Objects
Linearizability
Basic abstraction

- Process: an entity performing independent computation.

- Algorithm for $n$ processes $\{A_1, \ldots, A_n\}$ where $A_i$ is an automaton (states, inputs, outputs) and a sequential specification. May be deterministic or randomized.

- A process that never fails takes infinitely many steps
• Processes communicate by applying operations on and receiving response from shared objects

• A shared objects is defined by:
  • states
  • operations
  • sequential specification
Queue

Queue operations:
- `enq(x)`
- `enq(y)`
- `deq()`
- `ok`
Queue

- operations: \texttt{enq(item x)} ; item \texttt{deq()}
  
- states: sequence of items

- sequential specification:
  
\[
\begin{align*}
\{ \text{state} = f \} & \text{enq}(x) \{ \text{state} = f.x \} \\
\text{if} (\text{state} \neq \emptyset) & \{ \text{state} = a.f \} \text{deq()} \{ \text{state} = f \} \\
\end{align*}
\]

- invocation

- response

return \texttt{ok}

return \texttt{a}
Register

- Operations: read, write
- state: item
- sequential specification:
  \[
  \begin{align*}
  \{ \text{state} = y \} \text{write}(x) \{ \text{state} = x \} & \quad \text{return ok} \\
  \{ \text{state} = y \} \text{read()} \{ \text{state} = y \} & \quad \text{return y}
  \end{align*}
  \]
Implementing an object

base objects:
  e.g. registers, TAS()
Implementation of object O

• An operation on O is implemented using a sequence of accesses to base objects

• Correctness ?
Histories

- A history is a sequence of invocation and response

\[\text{enq}_1(1) \text{enq}_2(2) \text{ok}_1 \text{ok}_2 \text{enq}_1(0) \text{ok}_1 \text{deq}_2() 2_2\]
Histories

• A history is a sequence of invocation and response

\[ enq_1(1) \text{enq}_2(2)ok_1ok_2\text{enq}_1(0)\text{deq}_2()2_2 \]
• A history is **sequential** if every invocation is immediately followed by a corresponding response

• A sequential history has no concurrent operations

• A sequential history is **legal** if it satisfies the sequential specification of the shared object
• A operation op is **complete** in a history $H$ if $H$ contains both the invocation and the response of op.

• A completion of $H$ is a history $H'$ that includes all complete operations of $H$ and a subset of incomplete operation with their matching responses.
Histories

\[ H = enq_1(1) enq_2(2) ok_1 ok_2 enq_1(0) deq_2() 2_2 \]
\[ H' = enq_1(1) enq_2(2) ok_1 ok_2 enq_1(0) deq_2() 2_2 ok_1 \]
\[ H' = enq_1(1) enq_2(2) ok_1 ok_2 deq_2() 2_2 \]
\[ H' = enq_1(1) enq_2(2) ok_1 ok_2 enq_1(0) ok_1 deq_2() 2_2 \]
Histories H and H’ are equivalent if for all \( P_i \) H restricted to \( P_i \) (\( H|p_i \)) is equal to H’ restricted to \( P_i \).

- e.g

\[
\text{enq}_1(1) \text{enq}_2(2) \text{ok}_1 \text{ok}_2 \text{enq}_1(0) \text{deq}_2() 2_2 \\
\text{enq}_2(2) \text{ok}_2 \text{enq}_1(1) \text{ok}_1 \text{deq}_2() 2_2 \text{enq}_1(0)
\]
Linearizability (Atomicity)

- A history H is **linearizable** if there exists a **sequential legal** history S such that:
  - S preserves the precedence relation of H
    - if op1 precede op2 in H (i.e. the response of op1 is before the invocation of op2 in H) then op1 precede op2 in S
  - S is equivalent to some completion of H
Histories

\[ H = \text{enq}_1(1)\text{enq}_2(2)\text{ok}_1\text{ok}_2\text{enq}_1(0)\text{deq}_2()2_2 \]

\[ S = \text{enq}_2(2)\text{ok}_2\text{enq}_1(1)\text{ok}_1\text{deq}_2()2_2\text{enq}_1(0)\text{ok}_1 \]

S sequential, legal, equivalent to some completion of H
preserve precedence relation
Histories

\[ H = \text{enq}_1(1) \text{ok}_1 \text{deq}_1(\)\text{enq}_2(2) \text{ok}_2 \text{deq}_2(1)2 \\]
Alternative definition

