

Lecture: 13

Distributed Shared Memory



Rajiv Misra

Dept. of Computer Science &
Engineering

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Preface

Recap of Previous Lecture:

- In the previous lecture, we have discussed the **basic fundamentals of 'Distributed Deadlock Detection'** and
- Different classes of algorithms** such as path-pushing, edge-chasing, diffusion computation, and global state detection.

Content of this Lecture:

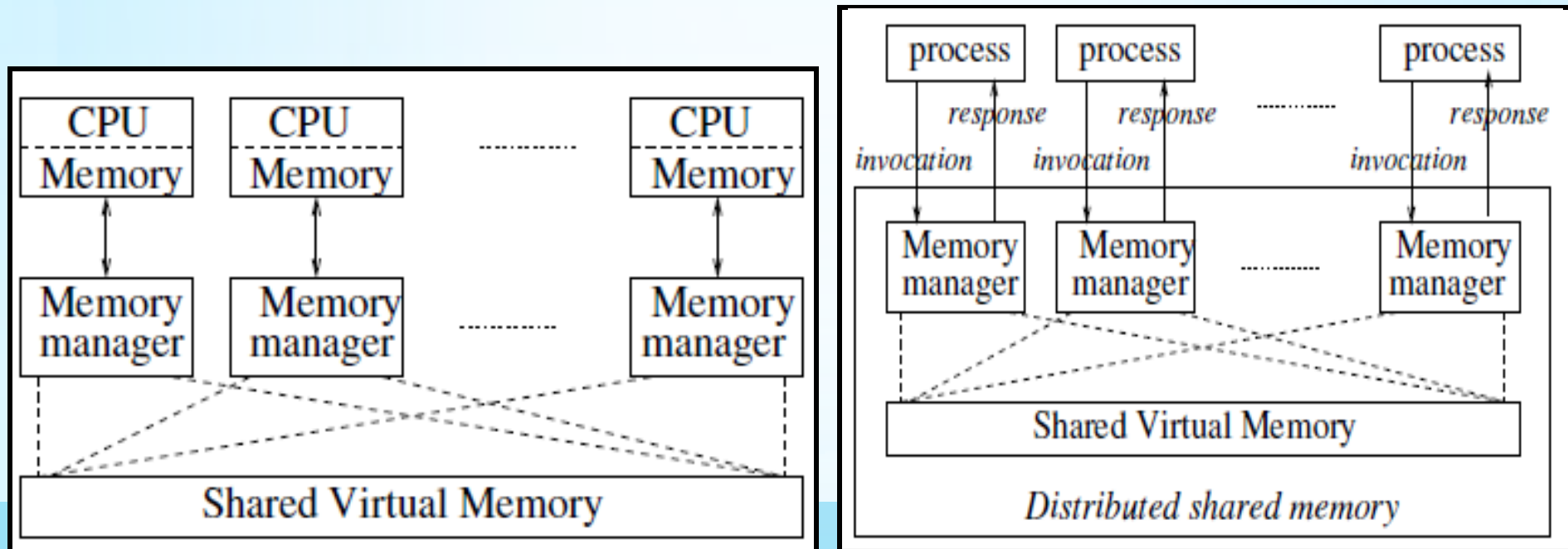
- In this lecture, we will discuss about the concept of **'Distributed Shared Memory (DSM)'** as well as provide different ways to classify DSMs and **'consistency models'** of DSM and
- Also Discuss **'Lamport's Bakery Algorithm'** for Shared Memory Mutual Exclusion.

Introduction

- **Distributed shared memory (DSM)** is an abstraction provided to the programmer of a distributed system. It gives the impression of a single monolithic memory, as in traditional von Neumann architecture.
- Programmers access the data across the network using only *read* and *write* primitives, as they would in a uniprocessor system.
- Programmers do not have to deal with *send* and *receive* communication primitives and the ensuing complexity of dealing explicitly with synchronization and consistency in the message passing model.

Distributed Shared Memory Abstractions

- Communicate with **Read/Write** operations 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!
- $\Rightarrow$ weaker consistency models (weaker than von Neumann) required



Advantages of DSM

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

Disadvantages of DSM

1. Programmers **need to understand consistency models**, to write correct programs
2. DSM implementations **use asynchronous message-passing**, and hence cannot be more efficient than message-passing implementations
3. By yielding **control to DSM manager software**, programmers cannot use their own message-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, no. of messages** to implement the semantics of concurrent access
- Data is **replicated or cached**
- **Remote access** by Hardware or Software
- **Caching/replication** controlled by Hardware or Software
- DSM **controlled** by memory management Software, 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 Consistency Models

Memory Coherence

- **Memory coherence** is the ability of the system to execute memory operations correctly.
- Assume **n processes** and **s_i memory operations** by per **process P_i**
- Also assume that all the operations issued by a process are executed sequentially (that is, pipelining is disallowed), as shown in figure 13.3.

