Max Registers, Counters and Monotone Circuits

James Aspnes[1] Hagit Attiya[2] Keren Censor[2]

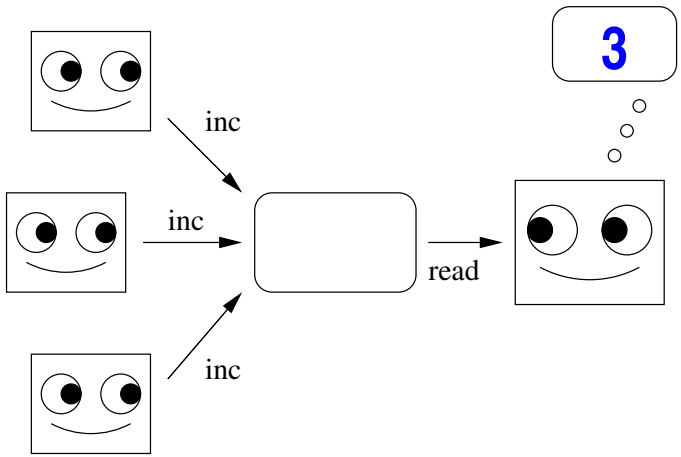
[1]Yale

[2]Technion

November 18th, 2009

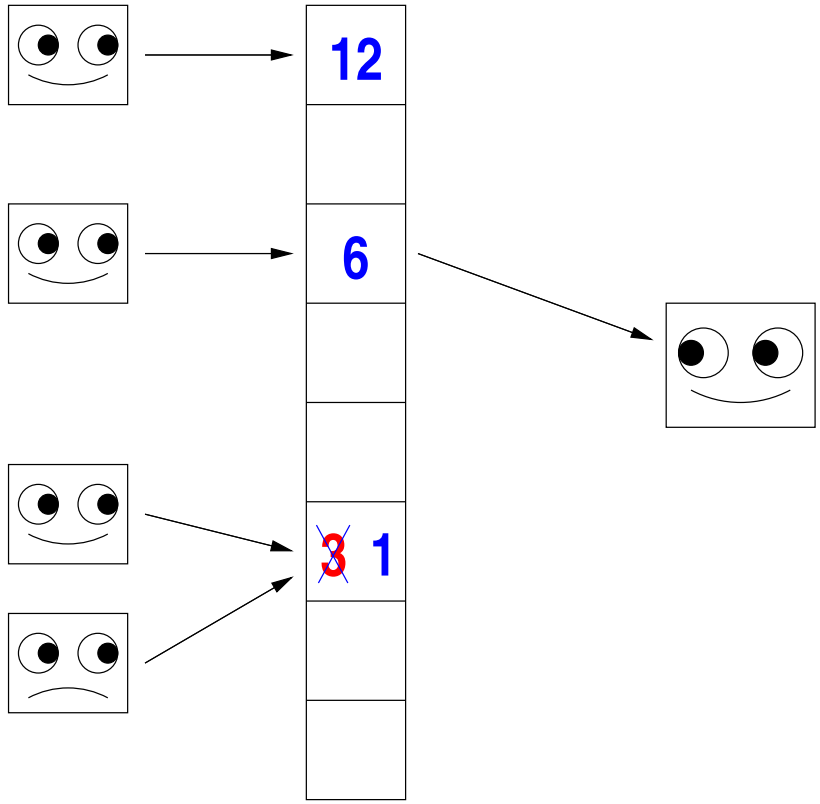
Counters

- Our goal: build a cheap **counter** for an asynchronous shared-memory system.
- Two operations: increment and read.
- Read returns number of previous increments.

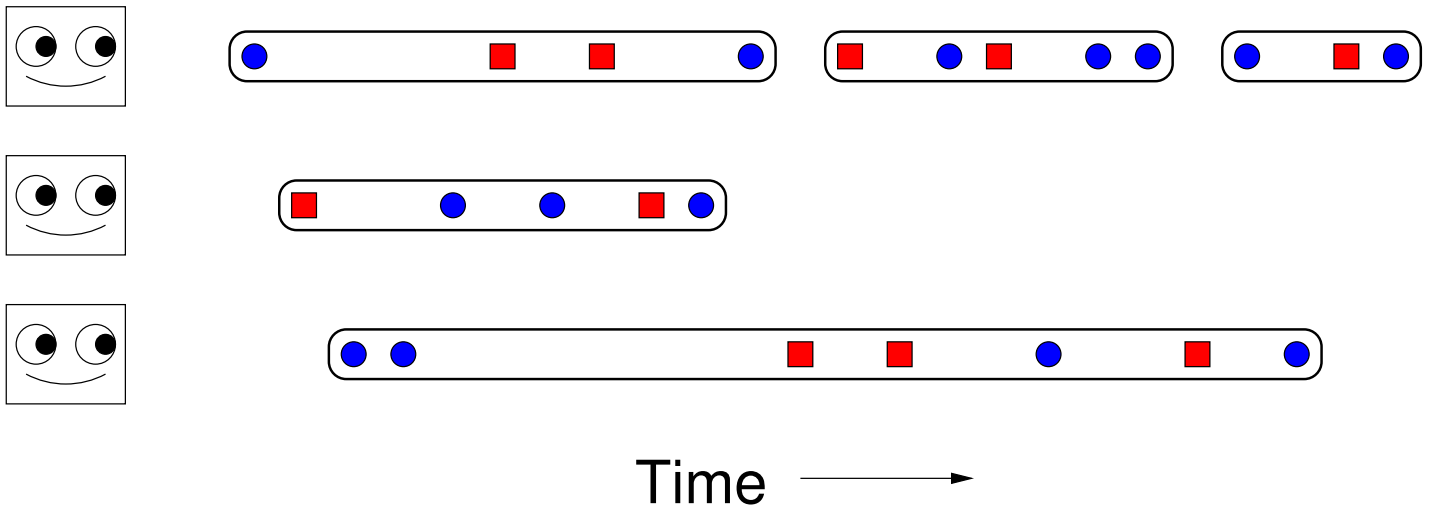


Model

- Processes can read and write shared **atomic registers**.
- Read on an atomic register returns value of last write.
- Timing of operations is controlled by an adversary.
- Cost of a high-level operation is number of low-level operations (register reads and writes) used.

