Chapter 12: Distributed Shared Memory

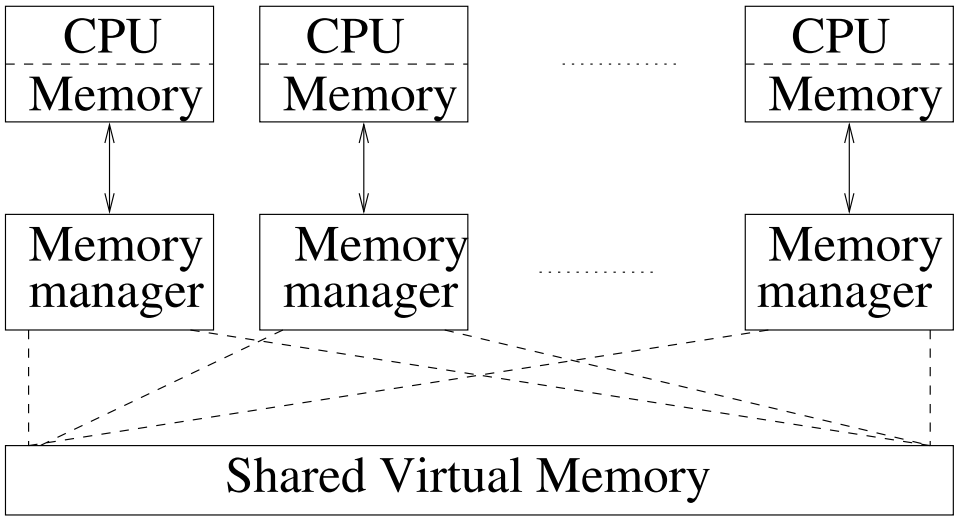
Ajay Kshemkalyani and Mukesh Singhal

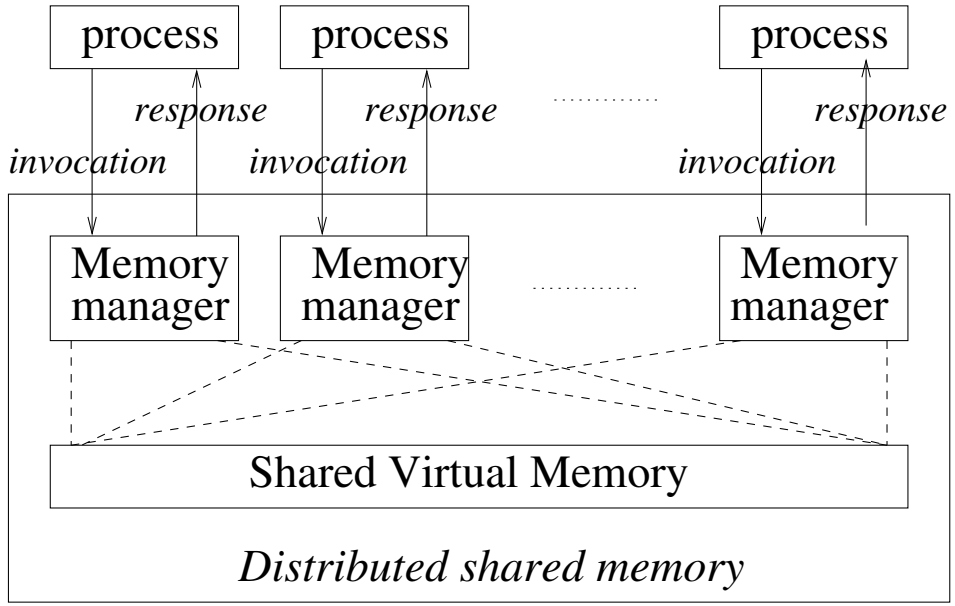
Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Distributed Shared Memory Abstractions

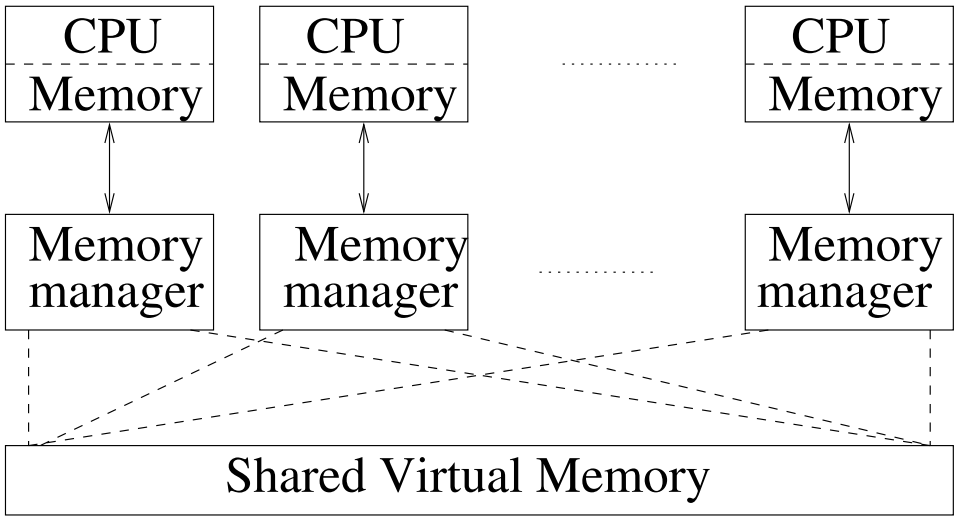
- communicate with Read/Write ops in shared virtual space
- No Send and Receive primitives to be used by application
  - Under covers, Send and Receive used by DSM manager
- Locking is too restrictive; need concurrent access
- With replica management, problem of consistency arises!
- $\Longrightarrow$ weaker consistency models (weaker than von Neumann) reqd

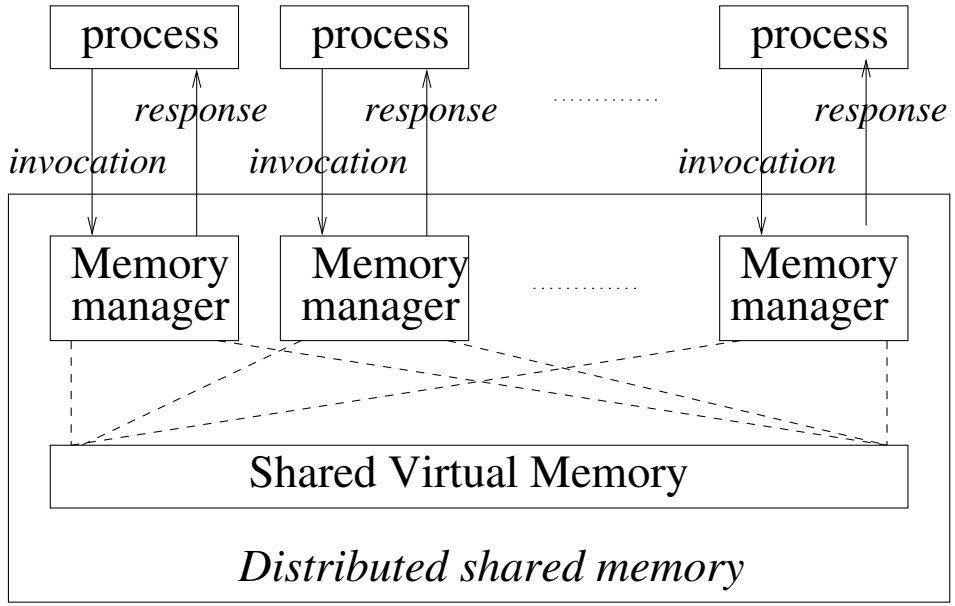




Distributed Shared Memory Abstractions

- communicate with Read/Write ops in shared virtual space
- No Send and Receive primitives to be used by application
  - Under covers, Send and Receive used by DSM manager
- Locking is too restrictive; need concurrent access
- With replica management, problem of consistency arises!
- $\Longrightarrow$ weaker consistency models (weaker than von Neumann) reqd





Advantages/Disadvantages of DSM

Advantages:

- Shields programmer from Send/Receive primitives
- Single address space; simplifies passing-by-reference and passing complex data structures
- Exploit locality-of-reference when a block is moved
- DSM uses simpler software interfaces, and cheaper off-the-shelf hardware. Hence cheaper than dedicated multiprocessor systems
- No memory access bottleneck, as no single bus
- Large virtual memory space
- DSM programs portable as they use common DSM programming interface

Disadvantages:

- Programmers need to understand consistency models, to write correct programs
- DSM implementations use async message-passing, and hence cannot be more efficient than msg-passing implementations
- By yielding control to DSM manager software, programmers cannot use their own msg-passing solutions.

Advantages/Disadvantages of DSM

Advantages:

- Shields programmer from Send/Receive primitives
- Single address space; simplifies passing-by-reference and passing complex data structures
- Exploit locality-of-reference when a block is moved
- DSM uses simpler software interfaces, and cheaper off-the-shelf hardware. Hence cheaper than dedicated multiprocessor systems
- No memory access bottleneck, as no single bus
- Large virtual memory space
- DSM programs portable as they use common DSM programming interface

Disadvantages:

- Programmers need to understand consistency models, to write correct programs
- DSM implementations use async message-passing, and hence cannot be more efficient than msg-passing implementations
- By yielding control to DSM manager software, programmers cannot use their own msg-passing solutions.

