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

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

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

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

invocation
response

return ok
return a
Queue

enq(x)

enq(y)

deq()

x

deq()

ok

enq(x)

ok

enq(y)
Register

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

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

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

\[ enq_1(1) \, enq_2(2) \, ok_1 \, ok_2 \, enq_1(0) \, ok_1 \, deq_2() \, 2_2 \]
Histories

- A history is a sequence of invocation and response

\[ enq_1(1)enq_2(2)ok_1ok_2enq_1(0)deq_2()2 \]
• 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 include all complete operations of H and a subset of incomplete operation with their matching responses.
Histories

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

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

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

S is sequential, legal, equivalent to some completion of \( H \)

preserve precedence relation
Histories

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

- store values: binary? multivalued?
- two operations:
  - read: one reader? many readers?
  - write: one writer? many writers?
- safety property
Registers

• Safety: if operations don’t overlap every read return the last written value (or else the initial value)

• if operation overlap:

• safe register: any value

• regular register last written value or concurrently read value

• atomic registers: the operations can be totally ordered preserving legality and precedence (linearizability)
• weaker one: SRSW safe boolean register

• stronger one: MRMW Atomic multivalued register
Register Space

- MRMW
- MRSM
- SRSW
- Safe
- Regular
- Atomic
- m-valued
- Boolean
• Theorem: It is possible to implement multivalued MWMR atomic register from SWSR safe binary register

  • From binary safe SRSW to binary safe MRSW
  • From binary **safe** SWMR to binary **regular** SWMR
  • From **binary** regular MRSW to **multivalued** regular MRSW
  • From multivalued **regular** SRSW to multivalued **atomic** SRSW
  • From multivalued atomic SRSW to multivalued atomic MRSW
  • From multivalued atomic MRSW to multivalued atomic MRMW
Binary safe **SRSW** to binary safe **MRSW**

- P0 is the only writer, v is the initial value

`initially shared` array R of n SWSR binary safe register init v

**Code of P0**
```c
write(w){
    for ( int i=0; i<n; i++) R[i].write(w);
    return (ok)
}
```

**Code for Pi**
```c
read(){
    return R[i].read()
}
```

Work also:
- for multivalued regular

doesn’t work for atomic register
SWMR binary safe to SWMR binary regular

- P0 is the only writer and v is the initial value

```
initially shared R SWMR binary safe register init
local to P0 lw:=v //last written value

Code of P0
write(w){
    if w\neq lw then
        lw:=w; R.write(w)
    return (ok)
}

Code for all processes
read()
    return R.read()

```
From **binary** regular MRSW to **multivalued** regular MRSW
Representing $m$ Values

Unary representation:

bit[i] means value i

1 0 0 0 0 0 0 0

0 1 2 3 4 5 6 7
Writing $m$-Valued Register

Write 5

0 0 0 0 0 1

0 1 2 3 4 5 6 7
RegBoolMRSWRegister[M] bit;

public void write(int x) {
    bit[x].write(true);
    for (int i=x-1; i>=0; i--)
        bit[i].write(false);
}

public int read() {
    for (int i=0; i < M; i++)
        if (bit[i].read())
            return i;
}
}
SWSR regular to SWSR atomic

- Where is the problem?
- regular but not linearizable

Write(1) Write(2)

P0

P1 Read(2)

Read(1)

Timestamp
SWSR regular to SWSR atomic

- P0 is the only writer, P1 the only reader and v is the initial value

```
initially shared R SWMR binary safe register init
   local to P0 t:=0 //ltimestamp
   local to P1 lt:=0; lw //last timestamp, last written value

Code of P0
write(w){
   t:=t+1; R.write(t, w)
   return (ok)
}

Code for P1
read(){
   (t',w')=R.read()
   if ( lt<t' ) then (lt,lw):= ( t',w )
   return lw
}
```
SWSR atomic to SWMR atomic

Initially shared table[n,n] SWSR atomic register
local to P0 t:=0 //timestamp
local to readers lt:=0; lw //last timestamp, last written value

Code of writer
write(w){
    t:=t+1; for (int i=0;i>n; i++) table[i,i].write(w,t) //write diagonal
    return (ok)
}

Code for Preader I
read(){
    Read the row I, take the value lw with the maximum timestamp lt
    Write the column with this (lw, lt) except the diagonal
    return lw
}
SWMR atomic to MWMR atomic

initially shared table[n] SWMR atomic register
local to P0 t:=0 //ltimestamp

Code of writer I
write(w){
    Read table, take the maximum timestamp lt in tablz
    table[i]=(w,lt+1)
    return (ok)
}

Code for Preader I
read(){
    Read table, take the value lw with the maximum timestamp lt
    return lw
}
• All registers are (computationally) equivalent