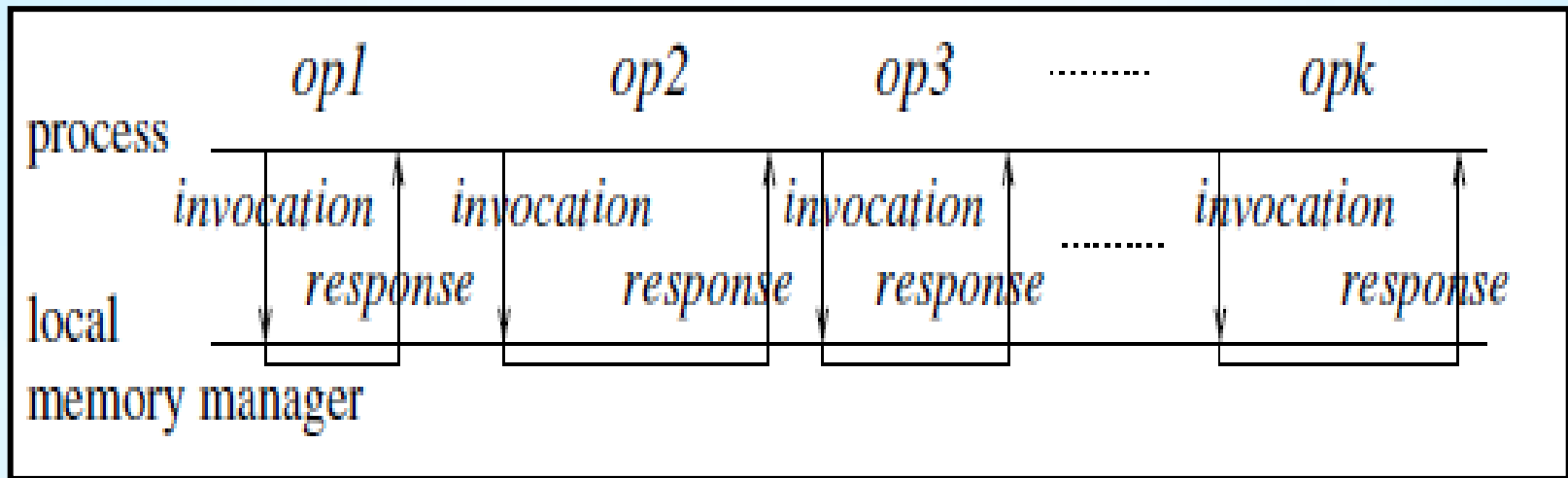


Figure 13.3 Sequential invocations and responses in a DSM system, without any pipelining.

Memory Coherence

- Observe that there are a total of

$(s_1 + s_2 + \dots s_n)! / (s_1! s_2! \dots s_n!)$ 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

Consistency Models

Author	Year	Consistency Model
L. Lamport	1979	Sequential consistency model
L. Lamport	1986	Linearizability model
Lipton and Sandberg	1988	PRAM model
M. Herlihy and J. Wing	1990	Linearizability model
P. Hutto and M. Ahamad	1990	Slow memory model
M. Dubois and C. Scheurich	1990	Weak consistency model
K. Gharachorloo et al.	1990	Release consistency model
H. Bal et al.	1992	Sequential consistency model
B. Bershad et al.	1993	Entry consistency model
H. Attiya and J. Welch	1994	Linearizability and Sequential consistency
M. Ahamad et al.	1995	Causal consistency model

1. Strict Consistency/Linearizability/Atomic Consistency

Strict consistency (SC)

1. Any **Read** to a location (variable) is required to return the value written by the most recent **Write** to that location (variable) as per a global time reference.
2. All operations appear to be executed atomically and sequentially.
3. All processors see the same ordering of events, which is equivalent to the global-time occurrence of non-overlapping events.

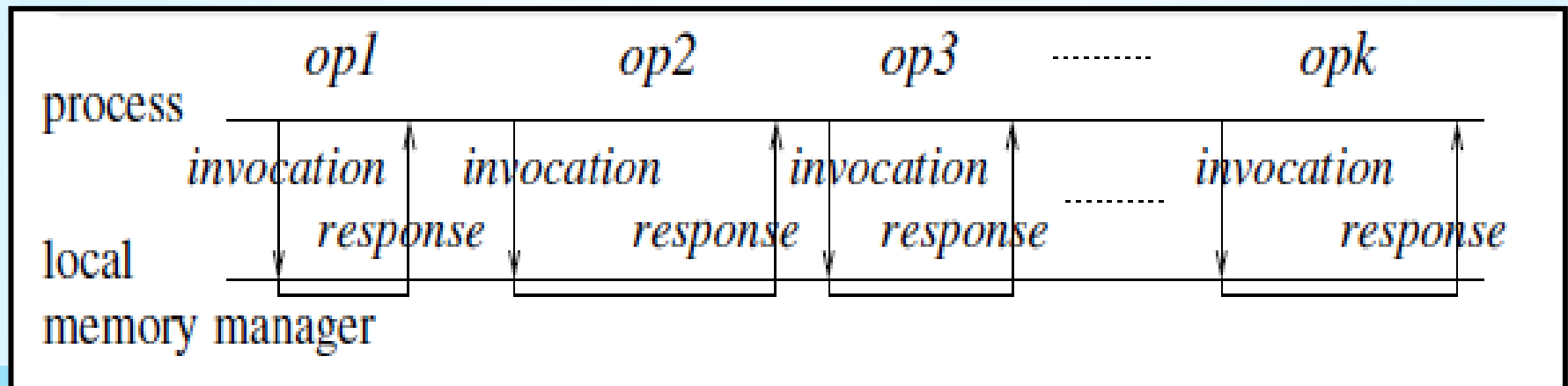


Fig 13.4 Sequential invocations and responses to each Read or Write operation.

