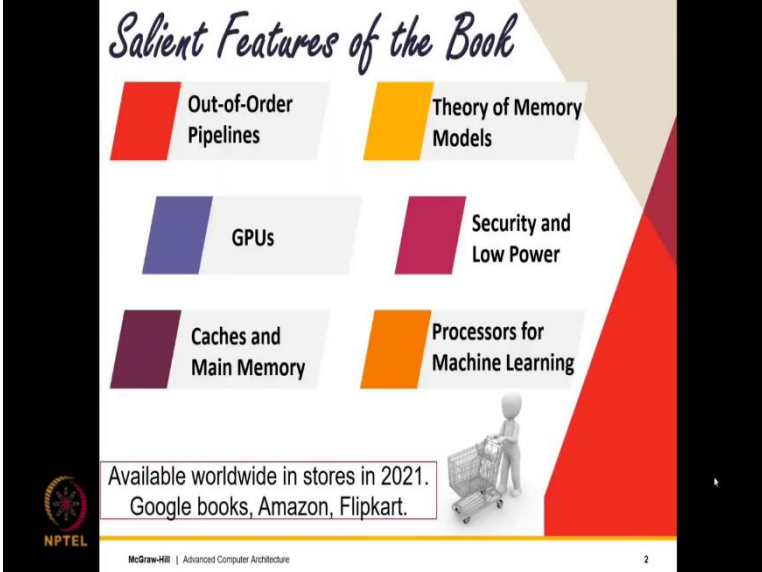


Advanced Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 32
Multicore Systems Part - VIII

(Refer Slide Time: 00:23)



Salient Features of the Book

- Out-of-Order Pipelines
- Theory of Memory Models
- GPUs
- Security and Low Power
- Caches and Main Memory
- Processors for Machine Learning

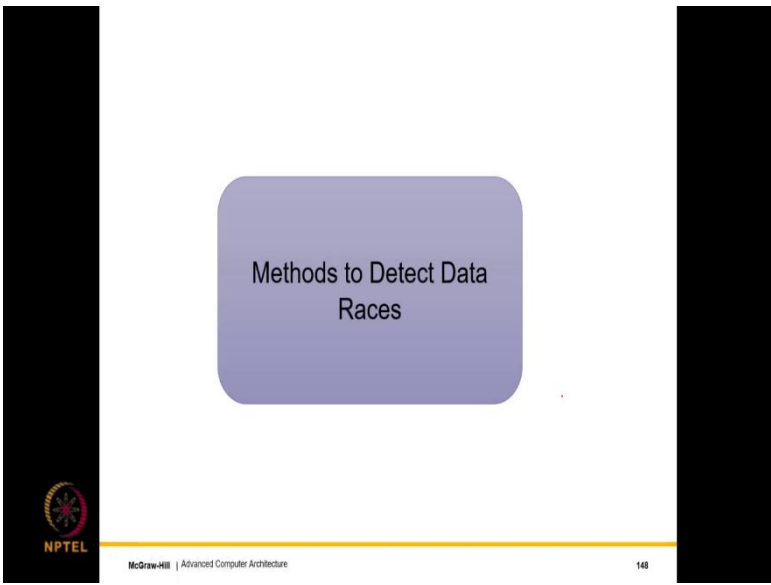
Available worldwide in stores in 2021.
Google books, Amazon, Flipkart.

NPTEL

McGraw-Hill | Advanced Computer Architecture

2

(Refer Slide Time: 00:38)



Methods to Detect Data Races

NPTEL

McGraw-Hill | Advanced Computer Architecture

148

Welcome to lecture 8 of the chapter Coherence Consistency and Transaction Memory in Multiprocessor systems. So, in this lecture we will discuss the Different Methods to Detect Data Races in a Program.

So, if you would have looked at the previous lecture, you would have realized that a data race is an inherent property of a program regardless of the memory model. So, the aim is to find a data race in a program.

(Refer Slide Time: 01:18)

Data Race Detection Algorithm

? How do we detect data races?

- We need an **algorithm** that tests a piece of code for data races
- **Exercise** all possible control paths
- **Create** as many interleavings as possible
- A data race must show up in an SC execution
- Try to find it ...

static

dynamic

NPTEL

McGraw-Hill | Advanced Computer Architecture

149

So, the question is how do we detect data races. So, what we need is we need an algorithm that tests a piece of code for data races. So, this means that it needs to exercise all possible control paths give it all possible inputs, create as many inter leavings as possible and finally, what we have proven in the last chapter is that if there is a data race in a program it will definitely show up in an SC execution.

