FDS 2012: Implementing an atomic bit

Administrivia

- Static web page:
- Next tutorial: May 7
- Next class: May 8

The space of registers

- Nb of writers and readers: from 1W1R to NWNR
- Size of the value set: from binary to multi-valued
- Safety properties: safe, regular, atomic

All registers are (computationally) equivalent!

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)
VI. From 1WNR to NWNR (multi-valued atomic)
This class

- Implementing an atomic bit
- Atomic snapshot: reading multiple locations atomically

How to implement an atomic bit?

The problem: implement a binary 1W1R atomic register (atomic bit) from binary 1W1R safe ones (safe bits).

A brute-force approach

- Binary 1W1R safe => binary 1W1R regular
- Binary 1W1R regular => multi-valued 1W1R regular
- Multi-valued 1W1R regular => multi-valued 1W1R atomic

The reader never writes (to the base objects), but...

- Unbounded # of base safe bits (to account for sequence numbers)

An optimal solution

- No sequence numbers?
- Bounded number of safe bits, O(1)?
- Bounded number of base actions, O(1)?

Can we do it if the reader does not write?
Safe bit to regular bit? Easy

- the writer is allowed only to change the value
- What about atomic?

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

Impossible if the reader does not write for bounded # of regular bits!

**Proof sketch** (by contradiction):

- Suppose only the writer executes writes on the base (regular) bits.
- Every write operation \( W(1) \) is a sequence of writes \( w_1, \ldots, w_k \) on base regular bits
  - Corresponds to the sequence of shared-memory states \( s_0, s_1, \ldots, s_k \) (defined for sequential runs)

\[ \text{write}(1) \]

\[ \text{p}_1 \quad s_0 \quad w_1 \quad s_1 \quad w_2 \quad \ldots \quad w_k \quad s_k \]

Proof (contd): digests

- There are only finitely many states!
  (bounded # of base registers)
- Each sequence \( s_0, s_1, \ldots, s_k \) of states (though possibly unbounded) defines a digest \( d_0, d_1, \ldots, d_m \)
  - \( d_0 = s_0 \), \( d_m = s_k \) (same global state transition)
  - \( d_0 = s_0 \Rightarrow i=j \) (all digest elements are distinct)
  - for all \( (d_i, d_{i+1}) \), exists \( (s_j, s_{j+1}) \) such that \( s_j = d_i \) and \( s_{j+1} = d_{i+1} \)
    - \( 7, 4, 8, 4, 2, 8, 3 \Rightarrow 7, 4, 8, 3 \)
- Each write operation “looks” like its digest
- There are only finitely many digests!

Proof (contd.): counter-example

- Consider a run with infinitely many alternating writes: \( W_1(1), W(0), W_2(1), \ldots \) (no reads)
  - Writes \( W_1, W_2, \ldots \) give an infinite sequence of digests \( D_1, D_2, \ldots \)
- At least one digest \( D = d_0, d_1, \ldots, d_m \) appears infinitely often in \( D_1, D_2, \ldots \)
  - Why?
- We can amend our run with a sequence of reads \( R_0, R_1, \ldots, R_m \) (in that order), each \( R_i \) “sees” state \( d_{m-i} \)
  - How?
Proof (contd.): the “switch”

- \( R_0 \) “sees” \( d_m \) and, thus, returns 1
  - Could have happened right after \( W(1) \)
- \( R_m \) “sees” \( d_0 \) and, thus, returns 0
  - Could have happened right before \( W(1) \)

\[ \Rightarrow \text{There exists } i \text{ such that } R_i \text{ returns 1 and } R_{i+1} \text{ returns 0 (by induction on } i=0,\ldots,m) \]

Proof (contd.): contradiction

- The (sequential) execution of \( R_i \) and \( R_{i+1} \) is indistinguishable (to the reader) from a concurrent one

```
\begin{array}{c}
\text{write}(1) \rightarrow \text{write to a base bit} \rightarrow \text{ok}
\end{array}
```

```
\begin{array}{cccc}
p_1 & \ldots & d_{m-1} & d_m & \ldots \\
& R_i & 1 & R_{i+1} & 0 \\
p_2 & d_m & \ldots & d_{m-1} & \end{array}
```

New-old inversion!

The reader must write

- And the writer must read
- But how the writer would tell what it read?
  - The writer needs at least two bits!
  - Why?
- Suppose the writer writes to one bit only
  - there are exactly two digests 0,1 and 1,0
  - suppose infinitely many \( W(1) \) operations export digests 0,1
  - new-old inversion:

```
\begin{array}{c}
\text{write}(1) \rightarrow \text{change the base bit from 0 to 1} \rightarrow \text{ok}
\end{array}
```

```
\begin{array}{c}
\text{read() 1 read() 0}
\end{array}
```

Optimal construction?

