Distributed Systems
Dr. Rajiv Misra
Department of Computer Science and Engineering
Indian Institute of Technology, Patna

Lecture – 13
Distributed Shared Memory

(Refer Slide Time: 00:13)

Lecture 13 distributed shared memory; preface, Recap of previous lecture.

(Refer Slide Time: 00:22)

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.

In 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 to basically see the distributed deadlock in distributed systems. Content of this lecture: In this lecture, we will discuss about the concept of distributed shared memory as well as provide different ways to classify distributed shared memories and consistency models of distributed shared memory and also discuss Lamport's Bakery algorithm for shared memory mutual exclusion.

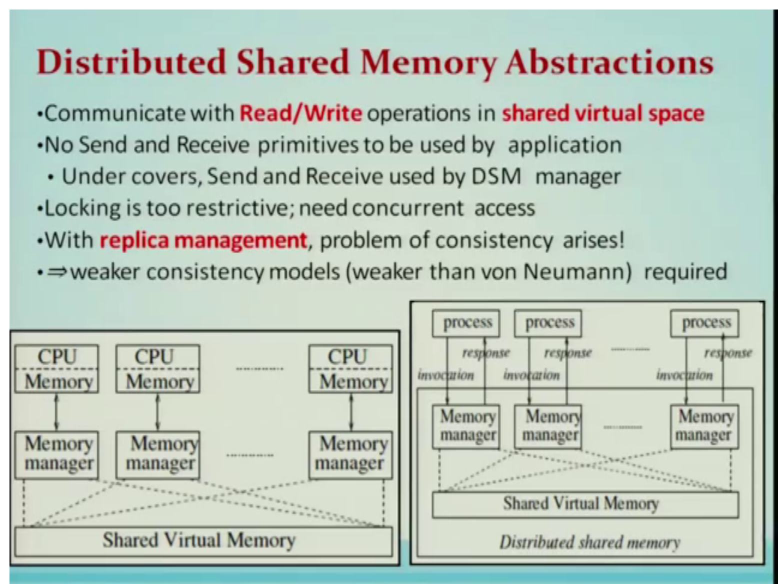
(Refer Slide Time: 01:06)

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.

Introduction: Distributed shared memory is an abstraction provided to the programmer of a distributed system. It gives impression of a single monolithic memory, as in the traditional von Neumann architecture. Programmers access the data across the network using only read and write primitives, and they would as they would do it in uniprocessor system. Programmers do not have to deal with send and receive communication primitives and also ensuring the complexity of dealing explicitly with synchronization and consistency in the message passing model. So, all the intricacies will be bypassed, if the programmers are given a high level abstraction which is called a distributed shared memory.

(Refer Slide Time: 02:01)



So, distributed shared memory abstractions. They communicate they provide the abstraction to the programmers so that they can communicate using read and write operations in the shared virtual space. No send and receive primitives to be used by the application, under the covers, send and receive used by the distributed shared memory manager, here locking is too restrictive and also need the concurrent access. So, with the replica management, problem of consistency arises. So, weaker consistency model that weaker than von Neumann architecture is required in this particular scenario.

Let us understand this particular picture to understand the place or the placement of a distributed shared memory in the system architecture. Now, we see that in this particular figure, we have seen that every process or every processor has its memory within it. So, out of this particular memory which is available with the processor in the system, some part of the memory is basically assigned for the distribute shared memory and remaining will be used as the local memory. This particular memory which is now spared by the different processors will be managed by a module which is called a memory manager. And this particular memory manager will give a complete one view of a monolithic memory, that is called a shared memory and that is realized using memory manager in the system.

So, that was an architecture, so here the placement of the shared virtual memory you can see is lying over here in the distributed shared memory and this memory manager and

now basically the processes of the application. They communicate with the memory manager through two different constructs one is called invocation, the other is called response.

(Refer Slide Time: 04:38)

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.

So, invocation and response are basically the primitive to access the memory as if it is there in the von Neumann architecture.

The advantage of distributed shared memory: It shields the programmer from doing the send receive primitives; that means, the programmers have to only use the read and write primitive. Regarding realization of read and write primitives on the distributed system that is the send and receive will be completely abstracted. Single there will be a single address space and it simplifies the passing by reference and passing complex data structure. So, once a single address space is available, the programming becomes easier and the constructs like passing by reference and passing complex data structure will become a convenient for the programmer to use the distributed shared memory.

It exploits the locality-of-reference when a block is moved. Distributed shared memory uses simpler software interfaces, and cheaper off-the-shelf hardware hence cheaper than dedicated multiprocessor systems are realized. No memory access bottleneck, here as no single bus large virtual memory space is available. Distributed shared memory programs are portable as they use the common distributed shared memory programming interface.

(Refer Slide Time: 06:04)

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.

The disadvantages of distributed shared memory: here programmer need to understand the different consistency models, to write the correct programs.

Distributed shared memory implementations use asynchronous message-passing, and hence cannot be more efficient than the message-passing implementations. By yielding control to the distributed shared memory manager, software programmers cannot use their own message-passing solutions. So, it is an abstraction and the programmer has to work in that APIs are at interfaces which is being provided by the distributed shared memory.

(Refer Slide Time: 06:46)

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

Issues in implementing distributed shared memory software. So, there are several issues in implementation and we will be touching upon in more details in this slides.

So, some of the issues which are listed over here as semantics for concurrent access must be clearly specified, semantics of replication, location for the replication, for optimization and the it will it should reduce the delays, and number of messages to implement the constructs. Similarly the data is replicated or cached, this particular aspect is a decision aspect in the design part remote access by the hardware or a software, this also is a design aspect and this will be a major issue caching oblique replication whether it is controlled by the hardware or software, is also a design issue and this will be dealt for different applications. Distributed shared memory controlled by the memory management software, operating system, and language compilers.

