the objects)

• Linearizability has the useful property of being composable
  • If $H|A$ is linearizable for any object $A$, then $H$ is linearizable
Is this execution linearizable?
Is this execution linearizable?
Is this execution linearizable?

X.write(1), X.read() → 1,
Is this execution linearizable?

\[
\begin{align*}
\text{X.write}(1), \text{X.read()} & \rightarrow 1, \\
\text{X.write}(2), \text{X.write}(3), \\
\text{X.write}(1), \text{X.read()} & \rightarrow 3
\end{align*}
\]
Is this execution linearizable?

\[
\begin{align*}
X.\text{read()} & \rightarrow 1 \\
X.\text{write}(1) & \\
X.\text{write}(2) & \\
X.\text{write}(3) & \\
X.\text{read()} & \rightarrow 3 \\
X.\text{read()} & \rightarrow 3
\end{align*}
\]
Is this execution linearizable?

X.write(1), X.read() → 1, X.write(2), X.read() → 2, X.write(3), X.read() → 3
Is this execution linearizable?

$X$.write(1), $X$.read() $\rightarrow$ 1, $X$.write(2), $X$.read() $\rightarrow$ 2, $X$.write(3), $X$.read() $\rightarrow$ 3
Fancier registers

- Other types of shared objects exist, which are more powerful in the operations they perform
- They are usually called Read-Modify-Write (RMW) objects
  1. Returns the current value of the shared object $V$
  2. Computes a new value $V'$ as a function $h$ of the current value of $V$
  3. Write the new value to $V$

- Examples of RMW
Fancier registers

- **Swap(u)**: allows to atomically
  1. Returns the current value of the shared object V
  2. Sets the current value of V to u
- $f(Swap(u)) = (v)$
Fancier registers

- **Compare&Swap**(u,w): allows to atomically performs a conditional write.
  1. Returns the current value of the shared object V
  2. If the current value of V is equal to u then sets V to w
- f(Compare&Swap(u,w)) = (v)
Fancier registers

• Test&Set variable supports two operations
  • `test&set()`: allows to atomically
    1. Returns the current value of the shared object V
    2. Sets the current value of V to 1
  • \( f(\text{Test}&\text{Set}()) = (1) \)
  • Reset():
    1. Sets the current value of V to 0
    \( f(\text{Reset}()) = (\text{ack}) \)
Fancier registers

- **Fetch-and-Add(u):** allows to atomically
  1. Returns the current value of the shared object V
  2. Add u to the current value of V
- \( f(\text{Fetch-and-Add}(u)) = (v) \)
The mutual exclusion problem

• We now see a classic problem that classically arrives in distributed computing
• The mutual exclusion problem that allows to access a given resource in a mutually exclusive way
Mutual exclusion problem

A group of processors need to access some resource that cannot be used simultaneously by more than a single processor.

- 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 solves for systems without crash what consensus solves for systems with crash
Mutual exclusion problem

- One can partition the code of a processor as follows
  1. **Remainder**: the rest of the code
  2. **Entry**: the code executed in preparation for entering the critical section
  3. **Critical**: the code to be protected from concurrent execution
  4. **Exit**: the code executed on leaving the critical section

The problem is to design the **entry** and **exit** codes in a way that guarantee that the mutual exclusion and deadlock-prevention properties are satisfied.
Mutual exclusion problem

• We assume that 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 section (no crash)
An algorithm for a shared memory system solves the mutual exclusion problem with no deadlock 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 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 (lockout-freedom)**: In every 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.
Mutual exclusion problem

• There are several ways to design mutual exclusion algorithms.
• They depend on the operations and properties provided to the processors by the underlying shared memory communication system
• We distinguish 3 different families of mutual exclusion algorithms
1. **Strong primitives**
   - Hardware primitives are offered by the multiprocessors architectures (e.g. binary test&set variables).
   - Those primitives are more sophisticated (i.e. stronger) than simple r/w variables.