If it shows up in an SC execution, then this basically motivated us or gave us an algorithm or a method of how to create the inter leavings because you think about it if there is a complex memory model and we are creating inter leavings as per the memory model that is going to be a lot of work.

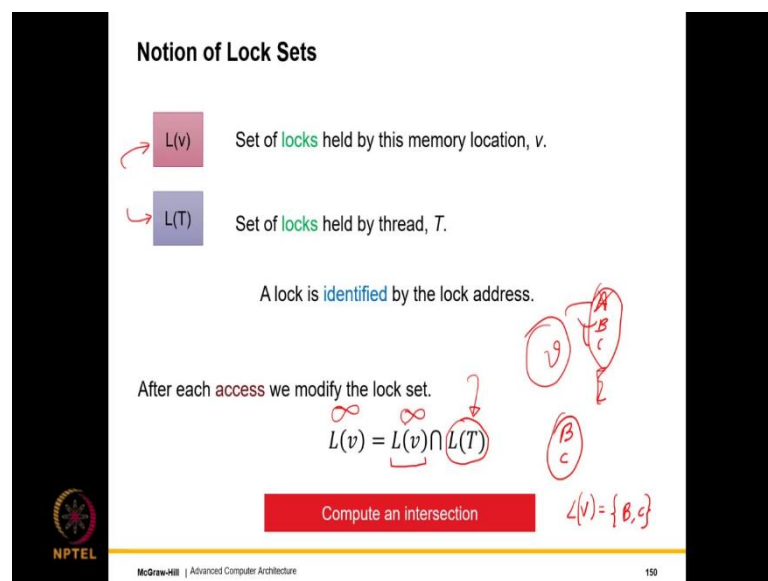
But, if you assume SC then creating inter leavings and different executions is far easier, but even with that we need to exercise as many paths as possible and conclusively prove that there is no data race and that is quite hard. I mean there are two ways of course, but

both the ways are reasonably hard we will discuss the easier one. So, one is the static approach where we analyze the code using a compiler and try to find out if there is a data race or not.

So, this is computationally hard. It is not easy to prove. So, what is computationally hard? Well, that is normally dealt with in a course and computational complexity but let us just say that within a polynomial number of tries we are not going to get the answer and in some of these problems are also so hard to solve that it is also possible that if we start solving it the algorithm may never terminate.

So, that is the reason what we do is we do a dynamic approach where we instrument the program; instrumenting basically means we just record which variables we are reading and writing to and try to see if there is a data race and say if you run a program millions of times if there is a data race there is a very high likelihood that it will show up.

(Refer Slide Time: 03:35)



So, I will discuss two algorithms one is the algorithm based on lock sets and the other is the algorithm based on vector clocks. So, at first I will discuss the algorithm based on lock sets. So, let us define these two terms here L_v and L_T . So, L_v is the set of locks held by a given memory location. So, this is something that our algorithm has to simulate and L_T is a set of logs held by a thread T . Again, an algorithm we will have to simulate this.

So, lock is identified by the lock address. In the sense that every lock has a unique lock address which as we have seen in the previous lectures we atomically set and unset. So, after each after each access memory location v , we modify the lock set. So, what we do is we compute an intersection say L_T of course, is a lock currently held by the by the thread T .

So, what we do is we compute L_v is equal to we set $L(v) = L(v) \cap L(T)$ which basically means that initially if let us say we assume that every location is associated with all possible locks. So, initially let us say the set of all possible locks is infinity. What will happen is let us say the first thread comes and it accesses this memory location. This is essentially being infinity intersection all the locks held by L_T which will automatically become L_T .

And, subsequently this set will get more and more and more refined which basically means that for every lock variable we want to find those lock addresses that you have to acquire those locks that you have to acquire to access this location over here. So, let us say that we have found out that may be locks A, B and C are what you need to acquire, but then let us say another thread comes by which has only acquired locks B and C.

Then when we compute an intersection between these two sets we realize that maybe lock A is not required because the second thread did not have lock A in its acquired lock set. So, let us remove this and let us set $L(v) = \{B, C\}$. So, what we do is that we continue simulating this algorithm. So, the key thing is that at the end of the program for every variable for every shared variable its lock set should not be empty. If its lock set is empty basically means that some thread has access this variable without actually acquiring any lock.