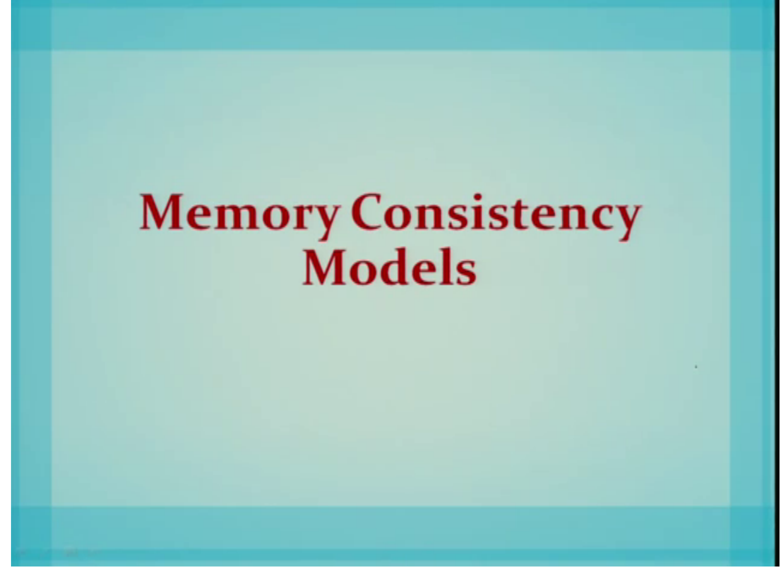
(Refer Slide Time: 08:04)

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

Now, this particular chart or a table matrix this will give a comparison of early distributed shared memory systems. Now the type of shared memory systems you have seen in the or we are seeing that in this matrix, that there is a single bus multiprocessor and multiprocessors and paged base shared memory, shared memory, shared variable, a distributed shared memory, shared object, distributed shared memory. So, these are a different type of distributed systems and they use a different kind of caching methods. Some are using the hardware and the some are using the software and remote access also some are doing by hardware the others are realizing using software.

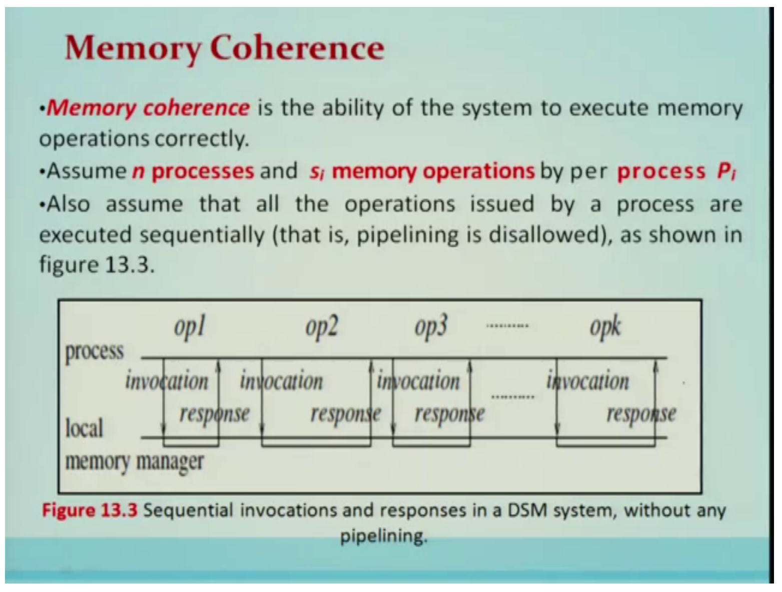
(Refer Slide Time: 08:58)



So, all these comparison will show that distributed shared memory requires a lot of system level intricacies and depends upon different applications, how efficiently they are going to exploit hardware versus software, replication versus a caching. So, these are the major issues which we are going to see and different consistency models. So, that distributed memory will be realized as single monolithic memory at the time of access using read and write. How that is all done we are now going to see the memory consistency model, because this model is now given to the programmer, and programmer will see this particular model and write a programs.

These models how they are implemented we will see in this particular discussion. Memory consistency model.

(Refer Slide Time: 09:58)

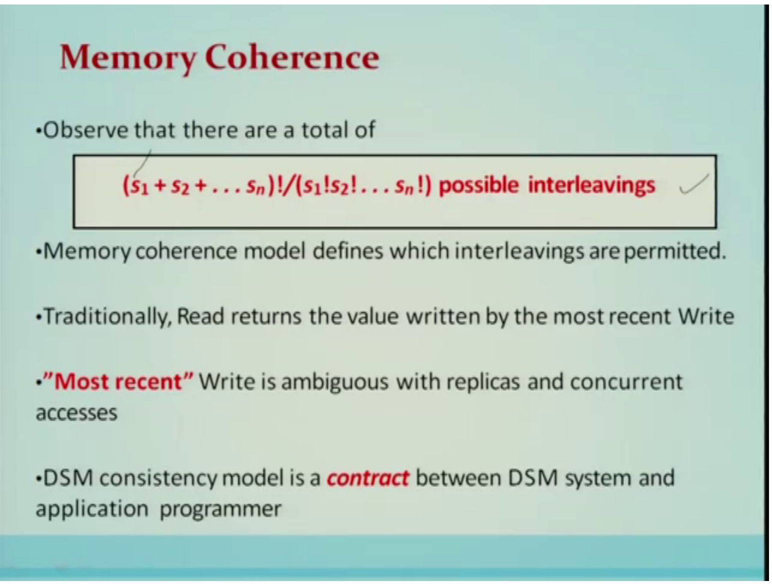


Memory coherence; memory coherence is the ability of the system to execute memory operations correctly. Assume n processes and s i the memory operation per process P i. So, also assume that all the operations issued by the process are executed sequentially and pipelining is disallowed. So if we see this particular figure 13, it shows the sequential invocation and responses in a distributed shared memory. So, in this particular model, one thing we have to understand is that; there is the interaction between the process and the local memory manager, the placement we have shown you in the previous slide.