Conditions of Linearizability

- More formally, a sequence Seq of invocations and responses is **linearizable (LIN)** if there is a permutation Seq' of adjacent pairs of corresponding $\langle \text{invoc}, \text{resp} \rangle$ events satisfying:
 - C1.** For every variable v , the projection of Seq' on v , denoted Seq'_v , is such that **every Read (adjacent $\langle \text{invoc}, \text{resp} \rangle$ event pair) returns the most recent Write (adjacent $\langle \text{invoc}, \text{resp} \rangle$ event pair) that immediately preceded it.**
 - C2.** If the response $op1(\text{resp})$ of operation $op1$ occurred before the invocation $op2(\text{invoc})$ of operation $op2$ in Seq , then $op1$ (adjacent $\langle \text{invoc}, \text{resp} \rangle$ event pair) occurs before $op2$ (adjacent $\langle \text{invoc}, \text{resp} \rangle$ event pair) in Seq' .
- **Condition 1** specifies that every processor sees a common order Seq' of events, and that in this order, the semantics is that **each Read returns the most recent completed Write value. Condition 2** specifies that the **common order Seq' must satisfy the global time order of events, viz.,** the order of non-overlapping operations in Seq must be preserved in Seq' .

Strict Consistency/Linearizability: Examples

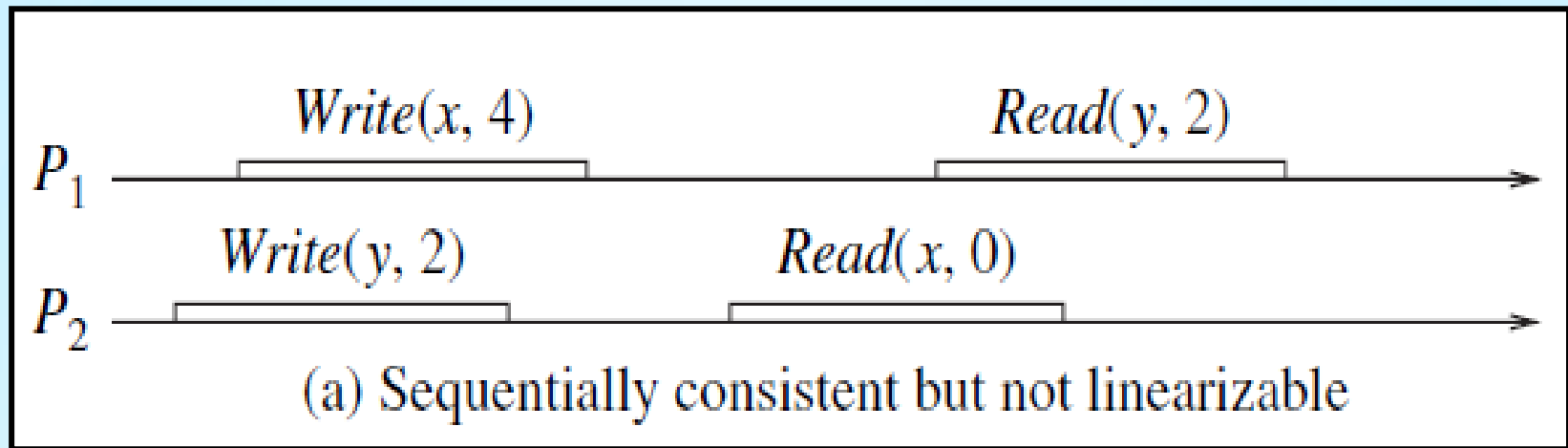


Figure 13.5(a) The execution is not linearizable because although the Read by P_2 begins after $Write(x, 4)$, the Read returns the value that existed before the Write. Hence, a permutation Seq' satisfying the condition 2 above on global time order does not exist.

Contd...