(Refer Slide Time: 06:30)

Notion of Lock Sets

$L(v)$ Set of **locks** held by this memory location, v .

$L(T)$ Set of **locks** held by thread, T .

A lock is **identified** by the lock address.

After each **access** we modify the lock set.

$$L(v) = L(v) \cap L(T)$$

Compute an intersection

$[] \cap [] = \phi$

NPTEL

McGraw-Hill | Advanced Computer Architecture

150

Or it is the case that there are two accesses to the same variable, but the intersection of the lock sets is null. So, this effectively means that the variable is not protected by the same set of locks.

(Refer Slide Time: 06:48)

Standard Approach

1. Add **annotations** to multithreaded code. These annotations **modify** the lock set.
2. The locksets of threads are **initially empty**.
3. For each variable, its lockset initially has **all the locks**.
4. When a thread **acquires** a lock, we **add** the lock to its lock set.
5. When it releases a lock, we **remove** the lock.
6. As the program executes, $L(v)$ keeps getting **updated**.
7. At the **end**, if there is a **variable** with an empty lock set, it is probably involved in a data race.

If the lock set is **empty**, it means that there is no **synchronization** between accesses to the variable.

NPTEL

McGraw-Hill | Advanced Computer Architecture

151


So, what is the standard approach? The standard approach for using the lock set algorithm is to add annotations to the multithreaded code. So, these annotations are the ones that help track the lock set in the sense the annotations are added to every read and write of a shared variable. And the moment we have a read and write we track the lock sets and furthermore

when we acquire a lock by a thread or we release a lock the lock sets of the threads are also tracked.

So, lock sets of threads are initially empty and the lock sets of every variable is initially infinite. So, for each variable it is lock set initially has all the locks. So, when the thread acquires a lock we add the lock to its lock set when it releases we remove. As the program executes for every variable v , L_v keeps getting updated at the end if there is a variable with an empty lock set may be it will involved in the data race. So, we will see that there could be many instances where actually it is not, but it is possibly involved in a data race.

So, basically this also means that if a lock set is empty it means that there is no synchronization between accesses to the variable, but as I said all such instances where the lock set is empty are not harmful there are some harmless ones as well. So, we will discuss that.

(Refer Slide Time: 08:12)



Notion of False Positives

Note that we will detect many scenarios that are actually not data races. For example, the basic algorithm will flag a data race when we consider read-only variables.

A few more examples

Example	Description
Initialization	We typically initialize variables without using locks.
Read-only variables	Written once (during initialization) and read many times
Reader-writer pattern	Multiple threads read a variable concurrently.

McGraw-Hill | Advanced Computer Architecture

152

So, there are of course, a notion of false positives in the sense we will detect many scenarios that are actually not data races. For example, the basic algorithm will flag a data race for read only variables. So, let us look at some of these patterns where you will see that a problem will be flagged.

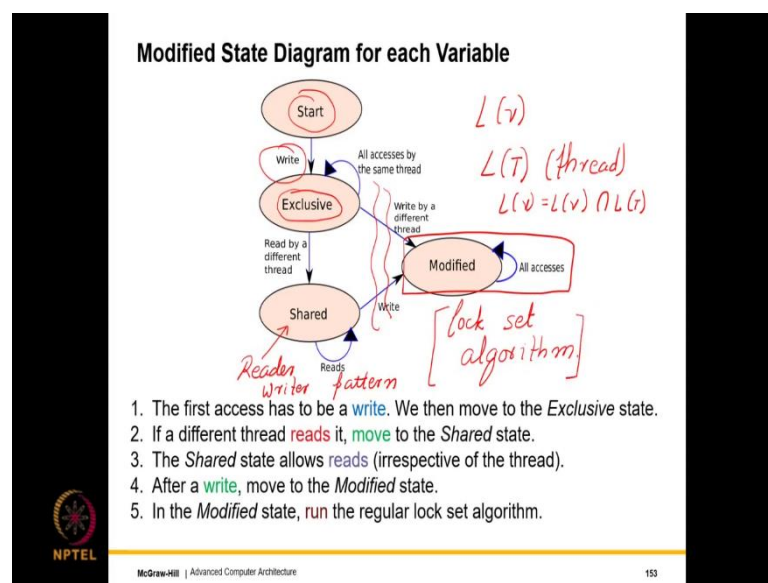
For example, when we are initializing a variable we will typically initialize variables without using locks. So, the first access some thread at the beginning will just initialize.