Issues in Implementing DSM Software

- Semantics for concurrent access must be clearly specified
- Semantics – replication? partial? full? read-only? write-only?
- Locations for replication (for optimization)
- If not full replication, determine location of nearest data for access
- Reduce delays, # msgs to implement the semantics of concurrent access

- Data is replicated or cached
- Remote access by HW or SW
- Caching/replication controlled by HW or SW
- DSM controlled by memory management SW, OS, language run-time system

Issues in Implementing DSM Software

- Semantics for concurrent access must be clearly specified
- Semantics – replication? partial? full? read-only? write-only?
- Locations for replication (for optimization)
- If not full replication, determine location of nearest data for access
- Reduce delays, # msgs to implement the semantics of concurrent access

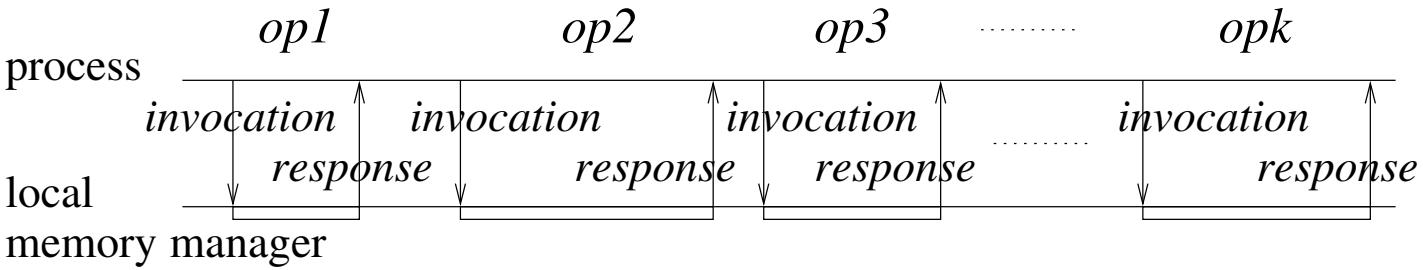
- Data is replicated or cached
- Remote access by HW or SW
- Caching/replication controlled by HW or SW
- DSM controlled by memory management SW, OS, language run-time system

Comparison of Early DSM Systems

Type of DSM	Examples	Management	Caching	Remote access
single-bus multiprocessor	Firefly, Sequent	by MMU	hardware control	by hardware
switched multiprocessor	Alewife, Dash	by MMU	hardware control	by hardware
NUMA system	Butterfly, CM*	by OS	software control	by hardware
Page-based DSM	Ivy, Mirage	by OS	software control	by software
Shared variable DSM	Midway, Munin	by language runtime system	software control	by software
Shared object DSM	Linda, Orca	by language runtime system	software control	by software

Memory Coherence

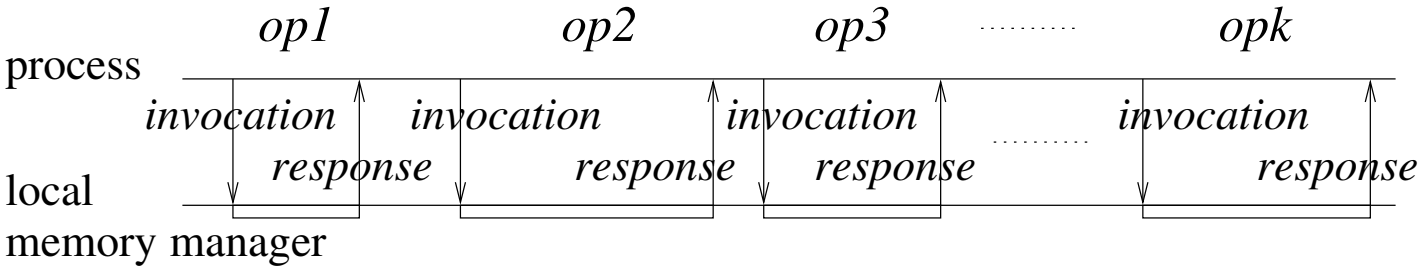
- s_i memory operations by P_i
- $(s_1 + s_2 + \ldots s_n)!/(s_1!s_2! \ldots s_n!)$ possible interleavings
- Memory coherence model defines which interleavings are permitted
- Traditionally, Read returns the value written by the most recent Write
- "Most recent" Write is ambiguous with replicas and concurrent accesses
- DSM consistency model is a *contract* between DSM system and application programmer



Strict Consistency/Linearizability/Atomic Consistency

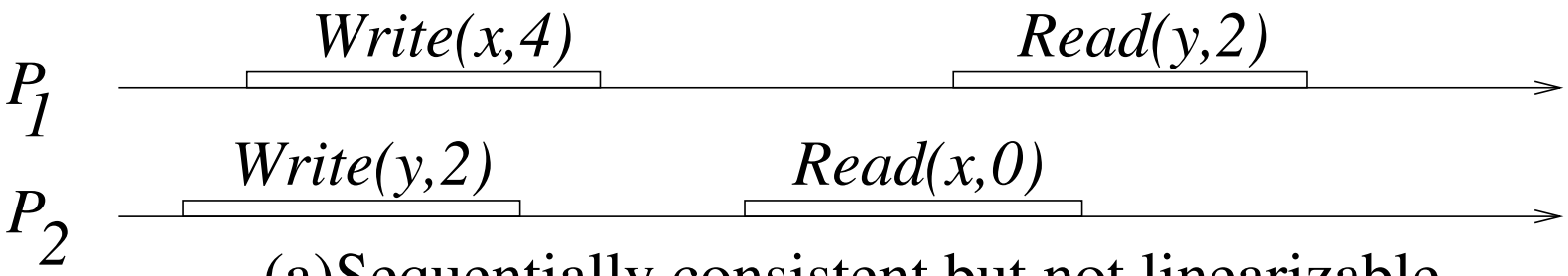
Strict consistency

1. A Read should return the most recent value written, per a global time axis. For operations that overlap per the global time axis, the following must hold.
2. All operations appear to be atomic and sequentially executed.
3. All processors see the same order of events, equivalent to the global time ordering of non-overlapping events.

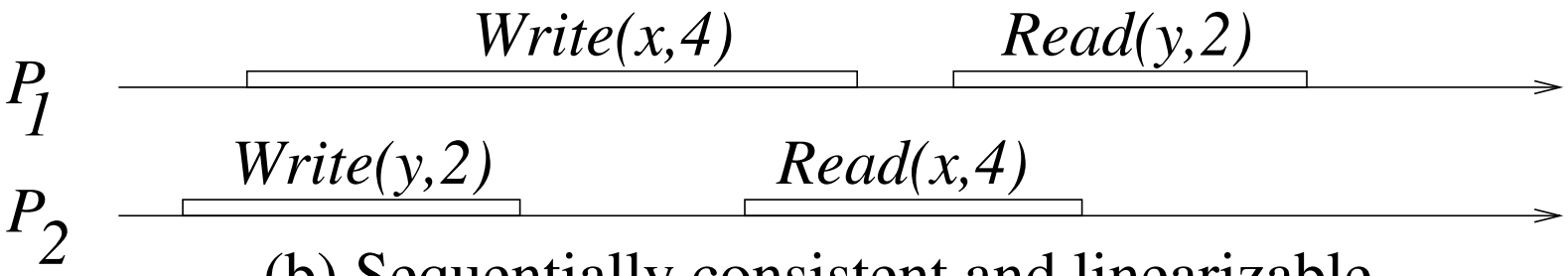


Sequential invocations and responses to each Read or Write operation.

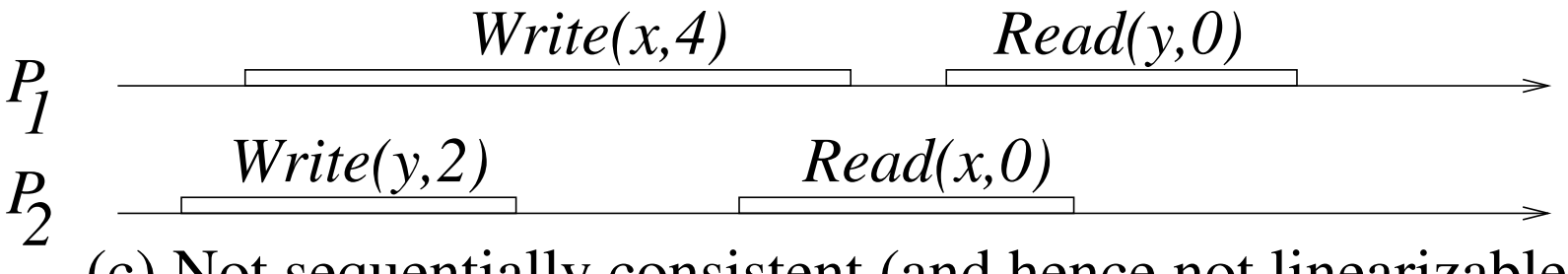
Strict Consistency / Linearizability: Examples



(a)Sequentially consistent but not linearizable



(b) Sequentially consistent and linearizable



(c) Not sequentially consistent (and hence not linearizable

Initial values are zero. (a),(c) not linearizable. (b) is linearizable

Linearlzability: Implementation

- Simulating global time axis is expensive.
- Assume full replication, and total order broadcast support.