Now, the process will through the operations, it will do the invocation for the shared memory used and this particular invocation in turn will make a call to the local memory manager. Local memory manager will basically handle these invocations, through the internal details of that we are going to see and provide the response to the operation. So, rest of the internal intricacies are hidden from the programmer or is being abstracted only in the form of invocation and response. So, these particular every invocation will lead to a different memory operations.

Now, you see that there are so many number of operation, simultaneously at the same point of time are issued on the distributed systems. So, basically each processor will have its own memory operation. So, many memory operations will be overlapping or a non-overlapping and so many number of permutations are possible which one is basically the correct one or a and which one is not allowed or a not correct. So, it depends upon different memory models that we are going to see. And this memory model is basically useful to the programmer to design the correct application or programs.

(Refer Slide Time: 12:09)



So, now we are going to see the memory coherence. Observe that there are total numbers of so many possible interleavings. So, s i is basically the memory operations so many possible permutations are there. So, memory coherence model defines which interleavings are permitted. So, as you see that not all permutations or not all interleavings are allowed in the system. So, some are basically allowed, so the

interleavings which are permitted only they will be captured by the model. So, memory coherence model will define those interleavings which are permitted.

Traditionally, read returns the value which is written by the most recent write. So, most recent write is ambiguous with the replica and the concurrent accesses. So, distributed shared memory, consistency model is a contract between the distributed shared memory system and the application programmer.

(Refer Slide Time: 13:09)

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

So, different consistency models are used by different scientists that we can list out here. Different consistency models are listed as a sequential consistency model by Lamport, linearizability model by again Lamport, a PRAM model and linearizability, slow memory, weak consistency, a release consistency, sequential consistency and so on casual consistency model. These models consistency models are important, why because, they give as an abstract to the programmer and programmer will use this model for.

(Refer Slide Time: 13:48)

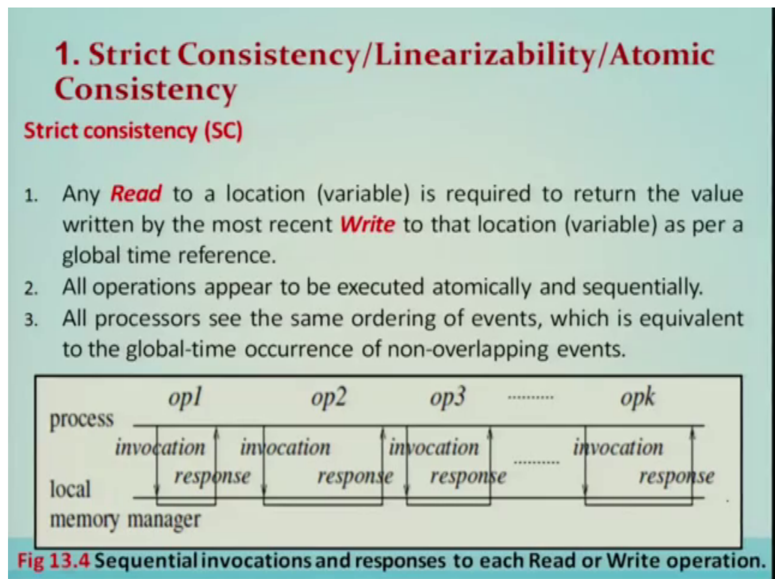


Fig 13.4 Sequential invocations and responses to each Read or Write operation.

So, let us go in more detail of these consistency models, because they are the most important features of the distributed shared memory. The first model is called strict consistency, it is also called a linearizability, it is also called as atomic consistency. So, strict consistency model says that any read to a location is required to return the value written by the most recent write to that location as per the global time reference. So, basically here there are two important thing is that, whenever a read is issued it has to be dependent on the most recent write, second issue; we have to see that this particular dependency has to be linked in the global time scale or a global time frame.

So, all the operations appear to be executed atomically and sequentially. All the a processors see the same ordering of the events, which is equivalent to a global-time occurrence of non-overlapping events. So, here in this strict consistency, the association of the read to the most recent right and also the global time reference is going to be very very important notions.

(Refer Slide Time: 15:43)

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 <invoc, resp> events satisfying:

C1. For every variable *v*, the projection of *Seq'* on *v*, denoted *Seq'v*, is such that **every *Read (adjacent <invoc, resp> event pair) returns the most recent Write (adjacent <invoc, resp> event pair)*** *that* immediately preceded it.

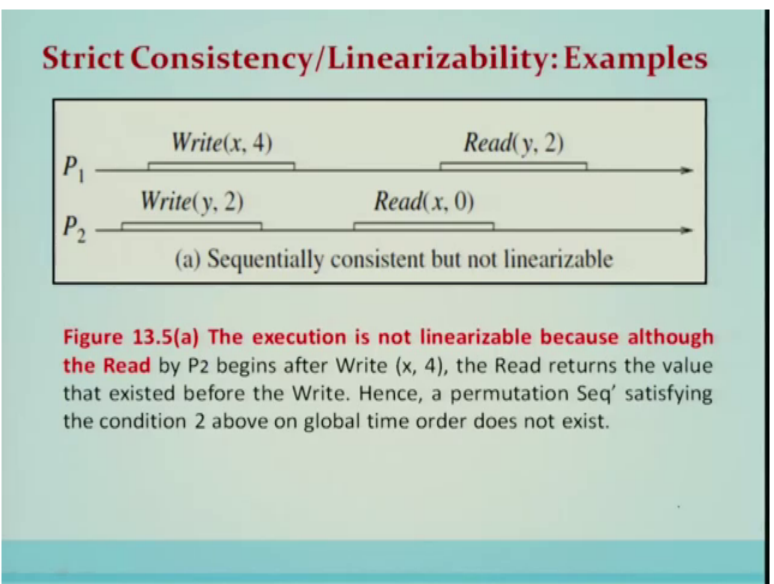
C2. If the response *op1(resp)* of operation op1 occurred before the invocation *op2(invoc)* of operation *op2* in *Seq*, then *op1* (adjacent <*invoc, resp*> event pair) occurs before *op2* (adjacent <*invoc, resp*> event pair) in *Seq'*.