2. **Atomic read/write variables**
   - The only way processors communicate is through those variables.

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 variables than atomic ones.

Mutual exclusion problem
Solving the mutual exclusion problem with the Test & Set object

• We will see that with Test&Set objects, one bit is sufficient to guarantee mutual exclusion to a critical section with $n$ competing processes with no deadlock
Solving the mutual exclusion problem with the Test & Set object

- A Binary Test&Set object x is an object that supports 2 atomic operations
  1. test&set (TAS)
  2. reset

\[
\text{TAS}(x: \text{memory address}):\{
    \text{temp:=read}(x) \\
    \text{write}(x,1) \\
    \text{return (temp)}
\}
\]

\[
\text{reset}(x: \text{memory address}):\{
    \text{write}(x,0) \\
    \text{return}
\}
\]

- If two processes both try to perform a test-and-set on the same object, only one of them will see a return value of 0
Solving the mutual exclusion problem with the Test & Set object

- $n$ processors $p_1 \ldots p_n$ want to access to the critical section

**Algorithm:**

```plaintext
{ // Mutual_Exclusion using T&S object x – executed by pi, i=1 \ldots n
    Initially x = 0 // T&S Shared object
    while true do
        <Entry>:
            wait until (TAS(x) = 0)
        <Critical Section>:
            /* execute critical section */
        <Exit>:
            reset(x)
        <Remainder>:
            /* execute the rest of the code */
}
```
Solving the mutual exclusion problem with the Test & Set object

Algorithm: { // Mutual_Exclusion using T&S
Initially x = 0
while true do

<Entry>: wait until (TAS(x)= 0)
<Critical Section>: /* execute critical section */
<Exit>: reset(x)
<Remainder>: /* execute the rest of the code */
}

Assume that initially x=0

- In the entry section, \( p_i \) repeatedly tests \( x \) until it returns 0
- The first successful test by \( p_i \) assigns 1 to \( x \), which causes the next test to return 1, prohibiting any \( p_j, i \neq j \), from entering the CS
Solving the mutual exclusion problem with the Test & Set object

**Algorithm:**

```plaintext
Algorithm: { // Mutual_Exclusion using T&S
Initially x = 0
while true do
    <Entry>: wait until (TAS(x) = 0)
    <Critical Section>: /* execute critical section */
    <Exit>: reset(x)
    <Remainder>: /* execute the rest of the code */
}
```

**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
- Once $p_i$ has entered the CS, $x$ is set to 1
- $x$ remains equal to 1 until some processor leaves the CS
- By assumption of $t$, no processor other than $p_i$ is in the CS when $p_j$ enters the CS
- Thus $x$ is always equal to 1 up to the time $p_j$ enters the CS
- Thus when $p_j$ tests $x$ it should return 1 and not 0
- Thus $p_j$ cannot enter the CS
- A contradiction
Solving the mutual exclusion problem with the Test & Set object

Th: No Deadlock is guaranteed

Proof: by contradiction
- Suppose that there is an execution in which, after some time at least \( p_i \) is in the Entry section but no processor ever enters the CS
- Since no processor remains forever in the CS then there is some time from which at least \( p_i \) is in the Entry section but no processor is in the CS
- \( x = 0 \) iff no processor is in the CS
- Thus any processor \( p_i \) that executes the first line of the algorithm discovers that \( x = 0 \) and enters the CS
- A contradiction

---

**Algorithm:**

```plaintext
{ // Mutual_Exclusion using T&S
Initially x = 0
while true do
    <Entry>:
        wait until (TAS(x) = 0)
    <Critical Section>:
        /* execute critical section */
    <Exit>:
        reset(x)
    <Remainder>:
        /* execute the rest of the code */
}
```
Solving the mutual exclusion problem with the Test & Set object

Algorithm: 

// Mutual_Exclusion using T&S
Initially x = 0
while true do
  
  <Entry>:
  
  wait until (x.test&set() = 0)
  