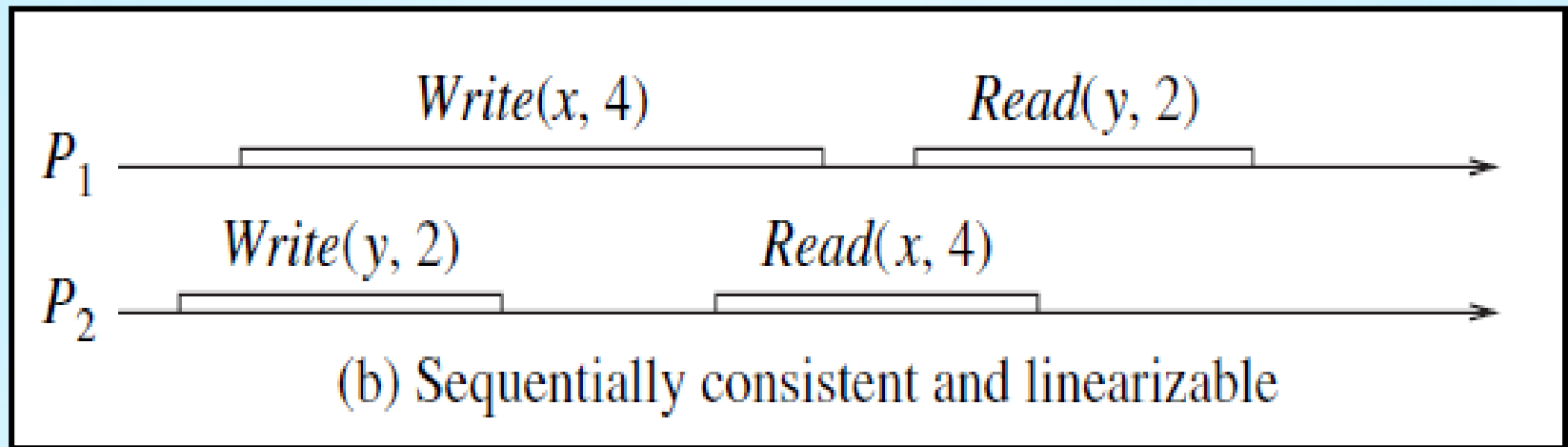


Figure 13.5(b) The execution is linearizable. The global order of operations (corresponding to $\langle \text{invocation}, \text{response} \rangle$ pairs in Seq'), consistent with the real-time occurrence, is: $Write(y, 2)$, $Write(x, 4)$, $Read(x, 4)$, $Read(y, 2)$. This permutation Seq' satisfies conditions 1 and 2.

Contd...

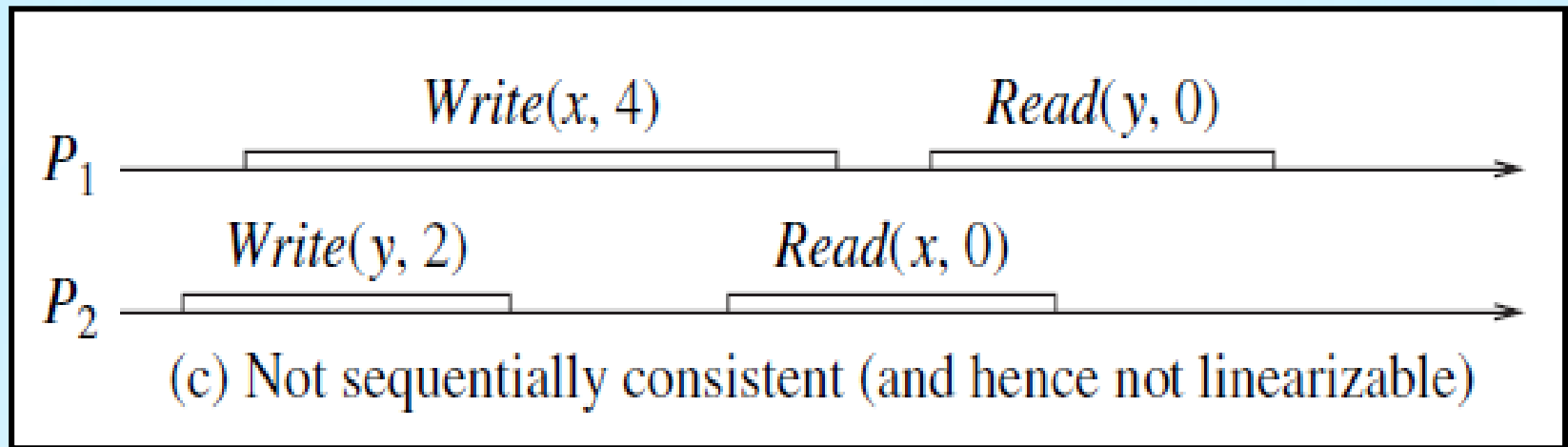


Figure 13.5(c) The execution is not linearizable. The two dependencies: $Read(x, 0)$ before $Write(x, 4)$, and $Read(y, 0)$ before $Write(y, 2)$ cannot both be satisfied in a global order while satisfying the local order of operations at each processor. Hence, there does not exist any permutation Seq' satisfying conditions 1 and 2.

Linearizability: 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**.

Algorithm 13.1 Implementing linearizability (LIN) using total order broadcasts .

Code shown is for P_i , $1 \leq i \leq n$.

Linearizability: Implementation (2)

- For a Read operation, when the memory managers system wide receive the total order broadcast, they do not perform any action.

- **Why is the broadcast then necessary?**

- The reason is this. If Read operations do not participate in the total order broadcasts, they do not get totally ordered with respect to the Write operations as well as with respect to the other Read operations.