So, that may not use locks where we flag by a problem by our system which is not the case.

We will have some read only variables which are just written once during initialization and subsequently, they are only read. Again that will be flagged as a problem, but that is still not the case and a reader writer pattern remember multiple threads read variable correct concurrently. See even that will be flagged as an issue, but that is also not the case. So, basically once the reader writer pattern could be that a writer writes to a variable multiple readers keep reading to it that is not a problem.

So, to ensure that this read only variables which are constant the reader writer pattern or the initialization pattern, all these patterns are effectively captured we need to do something. So, here is what we do?

(Refer Slide Time: 09:48)



So, we have a modified state diagram for each variable. So, for every variable we have this state diagram or this FSL where the variable starts at the start state; the first access to the variable has to be a write because the first access you have to initialize the value of the variable. So, that has to be a write.

So, moment we have a write to the variable, the variable moves to exclusive state which means it is the exclusive property of one thread. In that state all the accesses by the same thread whether it is a read access or a write access. So, after a thread has made all the

accesses write or putting in another words if you are in the exclusive state regardless of the read or write by the same thread, we will still continue to remain here. So, there is no problem up till this points.

Now, if there is a read by a different thread then this could possibly signal the reader writer pattern. So, from here we will go to the shared state where shared states the connotation is similar to cache coherence where if there are reads, reads, reads and reads we will continue to be in the shared state. Say here again there is no problem because this shared state for us is the reader writer pattern where you write once then you keep on reading.

The problem arises when there is a write by a different thread which means that we are starting a regular shared access. So, there is a write by a different thread either when the variable is in the exclusive state which means that other threads also are showing an interest in the variable or from the shared state. So, when this happens we enter the modified state and in the modified state our regular lock set algorithm runs.

So, in this case, our regular lock set based algorithm that runs because if you think about it we have covered all the basic cases. So, the initialization state is exclusive. So, we do not detect any data locks here. So, go back to the three things that we said read only variables. So, read only variables and also reader writer patterns both are being captured by the shared state, but let us see if other threads are writing to the variable then of course, there is an issue.

And, in this case we enter the modified state and we remain there for every subsequent access, whether it is the read or write it does not matter we remain here in the modified state, for all subsequent accesses and then we run the regular lock set algorithm on the modified state.

(Refer Slide Time: 12:44)

Notion of Vector Clocks

- We can think of a multithreaded execution environment as a **classical** distributed system
- Here, there is no notion of **global time**.
- Every thread has a **local clock**.
- The **local clocks** are updated any time there is an **interaction** between threads.
- Assume there are n threads. Every thread maintains an n -element vector clock. Thread i 's vector clock is V_i .
- $V_i[j]$ is the best **estimate** of j 's local clock

Handwritten notes:

- n -element vector.
- $C_1 \dots C_n$ (circled)
- $T_1 \dots T_n$ (circled)
- $V_i[j]$
- $V_i[i] \leftarrow \text{accurate}$
- $i \neq j \quad V_i[i] \quad V_i[j]$
- vector clock

NPTEL
McGraw-Hill | Advanced Computer Architecture
154

So, this completed the lock set algorithm for us. So, so what are the key take away point? So, let me just go back once again. So, the key takeaway points are it is a simple algorithm where all that we do is we dynamically track $L(v)$ which is the vector of locks held by a variable or vector of locks required to access a variable and L_T which is the lock held by a thread.

Every time there is an access we just compute this, but as I said that would give us too many of these false positives. So, that is the reason what we do is that the first we try to ignore and then subsequently reads the other threads also we ignore. So, that filters out a lot of the false positive cases.

And, the moment there is a write to another thread it comes to the modified state and subsequently we run the lock set algorithm. So, in this case this prunes out a large number of false positives. So, there is incidentally a huge body of research or there is a great amount of literature that talks about different kinds of data races false positives false negatives and so on.

But, I will not get into that I will just tell you that the key idea of introducing the FSM was to prune out some of these false positives such that we can remain with a finite set of possible data races. The other idea is a direct import or inspiration from the world of distributed systems. So, here we can think of a classical multithreaded execution environment as a distributed system.