- Linearization points
- Each method call should appear to take effect instantaneously at some moment between the invocation and response
Sequential consistency

• Weaker than linearizability

• A history $H$ is **sequential consistent** if there exists a sequential legal history $S$ such that:
  
  per process order

  • $S$ preserves the precedence relation of $H$

  • if $P_i$ executes op1 before op2 in $H$ then $P_i$ executes op1 before op2 in $S$

  • $S$ is equivalent to some completion of $H$
Histories

\[ H = enq_1(1)ok_1deq_1()enq_2(2)ok_22deq_2()1_2 \]
\[ S = enq_2(2)ok_2enq_1(1)ok_1deq_2()1_2deq_1()2_1 \]
S sequential, legal, equivalent to some completion of H
preserve per process order
Composition

• Theorem: H is linearizable if and only if, for each object x, Hlx is linearizable

• no true for sequential consistency
• no true for sequential consistency
How get linearizability ?

- execute all operations in mutual exclusion
  
  - linearizability !

- if all processes are correct : at least one process can complete its operation

- if all processes are correct and the mutual exclusion is fair : every process may complete their operation

- But what happen in case of failures/asynchrony??
Liveness properties

- **Non blocking (lock free):** at least one correct process makes progress
- **Wait free:** every correct process makes progress
- **Obstruction free:** every correct process makes progress if it executes alone
Test and Set Object

- one operation: TAS()
- state: binary value x (init 1)
- sequential specification
- TAS() :: if (x==0) then return 0 else x:=0 return 1
• Is it possible to implement **wait free** TAS from atomic registers?

**NO**
Code of Pi: $a_i = \text{tas}()$

- $a_0 = 0$
- $a_1 = 1$
- $a_0 = 0$
- $a_1 = 1$

Diagram:

- Node 0: $a_0 = 0$
- Node 1: $a_1 = 1$
- Node 2: $a_0 = 1$
- Node 3: $a_1 = 0$

---

27
Code of Pi : $a_i = \text{tas}(\cdot)$

- $a_0 = 0$
- $a_1 = 1$
- $a_0 = 1$
- $a_1 = 0$

Diagram:

- Node 1
- Node 2
- Node 3
- Node 4
Code of Pi : ai=tas()
Code of Pi : $a_i = \text{tas}()$

- $a_0 = 0$
- $a_1 = 1$
- $a_0 = 1$
- $a_1 = 0$
- $a_0 = 0$
- $a_1 = 1$
- $a_0 = 0$
- $a_1 = 1$
- $a_1 = 0$
- $a_0 = 1$
Code of Pi: ai = tas()
\begin{align*}
\text{P0} : & \text{R.read()} \quad \text{P1: R’\text{.read()}} \\
\text{P0} : & \text{R.write(x)} \quad \text{P1: R.write(y)} \\
\text{P0} : & \text{R.write(x)} \quad \text{P1: R’\text{.write(y)}} \\
\text{P0} : & \text{R.read()} \quad \text{P1: R.write(y)} \\
\text{P0} : & \text{R.read()} \quad \text{P1: R’\text{.write(y)}}
\end{align*}
a0=0  a1=0  a0=1  a1=1

P0 : R.read()  // P1: R’.read()
P0 : R.write(x) // P1: R.write(y)
P0 : R.write(x) // P1: R’.write(y)
P0 : R.read()  // P1: R’.write(y)
P0 : R.read()  // P1: R’.write(y)

contradiction
\[a_0 = 0\quad a_1 = 1\] 
\[a_0 = 1\quad a_1 = 0\]

\[P_0 : R.\text{read}() \quad \text{//} \quad P_1 : R'.\text{read}()\]
\[P_0 : R.\text{write}(x) \quad \text{//} \quad P_1 : R'.\text{write}(y)\]
\[P_0 : R.\text{write}(x) \quad \text{//} \quad P_1 : R'.\text{write}(y)\]
\[P_0 : R.\text{read}() \quad \text{//} \quad P_1 : R'.\text{write}(y)\]
\[P_0 : R.\text{read}() \quad \text{//} \quad P_1 : R'.\text{write}(y)\]

\[\text{contradiction}\]
(m,n) assignment objects

- dual of atomic snapshot object
- state: an array with n items
- two operations: \( \text{assign}(v_1, \ldots, v_m, i_1, \ldots, i_m), \text{read}(i) \)
- sequential specification: \( \text{assign}(v_1, \ldots, v_m, i_1, \ldots, i_m) \)
  write \( v_j \) in position \( i_j \)
(m,n) assignment objects