  <Critical Section>:
  /* execute critical section */
  
  <Exit>:
  x.reset(V)
  
  <Remainder>:
  /* execute the rest of the code */

Th: This algorithm guarantees mutual exclusion without deadlock with one bit
Solving the mutual exclusion problem with the Test & Set object

**Algorithm:**

```plaintext
Algorithm: {    // Mutual_Exclusion using T&S
Initially x = 0
while true do
    <Entry>:
        wait until (x.test&set() = 0)
    <Critical Section>:
        /* execute critical section */
    <Exit>:
        x.reset(V)
    <Remainder>:
        /* execute the rest of the code */
}
```

**Th: This algorithm does not starvation freedom**

Nothing prevents a single fast process from accessing x every time it goes through the outer loop. Lockout-freedom requires a more sophisticated turn-taking strategy.
A starvation-free algorithm using an atomic queue (FIFO)

Shared (FIFO) queue

- `enq(Q, v)`: inserts `v` at the end of the queue
- `deq(Q)`: removes the first element of the head of the queue
- `head(Q)`: returns the first element of the queue
A starvation-free algorithm using a read-modify-write variable

Basic idea:

- In the trying phase, each process enqueues itself at the end of a shared queue (assumed to be an atomic operation).

- When a process comes to the head of the queue, it enters the critical section, and when exiting it dequeues itself.
A starvation-free algorithm using an atomic queue (FIFO)

Algorithm: { // Mutual_Exclusion
  while true do
    <Entry>:
      enq(Q, myId)
      while (head(Q) not equal to myId) do nothing
    <Critical Section>:
      ....
    <Exit>:
      deq(Q)
  <Remainder>:
    ....
}

Mutual exclusion:
Idea: only the process whose id is at the head of the queue can enter its critical section.
The following invariant holds:
« any process that is between the inner while loop and the call to deq(Q) must be at the head of the queue »

Proof:
- a process can’t leave the while loop unless the test fails (i.e., it is at the head of the queue),
- no enqueue operation changes the head value (if the queue is nonempty),
- the dequeue operation (which changes the head value) can only be executed by a process already at the head.
A starvation-free algorithm using an atomic queue (FIFO)

Algorithm: { // Mutual _Exclusion

while true do
    <Entry>:
enq(Q,myId)
while (head(Q) not equal to myId) do nothing
    <Critical Section>:
        ....
    <Exit>:
deq(Q)
    <Remainder>:
        ....
}

Deadlock freedom

Proof:

- any element of the queue is the id of some process in the trying, critical, or exiting sections
- so eventually the process at the head of the queue passes the inner loop, executes its critical section, and dequeues its i

Starvation freedom

Proof:

- once a process is at position k in the queue, every execution of the CS reduces its position by 1
- When it reaches the head of the queue it gets the CS itself
A starvation-free algorithm using

- Formally, a rmw operation is defined as follows:

\[
\text{rmw}(x: \text{memory address}, h: \text{function }):\
\begin{align*}
\text{temp} & := x \\
\text{x} & := h(\text{x}) \\
\text{return} & \text{ (temp)}
\end{align*}
\]

- The rmw operation takes a function h that specifies how the new value is related to the old one.
- The Test&Set object is a special case of a RMW objet, where h(x)=1.
We now present a mutual exclusion problem that uses only one shared RMW object $V$

**Idea:**
- We do not need to keep track of process position in the queue
- We can hand out tickets to each process to remember where its place in line is

$V$ has 2 fields: first and last, both initialized to 0
- Incrementing last simulates an enqueue
- Incrementing first simulates a dequeue

A process remembers the value of last when it ``enqueued”, and waits for the first field to equal this value
A lockout-free algorithm using a RMW shared object

**Algorithm:** {  
  // Mutual_Exclusion with V
  Initially V = <0,0>
  while true do
    <Entry>:
      position := RMW(V, <V.first, V.last+1>)
      repeat
        queue := RMW(V, <V.first, V.last>)
        until (queue.first = position.last)
    <Critical Section>:
      ....
    <Exit>:
      RMW(V, <V.first+1, V>)
    <Remainder>:
      ....
  }

The RMW has two fields: first and last (initially both fields are equal to 0).