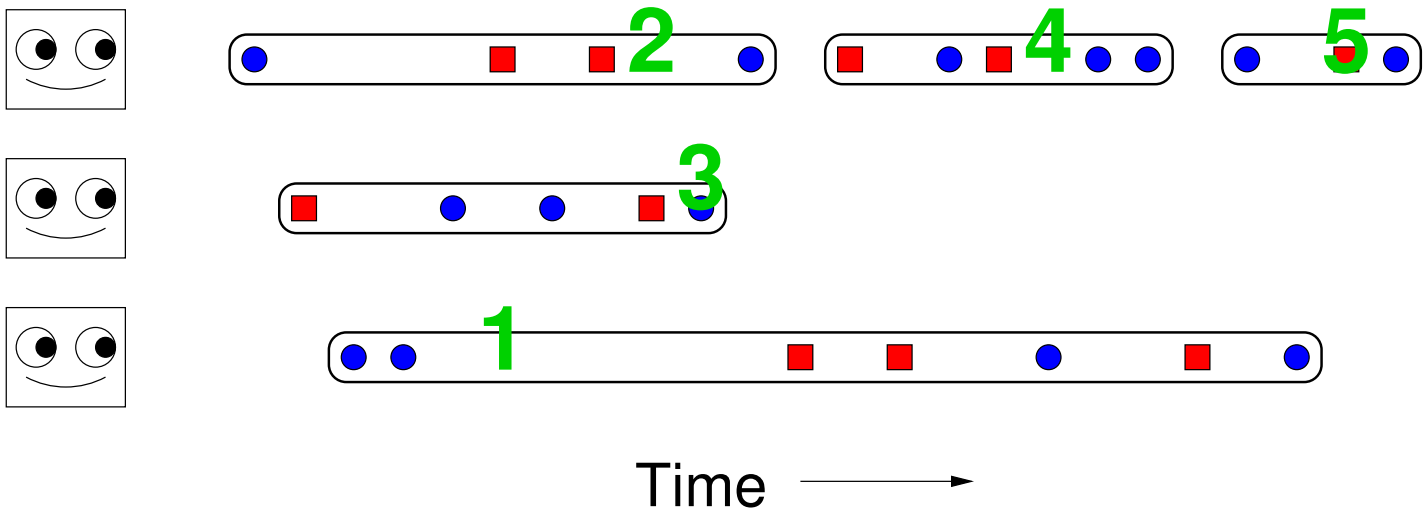


Executions



- **High-level operations** are overlapping sequences of **low-level operations**.
- **Wait-free** if all high-level operations finish in finite time for any interleaving.
- **Linearizable** if all high-level operations look like they happen atomically at some point in their execution interval.

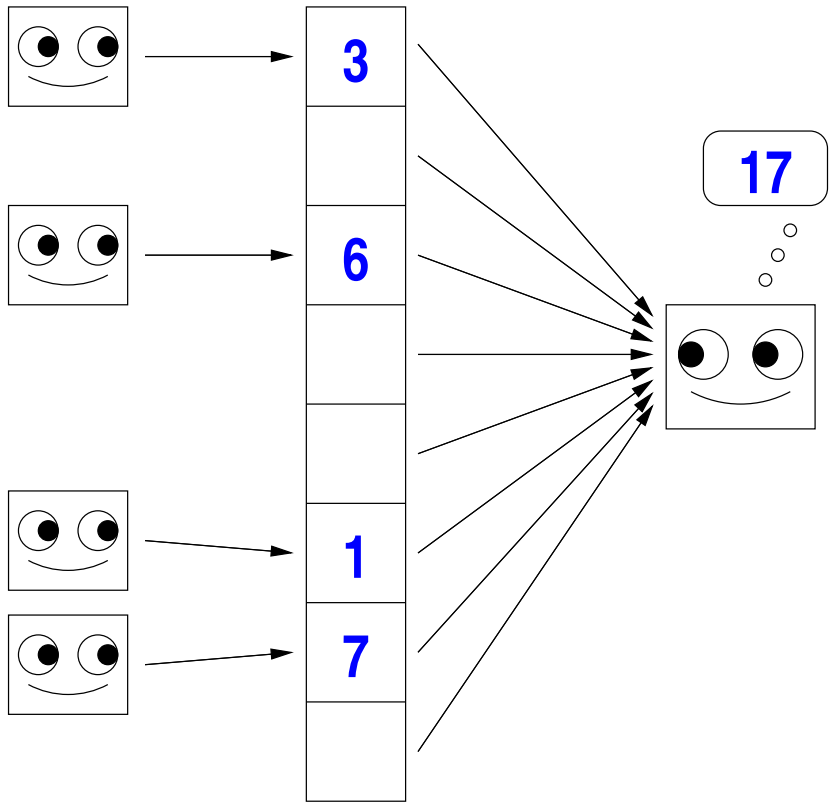
Executions



- **High-level operations** are overlapping sequences of **low-level operations**.
- **Wait-free** if all high-level operations finish in finite time for any interleaving.
- **Linearizable** if all high-level operations look like they happen atomically at some point in their execution interval.

Implementing a counter

- Each process writes its increments so far to a separate register.
- To read the counter, read all registers and add them up.
- Sum always includes writes that finish before read
- Cost: 1 for write, $n-1$ for read.
- Also works for other combining functions (e.g., max instead of sum).

