FDS 2012: 
Shared memory basics

Shared memory model
- Processes communicate by applying operations on and receiving responses from shared objects
- A shared object is a state machine
  - States
  - Operations/Responses
  - Sequential specification
- Examples: read-write registers, T&S, C&S, LLSC, ...

Read-write register
- Stores values (in a value set V)
- Exports two operations: read and write
  - Write takes an argument in V and returns ok
  - Read takes no arguments and returns a value in V

We assume well-formed executions:
A process never invokes an operation before its previous invocation returns

Liveness
- An operation is complete if its invocation is followed by a matching response
  - write(v) → ok
  - read() → a value in V
- A process invoking an operation may fail (stop taking steps) before receiving a response
- A process is correct (in a given run) if it never fails

Under which condition a correct process makes progress?

Shared memory guarantees
Processes invoke operations on the shared objects and:
- Liveness: the operations eventually return something
- Safety: the operations never return anything incorrect

Wait-freedom: unconditional progress
Every operation invoked by a correct process eventually completes
All objects considered in this class are wait-free
We consider well-formed runs: a process never invokes an operation before returning from the previous invocation
A shared memory run

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

\[\text{read}() \quad 1\]

\[\text{write}(0) \quad \text{ok}\]

\[\text{write}(2)\]

\[\text{read}() \quad ?\]

\[\text{write}(0) \quad \text{ok}\]

\[\text{read}() \quad ?\]

\[\text{read}() \quad ?\]

\[\text{read}() \quad ?\]

Operation precedence

- Operation \(\text{op1}\) precedes operation \(\text{op2}\) in a run \(R\) if the response of \(\text{op1}\) precedes (in global time) the invocation of \(\text{op2}\) in \(R\).

- If neither \(\text{op1}\) precedes \(\text{op2}\) nor \(\text{op2}\) precedes \(\text{op1}\) than \(\text{op1}\) and \(\text{op2}\) are concurrent.

Safety (registers)

Informally, every read operation returns the “last” written value (the argument of the “last” write operation).

- What does the “last” mean?
- What if operations overlap?

Safety criteria

- Safe registers: every read that does not overlap with a write returns the last written value.
- Regular registers: every read returns the last written value, or the concurrently written value (assuming one writer).
- Atomic registers: the operations can be totally ordered, preserving legality and precedence (linearizability).
  - If \(\text{read1}\) returns \(v\), \(\text{read2}\) returns \(v'\), and \(\text{read1}\) precedes \(\text{read2}\), then \(\text{write}(v')\) cannot precede \(\text{write}(v)\).
Safe register

- write(0) ok
- write(1) ok
- read() 1
- read() 3
- read() 2

Regular register

- write(0) ok
- write(1) ok
- read() 1
- read() 0
- read() 1

Atomic register

- write(0) ok
- write(1) ok
- read() 1
- read() 0
- read() 1

Space of registers

- Values: from binary (V={0,1}) to multi-valued
- Number of readers and writers: from 1-writer 1-reader (1W1R) to multi-writer multi-reader (NWNR)
- Safety criteria: from safe to atomic

1W1R binary safe registers can be used to implement an NWNR multi-valued atomic registers!

Transformations

From 1W1R binary safe to 1WNR multi-valued atomic

I. From safe to regular (1W1R)
II. From one-reader to multiple-reader (regular binary or multi-valued)
III. From binary to multi-valued (1WNR regular)
IV. From regular to atomic (1W1R)
V. From 1W1R to 1WNR (multi-valued atomic)

1WNR binary safe -> 1WNR binary regular

Let p1 be the only writer and 0 be the initial value

Code for process p1:

```plaintext```

```
initially:
    shared 1WNR safe register R := 0
    lv := 0        \ last written value

upon write(v)
    if v ≠ lv then
        lv := v
        R.write(v)
        return ok

upon read()
    return R.read()
```
```
1WNR binary safe -> 1WNR binary regular

- Correctness:
  ✓ R is touched only to change its value
  ✓ both 0 and 1 are legal values in case of concurrency!

1W1R (binary regular) -> 1WNR (binary regular)

Let p1 be the only writer and 0 be the initial value

Code for process pi:
initially:
  shared array R[1..N] of 1W1R binary regular registers := 0
  // For all i, R[i] is written by p1 and read by pi
upon read():
  return R[i].read();
upon write(v)  // if i=1
  for all j do R[j].write(v)
  return ok

Transformations

From 1W1R binary safe to 1WNR multi-valued atomic

I. From safe to regular (1W1R)
II. From one-reader to multiple-reader (regular binary or multi-valued)
III. From binary to multi-valued (1WNR regular)
IV. From regular to atomic (1W1R)
V. From 1W1R to 1WNR (multi-valued atomic)

Binary -> M-valued (1WNR regular)

Code for process pi:
initially:
  shared array R[0..M-1] of 1WNR registers := [1,0,...,0]
upon read():
  for j = 0 to M-1 do
    if R[j].read() = 1 then return j
upon write(v)  // if i=1
  R[v].write(1)
  for j=v-1 down to 0 do R[j].write(0)
  return ok