Incrementing last simulates an enqueue – ticket
Incrementing first simulates a dequeue
A lockout-free algorithm using a RMW shared object

Algorithm: { // Mutual Exclusion with V

Initially V =<0,0>

while true do

<Entry>:

position := RMW(V, <V.first, V.last+1>)
repeat
queue:= RMW(V, <V.first, V.last>)
until (queue.first = position.last)

<Critical Section>:

....

<Exit>:

RMW(V,<V.first+1,V>)

<Remainder>:

....

}

Mutual exclusion is satisfied

Proof:
• Only the process at the head of the queue can enter the CS
• Thus all the other processes cannot enter the CS

Lockout-freedom is satisfied

Proof:
• The FIFO construction together with the fact that processes do not stay forever in the CS provide the lock-out freedom property
A lockout-free algorithm using a RMW shared variable

Algorithm: {  // Mutual_Exclusion with V
Initially V =<0,0>
while true  do
  <Entry>:
    position := RMW(V, <V.first, V.last+ 1>)
    repeat
      queue:= RMW(V, <V.first, V.last>)
    until (queue.first =position.last)
  <Critical Section>:
    ....
  <Exit>:
    RMW(V,<V.first+1,V>)
  <Remainder>:
    ....
}
Mutual exclusion based on Read/Write variables

• 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 variables
Read/Write variable

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

2 – Who is allowed to access a variable?
   SWSR (Single Writer / Single Reader)
   SWMR (Single Writer / Multiple Readers)
   MWMR (Multiple Writer / Multiple Readers)

3 – How behave the variable when concurrently accessed?
   safe – regular - atomic
Safe read/write variable (only single writer)

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 variable we can think of to enable any 2 processes to communicate
Safe read/write variable (only single writer)

\[ X = \text{[1,..6]-valued safe register} \]

\[ p_i \begin{align*} \text{begin} & \quad \text{end} \quad \text{X.read-\rightarrow?} \\ \text{p}_j & \quad \text{X.write(5)} \quad \text{X.write(6)} \quad \text{X.read-\rightarrow?} \quad \text{X.read-\rightarrow?} \end{align*} \]
Safe read/write variable (only single writer)

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

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

- Provides stronger properties than the safe one
Regular read/write variable

\[ 1, \ldots, 6 \]-valued regular register \( X \)
Regular read/write variable

\[ p_i \]
\[ p_j \]
\[ X.\text{write}(5) \]
\[ X.\text{write}(6) \]
\[ X.\text{read}\rightarrow5 \]
\[ X.\text{read}\rightarrow5/6 \]
\[ X.\text{read}\rightarrow5/6 \]

\[ [1,..6]-\text{valued regular register} \, X \]
Atomic read/write variable

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 variable

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 variable

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 invocation event and its result event, and
- (b) the resulting sequence is such that any read invocation returns the value written by the closest preceding write invocation.

\[
\text{X.write(5)} \quad \text{X.read->5} \quad \text{X.read->?} \quad \text{X.read->?}
\]

\[
\text{X.write(6)}
\]

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

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 variable

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 variable

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.
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. The $\ell()$ instants are the linearization points we have seen.