- **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'.

Now, conditions of linearizability. More formally, a sequence of invocation and response is linearizable, if there is a permutation sequence of adjacent pairs of corresponding invocation and response even satisfying. First condition for every variable v, the projection of sequence prime on v, denotes sequence prime v, is such that every read returns the most recent write that immediately preceded it. So, this condition we have seen that the read has to be preceded with the most recent write on a global scale or in a global reference. Second part is says that, if the response of operation 1 occurred before the invocation of operation 2 in the sequence, then operation 1 occurs before operation 2 in the sequence prime; that means, in the globally scale.

So, if the operation 1 happened before operation 2. So, in the global scale it should reflect this happen before deletion and this is the condition number 2 and this has to be in a reference to the global time frame. So, condition 1 is specifies that every processor sees a common order of sequence prime of events that and that or and that in this order the semantics is that read returns the most recent in completed write value. Condition 2 specifies that the common order must satisfy the global time order of events that is the order of non overlapping operations in the sequence must be preserved.

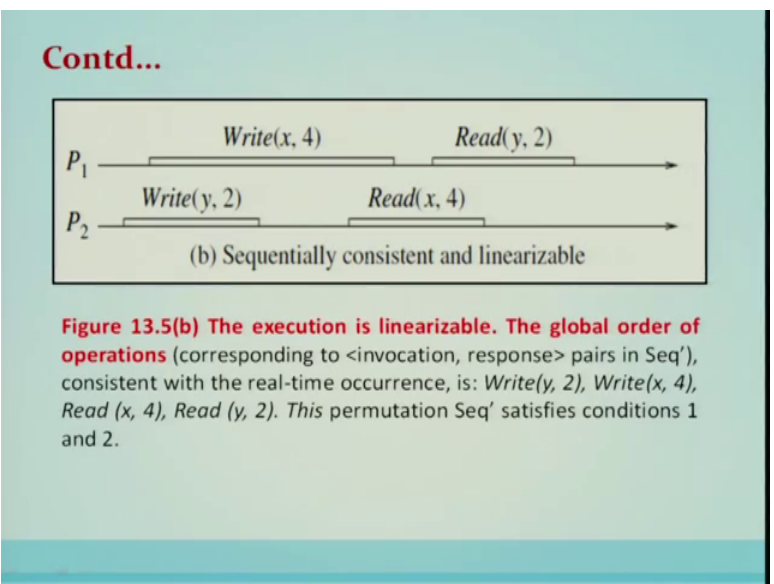
(Refer Slide Time: 17:30)



Strict sequence strict consistency or linearizability: example we can see over here is that, and in this particular figure the execution is not linearizable because the read by P 2 here gives the value 0; although the most recent write of x is 4, although it is taking that particular value not the most recent, but the old value where x was 0 in that case so P 2; So, here we can see that the P 2 issues the write P 2 begins after the write x, 4, so this particular read happens after the write. And so basically this read is not as shared with the most recent right.

Hence, this is not linearizable. This example shows that it is not linearizable; however, it is sequentially consistent. What is sequentially consistent? We will explain in a minute; in a next slide. Hence, the permutation this permutation or the ordering that is in sequence prime satisfying condition 2 above on the global time order does not exist. So, out of two condition; condition number 1 and 2 defined earlier, the it violates condition number 2 hence it is not linear is not linearizable.

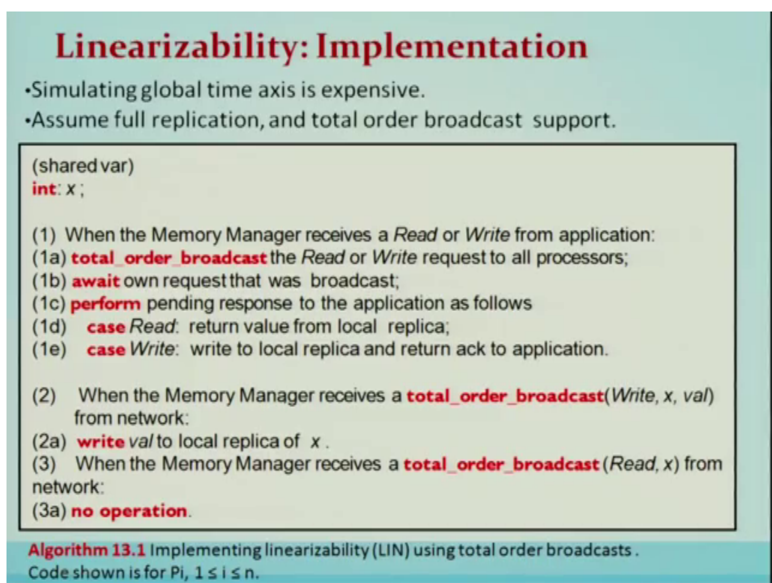
(Refer Slide Time: 19:02)



So, this particular example in figure 13.5 the execution is linearizable. So obviously, we can see over here that this particular read of x is drawn out of the most recent write. So, here the value 4 is written and the read is also basically able to fetch the same value or that value is basically now available whatever recent write has done similarly for y. So, y in most recent y has written nearly two and that is available to the read which is following the right. So, the this particular both are read operations so, hence it is linearizable.

And, it is also sequentially consistent. So, here it is written that it is consistent with the real time occurrence and that is write y 2 and write x 4, read x 4 and read y 2 is basically the sequence, and that is why it is linearizable? And this sequence is following the global timeframe or the real time occurrence. Hence this permutation sequence time satisfies the condition 1 and condition 2; hence it is very strictly consistent and linearizable.

(Refer Slide Time: 20:31)



The implementation of linearizability we can see here, requires two aspects to be taken into an account. The first aspect which we have seen is how to associate the read with the most decent write, the second one is how to evolve a global time reference. Although, there is no global clock which and in the distributed system and also there is no common memory. So, in spite of these two absence, we have to provide the global time frame reference to all the events which is occurring. Linearizability is implementation is a challenge.