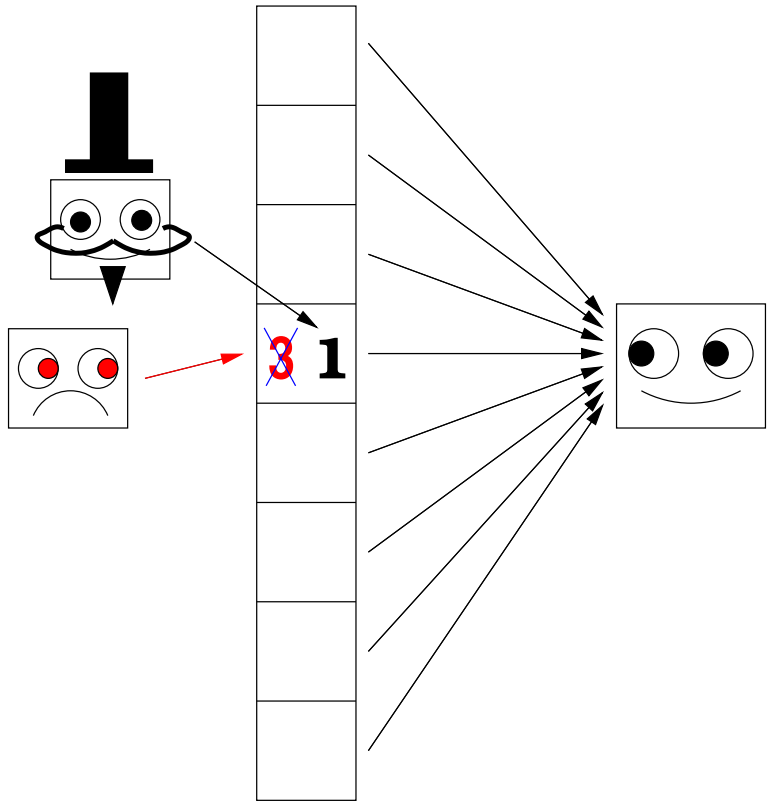


Properties of the collect-based counter

- Easy to show that it's **wait-free**: counter reads and writes are always 1 and $n-1$ operations each.
- Can also show **linearizability**:
  - For a counter increment, **linearization point** is time of its low-level write.
  - For a counter read that returns k, linearization point is either the start of the counter read, or just after the k-th increment, whichever comes later.
  - This works because a counter read that starts after k increments always returns at least k, while a counter read that finishes before k' increments always returns less than k'.
- But: still too expensive.

Lost update problem

- What if we try to use fewer registers?
- Two processes must write to same register.
- First write is lost.
- Second write may be very out-of-date.
- This is the **lost update problem**.
- Usually solved with locks (not wait-free) or stronger low-level objects (we don't have them).



Avoiding lost updates

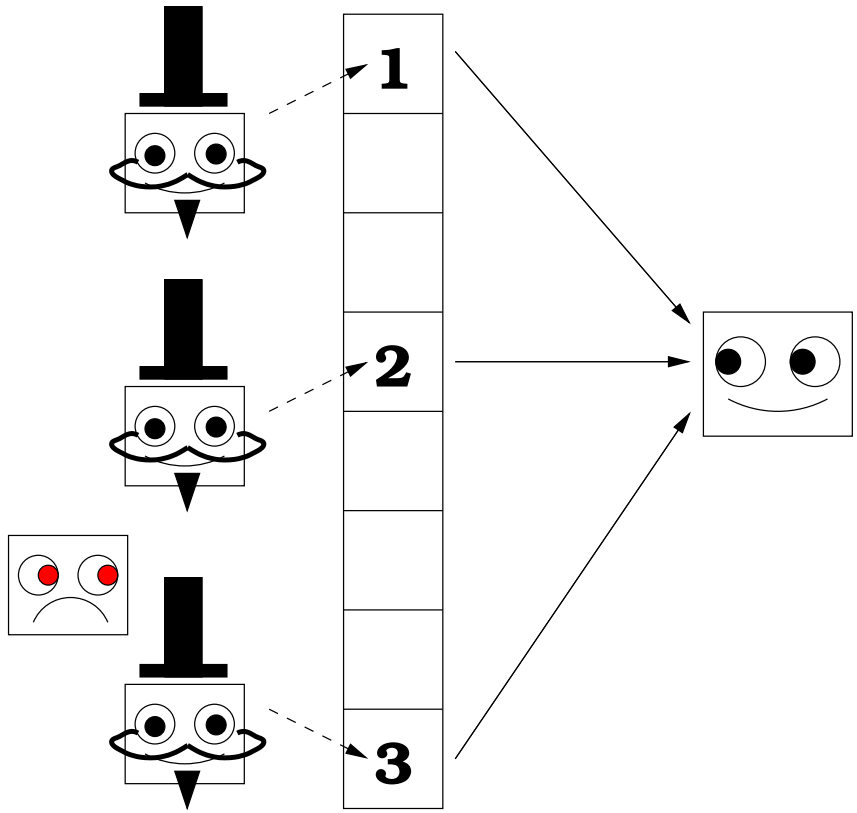
- Can we get around the lost update problem?
- No. (Jayanti, Tan, and Toeug, 2000):

  Any deterministic **solo-terminating**[1] implementation of a **perturbable** object[2] from registers[3] requires at least $n-1$ space and $n-1$ register operations for some high-level operation in the worst case.

