Objects
Linearizability
Basic abstraction

• Process: an entity performing independant 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 processes 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

- operations: enq(item x); item deq()
- states: sequence of items
- sequential specification:

\[
\begin{cases}
  \text{state} = f \\
  \text{enq}(x) \{ \text{state} = f.x \} \\
  \text{if} (\text{state} \neq \emptyset) \{ \text{state} = a.f \} \\
  \text{deq}() \{ \text{state} = f \}
\end{cases}
\]
Queue

\[\text{enq}(x)\]
\[\text{enq}(y)\]
\[\text{deq}()\]
\[x\]
\[\text{ok}\]
\[\text{ok}\]
\[\text{enq}(x)\]
\[\text{enq}(y)\]
Register

- Operations: read, write
- state: item
- sequential specification:

\[
\{ state = y \} \text{write}(x) \{ state = x \} \quad \text{return } \text{ok}
\]
\[
\{ state = y \} \text{read()} \{ state = y \} \quad \text{return } y
\]
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

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

1. `enq(1)` ok
2. `enq(2)` ok
3. `enq(0)` ok
4. `deq()` 2

---

P1

- `enq(1)`
- ok

P2

- `enq(2)`
- ok
- `deq()` 2
Histories

- A history is a sequence of invocation and response

\[ enq_1(1) \quad enq_2(2) \quad ok_1 \quad ok_2 \quad enq_1(0) \quad deq_2()2 \]

\[ enq(1) \quad ok \quad \text{enq(0)} \]

\[ P1 \quad P2 \]

\[ \text{enq}(2) \quad \text{deq}() \quad 2 \]
• 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 include all complete operations of H and a subset of incomplete operation with their matching responses
Histories

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

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

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

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

\[ enq_1(1)enq_2(2)ok_1ok_2enq_1(0)deq_2()2_2 \]

\[ enq_2(2)ok_2enq_1(1)ok_1deq_2()2_2enq_1(0) \]
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 $\text{op1}$ precedes $\text{op2}$ in $H$ (i.e., the response of $\text{op1}$ is before the invocation of $\text{op2}$ in $H$) then $\text{op1}$ precedes $\text{op2}$ 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

\[ H = enq_1(1)ok_1 deq_1()enq_2(2)ok_2 2_{1} deq_2() 1_{2} \]
Alternative definition

- Linearization points
- Each method call should appear at take affect 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

\[ H = enq_1(1)ok_1deq_1()enq_2(2)ok_2deq_2()1_2 \]

\[ S = enq_2(2)ok_2enq_1(1)ok_1deq_2()1_2deq_1()2_1 \]

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 $x$, $H[x]$ 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
Atomic snapshot

• Goal: reading multiple locations atomically
Two operations:

- update(v) return ok
- snapshot() return an array of n items

Sequential specification:

- state: an array A of n items
- update(v) by process pi writes v in A[i]
- snapshot() return an array X where X[i] is the last value written by Pi
One update per process

- Property: all the snapshots are ordered

\[ S_i \subseteq S_j \text{ or } S_j \subseteq S_i \]
first attempt

initially **shared** R array of atomic register init \([v,\ldots v]\)

**local to Pi** array C of n items init \([v,\ldots v]\)

Code of Pi
update(w){
    R.[i].write(w)
    return (ok)
}
snapshot(){
    for ( int j=0; j<n; j++) C[i]:=R[i].read()
    return C
}
initially **shared** R array of atomic register init [v,…v]

**local to Pi** array C of n items init [v,…v]

Code of Pi
update(w){
    R.[i].write(w)
    return (ok)
}
collect(){
    for ( int j=0; j<n; j++) C[i]:=R[i].read()
    return C
}
• $C_1=\text{collect}(); C_2= \text{collect} ()$

• if $C_1=C_2$ then $C_2$ is a snapshot!
collect C1 collect C2
initially shared R array of atomic register init [v, ... v]
local to Pi array C1, C2 of n items init [v, ... v]

Code of Pi
update(w){
    R[i].write(w)
    return ok
}
collect(){
    for (int j=0; j<n; j++) C[i] := R[i].read()
    return C
}
snapshot(){
    C1 := collect()
    forever
        C2 := collect();
        if (C1 == C2) return C1 else C1 := C2
}
Safety: linearization points

update

snapshot

collect

C1==C2

C1

C1 collect C2
Liveness

- If update() concurrent to collect, snapshot never terminate
- Obstruction free? YES
- Non blocking? YES
- Wait free? NO
initially **shared** $R$ array of atomic register init $[((v,..,v),v),(v,..,v),v]]$