So, let us see how the linearizability is implemented. So, as I mentioned simulating global time axis is expensive. Assume full replication is available, and a total order broadcast support is also available. Total order broadcast will be used here in implementation of linearizability. Now, here when the memory manager receives the read and write from the applications, it will issue a total order broadcast the read or write request to all the processes processors. So, it will await its own request that was broadcast perform the pending response as follows; when if the case is read, then it will return the value from the local replica; if it is write then it will write to the local replica and return acknowledgment to the application.

Now, when the memory manager receives the total order broadcast, that is; write x value from the network. Then it will write the value to the local replica x. Then the memory manager receives a total order broadcast from read x value from the network they will

not do any response. So, here you can see that either it is read or write in both the operations, it will issue a total order broadcast why it is issue a total order broadcast is to evolve a global time reference implementation.

(Refer Slide Time: 22:47)

Linearizability: Implementation (2)

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

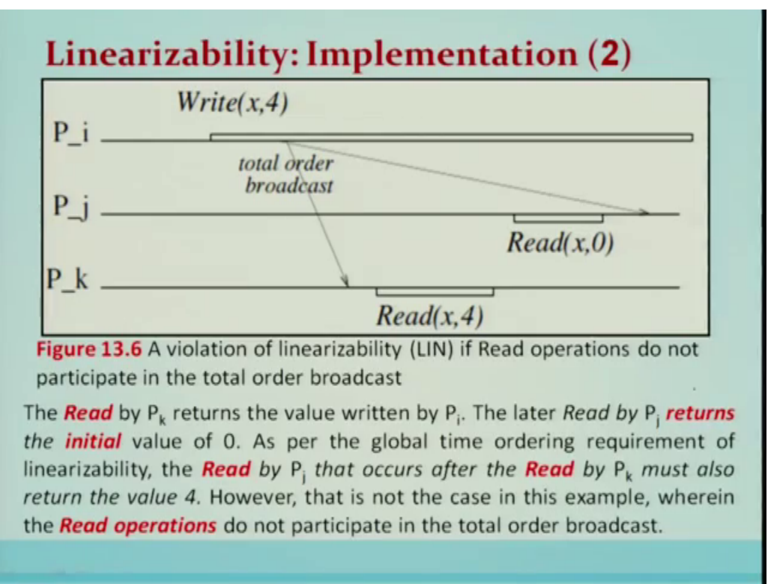
•**Why is the broadcast then necessary?**

•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**

So, for read operation, whenever a memory manager system wide receives a total order broadcast, they do not perform any action that we have seen in the algorithm. Then why is the broadcast necessary? The reason is this. If the read operations do not participate in total order broadcast, they do not get totally ordered with respect to the write operation as well as with respect to the other read operations. Hence, the read is to be associated with the most recent write is realized, because of this total order broadcast of read as well as write operations.

(Refer Slide Time: 23:24)



The example you can see over here in this particular figure is that, when I write issues a total order broadcast this message will reach P k earlier than P j. So, if it reaches P k earlier and then a read is issued to read this variable x which is written by the most recent write. So, that value is available whatever is recently written value for x.

However; for P j the read is happening before, because the total order broadcast is receiving at a later point of time. Hence, it is going to read the old value although, this is happening after this read even then it is able to only view the old values, because new value is not available hence it is a violation of linearizability. So, that is why the read operation have to basically participate in total order broadcast that I explained you.

(Refer Slide Time: 24:36)

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.

The next consistency model is called sequential consistency. Sequential consistency is specified as follows: The result of any execution is the same as if all operations of the processors who are executed in some sequential order. The operations of each individual processor appear in this sequence in the local program order. So, any interleaving of the operations from different processor is possible. But all processors must see the same interleaving. Even if the two operations from different processors do not overlap in a global timescale, they may appear in a reverse order in a common sequential order seen by all the processors.

So, here one thing we have to understand, that sequential consistency model is going to evolve a sequence or a some other sequence and that sequence should be visible to all the processors.

(Refer Slide Time: 25:43)

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.

So that order we are going to see how we are going to evolve in sequential consistency. So, here implementation of a sequential consistency model which is a weaker than the linearizable model or a strict consistency model it is weaker model. So, it only here you can see that only write participate in a total order broadcast. And reads do not because: all consecutive operations by the same processors are ordered in the same order, the read operations by different processors are independent of each other; and to be ordered only with respect to the write operations.

(Refer Slide Time: 26:30)