- This can lead to a violation of linearizability as shown in **Figure 13.6**

Linearizability: Implementation (2)

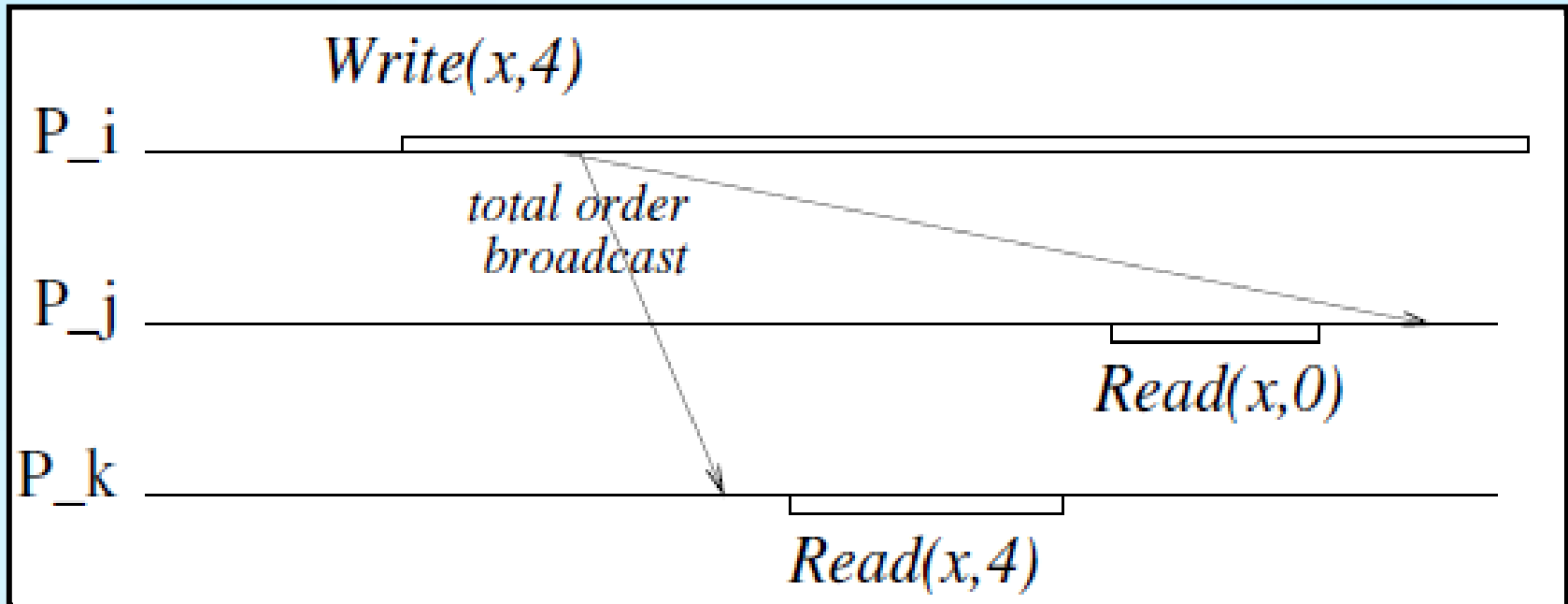


Figure 13.6 A violation of linearizability (LIN) if Read operations do not participate in the total order broadcast

The **Read** by P_k returns the value written by P_i . The later *Read* by P_j **returns the initial** value of 0. As per the global time ordering requirement of linearizability, the **Read** by P_j that occurs after the **Read** by P_k must also return the value 4. However, that is not the case in this example, wherein the **Read operations** do not participate in the total order broadcast.

2. Sequential Consistency

Sequential consistency is specified as follows:

- 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 the processors. See examples used for linearizability.

Contd...

Only Writes participate in total order broadcasts. 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 linearizability (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_j**(**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 network

(2a) **write** val to local replica of x ;

(2b) **if** $i = j$ **then return** acknowledgement to application.

Algorithm 13.2 Implementing sequential consistency (SC) using local Read operations . Code shown is for P_i , $1 \leq i \leq n$.

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) $counter \leftarrow counter + 1$;

(2b) **total_order_broadcast_i** the $Write(x, val)$;

(2c) **return** acknowledgement to the application.

(3) When the Memory Manager at P_i receives a **total order broadcast_j** ($Write, x, val$) from network

(3a) **write** val to local replica of x .

(3b) **if** $i = j$ **then**

(3c) $counter \leftarrow counter - 1$;

(3d) **if** ($counter = 0$ and any $Reads$ are pending) **then**

(3e) **perform** pending responses for the $Reads$ to the application.

Algorithm 13.3 Locally issued Writes get acked immediately. Local Reads are delayed until the locally preceding Writes have been acked. All locally issued Writes are pipelined.