[1] includes wait-free implementations
[2] includes counters
[3] or various other primitives

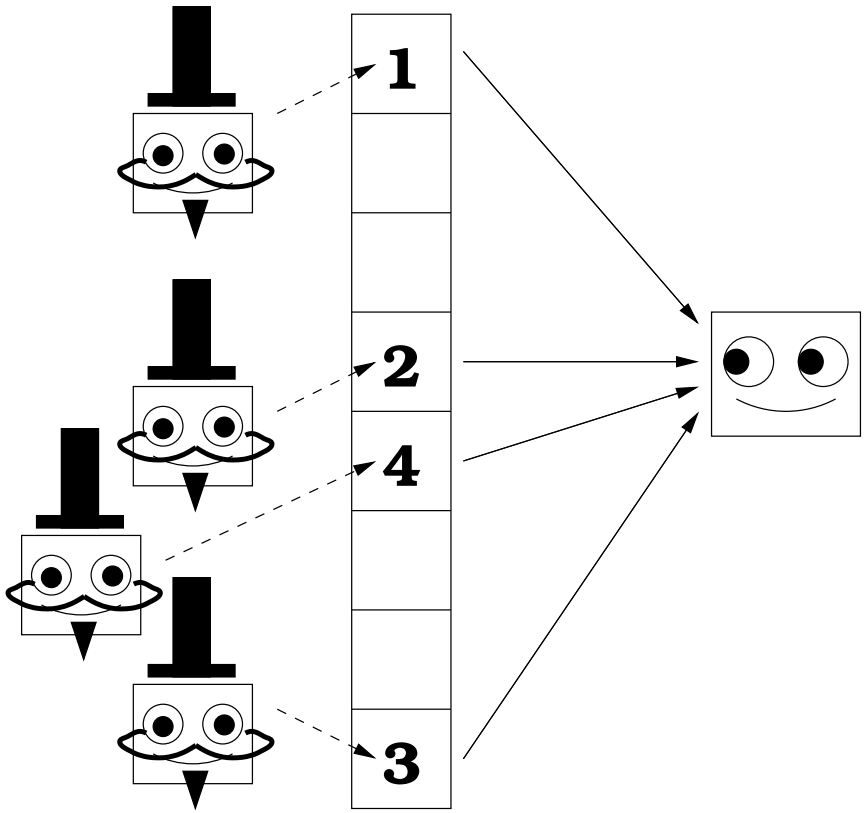
JTT lower bound: intuition

- Consider sequence of registers read by (deterministic) counter read.
- Build an execution where each of these registers is **covered** by a delayed write from an old increment.
- New increment must write to new register.
- Delay that write to cover another register.
- Continue until $n - 1$ registers covered.



JTT lower bound: intuition

- Consider sequence of registers read by (deterministic) counter read.
- Build an execution where each of these registers is **covered** by a delayed write from an old increment.
- New increment must write to new register.
- Delay that write to cover another register.
- Continue until $n - 1$ registers covered.



JTT lower bound: details

Bad execution $\Lambda_i \Sigma_i \Pi_i$ is constructed inductively:

- $\Lambda_i = \lambda_1 \lambda_2 \ldots \lambda_i$ consists of high-level operations by processes 1 through $n-1$, some of which are incomplete.
- $\Sigma_i = \sigma_1 \sigma_2 \ldots \sigma_i$ delivers pending write operations to distinct registers.
- Π_i is a counter read operation by process n that reads all registers written in Σ_i.

JTT lower bound: more details

We are building an execution

$$\Lambda_i \Sigma_i \Pi_i = \lambda_1 \lambda_2 \ldots \lambda_i \sigma_1 \sigma_2 \ldots \sigma_i \Pi_i$$

where Σ_i writes to i distinct registers, all read by Π_i.

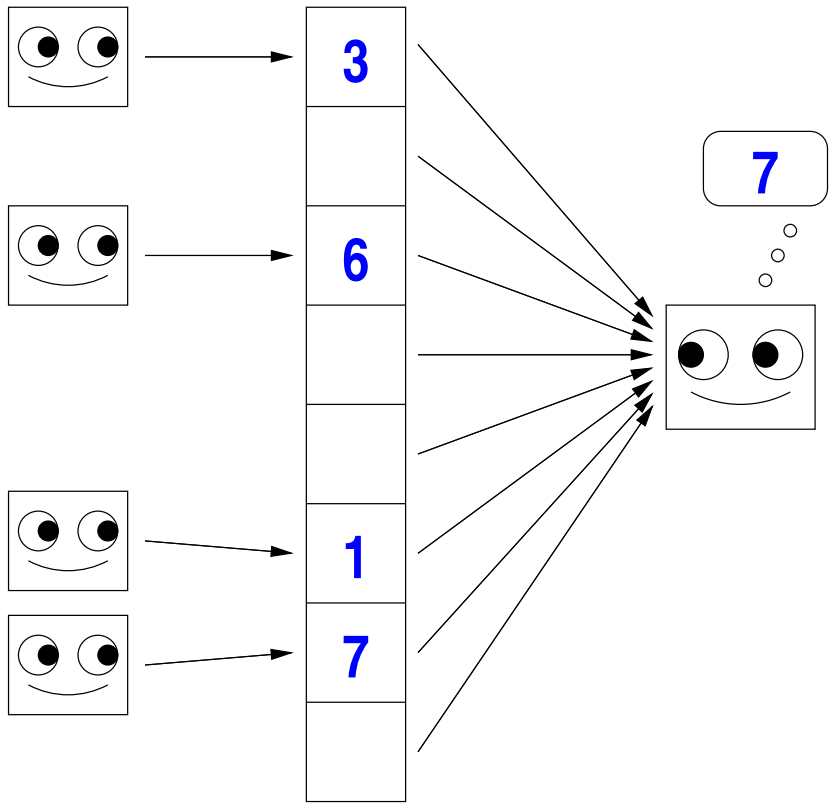
- Basis: $\Lambda_0 \Sigma_0$ is empty.
- Induction step:
  - Consider all sequences γ of operations by processes $< n$ and not in Σ_i such that the reader returns different values after $\Lambda_i \gamma \Sigma_i$ and $\Lambda_i \Sigma_i$. (**Perturbable** = some such sequence exists.)
  - Each of these must write to some uncovered register r. Choose the r that is read by the reader first and a γ that writes to it.
  - Write $\gamma = \lambda'_{i+1} \sigma_{i+1} \tau_{i+1}$ where σ_{i+1} writes to r.
  - Let λ_{i+1} extend λ'_{i+1} by finishing all other pending operations.
- At end of induction, Π_{n-1} reads $\geq n-1$ distinct registers.