Sequential Consistency using Local Reads

```
(shared var)
int: x ;
(1)When the Memory Manager at Pi receives a Read or Write
from application:
(1a) case Read: return value from local replica;
(1b) case Write(x,val): total_order_broadcasti(Write(x,val)) to all
processors including itself.

(2)When the Memory Manager at Pi receives a
total_order_broadcastj(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 Pi, 1 ≤ i ≤ n.

Direct simplification of linearizability algorithm is we are going to show you. So, sequential consistency using local reads. So, here when a memory manager P i receives read or write from the application in sequential consistency model we see that, what it will do? It will form the two cases read and write; if read then it will return the local replica. And for the write operation, if it want to write the value to the variable x, then it will issue a total order broadcast to all the processor including itself. When this memory manager at P i receives the total order broadcast from j from the network, then it will write the value to the local replica. And if it is the same process then it will send the acknowledgment to the application.

So, here we see that only the write will issue the total order broadcast and read basically is not required by, because it is a weaker model than strict consistency model we have seen all linearizability model. So, this algorithm issues the locally issued writes get acknowledgment locally read are delayed until the locally proceeding writes have been acknowledged or locally issued writes are pipeline.

(Refer Slide Time: 28:05)

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

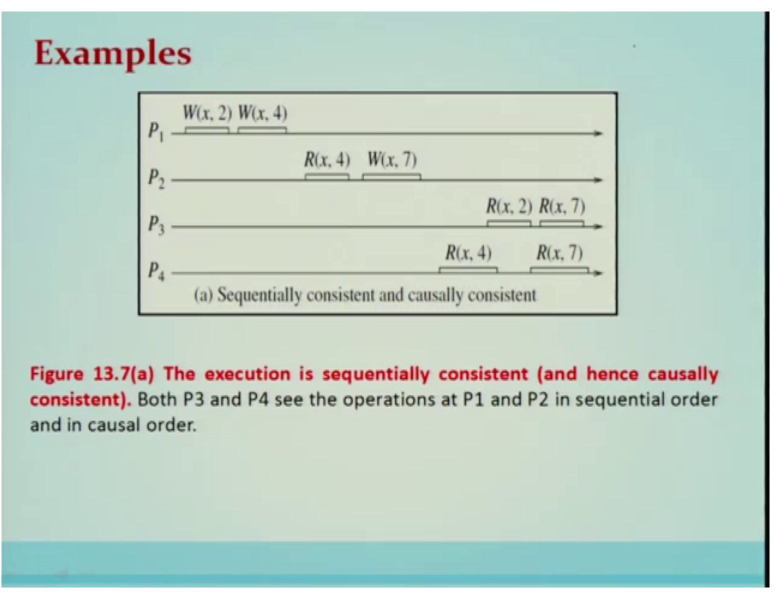
Total order broadcasts (for SC) also provide causal order in shared memory

So, this is an improvement using local writes. Now, the next consistency model for distributed shared memory is called casual consistency. Causal consistency is also a weaker model compared to the sequential consistency model.

In sequential consistency all write operation should be seen in a common order that we have seen that after issue the write, then a total broadcast message is performed total

order after the write operations. So, all the write operation should be seen in a common order in the sequential consistency. Now for causal consistency only causally related write should be seen in a common order. So, causal relation for a shared memory system: At a processor, local order of events is the causal order and write usually precedes read issued by another process if the read returns the value written by the write. The transitive closure of the above to order is causal order. Total order broadcast for the consistency sequential can also provide the causal order in the shared memory.

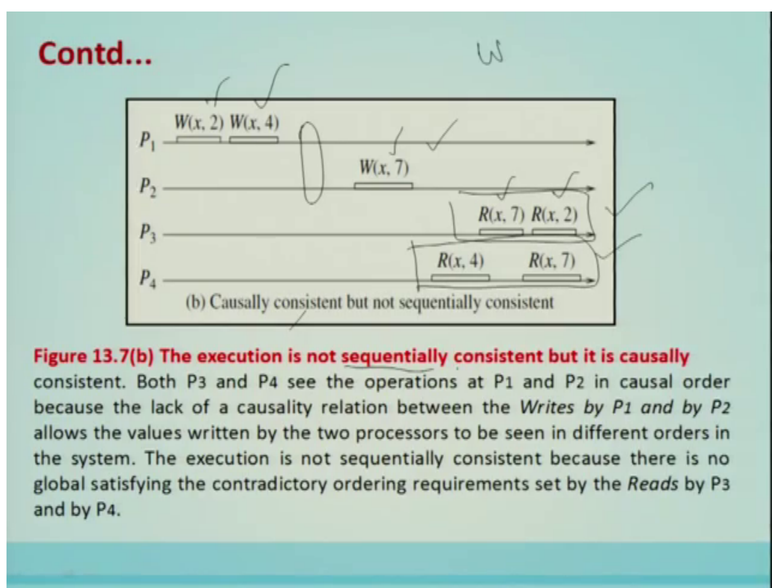
(Refer Slide Time: 29:21)



So, here we can see that, in this example the execution is sequentially consistent, hence it is causally consistent, why? Because casually consistency is a weaker than the sequential consistency. Here in this example you see both P 3 and P 4 see the operations at the P 1 and P 2 in a sequential order, hence in the casual order. So, that P 3 and P 4 the operations are basically the read operations of value x; here by P 3 so x value is written two which is available over here, similarly this x is writing 7 and this particular order is followed, why? Because, they are they are sequential order as well as causally related order.

Similarly, x 4 and x 7 so x 4 here x 4 it is there and x 7 is there. So, they are causally related as well as and so it is sequential consistency as well as; causal consistency model which is being taken in this example.

(Refer Slide Time: 30:44)

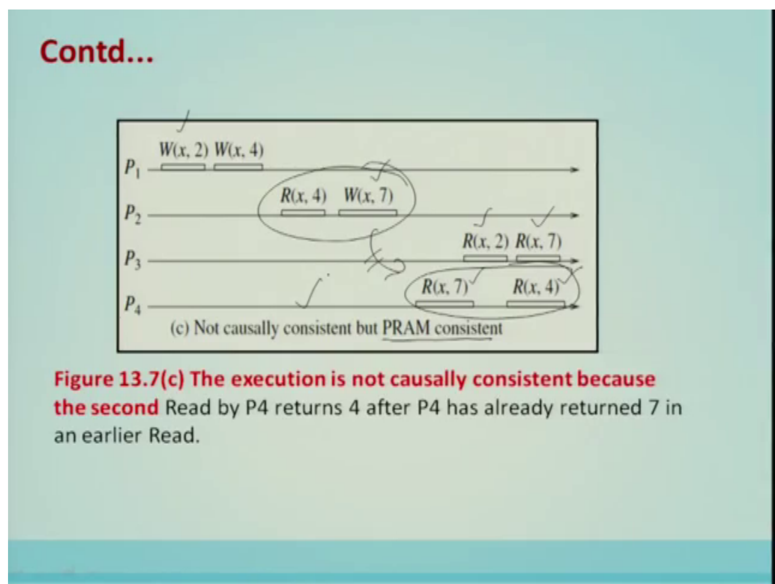


So, this example shows that the execution is not sequentially consistent, but causally consistent; that means, causally consistent is a weaker model this particular example will show, and whereas, the sequential consistency is not followed. So, here we can see that both P 3 and P 4 see the operations at P 1 and P 2 in a casual order, because the lack of causality relation between writes by P 1 and P 2 allows the values written by the two processors to be seen in different order of the system.