In a distributed system there is no notion of global time. Say, every thread has a local clock and there is as such no notion of a global time. So, what we have is we have a bunch of local times and whenever there is an interaction between threads the local clocks are updated and as I set a local there a certain rule for the local clock say let us say that there are n threads and each thread has its local clock. There is a certain rule of interaction and that binds this local clocks plus as such there is no global time disc.

So, assume there are n threads each thread what it maintains some n element vector is call as vector clock. So, each local clock in this case is not a scalar, but if there are n threads the each such local clock is actually an n element vector which is known as it is vector clock.

Let us say thread is vector clock is V_i let V_{ij} which is basically the j -th element of V_i . Let it be the best estimate of j 's local clock. So, let V_{ij} be the best estimate of j 's local clock. So, the key important point over here is that the element V_{ii} this is the only element which is accurate because this is i 's estimate of its own local clock which is bound to be the most accurate.

But, for any other element which is not equal to i such as V_{ij} this is essentially is estimate of j 's local lock which of course, may not be that up to date may not be that accurate, but that sudden when V_i is local clock and that is known as a vector clock. So, we will use this logic to detect data races.

(Refer Slide Time: 16:37)



Comparability of Clocks

- Two vector clocks are **equal** when

$$V_i = V_j \Leftrightarrow \forall k, V_i[k] = V_j[k]$$
- Two vector clocks are **totally ordered** when

$$V_i < V_j \Leftrightarrow (V_i \neq V_j) \wedge (\forall k, V_i[k] \leq V_j[k])$$

$$V_i \leq V_j \Leftrightarrow (V_i = V_j) \vee (V_i < V_j)$$
- Two vector clocks may not always be **comparable**

$V_i = \{3, 4, 5\}$


$V_j = \{4, 5, 6\}$

$V_i < V_j$

$V_i = \{3, 4, 5\}$

$V_j = \{2, 5, 6\}$

i



McGraw-Hill | Advanced Computer Architecture

155

So, what we can do is that we can define some more terminology and kind of define an algebra on vector clocks. So, two vector clocks are said to be equal when all their components are equal. So, we say that $V_i = V_j$ if and only if for all k which means for all elements $V_{i k} = V_{j k}$.

Two vector clocks are totally ordered. So, the precedence symbol over here indicates a total order where number one they are not equal and for all k for all k means for all elements $V_{i k} \leq V_{j k}$. So, let me explain this again $V_i < V_j$ or precedes V_j if and only if $V_i \neq V_j$ and for all elements $V_{i k} \leq V_{j k}$.

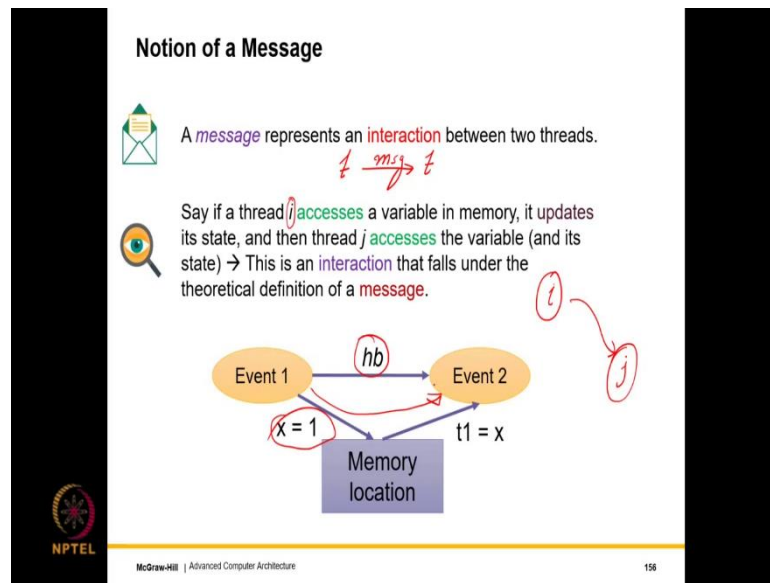
So, we can also have this kind of precedes or is the same or less than equal to. So, this basically means that either $V_i = V_j$ or put the or symbol V_i precedes V_j . So, two vector clocks may not always be comparable in the sense it is possible that I will give you two examples.