3. 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

Examples

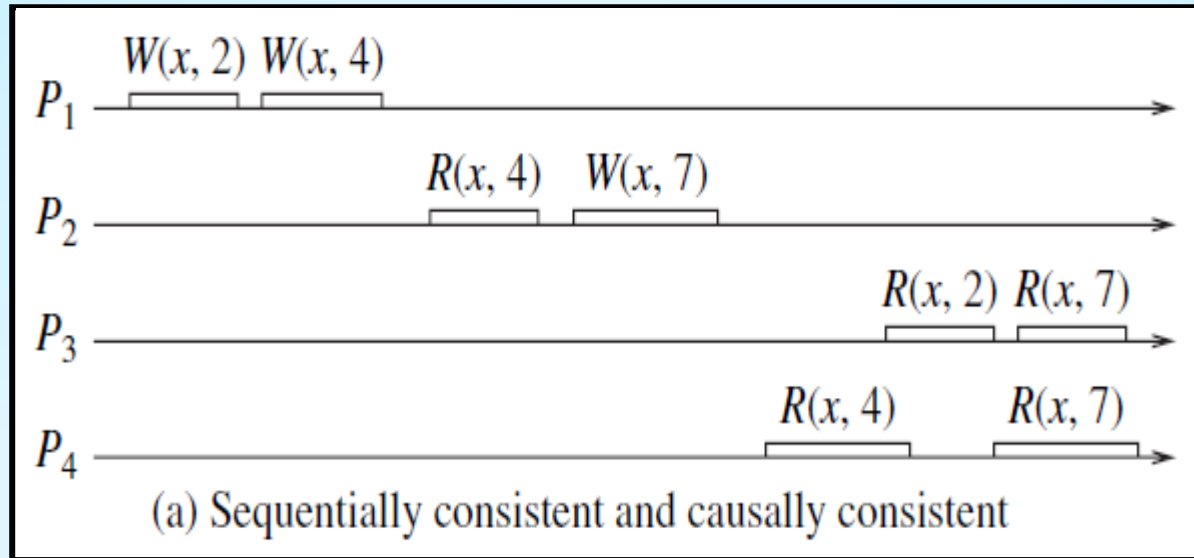


Figure 13.7(a) The execution is sequentially consistent (and hence causally consistent). Both P_3 and P_4 see the operations at P_1 and P_2 in sequential order and in causal order.

Contd...

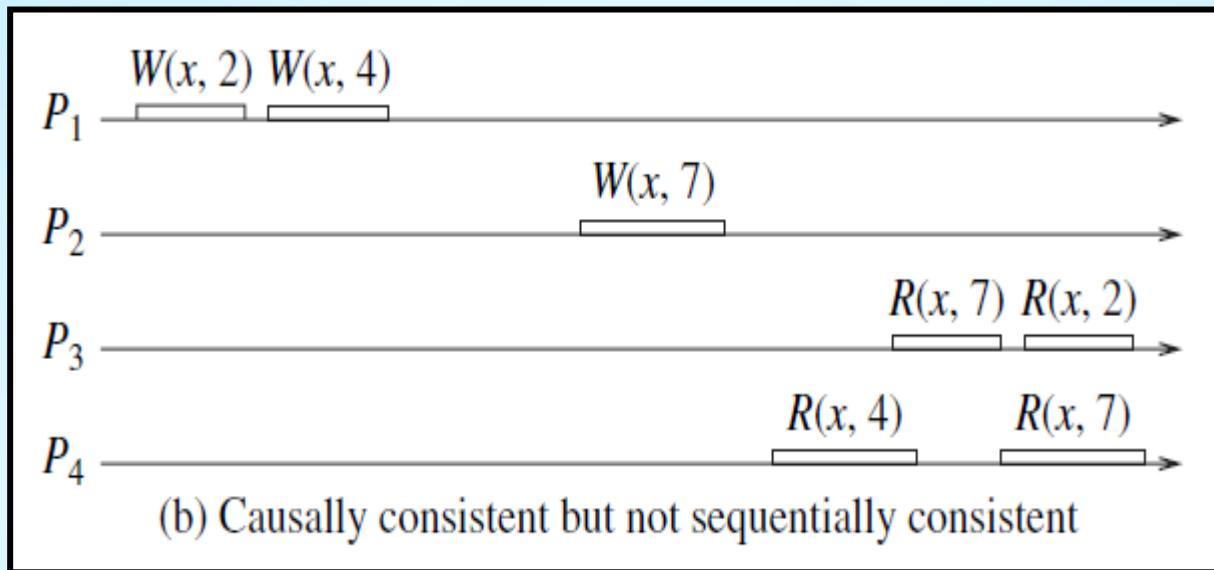


Figure 13.7(b) The execution is not sequentially consistent but it is causally consistent. Both P_3 and P_4 see the operations at P_1 and P_2 in causal order because the lack of a causality relation between the *Writes by P_1 and by P_2* allows the values written by the two processors to be seen in different orders in the system. The execution is not sequentially consistent because there is no global satisfying the contradictory ordering requirements set by the *Reads by P_3 and by P_4* .

Contd...