We are doomed!

- Nobody is going to pay $n-1$ operations to read a counter.
- But wait: maybe there is a way around JTT.

Max registers

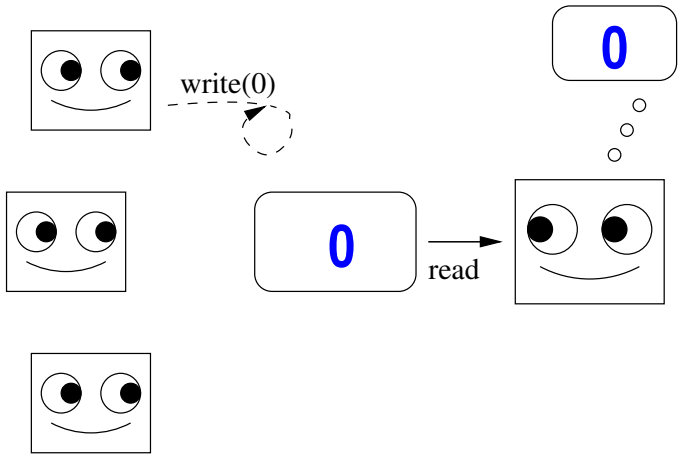
- **Max register** read operation returns the maximum value ever written to it.
- Solves the lost update problem: writes are (effectively) ordered by value, not time.
- Easy implementation using collects: read all registers and compute max.
- Problem: max registers are perturbable, so JTT applies to them too.



Bounded max registers

We will escape JTT by considering *bounded* max registers.

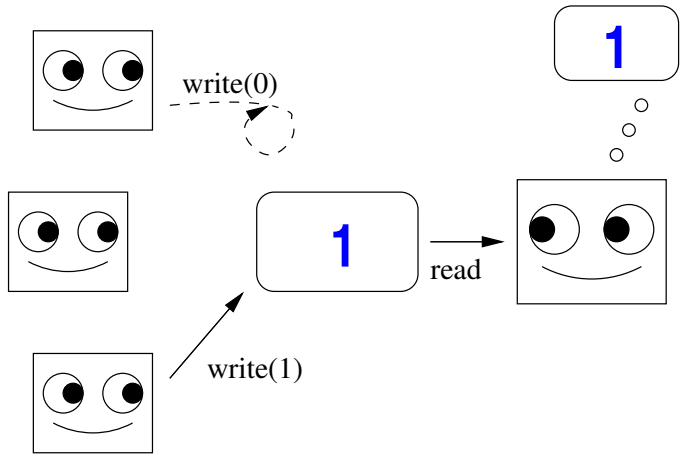
- 2-valued max register = 1-bit atomic register.
- Write(v) operation: If $v = 1$, write 1, else do nothing.
- Read operation: Just read the register.
- Trivially wait-free and linearizable.



Bounded max registers

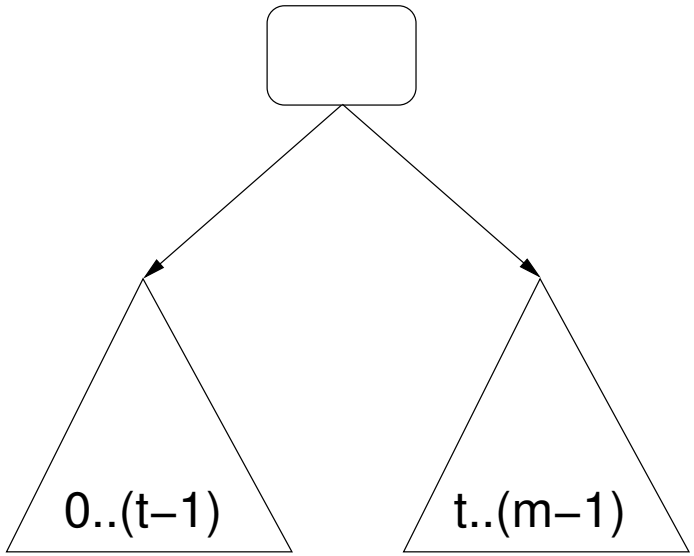
We will escape JTT by considering *bounded* max registers.

- 2-valued max register = 1-bit atomic register.
- Write(v) operation: If $v = 1$, write 1, else do nothing.
- Read operation: Just read the register.
- Trivially wait-free and linearizable.



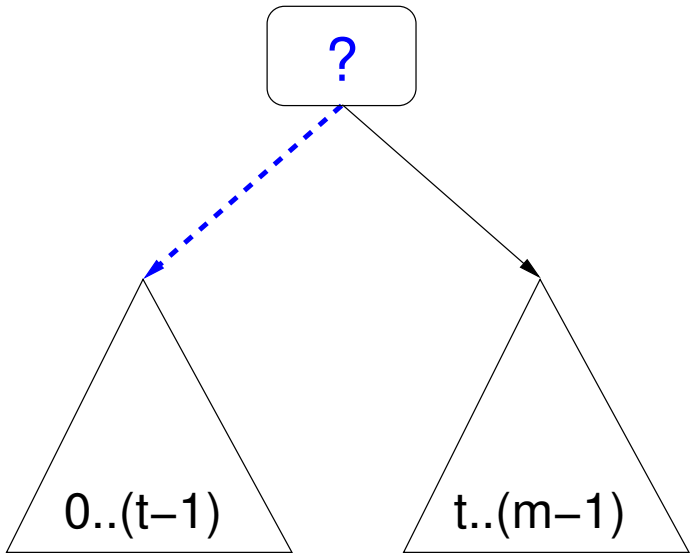
Tree-based max registers

- Multiplex two max registers through one selector bit.
- Left register holds values $0 \ldots t-1$.
- Right register holds values $\geq t$.