- This means that an atomic register is such that all its operations invocations appear as if they have been executed sequentially.
Atomic Read/Write variable

Properties of an atomic shared register
An atomic read-write 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
Atomic Read/Write variable

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 atomic objects 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 object
Mutual exclusion

- The algorithm ensures mutual exclusion without deadlock for two processors $p_0$ and $p_1$
- It is due to Peterson (1981)
- It uses 2 bits to communicate:
  - 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
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

If \( p_i \) executes this part of the code then \( \text{Want}[i]=1 \)

---

**Algorithm: Mutual exclusion for two processes \( p_0 \) and \( p_1 \)**

---

**Initially** \( \text{Want}[0]=0; \text{Want}[1]=0; \)

**Code executed by \( p_0 \)**

<Entry>:

1: \( \text{Want}[0] := 1 \)

2:

3: \( \text{Want}[0] := 1 \)

4:

5:

6: \( \text{wait until } \text{Want}[1]=0 \)

<Critical Section>

<Exit>:

7:

8: \( \text{Want}[0] := 0 \)

<Remainder>

**Code executed by \( p_1 \)**

<Entry>:

1: \( \text{Want}[1] := 0 \)

2: \( \text{wait until } (\text{Want}[0]=0) \)

3: \( \text{Want}[1] := 1 \)

4:

5: if \( (\text{Want}[0] = 1) \) then goto Line1

6:

<Critical Section>

<Exit>:

7:

8: \( \text{Want}[1] := 0 \)

<Remainder>
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 $p_0$'s last write to $Want[0]$, before entering the CS, follows $p_1$'s last write to $Want[1]$, before entering the CS
- We have $Want[0].write(1) < Want[1].read()$ (algo) and $Want[1].write(1) < Want[0].write(1)$ (assumption)
- thus $p_0$'s read of $Want[1]$ should return 1 ($Want$ registers are atomic, thus return the last written value).

A contradiction

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

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

Code executed by $p_0$

1. $Want[1]:=0$
2. wait until $(Want[0]=0)$
3. $Want[0]:=1$
4. 
5. if $(Want[0] = 1)$ then goto Line 1
6. wait until $Want[1]=0$

Code executed by $p_1$

1. $Entry$:
2. $Entry$:
3. $Want[1]:=0$
4. $Want[1]:=1$
5. 
6. $if (Want[0] = 1) then goto Line 1$

$Th$: 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 $p_0$'s last write to $Want[0]$, before entering the CS, follows $p_1$'s last write to $Want[1]$, before entering the CS
- We have $Want[0].write(1) < Want[1].read()$ (algo) and $Want[1].write(1) < Want[0].write(1)$ (assumption)
- thus $p_0$'s read of $Want[1]$ should return 1 ($Want$ registers are atomic, thus return the last written value).

A contradiction
“2-Mutual exclusion” using SWMR atomic variables

• The algorithm is deadlock-free
• 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 process gives priority to the other process on leaving the CS
“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic variables

- 3 bits to communicate
- Each processor $p_i$ uses 1 binary SWMR variable $\text{Want}[i]$
  - $\text{Want}[i] = 1$ if processor $p_i$ is interested in entering the critical section and
  - $\text{Want}[i] = 0$ otherwise
- and 1 MWMR variable $\text{Priority}$
  - $\text{Priority} = i$ if processor $p_i$ has priority at the moment.
    It is initialized to 0.

- The shared variable Priority is read and written by both processes
- The process that has priority plays the role of $p_0$ in the previous algorithm, so it will enter the critical section
- When exiting it will give priority to the other one, and will play the role of $p_1$ from the previous algorithm
Algorithm: 2-Mutual exclusion
Code executed by $p_i$
-------------------------------------------------------
Initially $\text{Want}[i]=0, \text{Want}[1-i]=0, \text{Priority}:0$

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

<Critical Section>:
   // code for the critical section

<Exit>:
7: $\text{Priority} =1-i$
8: $\text{Want}[i]=0$

<Remainder>
“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic variables

**Th:** The algorithm provides Mutual Exclusion

Proof: by contradiction
Assume both processes are in the CS
- By construction both $Want[i]=1, \; i=0, \; 1$
- Assume 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 if $p_0$ reads $Want[1]=0$.
- However we have
  - “$Want[0].write(1)$” $<$ “$Want[1].read()$” (algo)
  - “$Want[1].write(1)$” $<$ “$Want[0].write(1)$” (assumption)
- Thus $p_0$ reads $Want[1]$ that should return 1 (recall that variables are atomic)
A contradiction