So, let us say that $V_i = 3, 4, 5$ and $V_j = 4, 5, 6$. In this case, you can confidently say that V_i precedes V_j . However, if let us say $V_i = 3, 4, 5$, $V_j = 2, 5, 6$. So, in this case, it the one thing is clear that for all element let us say first element $3 > 2$, but for the second element $4 < 5$. So, V_i and V_j in this case are not comparable. It is not the case that $V_i < V_j$ or is not the case that $V_j < V_i$.

So, in this case they are not comparable. So, as I said two vector clocks may not always be comparable, but if they are if they are comparable there are some interesting properties we will look into this, but this is the broad idea of the relationships between vector clocks.

So, as I said what again is a vector clock? Well, we can consider each thread process; process is the basic entity in a distributed system every process is independent. So, we can think every thread as a process even though the terminology is quite different a process in operating system means something else a process in a distributed system means something else, but in this case we are talking about processes in distributed systems.

(Refer Slide Time: 19:41)



So, given our definition on vector clocks let us also define the notion of a message. A message represents an interaction between two threads. So, an interaction in a classical distributed system would be that let us say one thread sends a message to another thread over a network that would be a classical interaction.


In the multicore system we do not have such kind of interactions, but we have another kind of interactions which are nonetheless interactions. So, say if a thread i accesses a variable in memory then it will basically update its state at least conceptually and then thread j accesses the variable and its state.

So, this is a valid interaction that falls under the theoretical definition of a message which means that if thread i writes to a variable and thread j reads it. So, even though it is not a direct message, but thread i wrote a value and thread j read it, so, that is an interaction.


So, we have already looked at this in previous lectures. It is called happens before relationship and such happens for relationships can easily be enforced or modified in the memory model where if we said $x = 1$ then another thread reads $t1 = x$ there is a happens before relationship between them subject to the constraints imposed by the memory model.

So, that does not matter, but because we are primarily looking at SC over here. So, once we have a happens before model that would technically count as a message.

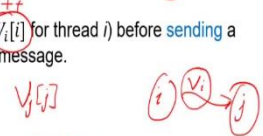
(Refer Slide Time: 21:30)




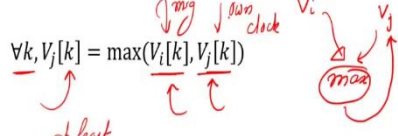
More about Events




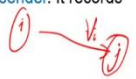
[Increment the local clock (e.g. $V_i[i]$ for thread i) before sending a message and after receiving a message.



 Let's say thread i sends a message to thread j . When j receives the message, it updates its clock as follows.



 The receiver is ^{at least} as up to date as the sender. It records an additional receive event.



McGraw-Hill | Advanced Computer Architecture

157

So, the moment we have such messages. so, let us say that it does not matter either indirectly through memory or directly via message right a message is sent from thread i to thread j . So, here is what we are going to do. So, what is the idea? The idea is a thread i is sending a message to thread j where a message is defined in a generic fashion in a generic sense as it was done in the on the previous slide.

So, what we do is we increment the local clock of i for example, we implement V_i i. so, whatever was its value we just do a $++$. So, this indicates that we are starting a new event. So, this is done any time before sending a message. So, before sending a message increment the lock and after receiving a message let us say j would also do that for V_j j.

So, this is basically indicating that any time there is either a message sends or a receive event we increment our local clocks. so, which is V_i i and V_j j respectively. Now, let us say the thread i sends a message to thread j when j receives the message along with incrementing its local clock which has already been mentioned over here it updates its vector clock as follows.

So, for all k V_j k is set as follows is the maximum of V_i k and V_j k. So, the maximum of these two quantities V_i k and V_j k. So, basically think of this that V_i is something that is being that is sent along with the message. V_j is the local clock, then we compute this max operation over here and the final result is used to increment V_j is used to replace V_j .

So, what is the key idea here? The key idea here is that if let us say thread i is sending a message to thread j , it is going to first increment its local clock before sending and then send the time stamp. Time stamp is the value of its vector clock along with the message. What V_j will do is that the moment it gets the message it will increment its local clock and along with that merge both the vector clocks, V_i which has come along with the message and V_j which is its own clock.