(shared var)
int: x;

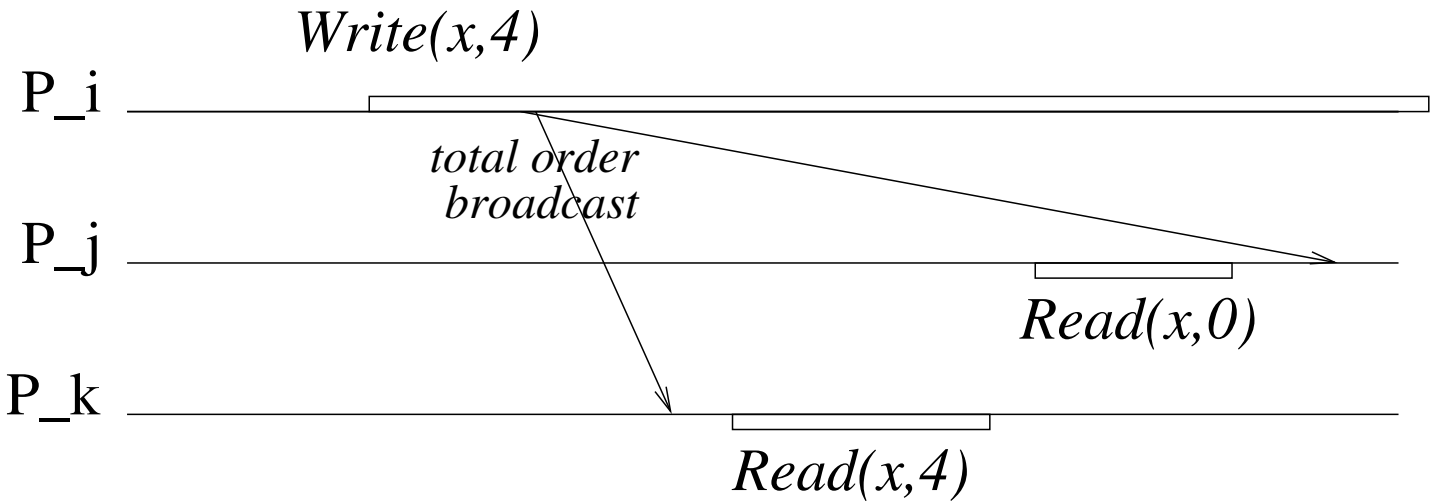
(1) When the Memory Manager receives a *Read* or *Write* from application:
(1a) **total_order_broadcast** the *Read* or *Write* request to all processors;
(1b) **await** own request that was broadcast;
(1c) **perform** pending response to the application as follows
(1d) **case** *Read*: return value from local replica;
(1e) **case** *Write*: write to local replica and return ack to application.

(2) When the Memory Manager receives a **total_order_broadcast**(*Write*, x, *val*) from network:
(2a) **write** *val* to local replica of x.

(3) When the Memory Manager receives a **total_order_broadcast**(*Read*, x) from network:
(3a) **no operation**.

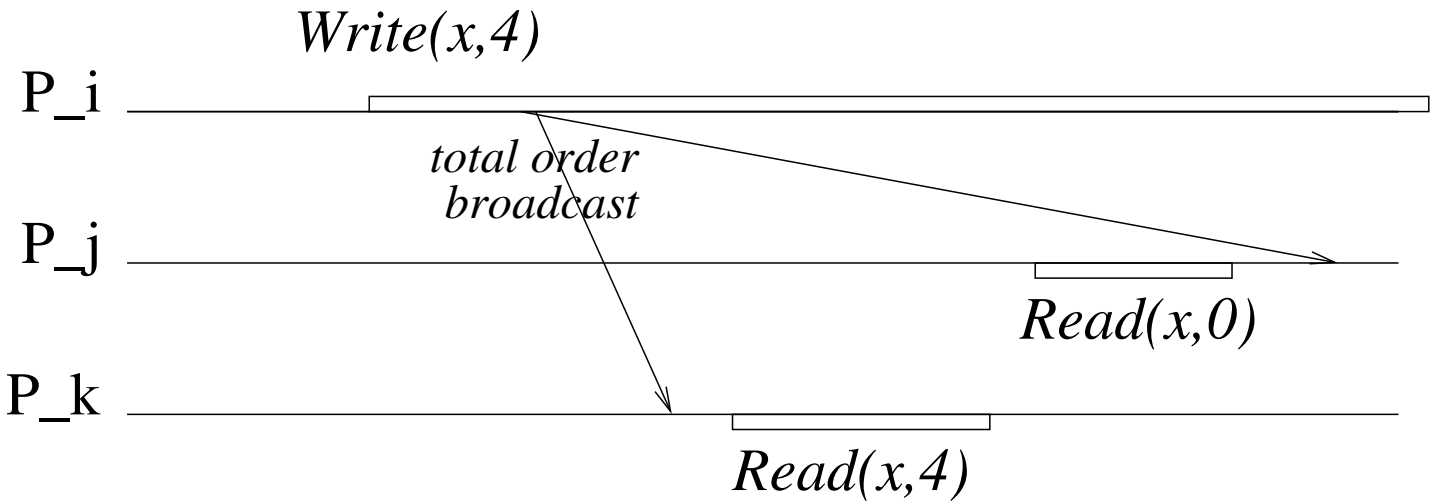
Linearizability: Implementation (2)

- When a Read in simulated at other processes, there is a no-op.
- Why do Reads participate in total order broadcasts?
- Reads need to be serialized w.r.t. other Reads and all Write operations. See counter-example where Reads do not participate in total order broadcast.



Linearizability: Implementation (2)

- When a Read in simulated at other processes, there is a no-op.
- Why do Reads participate in total order broadcasts?
- Reads need to be serialized w.r.t. other Reads and all Write operations. See counter-example where Reads do not participate in total order broadcast.



Sequential Consistency

Sequential Consistency.

- The result of any execution is the same as if all operations of the processors were executed in *some* sequential order.
- The operations of each individual processor appear in this sequence in the local program order.

Any interleaving of the operations from the different processors is possible. But all processors must see *the same* interleaving. Even if two operations from different processors (on the same or different variables) do not overlap in a global time scale, they may appear in reverse order in the *common* sequential order seen by all. See examples used for linearizability.

Sequential Consistency

Only Writes participate in total order BCs. Reads do not because:

- all consecutive operations by the same processor are ordered in that same order (no pipelining), and
- *Read* operations by different processors are independent of each other; to be ordered only with respect to the *Write* operations.

- Direct simplification of the LIN algorithm.
- Reads executed atomically. Not so for Writes.
- Suitable for Read-intensive programs.

Sequential Consistency using Local Reads

(shared var)
int: x;

(1) When the Memory Manager at P_i receives a *Read* or *Write* from application:
(1a) **case** *Read*: **return** value from local replica;
(1b) **case** *Write(x,val)*: **total_order_broadcast**$_i$(*Write(x,val)*) to all processors including itself.

(2) When the Memory Manager at P_i receives a **total_order_broadcast**$_j$(*Write*, x, *val*) from networl
(2a) **write** *val* to local replica of x;
(2b) **if** $i = j$ **then return** ack to application.

Sequential Consistency using Local Writes

(shared var)
int: x;

(1) When the Memory Manager at P_i receives a $Read(x)$ from application:
(1a) **if** $counter = 0$ **then**
(1b) **return** x
(1c) **else** Keep the *Read* pending.

(2) When the Memory Manager at P_i receives a *Write(x,val)* from application:
(2a) $counter \longleftarrow counter + 1$;
(2b) **total_order_broadcast**$_i$ the $Write(x, val)$;
(2c) **return** ack to the application.

(3) When the Memory Manager at P_i receives a **total_order_broadcast**$_j$(*Write*, x, val) from networ
(3a) **write** val to local replica of x.
(3b) **if** $i = j$ **then**
(3c) $counter \longleftarrow counter - 1$;
(3d) **if** ($counter = 0$ and any *Read*s are pending) **then**
(3e) **perform** pending responses for the *Read*s to the application.

Locally issued Writes get acked immediately. Local Reads are delayed until the locally preceding Writes have been acked. All locally issued Writes are pipelined.

Causal Consistency

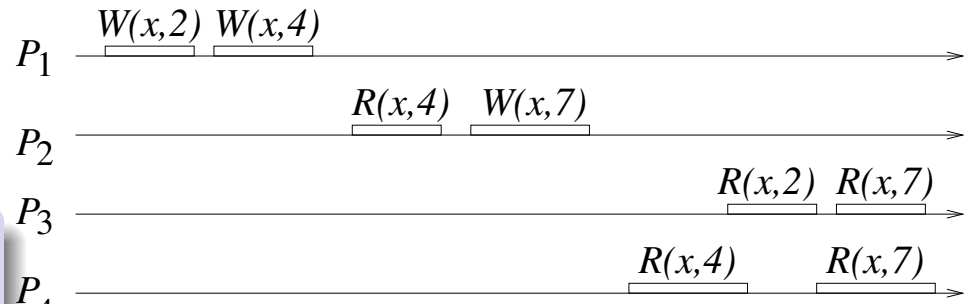
In SC, all Write ops should be seen in common order.

For *causal consistency*, only causally related Writes should be seen in common order.