---

Algorithm: 2-Mutual exclusion
Code executed by $p_i$

<table>
<thead>
<tr>
<th>Initially $Want[i]=0, ; Want[1-i]=0, ; Priority:=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Entry&gt;:</td>
</tr>
<tr>
<td>1: $Want[i]=0$</td>
</tr>
<tr>
<td>2: wait until ($Want[1-i]=0$ or $Priority = i$)</td>
</tr>
<tr>
<td>3: $Want[i]=1$</td>
</tr>
<tr>
<td>4: if ($Priority = 1-i$) then</td>
</tr>
<tr>
<td>5: if ($Want[1-i] = 1$) then goto Line 1</td>
</tr>
<tr>
<td>6: else wait until $Want[1-i]=0$</td>
</tr>
<tr>
<td>&lt;Critical Section&gt;</td>
</tr>
<tr>
<td>&lt;Exit&gt;:</td>
</tr>
<tr>
<td>7: $Priority = 1-i$</td>
</tr>
<tr>
<td>8: $Want[i]=0$</td>
</tr>
<tr>
<td>&lt;Remainder&gt;</td>
</tr>
</tbody>
</table>
“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic variables

Th: The algorithm is without deadlock

Proof: by contradiction
Suppose that there is a point after which some process 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 process is forever in the <Entry> section. Say $p_0$.

- Since $p_2$ does not stay forever in the CS or in the <Exit> section, it will set Priority = 0 and $Want[1]=0$
- Thus $p_0$ does not loop forever in the <Entry> section (lines 2,5,6)
- and enters the CS.
  A contradiction.
Th: The algorithm is starvation-free

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

Suppose that $p_1$ never executes Line 7 at some point later.

- Since there is no deadlock, $p_1$ must be forever in the remainder section
- Thus $Want[1]$ equals 0 forever
- Thus $p_0$ can never be stuck at Lines 2, 5 or 6 and enters the CS
  A contradiction

Suppose that $p_1$ executes line 7 at some later point.

- $Priority = 0$ after
- Thus $p_0$ passes the test Line 2 and skips Line 4
- Thus $p_0$ must be stuck in Line 6, waiting for $Want[1]$ to be 0, which never occurs
- Thus $p_1$ is always executing between Lines 3 and 8.
- But since $p_1$ does stay forever in the CS, this would mean that it is stuck in the entry section, which violates the no deadlock A contradiction.

“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic variables

Algorithm: 2-Mutual exclusion
Code executed by $p_i$

Initially $Want[i]=0$, $Want[1-i]=0$, $Priority:=0$

<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>
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 variables, that is variables 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 variables

• 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
• Algorithm that guarantees first-in-first-served property
• Safe SWMR 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” (which does not count as the smallest ticket)
Bakery Algorithm
Mutual exclusion problem using safe read/write registers

In practice

• Shared safe 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
  • after getting its ticket \(p_i\) waits until its ticket is the smallest one, and then enters the CS
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 variables 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
3. Note that several processes can be in the doorway at the same time. So to break ties, their tickets are set to (Number,index):
   p_i's ticket is the pair (Number[i],i)
   This makes each ticket unique
   The ordering between tickets is defined by the lexicographic order
Bakery Algorithm
Mutual exclusion problem using safe read/write variables

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>

- After choosing its ticket, p_i waits until its ticket is minimal:
- For each other p_j, p_i waits until p_j is not in the middle of choosing its ticket and then compares their tickets. If p_j’s ticket is smaller then p_i waits until p_j executes its critical section and leaves it.
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
Bakery Algorithm

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 (L3-7). Then
(Number[i],i) < (Number[j],j)

Proof
- As p_i is in the CS it read Flag[j] = false (L5)
- According to the time of that read and the time at
which p_j wrote Flag[j] (L1 or L3) we have two cases
  - Either t1 < t2
  - Or t3 < t4

Case 1: t1 < t2
Case 2: t3 < t4
Bakery Algorithm

Case 1: $t_1 < t_2$

- $p_i$ enters the bakery at time $t_1$.
- $p_j$ enters the bakery at time $t_2$.