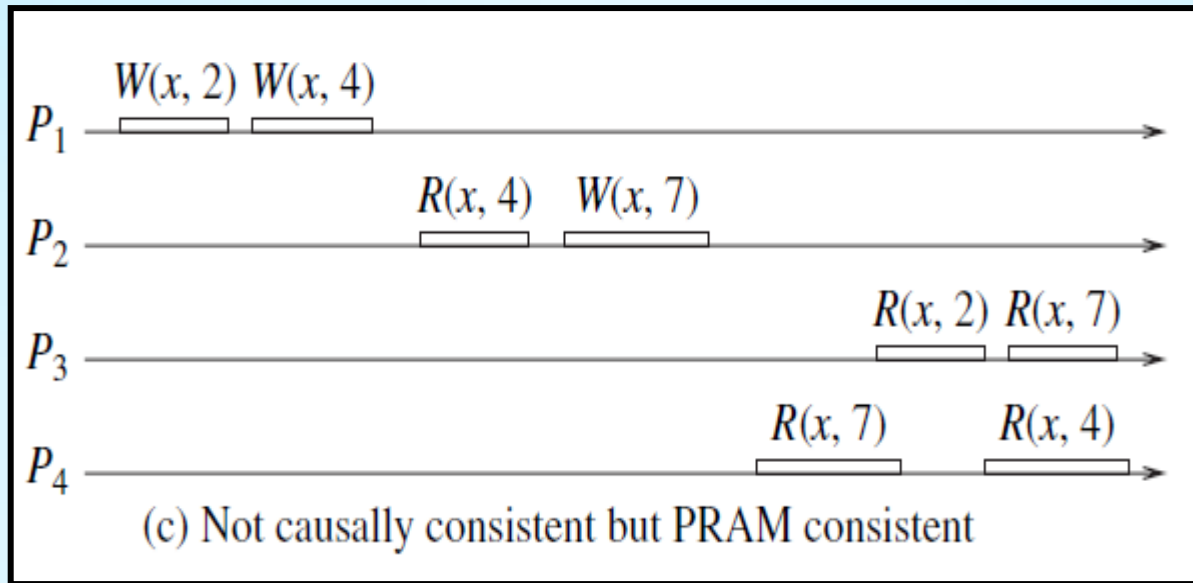


Figure 13.7(c) The execution is not causally consistent because the second Read by P_4 returns 4 after P_4 has already returned 7 in an earlier Read.

4. Pipelined RAM or Processor Consistency

PRAM memory

- Only Write operations 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) $x \leftarrow 4$;

(1b) if $y = 0$ then kill(P_2).

...

(2a) $y \leftarrow 6$;

(2b) if $x = 0$ then kill(P_1).

Algorithm 13.4 A counter-intuitive behavior of a PRAM-consistent program.

The initial values of variables are zero.

5. Slow Memory

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

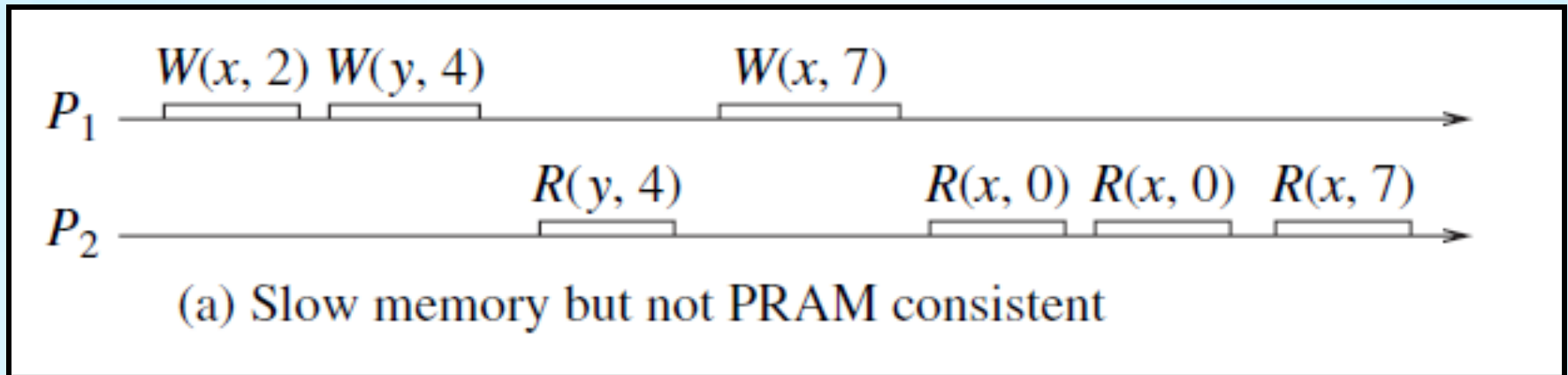


Figure 13.8(a) The updates to each of the variables are seen pipelined separately in a FIFO fashion. The “x” pipeline from P_1 to P_2 is slower than the “y” pipeline from P_1 to P_2 . Thus, the overtaking effect is allowed. However, PRAM consistency is violated because the FIFO property is violated over the single common “pipeline” from P_1 to P_2 – the update to y is seen by P_2 but the much older value of $x = 0$ is seen by P_2 later.

Contd...