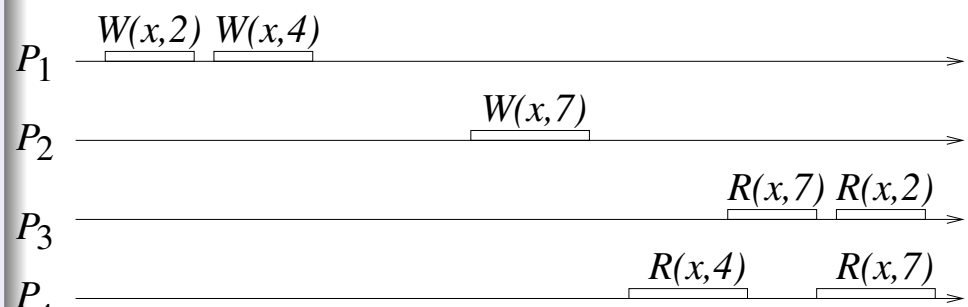
Causal relation for shared memory systems

- At a processor, local order of events is the causal order
- A Write causally precedes Read issued by another processor if the Read returns the value written by the Write.
- The transitive closure of the above two orders is the causal order

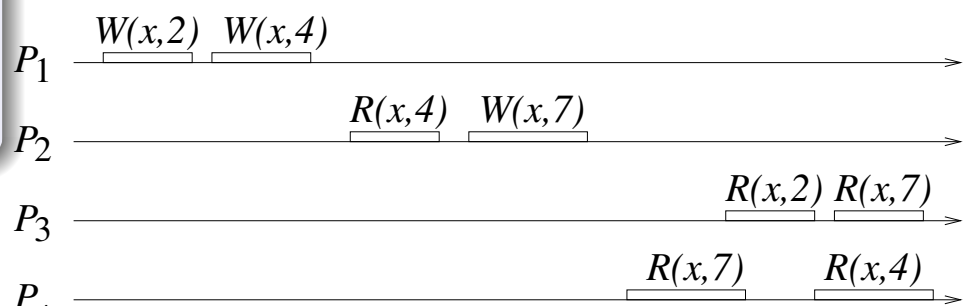
Total order broadcasts (for SC) also provide causal order in shared memory



(a)Sequentially consistent and causally consistent



(b) Causally consistent but not sequentially consistent



(c) Not causally consistent but PRAM consistent

Pipelined RAM or Processor Consistency

PRAM memory

Only Write ops issued by the same processor are seen by others in the order they were issued, but Writes from different processors may be seen by other processors in different orders.

PRAM can be implemented by FIFO broadcast? PRAM memory can exhibit counter-intuitive behavior, see below.

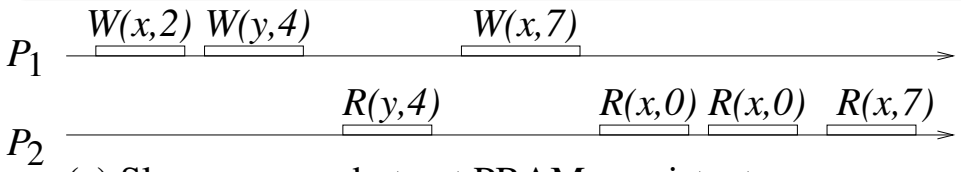
(shared variables)
int: x, y;

Process 1	Process 2
...	...
(1a) $x \longleftarrow 4$;	(2a) $y \longleftarrow 6$;
(1b) **if** $y = 0$ **then kill**(P_2).	(2b) **if** $x = 0$ **then kill**(P_1).

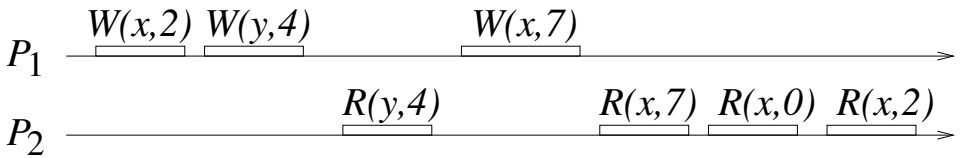
Slow Memory

Slow Memory

Only Write operations issued by the same processor and to the same memory location must be seen by others in that order.

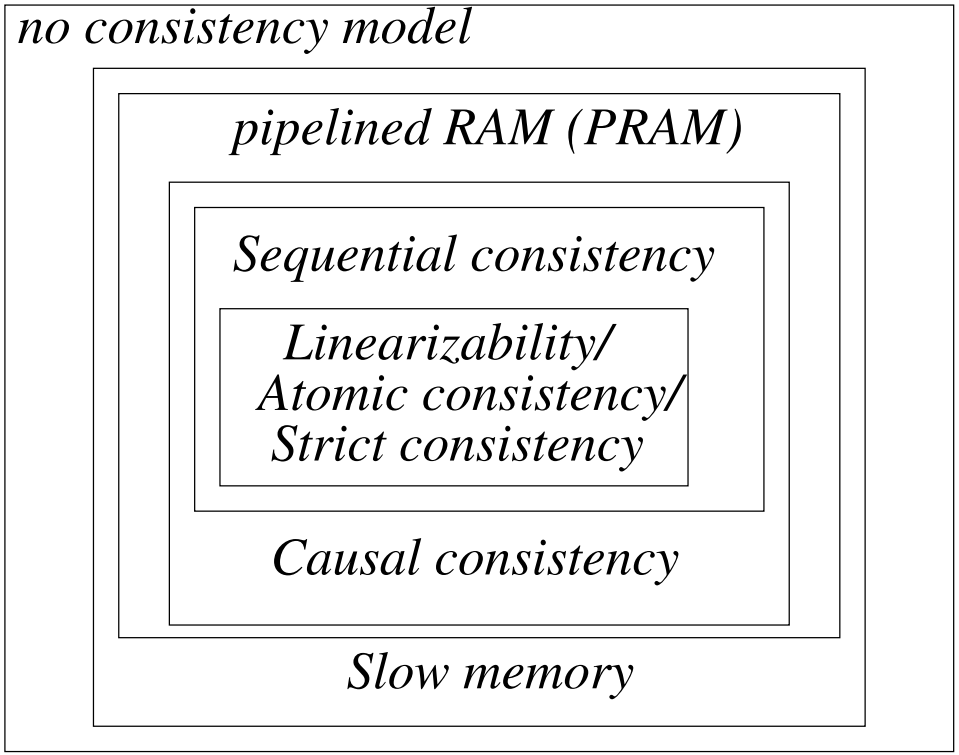


(a) Slow memory but not PRAM consistent



(b) Violation of slow memory consistency

Hierarchy of Consistency Models



Synchronization-based Consistency Models: Weak Consistency

- Consistency conditions apply only to special "synchronization" instructions, e.g., barrier synchronization
- Non-sync statements may be executed in any order by various processors.
- E.g.,weak consistency, release consistency, entry consistency

Weak consistency:

All Writes are propagated to other processes, and all Writes done elsewhere are brought locally, at a sync instruction.

- Accesses to sync variables are sequentially consistent
- Access to sync variable is not permitted unless all Writes elsewhere have completed
- No data access is allowed until all previous synchronization variable accesses have been performed

Drawback: cannot tell whether beginning access to shared variables (enter CS), or finished access to shared variables (exit CS).

Synchronization based Consistency Models: Release Consistency and Entry Consistency

Two types of synchronization Variables: *Acquire* and *Release*

Release Consistency

- *Acquire* indicates CS is to be entered. Hence all *Writes* from other processors should be locally reflected at this instruction
- *Release* indicates access to CS is being completed. Hence, all Updates made locally should be propagated to the replicas at other processors.
- *Acquire* and *Release* can be defined on a subset of the variables.
- If no CS semantics are used, then Acquire and Release act as barrier synchronization variables.
- Lazy release consistency: propagate updates on-demand, not the PRAM way.

Entry Consistency

- Each ordinary shared variable is associated with a synchronization variable (e.g., lock, barrier)
- For Acquire /Release on a synchronization variable, access to only those ordinary variables guarded by the synchronization variables is performed.

Shared Memory Mutual Exclusion: Bakery Algorithm

(shared vars)
array of boolean: $choosing[1 \ldots n]$;
array of integer: $timestamp[1 \ldots n]$;

repeat
(1) P_i executes the following for the **entry section**:
(1a) $choosing[i] \longleftarrow 1$;
(1b) $timestamp[i] \longleftarrow \max_{k \in [1 \ldots n]}(timestamp[k]) + 1$;
(1c) $choosing[i] \longleftarrow 0$;
(1d) **for** $count = 1$ **to** n **do**
(1e) **while** $choosing[count]$ **do** no-op;
(1f) **while** $timestamp[count] \neq 0$ **and** $(timestamp[count], count) < (timestamp[i], i)$ **do**
(1g) no-op.
(2) P_i executes the **critical section (CS)** after the **entry section**
(3) P_i executes the following **exit section** after the **CS**:
(3a) $timestamp[i] \longleftarrow 0$.
(4) P_i executes the **remainder section** after the **exit section**
until false;

Shared Memory Mutual Exclusion

- Mutual exclusion
  - Role of line (1e)? Wait for others' timestamp choice to stabilize ...
  - Role of line (1f)? Wait for higher priority (lex. lower timestamp) process to enter CS
- Bounded waiting: P_i can be overtaken by other processes at most once (each)
- Progress: lexicographic order is a total order; process with lowest timestamp in lines (1d)-(1g) enters CS

Space complexity: lower bound of n registers
Time complexity: (n) time for Bakery algorithm
Lamport's fast mutex algorithm takes $O(1)$ time in the absence of contention. However it compromises on bounded waiting. Uses $W(x) - R(y) - W(y) - R(x)$ sequence necessary and sufficient to check for contention, and safely enter CS

Lamport's Fast Mutual Exclusion Algorithm

(shared variables among the processes)
integer: x, y; // shared register initialized
array of boolean $b[1 \ldots n]$; // flags to indicate interest in critical section

repeat
(1) P_i $(1 \leq i \leq n)$ executes entry section:
(1a) $b[i] \longleftarrow true$;
(1b) $x \longleftarrow i$;
(1c) **if** $y \neq 0$ **then**
(1d) $b[i] \longleftarrow false$;
(1e) **await** $y = 0$;
(1f) **goto** (1a);
(1g) $y \longleftarrow i$;
(1h) **if** $x \neq i$ **then**
(1i) $b[i] \longleftarrow false$;
(1j) **for** $j = 1$ **to** N **do**
(1k) **await** $\neg b[j]$;
(1l) **if** $y \neq i$ **then**
(1m) **await** $y = 0$;
(1n) **goto** (1a);
(2) P_i $(1 \leq i \leq n)$ executes critical section:
(3) P_i $(1 \leq i \leq n)$ executes exit section:
(3a) $y \longleftarrow 0$;
(3b) $b[i] \longleftarrow false$;
forever.