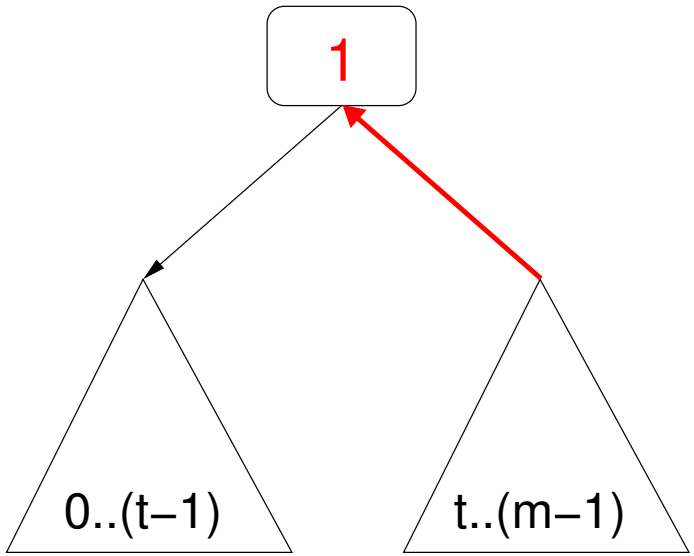
Tree-based max registers

- Multiplex two max registers through one selector bit.
- Left register holds values $0 \ldots t-1$.
- Right register holds values $\geq t$.
- To write k:
  - If $k < t$, read selector bit first: if 0, write k to left register (else do nothing).
  - If $k \geq t$, write $k - t$ to right register, then write 1 to selector bit.



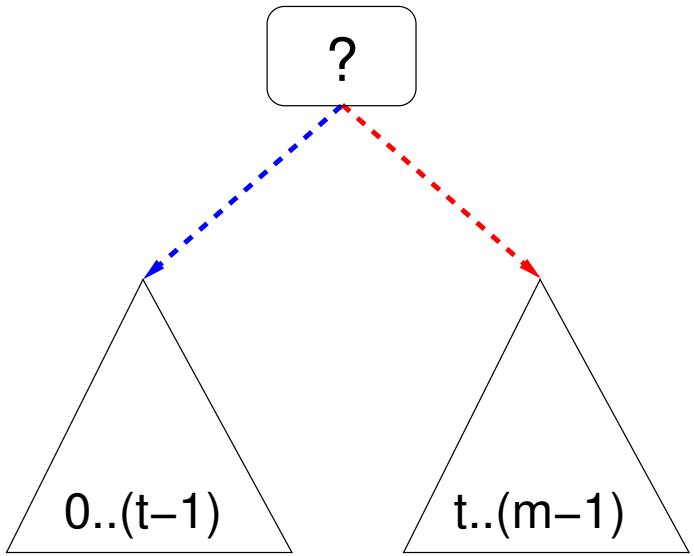
Tree-based max registers

- Multiplex two max registers through one selector bit.
- Left register holds values $0 \ldots t-1$.
- Right register holds values $\geq t$.
- To write k:
  - If $k < t$, read selector bit first: if 0, write k to left register (else do nothing).
  - If $k \geq t$, write $k - t$ to right register, then write 1 to selector bit.



Tree-based max registers

- Multiplex two max registers through one selector bit.
- Left register holds values $0 \ldots t-1$.
- Right register holds values $\geq t$.
- To read the register: If selector $= 0$, return read(left), else return read(right)$+t$.



Linearizability

- Claim: If child registers are linearizable, so is combined register.
- Proof: By constructing an explicit linearization.

Linearizability (continued)

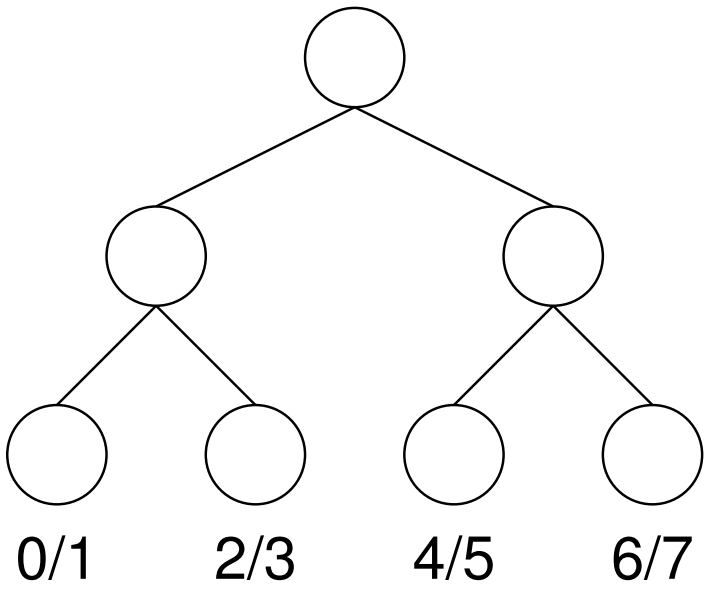
- Two categories of operations, based on value of the selector bit:

0	1
Read left	Read right
Write left	Write right
	Don't write left

- Linearize column 0 before column 1.
- Within each column, linearize operations using linearization order for left/right registers.
- This omits no-op writes. These have no effect, so put them anywhere consistent with timing.