Correctness:
✓ enough to consider a read that does not overlap with any write
✓ the last written value cannot be missed

Transformations

From 1W1R binary safe to 1WNR multi-valued atomic

I. From safe to regular (1W1R)
II. From one-reader to multiple-reader (regular binary or multi-valued)
III. From binary to multi-valued (1WNR regular)
IV. From regular to atomic (1W1R)
V. From 1W1R to 1WNR (multi-valued atomic)
Binary -> M-valued (1WNR regular)

- Correctness:
  ✓ only the last or concurrently written value can be returned
- HW what if:
  ✓ for j=0 to ν-1 do R[j].write(0)
  ✓ upon write(v) // if i=1
     for j=v-1 down to 0 do R[j].write(0)
     R[v].write(1)
     return ok

Transformations

From 1W1R binary safe to 1WNR multi-valued atomic

I. From safe to regular (1W1R)
II. From one-reader to multiple-reader (regular binary or multi-valued)
III. From binary to multi-valued (1WNR regular)
IV. From regular to atomic (1W1R)
V. From 1W1R to 1WNR (multi-valued atomic)

Histories

A history is a sequence of invocation and responses
E.g., p1-write(0), p2-read(), p1-ok, p2-0,...

A history is sequential if every invocation is immediately followed by a corresponding response
E.g., p1-write(0), p1-ok, p2-read(), p2-0,...

(A sequential history has no concurrent operations)

Legal histories

A sequential history is legal if it satisfies the sequential specification of the shared object

Read-write registers:
Every read returns the argument of the last write
(well-defined for sequential histories)
Complete operations and completions

Let $H$ be a history.
An operation $op$ is complete in $H$ 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 operations of $H$ followed with
matching responses.

Write(0)   ok
read()          1
write(0)   ok

read()   3
read()   3
read()   100

p1-write(0); p1-ok; p3-read(); p1-write(1); p3-3;
p3-read(); p1-ok; p2-read(); p2-1;

Equivalence

Histories $H$ and $H'$ are equivalent if for all $pi$
$H | p_i = H' | p_i$

E.g.:
$H = p_1-write(0); p_1-ok; p3-read(); p_3-3$
$H' = p_1-write(0); p_3-read(); p_1-ok; p_3-3$

Linearizability (atomicity)

A history $H$ is linearizable if there exists a legal
sequential history $S$ such that:

- $S$ is equivalent to some completion of $H$
- $S$ preserves the precedence relation of $H:
  \text{op1 precedes op2 in } H \Rightarrow \text{op1 precedes op2 in } S$
Atomic register

A register is atomic if every history it produces is linearizable.

Informally, the complete operations (and some incomplete operations) are seen as taking effect instantaneously at some time between their invocations and responses.

(The operations are atomic.)
From 1W1R regular to 1W1R atomic

Code for process p1:

iniAally:
shared 1W1R regular register R := 0
local variables t := 0, x := 0
upon read() (t',x') := R.read()
if t' > t then t:=t'; x:=x';
return(x)
upon write(v) // if i=1
t:=t+1
R.write(t,v)

Transformations

From 1W1R binary safe to 1WNR multi-valued atomic

I. From safe to regular (1W1R)
II. From one-reader to multiple-reader (regular binary or multi-valued)
III. From binary to multi-valued (1WNR regular)
IV. From regular to atomic (1W1R)
V. From 1W1R to 1WNR (multi-valued atomic)
Transformations-II

From one-reader to multiple-reader (regular binary or multi-valued)
• Every reader is assigned a dedicated register to read
• Writer writes in all
• Reader reads its own register

Transformations-III

From binary to M-valued (1WNR regular)
• Every value in \{0, ..., M-1\} is assigned a dedicated 1WNR register
• Write(v) sets R[v] to 1 and sets R[v-1] ... R[0] to 0
• Read returns the smallest v such that R[v]=1

Transformation IV

From regular to atomic (1W1R multi-valued)
• Write a timestamp with a value
• The reader returns the latest value and ignores the old one

Multiple readers?

Does not work either!

Multiple readers?

Does not work either!
We use:
- matrix $RR[1..N][1..N]$ of 1W1R atomic registers $= 0^{RN}$
  - for all $i, j$, $RR[i][j]$ is read by $pi$ and written by $pj$
- array $WR[1..N]$ of 1W1R atomic registers $= 0^N$
  - for all $i$, $WR[i]$ is written by $p1$ and read by $pi$

upon write($v$) // code for $p1$
  - $ts := ts + 1$
  - for all $j$ do $WR[j].write([v, ts])$
  - return ok

upon read() // code for $pi$
  - for all $j = 1, ..., N$ do $(t[j], x[j]) := RR[i][j].read()$
  - $(t[0], x[0]) := WR[i].read()$
  - $(t, x) := \text{highest}(t[...], x[...])$
  - for all $j$ do $RR[j][i].write([t, x])$
  - return($x$)

If read1 returns $v$ and read1 precedes read2 then
read2 cannot return a value that is older than $v$ –
sufficient for linearizability in the one-writer case

HW:
- What if readers don’t write?
- Multiple writers?