Shared Memory: Fast Mutual Exclusion Algorithm

Need for a boolean vector of size n: For P_i, there needs to be a trace of its identity and that it had written to the mutex variables. Other processes need to know who (and when) leaves the CS. Hence need for a boolean array $b[1..n]$.

Process P_i	Process P_j	Process P_k	variables
	$W_j(x)$		$\langle x = j, y = 0 \rangle$
$W_i(x)$			$\langle \mathbf{x = i}, y = 0 \rangle$
$R_i(y)$			$\langle x = i, y = 0 \rangle$
	$R_j(y)$		$\langle x = i, y = 0 \rangle$
$W_i(y)$			$\langle x = i, \mathbf{y = i} \rangle$
	$W_j(y)$		$\langle x = i, y = j \rangle$
$R_i(x)$			$\langle x = i, y = j \rangle$
		$W_k(x)$	$\langle x = k, y = j \rangle$
	$R_j(x)$		$\langle x = k, y = j \rangle$

Examine all possible race conditions in algorithm code to analyze the algorithm.

Hardware Support for Mutual Exclusion

Test&Set and *Swap* are each executed atomically!!

(shared variables among the processes accessing each of the different object types)
register: *Reg* ⟵ initial value; // shared register initialized
(local variables)
integer: *old* ⟵ initial value; // value to be returned

(1) *Test&Set*(*Reg*) returns *value*:
(1a) *old* ⟵ *Reg*;
(1b) *Reg* ⟵ 1;
(1c) **return**(*old*).

(2) *Swap*(*Reg*, *new*) returns *value*:
(2a) *old* ⟵ *Reg*;
(2b) *Reg* ⟵ *new*;
(2c) **return**(*old*).

Mutual Exclusion using *Swap*

(shared variables)
register: $Reg \longleftarrow false$; // shared register initialized
(local variables)
integer: $blocked \longleftarrow 0$; // variable to be checked before entering CS

repeat
(1) P_i executes the following for the **entry section**:
(1a) $blocked \longleftarrow true$;
(1b) **repeat**
(1c) $Swap(Reg, blocked)$;
(1d) **until** $blocked = false$;
(2) P_i executes the **critical section (CS)** after the **entry section**
(3) P_i executes the following **exit section** after the **CS**:
(3a) $Reg \longleftarrow false$;
(4) P_i executes the **remainder section** after the **exit section**
until false;

Mutual Exclusion using *Test&Set*, with Bounded Waiting

(shared variables)
register: $Reg \longleftarrow false$; // shared register initialized
array of boolean: $waiting[1 \ldots n]$;
(local variables)
integer: $blocked \longleftarrow$ initial value; // value to be checked before entering CS

repeat
(1) P_i executes the following for the **entry section**:
(1a) $waiting[i] \longleftarrow true$;
(1b) $blocked \longleftarrow true$;
(1c) **while** $waiting[i]$ **and** $blocked$ **do**
(1d) $\quad blocked \longleftarrow Test\&Set(Reg)$;
(1e) $waiting[i] \longleftarrow false$;
(2) P_i executes the **critical section (CS)** after the **entry section**
(3) P_i executes the following **exit section** after the **CS**:
(3a) $next \longleftarrow (i+1) mod\ n$;
(3b) **while** $next \neq i$ **and** $waiting[next] = false$ **do**
(3c) $\quad next \longleftarrow (next+1) mod\ n$;
(3d) **if** $next = i$ **then**
(3e) $\quad Reg \longleftarrow false$;
(3f) **else** $waiting[next] \longleftarrow false$;
(4) P_i executes the **remainder section** after the **exit section**
until false;

Wait-freedom

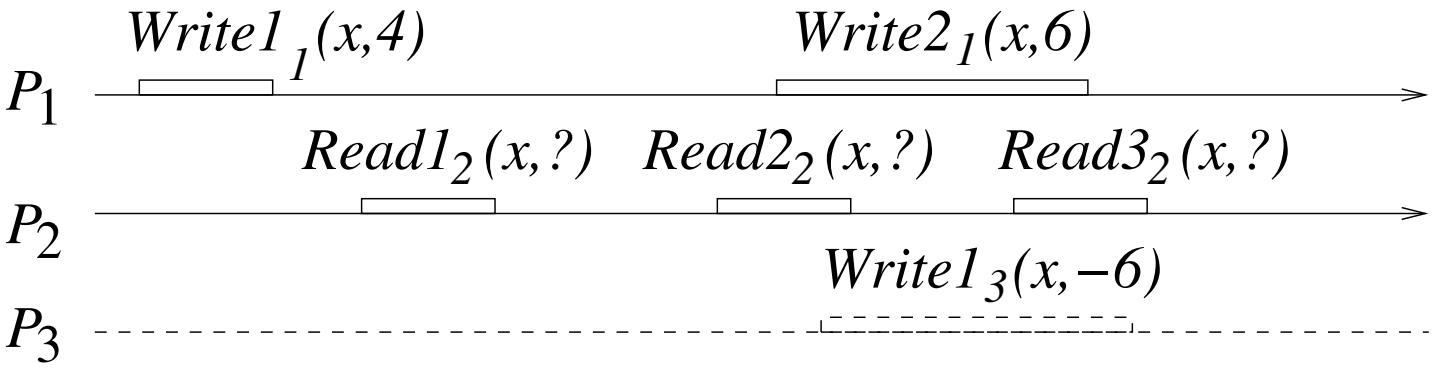
- Synchronizing asynchronous processes using busy-wait, locking, critical sections, semaphores, conditional waits etc. $\Longrightarrow$ crash/ delay of a process can prevent others from progressing.
- Wait-freedom: guarantees that any process can complete any synchronization operation in a finite number of low-level steps, irresp. of execution speed of others.
- Wait-free implementation of a concurrent object $\Longrightarrow$ any process can complete on operation on it in a finite number of steps, irrespective of whether others crash or are slow.
- Not all synchronization problems have wait-free solutions, e.g., producer-consumer problem.
- An $n-1$-resilient system is wait-free.

Register Hierarchy and Wait-freedom

- During concurrent access, behavior of register is unpredictable
- For a systematic study, analyze most elementary register, and build complex ones based on the elementary register
- *Assume* a single reader and a single writer

Safe register

A Read that does not overlap with a Write returns the most recent value written to that register. A Read that overlaps with a Write returns any one of the possible values that the register could ever contain.



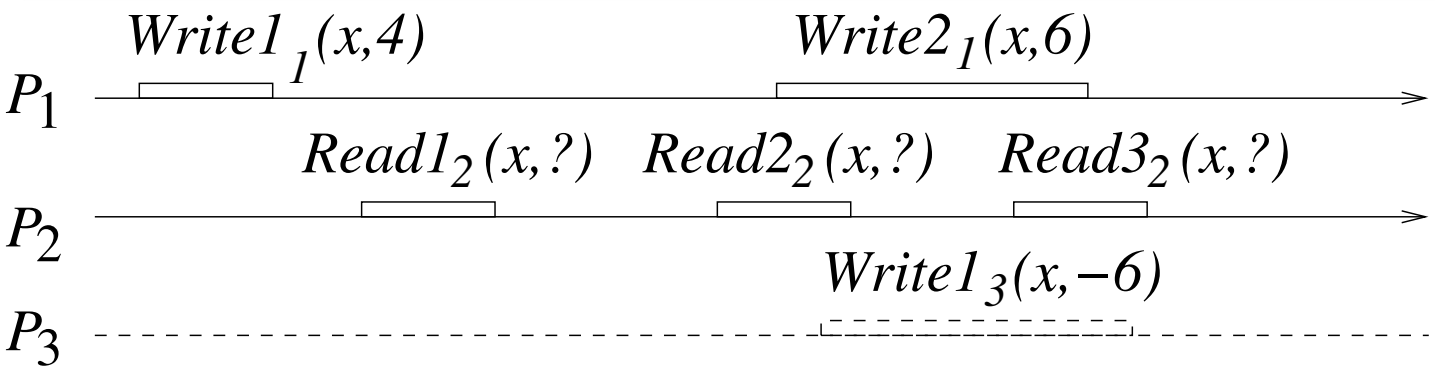
Register Hierarchy and Wait-freedom (2)

Regular register

Safe register + if a Read overlaps with a Write, value returned is the value before the Write operation, or the value written by the Write.

Atomic register

Regular register + linearizable to a sequential register

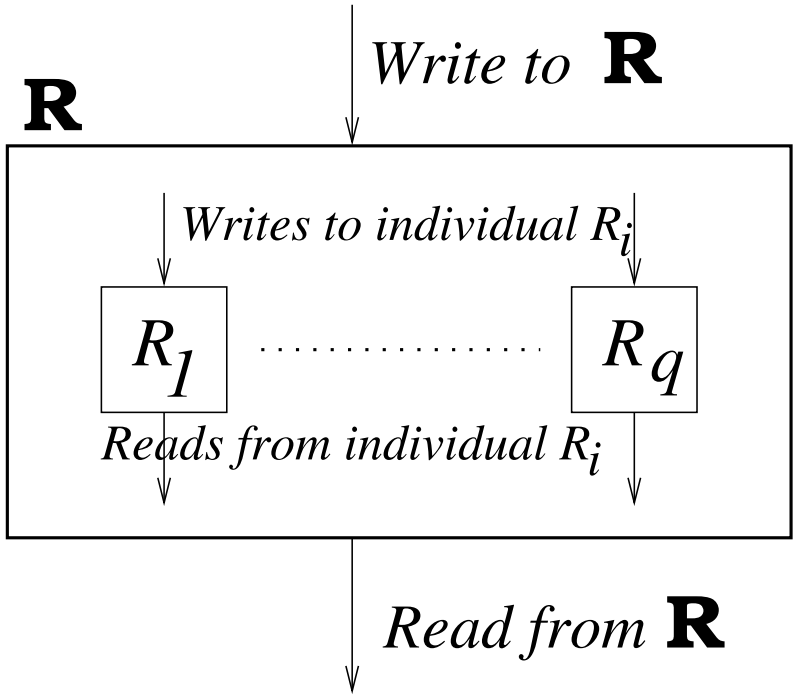


Classification of Registers and Register Constructions

Table 12.2. Classification by type, value, write-access, read-access

Type	Value	Writing	Reading
safe regular atomic	binary integer	Single-Writer Multi-Writer	Single-Reader Multi-Reader

$R_1 \ldots R_q$ are weaker registers that are used to construct stronger register types **R**. Total of n processes assumed.