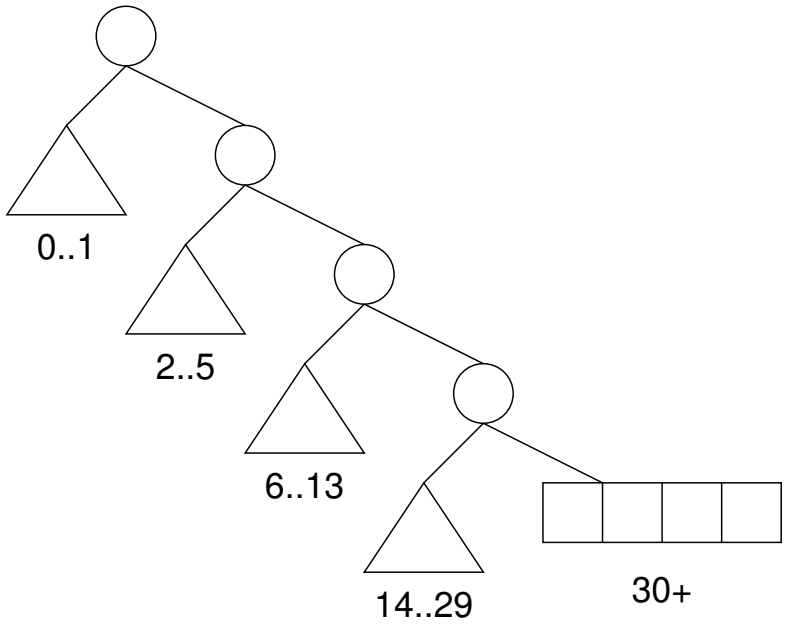
Cost of bounded max register

- Cost of read and write = depth of tree.
- For m-valued max register, use balanced tree of depth $\lceil \lg m \rceil$.
- If $m \geq 2^{n-1}$, use collects instead.
- Cost: $\min(\lceil \lg m \rceil, n-1)$.



Cost of unbounded max register

- Use *unbalanced* tree to make cost of write/read proportional to value v.
- Simplest scheme is to have left register double in size at each level.
- More efficient trees can be derived from prefix-free codes. (Bentley and Yao, 1976)
- Cost of operations: $O(\min(\lg(v+1), n))$ where v is value written (or read).



Lower bound for max registers

- Is there a lower bound for bounded max registers?
- Yes: And it *exactly* matches the $\min(\lceil \lg m \rceil, n-1)$ upper bound from the tree-based construction.
- Even stronger: given *any* solo-terminating, linearizable max register, we can extract an equally good tree-based max register from it.

Lower bound: details

- Consider executions consisting of (a) max-register writes Λ (possibly incomplete) by processes 1 through $n-1$ followed by (b) a single max-register read Π by process n. Let $T(m, n)$ be optimal reader cost for executions with this structure with m values.
- Let r be the first register read by process n.
- Let S_k be the set of all sequences of writes that only write values $\leq k$.
- Let t be the smallest value such that some execution in S_t writes to r.

Lower bound: two cases

Recall: r is first register read by process n, t is smallest value such that some execution α, with max value t, writes to r.
First case:

- Since t is smallest, no execution in S_{t-1} writes to r.
- If we restrict writes to values $\leq t-1$, we can omit reading r.
- Thus, $T(t,n) \leq T(m,n) - 1 \Rightarrow T(m,n) \geq T(t,n) + 1$

Lower bound: two cases

Recall: r is first register read by process n, t is smallest value such that some execution α, with max value t, writes to r.
Second case:

- Split α as $\alpha'\delta\beta$ where δ is first write to r, by some process p_i.
- Construct a new execution $\alpha'\nu$ by letting all max-register writes except the one performing δ finish.
- Now consider any execution $\alpha'\nu\gamma\delta$, where γ is any sequence of max-register writes with values $\geq t$ that excludes p_i and p_n.
- Reader always sees the same value in r following these executions, but otherwise (starting after $\alpha'\nu$) we have an $(n-1)$-process max-register with values t through $m-1$.
- Omit read of r to get $T(m,n) \geq T(m-t,n-1)+1$.

Lower bound: recurrence

We've shown this recurrence:

$$T(m,n) \geq 1 + \min_t \left(\max\left(T(t,n), T(m-t,n-1)\right)\right).$$

$$T(1,n) = 0.$$

$$T(m,1) = 0.$$

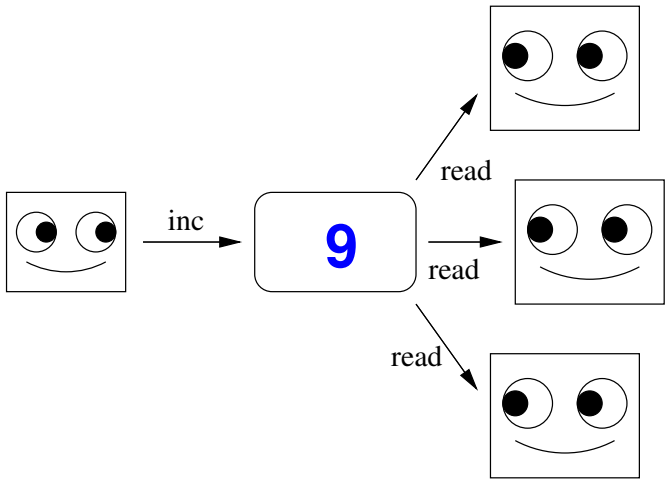
- Solution is exactly $T(m,n) = \min\left(\lceil \lg m \rceil, n-1\right)$.
- Also gives same recursive split as tree-based implementation.
- Same argument works for m-valued counters.

From max-registers to counters

- Now that we understand max-registers, how does this help us with counters?