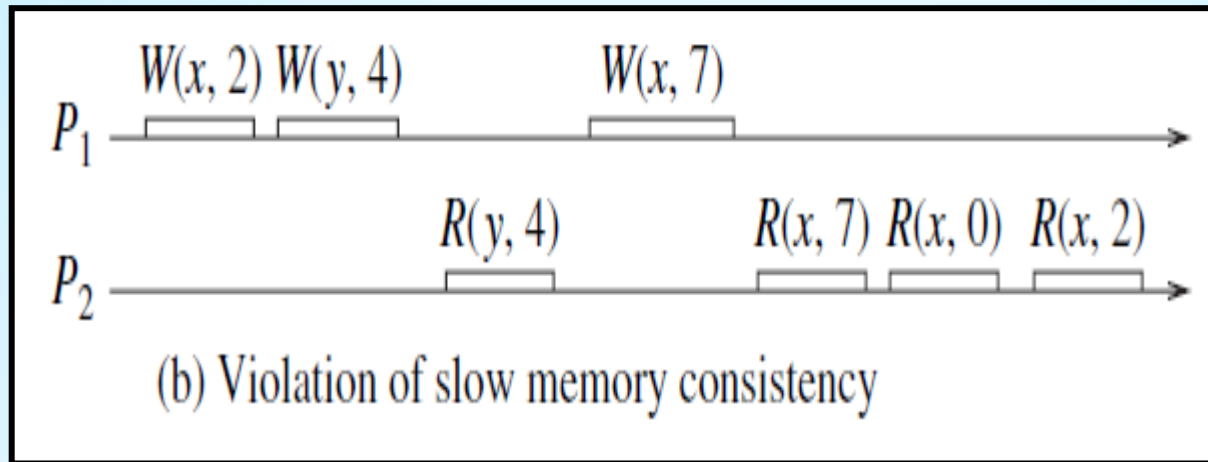
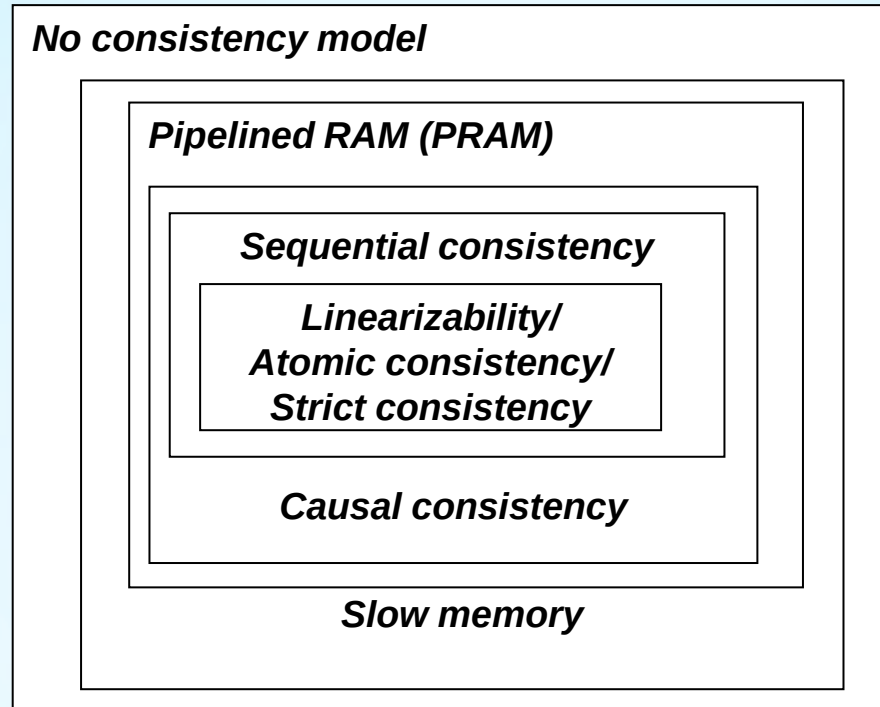


Figure 13.8(b) Slow memory consistency is violated because the FIFO property is violated for the pipeline for variable x . “ $x = 7$ ” is seen by P_2 before it sees “ $x = 0$ ” and “ $x = 2$ ” although 7 was written to x after the values of 0 and 2.

Strict Hierarchy of the Memory 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).

Contd...

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 (By L. Lamport, 1974)

- **Lamport** proposed the classical bakery algorithm for ***n-process*** mutual exclusion in shared memory systems.
- The algorithm is so called because it mimics the actions that customers follow in a ***bakery store***.
- A process wanting to enter the **critical section (CS)** picks a **token number** that is one greater than the elements in the array choosing $[1..n]$.
- Processes enter the critical section in the increasing order of the token numbers. In case of concurrent accesses to choosing by multiple processes, the processes may have the same token number.
- In this case, a **unique lexicographic order** is defined on the tuple **<token, pid>**, and this dictates the order in which processes enter the critical section.
- The algorithm for **process *i*** is given in Algorithm 13.4.
- The algorithm can be shown to satisfy the three requirements of the critical section problem: (i) mutual exclusion, (ii) bounded waiting, and (iii) progress.

Bakery Algorithm

```
(shared vars)
array of boolean: choosing[1...n];
array of integer: timestamp[1...n];

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

Algorithm 13.5 Lamport's *n*-process bakery algorithm for shared memory mutual exclusion. Code shown is for process P_i , $1 \leq i \leq n$.

Important Results

- **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:** $O(n)$ 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

Few Other Algorithms

Author	Year	Algorithm
L. Lamport	1987	Fast mutual exclusion algorithm
G. L. Peterson	1981	The two-process mutual exclusion algorithm
G. L. Peterson and M. Fischer	1977	Modified Mutual exclusion algorithm for two processes
L. Lamport	1977	Concept of Wait-freedom

Conclusion

- **Distributed shared memory (DSM)** is an abstraction whereby distributed programs can communicate with memory operations (***Read and Write***) *as opposed* to using message-passing.
- In this lecture, we have discussed the '**concept of DSM**' and several **consistency models** like linearizability, sequential consistency, causal consistency, pipelined RAM (PRAM), and slow memory.
- Then we have discussed the fundamental problem of '**Shared Memory Mutual Exclusion**' with the help of well-known '**Lamport's Bakery Algorithm**'
- In upcoming lecture, we will discuss about '**Distributed Minimum Spanning Tree**'.