Construction 1: SRSW Safe to MRSW Safe

- Single Writer P_0, Readers $P_1 \ldots P_n$. Here, $q = n$.
- Registers could be binary or integer-valued
- Space complexity: n times that of a single register
- Time complexity: n steps

(shared variables)
SRSW safe registers $R_1 \ldots R_n \longleftarrow 0$; // R_i is readable by P_i, writable by P_0

(1) $Write(R, val)$ executed by single writer P_0
(1a) **for all** $i \in \{1 \ldots n\}$ **do**
(1b) $R_i \longleftarrow val$.

(2) $Read_i(R, val)$ executed by reader P_i, $1 \leq i \leq n$
(2a) $val \longleftarrow R_i$
(2b) **return**(val).

Construction 2: SRSW Regular to MRSW Regular is similar.

Construction 3: Bool MRSW Safe to Integer MRSW Safe

- For integer of size m, $log(m)$ boolean registers needed.
- P_0 writes value in binary notation; each of the n readers reads $log(m)$ registers
- Space complexity $log(m)$. Time complexity $log(m)$.

(shared variables)
boolean MRSW safe registers $R_1 \ldots R_{log(m)} \longleftarrow 0$; // R_i readable by all, writable by P_0.

(local variable)
array of boolean: $Val[1 \ldots log(m)]$;

(1) $Write(R, Val[1 \ldots log\ m])$ executed by single writer P_0
(1a) **for all** $i \in \{1 \ldots log(m)\}$ **do**
(1b) $R_i \longleftarrow Val[i]$.

(2) $Read_i(R, Val[1 \ldots log(m)])$ executed by reader P_i, $1 \leq i \leq n$
(2a) **for all** $j \in \{1 \ldots log\ m\}$ **do** $Val[j] \longleftarrow R_j$
(2b) **return**$(Val[1 \ldots log(m)])$.

Construction 4: Bool MRSW Safe to Bool MRSW Regular

- $q = 1$. P_0 writes register R_1. The n readers all read R_1.
- If value is α before; Write is to write α, then a concurrent *Read* may get either α or $1 - \alpha$. How to convert to regular register?
- Writer locally tracks the previous value it wrote. Writer writes new value only if it differs from previously written value.
- Space and time complexity $O(1)$.
- Cannot be used to construct binary SRSW atomic register.

(shared variables)
boolean MRSW safe register: $R' \longleftarrow 0$; // R' is readable by all, writable by P_0.

(local variables)
boolean local to writer P_0: *previous* $\longleftarrow 0$;

(1) *Write*(R, val) executed by single writer P_0
(1a) **if** *previous* $\neq$ *val* **then**
(1b) $R' \longleftarrow val$;
(1c) *previous* $\longleftarrow$ *val*.

(2) *Read*(R, val) process P_i, $1 \leq i \leq n$
(2a) *val* $\longleftarrow R'$;
(2b) **return**(*val*).

Construction 5: Boolean MRSW Regular to Integer MRSW Regular

- $q = m$, the largest integer. The integer is stored in *unary* notation.
- P_0 is writer. P_1 to P_n are readers, each can read all m registers.
- Readers scan L to R looking for first "1"; Writer writes "1" in R_{val} and then zeros out entries R to L.
- Complexity: m binary registers, $O(m)$ time.

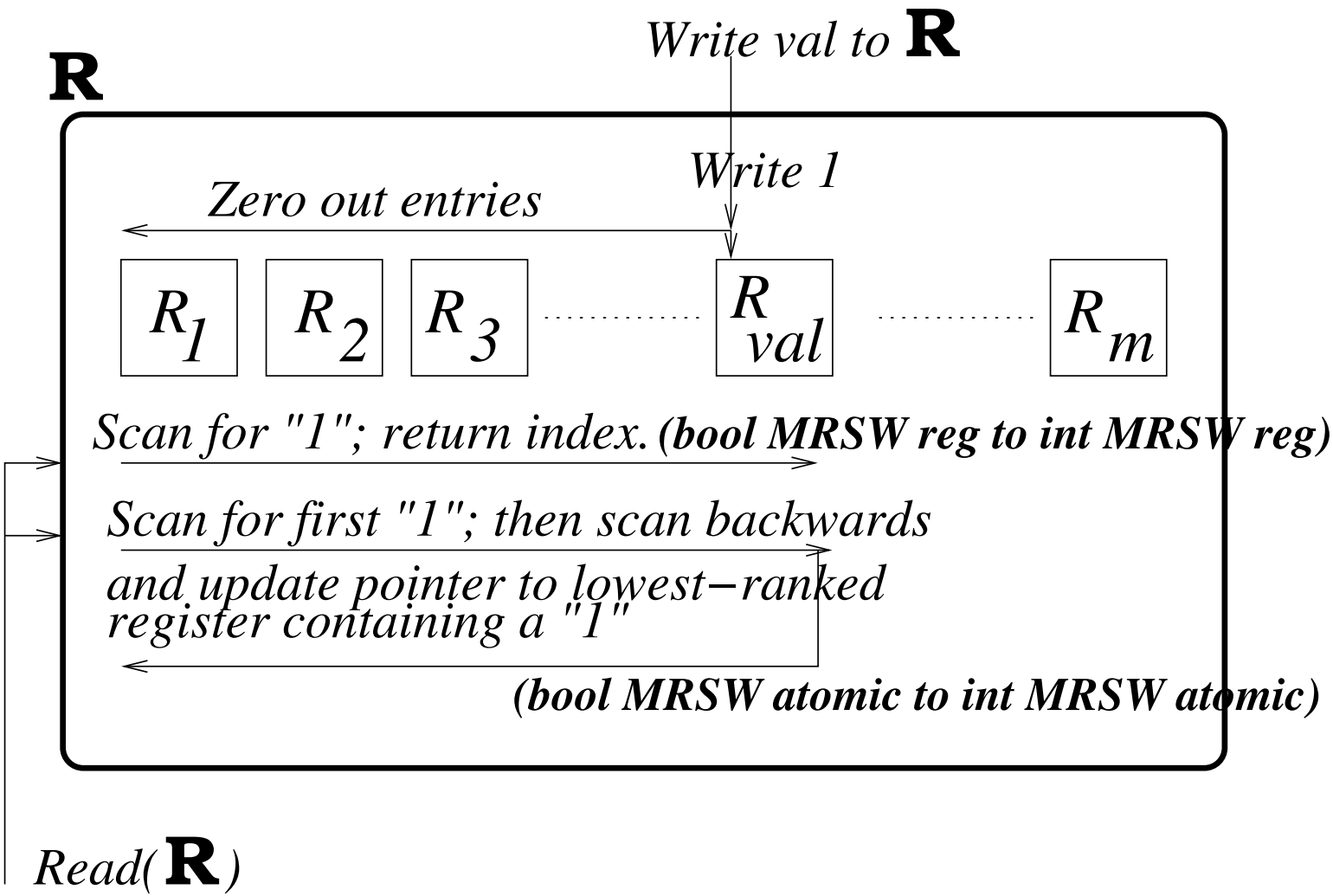
Construction 5: Algorithm

(shared variables)
boolean MRSW regular registers $R_1 \ldots R_{m-1} \longleftarrow 0$; $R_m \longleftarrow 1$;
// R_i readable by all, writable by P_0.
(local variables)
integer: *count*;

(1) $\underline{Write(R, val)}$ executed by writer P_0
(1a) $R_{val} \longleftarrow 1$;
(1b) **for** $count = val - 1$ **down to** 1 **do**
(1c) $\quad R_{count} \longleftarrow 0$.