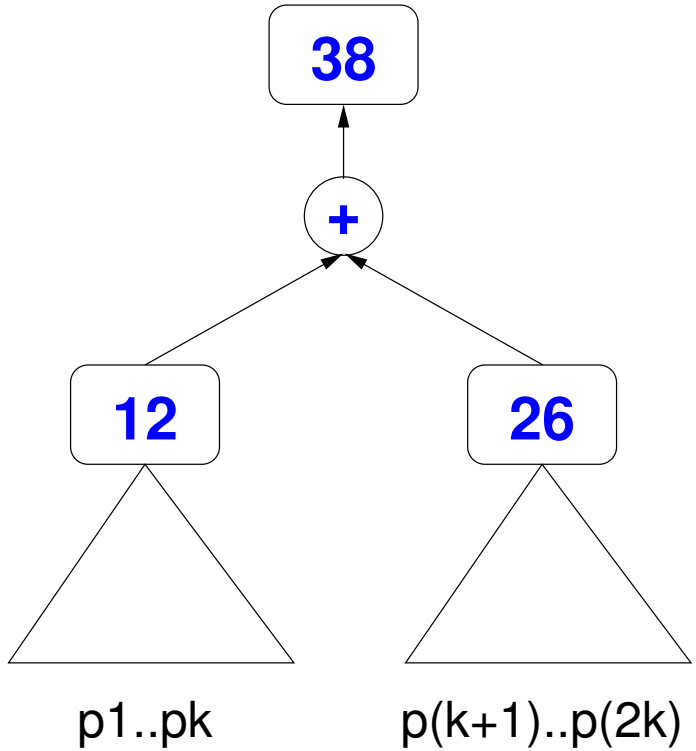
Counting with one incrementer

- 1-incrementer counter = one atomic register.
- Increment operation: $r \leftarrow r + 1$.
- Read operation: Just read the register.
- Trivially wait-free and linearizable.



Counter with more incrementers

- Combine two k-incrementer counters to get a $2k$-incrementer counter.
- Max register at root holds $C_1 + C_2$.
- Increment operation:
  1. Increment C_i.
  2. Read C_1 and C_2.
  3. Store $C_1 + C_2$ in max register.
- Read operation: just read the root.



Linearizability

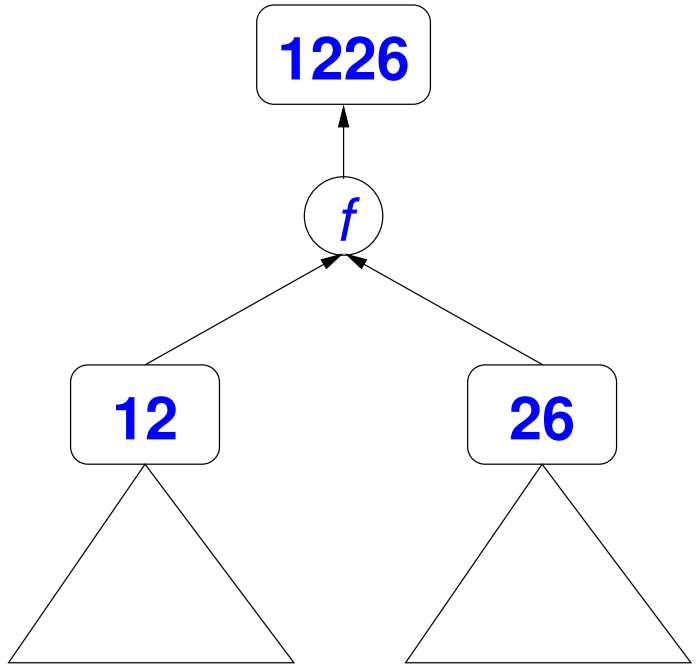
- Root register is max register and thus monotone increasing.
- Root register never exceeds $C_1 + C_2$.
- Each increment can be assigned the value of $C_1 + C_2$ when it finishes its sub-counter increment.
- The i-th increment writes at least i to root before it finishes.
- The i-th increment can be linearized at the first time a value $\geq i$ is written to root (sort by i if there are ties).

Cost

- Increment operation uses $O(\log n)$ max-register operations.
- Read operations uses 1 max-register read.
- So an m-valued counter costs $O(\log n \log m)$ for an increment and $O(\log m)$ for a read.
- For polynomial m, this is $O(\log^2 n)$ and $O(\log n)$.
- Read cost is tight (max register lower bound extends to counters).
- Write cost might not be tight.

General monotone circuits

- Same construction as used for counter works for any monotone combining functions.
- Update operation:
  1. Update C_i.
  2. Read C_1 and C_2.
  3. Store $f(C_1, C_2)$ in max register.
- Also works for non-trees if we propagate in the right order.
- Result is generally *not* linearizable.



Examples

- Generalized counters:
  - $f(x, y) = x + y$.
  - But no requirement that incrementers increase inputs by just 1.
- Approximate counters:
  - $f(x, y) = \left\lceil \log_{1+\epsilon}((1+\epsilon)^x + (1+\epsilon)^y) \right\rceil$.
  - These reduce the size of the intermediate max registers (and thus the cost of updates).
- Threshold objects:
  - Like a generalized counter, but final output is just $< t$ or $\geq t$.
  - Linearizable.

Monotone consistency

- If general monotone circuits aren't linearizable, what good are they?
- **Monotone consistency**:
  - Output is non-decreasing.
  - Output is always as least as big as it should be: $r \geq f(x_1 \ldots x_n)$, where $x_1 \ldots x_n$ are values of all updates that finish before the read starts.
  - Output is never bigger than it should be: $r \leq f(X_1 \ldots X_n)$, where $X_1 \ldots X_n$ are values of all updates that start before the read finishes.
- This is good enough for testing thresholds.

What we have

- New **max register** data structure with an optimal implementation from atomic registers.
- New implementation of **bounded counters** with polylogarithmic operations.
- General method for replacing linear-time collects and snapshots with polylogarithmic-time circuits, when computing monotone summary functions.

For more details, see our paper in PODC 2009.

Open problems

- Can we reduce the cost of a counter write?
- Does randomization help?
  - Lower bound: If writes cost at most w, randomized reads cost $\Omega(\log n/(\log w + \log\log n))$ operations with probability $1 - o(1)$.
  - Not clear if this is tight.
- Can we improve the monotone circuit construction?
  - A fast snapshot operation on pairs of max registers could give us linearizability instead of monotone consistency.