The execution is not sequentially consistent because there is no globally satisfying contrary contradictory ordering requirement by reads and write. So, here you can see that that the causal ordering is achieved, in the sense that if we see the read operations of P 3 and read operations of P 4. So, here the first read is able to read 7 and the second read is able to read the value 2. So, they are different processors as you see. So, as far as casual dependency is concerned they are satisfying as far as x 4 x is concerned it is getting 4 and then 7. So, causal consistency is allowed, but sequential consistency is not there, why? Because, here you can see that a first 7 is read and then 2 is read and here 4 is basically read and then 7 is read.

So, the ordering of ordering is cannot be organized as per the sequential consistency. Hence, it is a casual consistency, but not a sequential consistency.

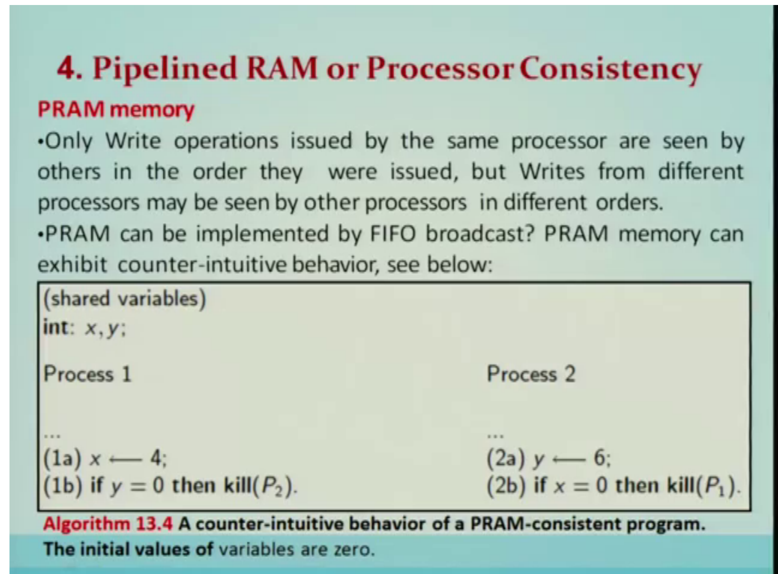
(Refer Slide Time: 32:42)



So, this example shows that it is not even a casual consistency. So, you will see a weaker model than causal consistency which is called a PRAM model; where that PRAM consistency will be there, but not causally consistent. So, casually consistent, why it is not there? So, you can see that x is basically 2 and then here x is basically read as 7. Now here x is read at 7 and x is then read at 7. So, this causal relation is violated here in this particular order.

So, 4 is proceeding 7, because it is happening with this particular relation is violated here. Hence, this is not causally consistent, but we will see another weaker model where it is PRAM consistent.

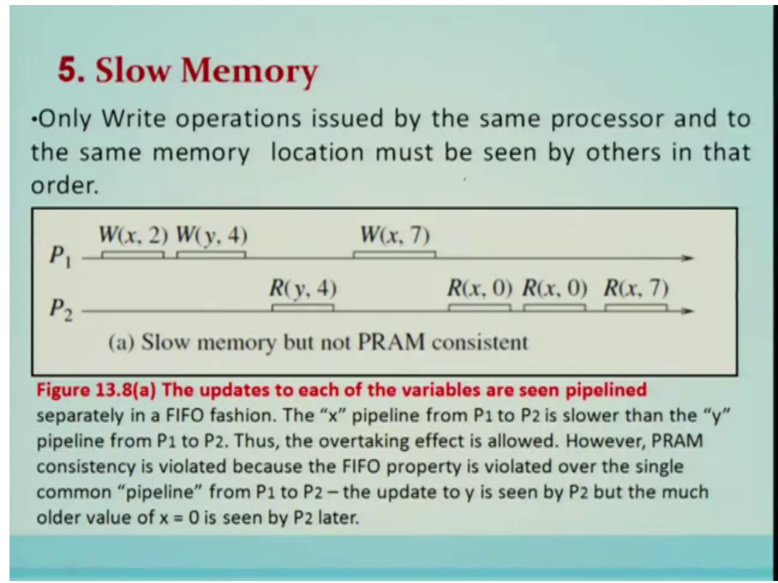
(Refer Slide Time: 33:43)



So, PRAM full form is called pipelined ram model or a processor consistency model, it is also called as a pc that is the processor consistency. That is the consistency at local level. So, only the write operation issued by the same processor are seen by the others in the order they were issued, that writes from different processors may be seen by the other processors in different order.

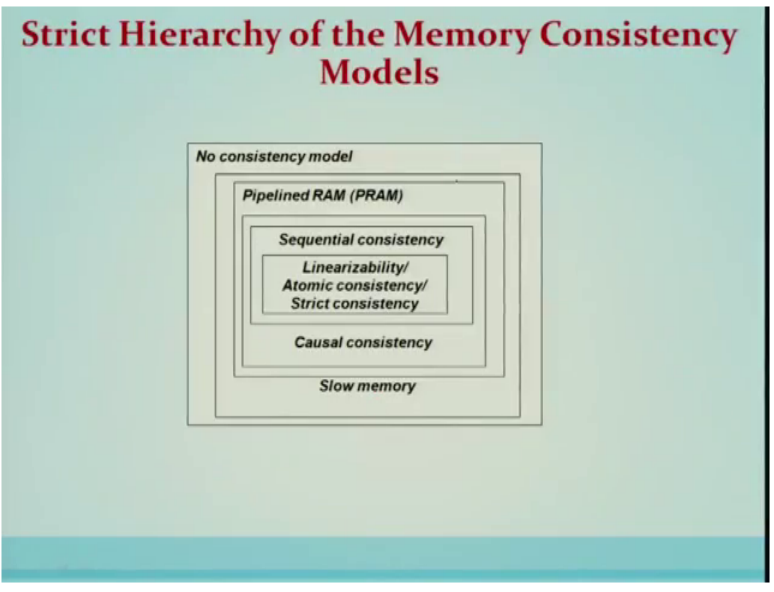
So, here the ordering by ordering of writes by the same processor is there unlike in casual ordering where the ordering of writes between different processors are also, if they are casually related there that also is a glaze and force. So, here processor consistency or a pipeline ram or a PRAM consistency model is a weaker form of a causal consistency model. Now this PRAM can be implemented using a FIFO broadcast.

(Refer Slide Time: 34:45)



Another consistency model is slow memory. So, only write operation issued by the same processor and to the same memory location must be seen by the others in that order. Slow memory, but not the PRAM so; obviously, we see that there is another weaker model that is called a slow memory model.

(Refer Slide Time: 35:02)



Now, after seeing so many number of consistency models starting from very strict consistency model, that is called the linearizability model or a strict consistency model. Then we have seen that a weakening of this model, we realized a another sequential

consistency model. And further weakening it we have obtained a causal consistency model weakening; further casual consistency we have obtained PRAM model pipelined ram model, weakening this pram model also we have received a slow memory model and we are slow memory there is no consistency model is also there; that means, consistency model is not assumed. So, this particular weakening will enforce a strict hierarchy of the memory consistency models which is shown here in this picture.

(Refer Slide Time: 36:04)

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 model. So here we are not going to see the synchronization based consistency model. The first one is called weak consistency. So, consistency conditions apply only to the special synchronization instructions and for example, barrier synchronization. Non-sync statement will be executed in any order by various processors. Example, weak consistency, release consistency, entry consistency. So, weak consistency all the writes are propagated to the other processes, and all writes done elsewhere are brought locally, at the sync instruction. Accesses to the sync variables are sequentially consistent. Access to the sync variables is not permitted until all writes elsewhere have completed. No data access is allowed until all previous synchronization variables. Accesses have been performed.

Drawback: cannot tell whether the beginning access to the shared variable enter critical section, or finished access to the shared variable that is exit critical section.

(Refer Slide Time: 37:08)

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.