Merging would basically mean that element wise take the maximum. Why would you do that? The reason you would do that is that this would ensure that the receiver is at least as up to date as the sender. So, this should ensure that the receiver is at least. So, maybe I should add the term at least over here. The receiver is at least as up to date as the sender in the sense the receivers view is not older than the senders view because whatever the sender has seen write of the system that is the receiver at least has seen that much if not more and along with that it is also recording an additional received event.

So, this is the key idea of merging of vector clocks where the moment there is the message from i to j . What j will do is it will take V_i which is i time stamp and merge it with its own which is just compute element twice maximum and element twice maximum that would basically tell j that it has the most up to date view of the system. When we are concentrating only i and j as compared to i it has taken is updates and merged with its own.

(Refer Slide Time: 25:14)

Vector Clocks and Causality

Let's say that there is a **happens-before** relationship between events e_i and e_j . We use **vector clocks** to track the interaction. $V_i < V_j$

Event	Time
e_i	Event by thread i at time V_i
e_j	Event by thread j at time V_j

We have the following relationship

$$[V_i < V_j \Leftrightarrow e_i \xrightarrow{hb} e_j]$$

This happens because the **receiver** has all the **sender's** updates and without a **chain** of happens-before edges two vector clocks will not remain **comparable**.

$V_i < V_k < V_{k'} < V_j$

McGraw-Hill | Advanced Computer Architecture

158

So, what about vector clocks and causality? Well, we have been looking at causality in the previous lectures as well. Causality basically means an event i happened and that led to event j and event j led to event k . So, we have a (Refer Slide Time: 25:30) dependence. So, let us see that there is a happens before relationship between the events e_i and e_j .

We will use vector clocks to track this interaction. So, if we have event e_i and event e_j let the time of the event in thread i be time V_i and in thread j be time V_j . So, we will have the following relationship and this relationship is being shown without proof. So, it is basically saying that let there be for thread i and event e_i and for thread j let there be an event e_j which is happening respectively at times V_i and V_j .

If let us say V_i precedes V_j then we can say that there is a happens before relationship between the e_i and e_j . similarly, if there is a happens before relationship between e_i and e_j then $V_i < V_j$. So, one side is easy to prove, the other side is slightly harder say if there is a happens before relationship between e_i and e_j then it means that there will be several intermediate nodes and because happens before relationship is a transitive relationship.

So, there will be several intermediate nodes and for each one of them if let us say this is k and k' and so on, the moment i is sending a message to k we will have V_i precedes V_k . So, this is my definition this is basically the way as you can see that in this case the moment i is sending a message to j . so, this is automatically implying that after we are doing this max merging V_i is precedes V_j . So, this follows the simple logic of the max.

So, once we have this we have V_k then we have $V_{k'}$ so on and so forth and then V_j and since it is a transitive relationship we have V_i precedes V_j . It is also easy to prove in the reverse direction that a V_i precedes V_j that will not happen unless there is a chain of happens before dependencies and the chain of happens before dependencies will ensure the other direction will ensure this as well. So, as we have said we have proven the other upper direction first the lower one is also easy to prove.

So, why does this happen? This happens, because the receiver has all the sender's updates. And without a chain of happens for edges the two vector clocks would remain incomparable, but since they are since they are comparable a chain of happens before relationships would exist.

(Refer Slide Time: 28:34)

Vector Clocks and Causality

Let's say that there is a **happens-before** relationship between events e_i and e_j . We use **vector clocks** to track the **interaction**.

Event	Time
e_i	Event by thread i at time V_i
e_j	Event by thread j at time V_j

We have the following relationship

$$V_i < V_j \Leftrightarrow e_i \xrightarrow{hb} e_j$$

Vector clock

! This happens because the **receiver** has all the **sender's** updates and without a **chain** of happens-before edges two vector clocks will not remain **comparable**.

NPTEL | McGraw-Hill | Advanced Computer Architecture | 158

So, there is of course, there are many proves of this fact, but one thing that you need to kind of believe me is that if V_i precedes V_j there is a happens before relationship and vice versa as well. Now, this is a very famous and classical result of distributed systems and the vector clock is also known as Lamports vector clock.

(Refer Slide Time: 28:51)

Vector Clock based Algorithm

Symbol	Meaning
C_T	Vector clock of the <u>current thread</u>
C_L	Vector clock of the <u>current lock</u>
R_v	Read clock of variable v
W_v	Write clock of variable v
tid	Thread id

Handwritten notes: C_T and C_L are circled. A red arrow points from C_T to C_L with the word "max".