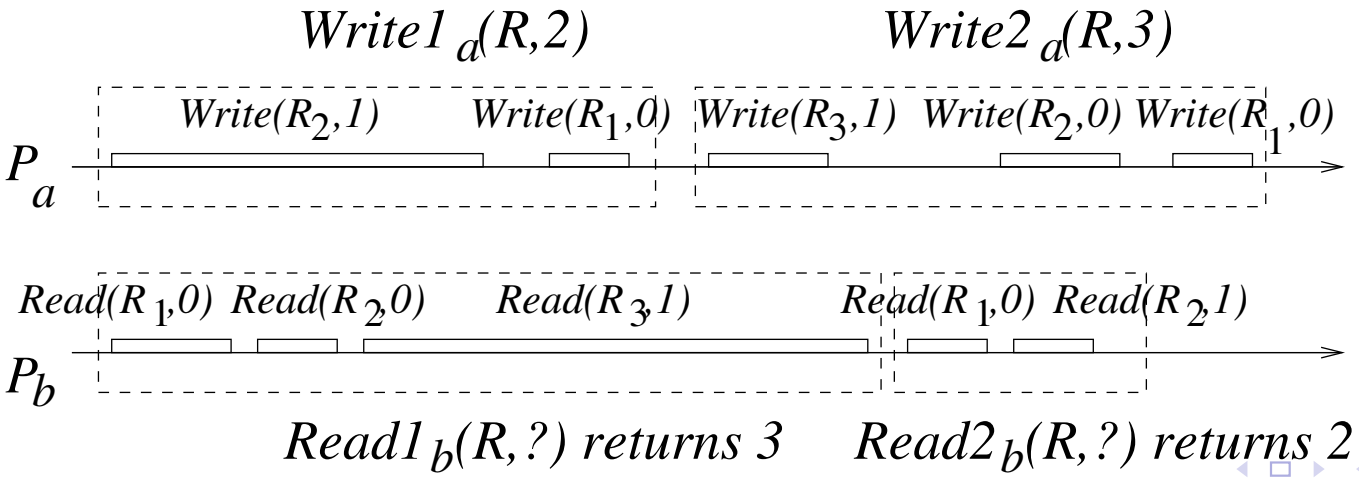
(2) $\underline{Read_i(R, val)}$ executed by P_i, $1 \leq i \leq n$
(2a) $count = 1$;
(2b) **while** $R_{count} = 0$ **do**
(2c) $\quad count \longleftarrow count + 1$;
(2d) $val \longleftarrow count$;
(2e) **return**(val).

Illustrating Constructions 5 and 6:



Construction 6: Boolean MRSW regular to integer-valued MRSW atomic

- Construction 5 cannot be used to construct a MRSW atomic register because of a possible inversion of values while reading.
- In example below, $Read2_b$ returns 2 after the earlier $Read1_b$ returned 3, and the value 3 is older than value 2.
- Such an inversion of read values is permitted by regular register but not by an atomic register.
- One solution is to require Reader to also scan R to L after it finds "1" in some location. In the backward scan, the "smallest" value is returned to the Read.
- Space complexity: m binary registers, Time complexity $O(m)$



Construction 6: Algorithm

(shared variables)
boolean MRSW regular registers $R_1 \ldots R_{m-1} \longleftarrow 0$; $R_m \longleftarrow 1$.
// R_i readable by all; writable by P_0.

(local variables)
integer: *count*, *temp*;

(1) $\underline{Write(R, val) \text{ executed by } P_0}$
(1a) $R_{val} \longleftarrow 1$;
(1b) **for** $count = val - 1$ **down to** 1 **do**
(1c) $\quad R_{count} \longleftarrow 0$.

(2) $\underline{Read_i(R, val) \text{ executed by } P_i,\ 1 \leq i \leq n}$
(2a) $count \longleftarrow 1$;
(2b) **while** $R_{count} = 0$ **do**
(2c) $\quad count \longleftarrow count + 1$;
(2d) $val \longleftarrow count$;
(2e) **for** $temp = count$ **down to** 1 **do**
(2f) $\quad$ **if** $R_{temp} = 1$ **then**
(2g) $\quad\quad val \longleftarrow temp$;
(2h) **return**(val).

Construction 7: Integer MRSW Atomic to Integer MRMW Atomic

- $q = n$, each MRSW register R_i is readable by all, but writable by P_i
- With concurrent updates to various MRSW registers, a global linearization order needs to be established, and the Read ops should recognize it.
- Idea: similar to the Bakery algorithm for mutex.
- Each register has 2 fields: $R.data$ and $R.tag$, where $tag = \langle pid, seqno \rangle$.
- The Collect is invoked by readers and the Writers The Collect reads all registers in no particular order.
- A Write gets a tag that is lexicographically greater then the tags read by it.
- The Writes (on different registers) get totally ordered (linearized) using the tag
- A Read returns data corresp. lexicographically most recent Write
- A Read gets ordered after the Write whose value is returned to it.

Construction 7: Integer MRSW Atomic to Integer MRMW Atomic

(shared variables)
MRSW atomic registers of type $\langle data, tag \rangle$, where $tag = \langle seq\_no, pid \rangle$: $R_1 \ldots R_n$;

(local variables)
array of MRSW atomic registers of type $\langle data, tag \rangle$, where $tag = \langle seq\_no, pid \rangle$: $Reg\_Array[1 \ldots n]$;
integer: seq_no, j, k;

(1) $Write_i(R, val)$ executed by P_i, $1 \leq i \leq n$
(1a) $Reg\_Array \longleftarrow Collect(R_1, \ldots, R_n)$;
(1b) $seq\_no \longleftarrow \max(Reg\_Array[1].tag.seq\_no, \ldots Reg\_Array[n].tag.seq\_no) + 1$;
(1c) $R_i \longleftarrow (val, \langle seq\_no, i \rangle)$.

(2) $Read_i(R, val)$ executed by P_i, $1 \leq i \leq n$
(2a) $Reg\_Array \longleftarrow Collect(R_1, \ldots, R_n)$;
(2b) **identify** j such that for all $k \neq j$, $Reg\_Array[j].tag > Reg\_Array[k].tag$;
(2c) $val \longleftarrow Reg\_Array[j].data$;
(2d) **return**(val).

(3) $Collect(R_1, \ldots, R_n)$ invoked by *Read* and *Write* routines
(3a) **for** $j = 1$ **to** n **do**
(3b) $\quad Reg\_Array[j] \longleftarrow R_j$;
(3c) **return**(Reg_Array).

Construction 8: Integer SRSW Atomic to Integer MRSW Atomic

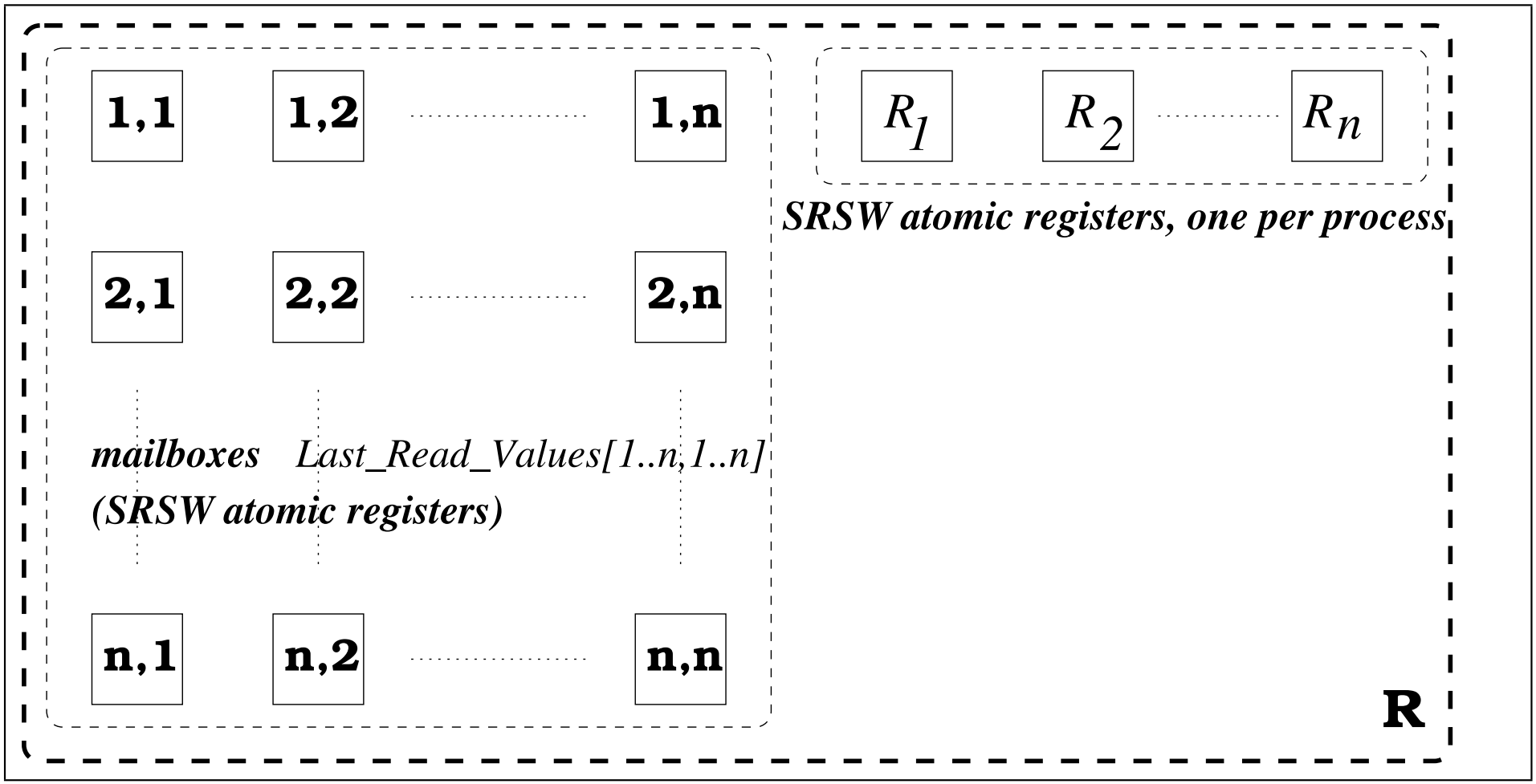
Naive solution: $q = n$. n replicas of R and the Writer writes to all replicas.

- Fails! $Read_i$ and $Read_j$ are serial, and both concurrent with Write. $Read_i$ could get the newer value and $Read_j$ could get the older value because this execution is non-serializable.

Each reader also needs to know what value was last read by each other reader!