1. $Flag[j]$.write(true) (Line 1)
2. $Flag[j]$.read() -> false (Line 5)
Bakery Algorithm

Case 1: $t_3 < t_4$

Flag[j].read() -> false (Line 5)

Flag[j].write(false) (Line 3)
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: $\text{Number}[i] = \max(\text{Number}[1], \ldots, \text{Number}[n])+1$
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
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: \( Number[i] = \text{max}(Number[1], ..., Number[n]) + 1 \)
3: \( Flag[i] = false \)
4: for \( j = 1 \) to \( n \) (\( j \neq i \)) do
5: \( \text{wait until} \ Flag[j] = false \)
6: \( \text{wait until} \ Number[j] = 0 \) 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 \( t_3 < t_4 \)
  - i.e., \( t_{j,s}(3) < t_{i,e}(5) \) (H1)
  - \( p_j \) is sequential thus \( p_j \) ended L2 < \( t_3 \)
    - i.e \( t_{e}(2) < t_{j,s}(3) \) (P1)
  - \( p_i \) is sequential thus
    - \( t_{i,s}(5) < t_{j,s}(6) \) (P2)
  - \( (P1) + (H1) + (P2) : t_{j,s}(2) < t_{j,s}(3) < t_{i,e}(5) < t_{j,s}(6) \)
  - \( t_{j}(2) < t_{j,s}(3) < t_{i,s}(5) < t_{j,s}(6) \)
  - \( t_{j}(2) < t_{j,s}(6) \) (P3)

- Thus the read by \( p_i \) of \( Number[j] \) in Line 6 occurred after \( Number[j] \) was written by \( p_j \) on Line 2
- As \( p_i \) is in the CS it exited the 2\(^{nd} \) wait statement, i.e., \( Number[j] = 0 \) or \( (Number[j],j) > (Number[i],i) \)
- By \( t_{j}(2) < t_{j,s}(6) \) (P3) we have \( Number[j] \neq 0 \)
- Thus \( (Number[j],j) > (Number[i],i) \)

- This ends this case and the proof of the Lemma
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: \( \text{Number}[i] = \text{max}(\text{Number}[1], \ldots, \text{Number}[n]) + 1 \)
3: \( \text{Flag}[i] = \text{false} \)
4: for \( j = 1 \) to \( n \) (\( j \neq i \)) do
5: \quad \text{wait until } \text{Flag}[j] = \text{false} \)
6: \quad \text{wait until } \text{Number}[j] = 0 \text{ or } (\text{Number}[j], j) > (\text{Number}[i], i))

<Critical Section>

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

<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 \((\text{Number}[i], i) < (\text{Number}[j], j)\)
- Similarly, as \( p_j \) is in the CS and \( p_i \) is in the bakery, by Lemma 2, we have \((\text{Number}[j], j) < (\text{Number}[i], i)\)
- As \( j \neq i \) both pairs are totally ordered. It follows that each pair contradicts the other, from which the mutual exclusion property holds.
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 starvation-free
Proof
By contradiction
- Let (Number[i],i) be the smallest pair of p_i that is starved before the CS
- All the processes that will enter the entry section after p_i will get a ticket greater than p_i’s one
- Thus they will not enter the CS before p_i
- All the processes that have a smaller ticket than p_i will enter the CS (by assumption they are not starved)
- They will exit it (since no process stays in the CS forever)
- So p_i will pass the for loop and enter the CS, a contradiction
Bibliography

• Specification part comes from

• Mutual exclusion part comes from