Two types of synchronization variables: acquire and release. Release consistency acquire indicates CS is to be entered. Hence all writes from the other process should be locally reflected at this instruction. Release indicates access to the critical section is being completed. Acquire and release can be defined on a subset of the variables. Lazy release consistency propagates the updates on-demand, and not in PRAM way.

So, entry consistency each ordinary shared variable is associated with a synchronization variable lock or barrier.

(Refer Slide Time: 37:56)

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.

Now, we are going to see a shared memory mutual exclusion algorithm which is given by the Leslie Lamport and it is also called the Bakery algorithm. So, Lamport proposed the classical bakery algorithm for n-process mutual exclusion in the shared memory system. The algorithm is so called because it mimics the action that the customers follow in a bakery store. A process wanting to enter critical section picks a token number that is one greater than the elements in the array choosing from 1 to n.

So, processors processes enters 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 obtained. In this case, a unique lexicographic order is defined on a tuple token and pid, and this will give a total order and this dictates the order in which the processes are entering the critical section, the algorithm for process i is given in the next slide. The algorithm can be shown to satisfy three requirements of the critical section problem the first is mutual exclusion, bounded waiting and progress.

(Refer Slide Time: 39:07)

Bakery Algorithm

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

repeat
(1) Pi executes the following for the entry section:
(1a) choosing[i] ⟵ 1;
(1b) timestamp[i] ⟵ max_{k∈[1...n]}(timestamp[k]) + 1;
(1c) choosing[i] ⟵ 0;
(1d) for count = 1 to n do
(1e)      while choosing[count] do no-op;
(1f)      while timestamp[count] ≠ 0 and (timestamp[count], count) < (timestamp[i], i) do
(1g)           no-op.
(2) Pi executes the critical section (CS) after the entry section
(3) Pi executes the following exit section after the CS:
(3a) timestamp[i] ⟵ 0.
(4) Pi 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 Pi, 1 ≤ i ≤ n.

This is the bakery algorithm Lamports and process Bakery algorithm for shared memory mutual exclusion. So, here we can see that this introduces the timestamp or this ordering. So, mutual exclusion the role of line 1 e wait for others timestamp choice to stabilize and the use of timestamp is to order them according to the priority.

(Refer Slide Time: 39:30)

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

So, highest priority will be allowed to go into a critical section and this will ensure the mutual exclusion. Bounded waiting means P i can be overtaken by the other process at most once the progress means lexicographic order is the total order process with the lowest timestamps will enter the critical section that becomes the high priority.

So, a space complexity here the lower bound of n registers time complexities of the order n time of Bakery algorithm Lamports fast Mutex algorithm takes of the order one in the absence of the contention; however, it compromises unbounded waiting it uses write read write write a write and a read. So, write of x followed by read of y, then write of y followed by read of x, this sequence necessary and sufficient condition to check for the contention and safe safely enter the critical section

(Refer Slide Time: 40:41)

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

There are few other algorithms in this particular problem. The another algorithm as I as I mentioned that is called fast mutual exclusion algorithm, the two process mutual exclusion, modified mutual exclusion algorithm for to process, concept for wait freedom.

(Refer Slide Time: 40:58)

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**'.

Conclusion; Distributed shared memory is an abstraction whereby distributed programs can communicate with the memory operations that is through read and write as opposed to using message-passing intricacies. So, in this lecture we have discussed the concept of distributed shared memory and we have also seen several consistency model like

linearizability, see the sequential consistency, casual consistency, pipeline ram, and slow memory. We have also discussed the fundamental problem of shared memory mutual exclusion with the help of the Lamport's Bakery algorithm. In the upcoming lecture, we will discuss about distributed minimum spanning tree.

Thank you