- It is possible to wait free implement TAS from (m,n) assignment n>m>1?
Implementing T&S object from (2,3) assignment object (two processes)

shared object ass (2,3) assignment object init -1

code of P0

TAS(){
  ass(0,0,0,1)
  if read(2)=-1
  then //I am the first
   return 1
  else m:=read(1)
   if m ==1
   then //I am the first
    return 1
   else //I am the second
    return 0
}

37
code of P1

TAS(){
  ass(1,1,1,2)
  if read(0)=-1
  then //I am the first
   return 1
  else m:=read(1)
   if m ==0
   then //I am the first
    return 1
   else //I am the second
    return 0
}
• safety: linearization point of TAS() == linearization point of assign()

• liveness: wait free
• It is possible to wait free implement snapshot from atomic registers

• It is possible to implement TAS from assignment

• **but** it is possible to wait free implement TAS register

• Then it is impossible to wait free implement (2,3) assignment from atomic registers

**some objects are stronger than others??**
Consensus and shared objects
Objects

- Shared Objects:
  - sequential specification + linearizability (ex: Register, Compare&swap, Stack …)
  - Wait-free implementation (every method call completes in finite number of steps)
  - How powerful are objects (concurrent data structures), like Test&Set, Compare&Swap, Queue, Stack …?
  - Use the consensus to compare objects
Consensus
Consensus:

As (one-shot) object: method $\text{propose}(v)$ returns the decided value:

0

propose(0) returns 0

⊥

propose(1) returns 1

1

propose(x) returns 1
**n-Consensus**

*n-consensus* (= consensus between n processes) to get a hierarchy of (wait-free) objects:

There is no wait-free implementation of 2-consensus with atomic registers (or no implementation of 2-consensus object)

- But with Queue and registers is 2-consensus solvable?

*consensus number*: among how many processes an object (and registers) may solve Consensus?
Consensus number

- A class $C$ of objects solves **$n$-consensus** if there exists a consensus protocol using any number of objects of class $C$ and atomic registers for $n$ processes.

- The **consensus number** of a class $C$: $(h(C))$ is the largest $n$ for which that class solves $n$-consensus.

- $h(C)$ defines a hierarchy of objects:
  - If one can implement an object of class $C$ from objects of class $D$ (and registers) then $h(C) \leq h(D)$
  - (if $h(C) > h(D)$ there is no implementation of objects of $C$ with objects of $D$).
Compare objects

• Theorem
  There is no wait-free implementation of $n$-consensus from read-write registers for $n \geq 2$

By contradiction, assume there is such for $n=2$
we are going to model the runs of an algorithm
assume only 2 processes (A and B)
assume binary consensus (initial value 0 or 1 only)
  • « step » of a process (here read or write of some register)
  • « configuration » (state of memory, state of process...)
  • « run » of the algorithm: sequences of configurations + steps
  • schedule: sequence of process id (wait-free: at each time any process may take a step)
  • an initial configuration and a schedule define a run
moves: atomic step

state

state after B moves

A moves

B moves

Initial state

Final states

C. Delporte
bivalent: from here 0 and 1 are possible
0-valent: from here only 0
univalent: from here only one value
1-valent: from here only 1
Model

- Wait-free computation is a tree (a forest) (node: configurations link: steps) representing all possible runs from some initial configuration (from all initial configurations)
- (wait free means that in each configuration (exactly) one step of any process is possible)
- Configurations Valences:
  - in \textit{Bivalent} configurations:
    - Outcome not fixed
  - in \textit{Univalent} states
    - Outcome is fixed
    - (but may not be “known” yet)
  - \textit{1-Valent} and \textit{0-Valent} states = univalent for 1 and 0
critical state:

Properties:
- Some initial state is bivalent
- If there is a consensus protocol then there is a critical state

in a critical state
the next step of any process
gives a configuration in which
the decision is fixed
There is an initial bivalent state?

S: A initial value 0, B initial value 1
A solo decides (wait-free), A must decide 0 (the run is indistinguishable from run A alone for A and B initial have both value 0 as initial value)
Same argument B solo decides 1
Then S is bivalent.
there is a critical state?