- Two bits for the writer
  - REG: for storing the current value
  - WR: for signaling to the reader
- One bit for the reader
  - RR: for signaling to the writer

Necessary, but is it also sufficient?
Evolutionary approach: Iteration 1

The reader should be able to distinguish the two cases:
✓ A new value was written: WR≠RR:
✓ The value is unchanged: WR=RR:

Writer: Reader:
change REG
if WR=RR then change WR
if WR≠RR then change RR
val:= REG
return val

Does not work: the read value does not depend on RR

Iteration 2

Return the “old” value if nothing changed
(local variable val initialized to the initial value of REG)

Writer: Reader:
change REG
if WR=RR then change WR
if WR=RR then return val
change RR
val:= REG
return val

Counter-example 2

Does not work: a read finds WR≠RR, a subsequent read finds WR=RR and reads an old value in REG (new-old inversion)

1. \(w_1(1)\) completes
2. \(r_1\) reads WR, finds WR≠RR and changes RR
3. \(w_2(0)\) begins, changes REG to 0, reads RR, finds WR = RR, changes WR, restoring the predicate WR≠RR, and completes
4. \(w_3(1)\) begins and starts changing REG from 0 to 1
5. \(r_1\) concurrently reads REG and returns the new value 1
6. \(r_2\) begins, finds RR≠WR, reads REG and returns the old value 0
Iteration 3

Read WR twice, if WR changed while the read is executed, return a conservative (old) value.

**Writer:**
- change REG
- if WR=RR then change WR

**Reader:**
- return val
- aux := REG
- change RR
- val := REG
- if WR=RR then return val
- return aux

Counter-example 3

Still a problem:
- *aux* gets a **new** value and *val* gets an **old** value (the two reads of REG are concurrent with a write on REG)
- succeeding read observes RR=WR (the two reads of WR are concurrent with a write on WR)

**Homework:** construct the counter-example

---

Iteration 4

Evaluate *val* again before returning the “old” value (*aux*): to make sure *val* is at least as old as *aux*

**Writer:**
- change REG
- if WR=RR then change WR

**Reader:**
- return val
- aux := REG
- change RR
- val := REG
- if WR=RR then return val
- return aux

Counter-example 4

Still (!) a problem:
- Unconditionally changing RR may be invalidated by a concurrent update of WR

**Solution:** check if WR≠RR again before changing RR
Final solution [Tromp, 1989]

**Writer protocol**
- change REG
- if WR=RR then return val
- change WR

**Reader protocol**
- (1) if WR=RR then return val
- (2) aux := REG
- (3) if WR≠RR then change RR
- (4) val := REG
- (5) if WR=RR then return val
- (6) val := REG
- (7) return aux

---

**Proof sketch: reading functions**

A *reading function* \( \pi \): for each complete read operation \( r \) (returning \( v \)), \( \pi(r) \) is a write operation \( w(v) \)

Show that for every run of the algorithm, there exists an atomic reading function \( \pi \):

(A0) No read \( r \) precedes \( \pi(r) \)
- No read returns a value not yet written

(A1) \( w \) precedes \( r \) => \( w=\pi(r) \) or \( w \) precedes \( \pi(r) \)
- No read obtains an overwritten value

(A2) \( r_1 \) precedes \( r_2 \) => \( \pi(r_2) \) does not precede \( \pi(r_1) \)
- No new/old inversion

A run is linearizable iff an atomic reading function exists (Section 4.3 of the lecture notes)

---

**Proof: constructing \( \pi \)**

- Let \( r \) return a value \( v \)
- Let \( \rho_r \) be the read of REG that got the value of \( r \)
  - If \( r \) returns in line 7, \( \rho_r \) is the read action in line 2 of \( r \)
  - If \( r \) returns in line 5, \( \rho_r \) is the read action in line 4
  - If \( r \) returns in line 1, \( \rho_r \) is the read in line 4 or 6 of some previous \( r' \) (depending on how \( r' \) returns)
- Let \( \phi_r \) be the last write action on REG that precedes or is concurrent to \( \rho_r \) and writes the value returned by \( r \) (and \( \rho_r \))
- Define \( \pi(r) \) as the write operation that contains \( \phi_r \)

---

**Proof: show that \( \pi \) is atomic**

- A0 is easy: by construction of \( \pi \), \( \pi(r) \) precedes or is concurrent to \( r \)
- A1? A2? Homework

Hint: assume the contrary and come to absurdum