**local to Pi** array $C1,C2$ of $n$ items init $[((v,..,v),v),…((v,..,v),v)]$

array moved of boolean init [false,..false]

array snap of items

Code of Pi

update(w){
    snap=snapshot()
    $R[i].write(snap, w)$
    return (ok)
}

snapshot(){
    $C1:=collect()$
    **for** forever
    $C2:=collect();$
    **for** ( int $j=0; j<n; j++}$
    {if ( $C1[j].val != C2[j].val$ )
        if ( $moved[j]$ ) then return $C2[j].snap$
        else $moved[j]:=$true
        $C1:=C2$
        break // go to forever
    }
    // $C1.val==C2.val$
    snap= extract_val(C2) return snap
}

collect(){
    **for** ( int $j=0; j<n; j++}$
    $C[i]:=$R[i].read()
    return $C$
}

liveness

- snapshot: if not already terminate what happens after \( n+1 \) double collect?

- either terminate by clean double collect (the two collect return the same value)

- or each time for some \( i \) \( C1[i].val \neq C2[i].val \)
  - but after \( n+1 \) double collect at least one cell moved twice
  - terminate by moved\([i]\) and \( C1[i].val \neq C2[i].val \)
• update: if snapshot terminates then update terminates

Wait free
Safety

• If a scanning thread makes a clean double collect then same linearization point as before

• If a scanning thread Pi observes a change in another thread Pj during two different double collect then the value of R[j] read during the last collect was written by an update () call that began after the first of the collects started. The linearization point is the linearization point of the scan embedded in this update.
One move

snapshot  write R[i]

update

snapshot  read R[i]
snapshot  read R[i]
collect  C1  collect  C2
two moves

snapshot

update

write

snapshot

read
Counter

- State int ; init 0
- Operations incr(), int val()

\[
\{\text{state} = a\}\text{incr()}\{\text{state} = a + 1\}\text{return ok}
\]
\[
\{\text{state} = a\}\text{val()}\{\}\text{return a}
\]
(m,n) assignment objects

- dual of atomic snapshot object
- state: an array with n items
- two operations: \( assign(v_1, \ldots v_m, i_1, \ldots, i_m), read(i) \)
- sequential specification: \( assign(v_1, \ldots v_m, i_1, \ldots, i_m) \)

write \( v_j \) in position \( i_j \)
Test and Set Object

- one operation: TAS()

- state: binary value x (init 0)

- sequential specification

- TAS(x) :: if (x==1) then return 1 else x:=1 return 0
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 0
    else m:=read(1)
        if m ==1
            then //I am the first
                return 0
        else //I am the second
            return 1

code of P1

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

• liveness: wait free
• Is it possible to implement wait free (2,3) assignment from atomic registers?

• Is it possible to implement \textbf{wait free} T&S from atomic registers?

\textbf{NO}
Code of Pi : ai=tas()
Code of Pi : ai=tas()
Code of Pi : ai=tas()
Code of Pi: \( a_i = \text{tas()} \)

- \( a_0 = 0 \)
- \( a_1 = 1 \)
- \( a_1 = 0 \)
- \( a_0 = 1 \)
- \( a_1 = 0 \)
- \( a_0 = 1 \)
- \( a_0 = 0 \)
- \( a_1 = 1 \)

\( \text{tas()} \): Function for the code of Pi.
Code of Pi: $ai = tas()$

a0 = 0
a1 = 1

53
a0 = 1
a1 = 0
\[
\begin{align*}
\text{P0 : R.read()} & \quad \text{P1: R’.read()} \\
\text{P0 : R.write(x)} & \quad \text{P1: R.write(y)} \\
\text{P0 : R.write(x)} & \quad \text{P1: R’.write(y)} \\
\text{P0 : R.read()} & \quad \text{P1: R.write(y)} \\
\text{P0 : R.read()} & \quad \text{P1: R’.write(y)}
\end{align*}
\]
\[
\begin{align*}
P_0 &: R.\text{read()} \quad // \quad P_1: R'.\text{read()} \\
P_0 &: R.\text{write}(x) \quad // \quad P_1: R.\text{write}(y) \\
P_0 &: R.\text{write}(x) \quad // \quad P_1: R'.\text{write}(y) \\
P_0 &: R.\text{read()} \quad // \quad P_1: R.\text{write}(y) \\
P_0 &: R.\text{read()} \quad // \quad P_1: R'.\text{write}(y)
\end{align*}
\]

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

• It is impossible to wait free implement (2,3) assignment, T&C from atomic registers

• **but** it is possible to wait free implement T&S from (2,3) assignment

  
  some objects are stronger than others???