- Starting from a bivalent initial state
- The protocol can reach a critical state
  - Otherwise we could stay bivalent forever
  - And the protocol is not wait-free
Then: for every consensus protocol there exists a critical step
• We are now ready to prove that it is impossible to achieve 2-consensus with registers (h(register)=1))
States look the same to A and B and

\[ 0 \]

A runs solo, eventually decides 0

\[ 1 \]

A runs solo, eventually decides 1

Contradiction

A writes y

B writes x

Contradiction

\[ 0 \]

B writes x

A writes y

Contradiction

States look the same to A and B and

read

write on different registers
Write on the same register

Theorem: \( h(\text{Register}) = 1 \)

no way to get consensus with register!
Fifo -> 2 consensus

• Fifo: enq(v) and deq()

• Algo for 2 processes:
  Queue q: initialized to Red, Green
  process i :
  writes its value \( v_i \) in proposed[i]
  \( V := \text{deq()} \) if \( V = \text{Red} \) then decide \( v_i \)
  else decide proposed[1-i]
if you like Java…

```java
public interface Consensus {
    Object decide(object value);
}

abstract class ConsensusProtocol<T> implements Consensus {
    protected T[] proposed = new T[N];
    protected void propose(T value) {
        proposed[ThreadID.get()] = value;
    }
    abstract public T decide(T value);
}

public class QueueConsensus extends ConsensusProtocol {
    private Queue queue;
    public QueueConsensus() {
        queue = new Queue();
        queue.enq(Ball.RED);
        queue.enq(Ball.GREEN);
    }
    public decide(object value) {
        propose(value);
        Ball ball = this.queue.deq();
        if (ball == Ball.RED) return proposed[i];
        else return proposed[1-i];
    }
}
```

⭐️ only one process gets RED it is the winner
⭐️ the winner decides its value
⭐️ the loser can find the winner’s value in the array

theorem: h(Queue) ≥ 2

there is no wait-free implementation of queue with registers
enq(a) and deq()? deq() and deq():

enq() and enq():

theorem: h(queue)=2
Read-Modify-Write RMW

F a set of functions of integer to integer
Variable interne \( x \)
method for a RMW object: \( \text{method}(v) \)
  • returns the previous value \( x \)
  • replace \( x \) with \( f_v(x) \) for some \( f_v \) in \( F \)
RMW

Examples:

- `getAndSet(v)` (or swap) \( f_v(x) = v \)
- `getAndIncrement()` \( f(x) = x + 1 \)
- `getAndAdd(k)` \( f_k(x) = x + k \)
- `compareAndSet(e,u)` if the value of the register is equal to \( e \) then sets the value to \( u \) and returns true else doesn’t change and returns false
- `get()` identity

RMW is *non trivial* if the set of functions includes at least one function different from identity.
Theorem

Any non-trivial RMW object has consensus number at least 2

(Hence No wait-free implementation of RMW registers from atomic registers)
Proof:

there is a value v and a function f in F such that f(v) ≠ v

```java
public class RMWConsensus 
    extends ConsensusProtocol {
private RMWRegister r = v;
public Object decide(object value) {
    proposed[i]=value;
    if (r.f() == v)
        return proposed[i];
    else
        return proposed[j];
}
RMW

- Let $F$ be a set of functions such that for all $f_i$ and $f_j$, either
  - Commute: $f_i(f_j(v)) = f_j(f_i(v))$
  - Overwrite: $f_i(f_j(v)) = f_i(v)$

- Theorem: Any set of RMW objects that commutes or overwrites has consensus number exactly 2
RMW

Critical state

\[ f_A \]

\[ f_B \]

0-valent

1-valent

C. Delporte
Commuting

\[ \text{indistinguishable for } C \]
Overwrite

A applies $f_A$

B applies $f_B$

C runs solo

C runs solo

contradiction

0-valent

1-valent

C. Delporte
public abstract class RMWRegister {
    private int value;
    public boolean synchronized compareAndSet(int expected, int update) {
        int prior = this.value;
        if (this.value == expected) {
            this.value = update;
            return true;
        }
        return false;
    }
    ...
}
public class RMWConsensus extend ConsensusProtocol {
private AtomicInteger r = new AtomicInteger(-1);
public Object decide(Object value) {
    proposed[i]=value;
    r.compareAndSet(-1,i);
    return proposed[r.get()];
}
}