- Due to SRSW registers, construction needs n^2 mailboxes for all reader process pairs
- Reader reads value set aside for it by other readers, as well as the value set aside for it by the writer (n such mailboxes; from Writer to each reader.
- $Last\_Read[0..n]$ is local array.
- $Last\_Read\_Values[1..n, 1..n]$ are the reader-to-reader mailboxes.

Construction 8: Data Structure



Construction 8: Algorithm

(shared variables)
SRSW atomic register of type $\langle data, seq\_no \rangle$, **where** $data, seq\_no$ **are integers**: $R_1 \ldots R_n \longleftarrow \overline{\langle 0, 0 \rangle}$;
SRSW atomic register array of type $\langle data, seq\_no \rangle$, **where** $data, seq\_no$ **are integers**: $Last\_Read\_Values[1 \ldots n, 1 \ldots n] \longleftarrow \overline{\langle 0, 0 \rangle}$;

(local variables)
array of $\langle data, seq\_no \rangle$: $Last\_Read[0 \ldots n]$;
integer: $seq, count$;

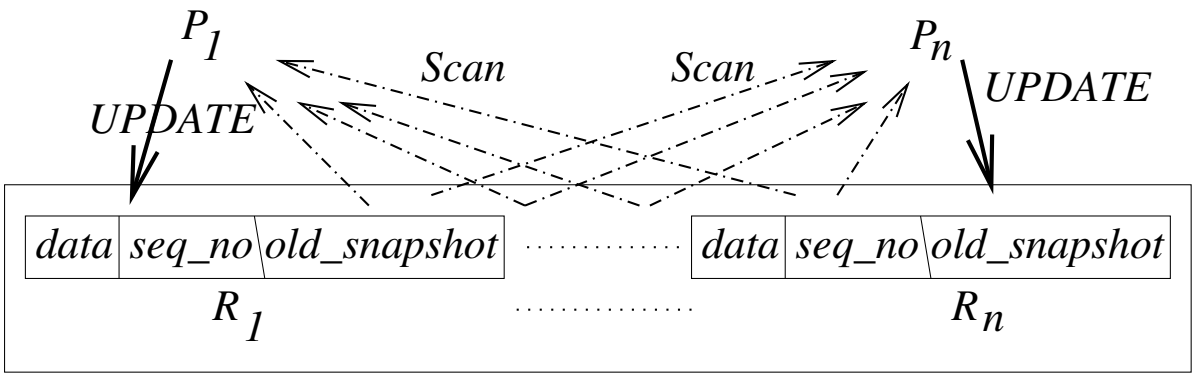
(1) $Write(R, val)$ executed by writer P_0
(1a) $seq \longleftarrow seq + 1$;
(1b) **for** $count = 1$ **to** n **do**
(1c) $\quad R_{count} \longleftarrow \langle val, seq \rangle$. // write to each SRSW register

(2) $Read_i(R, val)$ executed by P_i, $1 \leq i \leq n$
(2a) $\langle Last\_Read[0].data, Last\_Read[0].seq\_no \rangle \longleftarrow R_i$; // $Last\_Read[0]$ stores value of R_i
(2b) **for** $count = 1$ **to** n **do** // read into $Last\_Read[count]$, the latest values stored for P_i by P_{count}
(2c) $\quad \langle Last\_Read[count].data, Last\_Read[count].seq\_no \rangle \longleftarrow$
$\quad\quad \langle Last\_Read\_Values[count, i].data, Last\_Read\_Values[count, i].seq\_no \rangle$;
(2d) **identify** j such that for all $k \neq j$, $Last\_Read[j].seq\_no \geq Last\_Read[k].seq\_no$;
(2e) **for** $count = 1$ **to** n **do**
(2f) $\quad \langle Last\_Read\_Values[i, count].data, Last\_Read\_Values[i, count].seq\_no \rangle \longleftarrow$
$\quad\quad \langle Last\_Read[j].data, Last\_Read[j].seq\_no \rangle$;
(2g) $val \longleftarrow Last\_Read[j].data$;
(2h) **return**(val).

Wait-free Atomic Snapshots of Shared Objects using Atomic MRSW objects

Given a set of SWMR atomic registers $R_1 \ldots R_n$, where R_i can be written only by P_i and can be read by all processes, and which together form a compound high-level object, devise a *wait-free* algorithm to observe the state of the object at some instant in time. The following actions are allowed on this high-level object.

- $Scan_i$: This action invoked by P_i returns the atomic snapshot which is an instantaneous view of the object $(R_1, \ldots, R_n)$ at some instant between the invocation and termination of the *Scan*.
- $Update_i(val)$: This action invoked by P_i writes the data *val* to register R_i.



snapshot object composed of n MRSW atomic registers

Wait-free Atomic Snapshot of MRSW Object

- To get an instantaneous snapshot, double-collect (2 scans) may always fail because Updater may intervene.
- Updater is inherently more powerful than Scanner
- To have the same power as Scanners, Updater is required to first do double-collect and then its update action. Additionally, the Updater also writes the snapshot it collected, in the Register.
- If a scanner's double collect fails (because some Updater has done an Update in between), the scanner can "borrow" the snapshot recorded by the Updater in its register.
- $changed[k]$ tracks the number of times P_k spoils P_i's double-collect.
- $changed[k] = 2$ implies the second time the Updater spoiled the scanner's double-collect, the update was initiated after the Scanner began its task. Hence the Updater's recorded snapshot is within the time duration of the scanner's trails.
- Scanner can borrow Updater's recorded snapshot.
- Updater's recorded snapshot may also be borrowed. This recursive argument holds at most $n - 1$ times; the nth time, some double-collect must be successful.
- Scans and Updates get linearized.
- Local and shared space complexity both are $O(n^2)$. Time complexity $O(n^2)$

Wait-free Atomic Snapshot of MRSW Object: Algorithm

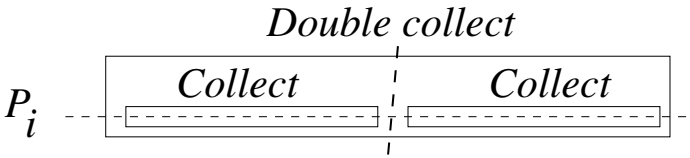
(shared variables)
MRSW atomic register of type $\langle data, seq\_no, old\_snapshot \rangle$, **where** $data, seq\_no$ **are of type integer, and** $old\_snapshot[1 \ldots n]$ **is array of integer**: $R_1 \ldots R_n$;

(local variables)
array of int: $changed[1 \ldots n]$;
array of type $\langle data, seq\_no, old\_snapshot \rangle$: $v1[1 \ldots n], v2[1 \ldots n], v[1 \ldots n]$;

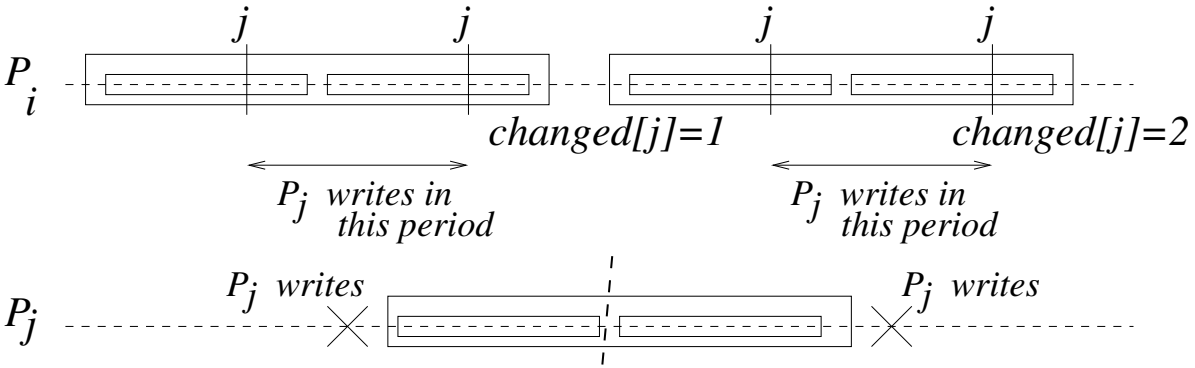
(1) $\underline{Update_i(x)}$
(1a) $v[1 \ldots n] \longleftarrow Scan_i$;
(1b) $R_i \longleftarrow (x, R_i.seq\_no + 1, v[1 \ldots n])$.

(2) $\underline{Scan_i}$
(2a) **for** $count = 1$ **to** n **do**
(2b) $\quad changed[count] \longleftarrow 0$;
(2c) **while** *true* **do**
(2d) $\quad v1[1 \ldots n] \longleftarrow collect()$;
(2e) $\quad v2[1 \ldots n] \longleftarrow collect()$;
(2f) $\quad$ **if** $(\forall k, 1 \leq k \leq n)(v1[k].seq\_no = v2[k].seq\_no)$ **then**
(2g) $\quad\quad$ **return**$(v2[1].data, \ldots, v2[n].data)$;
(2h) $\quad$ **else**
(2i) $\quad\quad$ **for** $k = 1$ **to** n **do**
(2j) $\quad\quad\quad$ **if** $v1[k].seq\_no \neq v2[k].seq\_no$ **then**
(2k) $\quad\quad\quad\quad changed[k] \longleftarrow changed[k] + 1$;
(2l) $\quad\quad\quad\quad$ **if** $changed[k] = 2$ **then**
(2m) $\quad\quad\quad\quad\quad$ **return**$(v2[k].old\_snapshot)$.

Wait-free Atomic Snapshots of Shared Objects using Atomic MRSW Objects



(a) Double collect sees identical values in both Collects



(b) P_j's Double−Collect nested within P_i's SCAN. The Double−Col is successful, or P_j borrowed snapshot from P_k's Double−Collect nested within P_j's SCAN. And so on recursively, up to n times.