$C_T[tid] \leftarrow C_T[tid] + 1$
 $C_T \leftarrow C_T \cup C_L$ (*max*)
 $C_L \leftarrow C_T$
 $C_T.inLock \leftarrow \text{True}$

Handwritten notes: "Lock" with a blue lock icon and "Unlock" with a blue unlock icon.

Increment the vector clock of C_T and C_L

$C_T.inLock \leftarrow \text{False}$

NPTEL | McGraw-Hill | Advanced Computer Architecture | 159

So, let us now look at the vector clock based algorithm to detect data races. So, we will use all the concepts that we are studied in the previous slides. So, the terminology is like this. Let C_T be the vector clock of the current thread. this is quite similar to the lock set

algorithm. Let C_L be the vector clock of the current lock. So, in this case, we assign a vector lock a vector clock with the current thread and a vector clock with a current clock or the lock in consideration.

R_v is the read clock of a variable v and W_v is the write clock of variable. So, with every variable we have two clocks a read clock and a write clock and tid is a thread id. So, let us now look at the lock operation. So, in the lock operation here is what we do? So, what we do? here that the first thing that we do is that we increment the current time of the thread which is basically $C_T tid$. So, recall that C_T is the vector clock of the current thread.

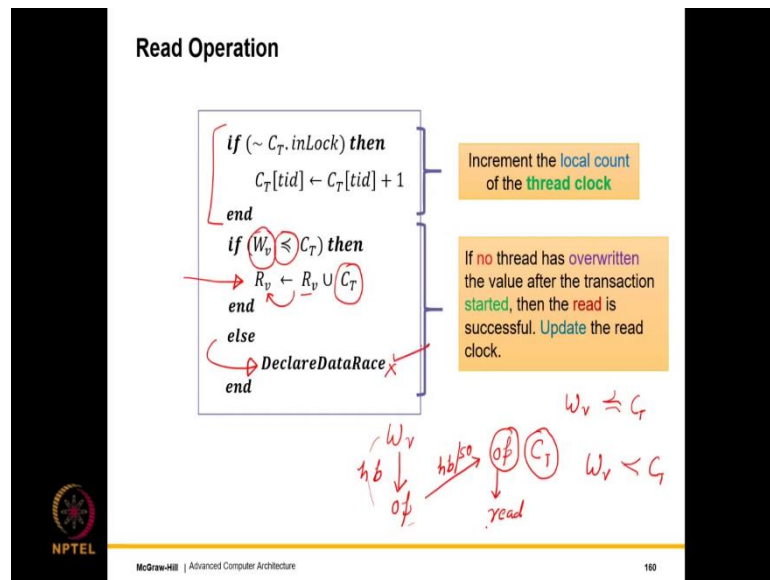
So, we go to it is tid -th element can we increment it which is quite similar to what we were doing when we were sending a message So, in this case we are doing something very very similar. After incrementing the current clock of the thread here is what we do. We do $C_T \cup C_L$. So, $C_T \cup C_L$ is basically nothing, but the max operation.

So, we perform the max operation with the vector clock of the lock that we are acquiring and this is set as the value of both C_T and as well as the value of both C_L . Say, in the sense that the current thread can have some clock. The lock can have some clock we compute the max of both which is nothing, but the union operation that I have shown. So, it is the thing of it is a modified union operation and then finally, the max operation is used to set the values of C_T and C_L . So, consider this is a rendezvous point which means that at this point the lock and the thread their information is up to date.

So, after doing this rendezvous we set the in lock field of C_T . So, in this case, the vector clock is an additional field as well for inLock which means that it is within a critical section. So, we set this to true. Exiting a lock or releasing a lock where exiting a particular section is quite easy; in this case we unlock. So, we set the value of inLock as false which basically means we unlock or we release the lock.

So, what is the most important point here? or the operative part is that we actually do a rendezvous between the thread and the lock which means that we perform a max operation and set them to do the same which essentially indicates to us that now, both the thread and the lock are at the same point, they are up to date.

(Refer Slide Time: 32:21)



So, now let us keep going forward let us look at the read operation. So, when we are reading something if we are not reading a variable within the scope of a lock say if not $C_T.inLock$ then what we do is that we increment the vector clock of C_T . So, in we set $C_T[tid]$ and $C_T[tid] + 1$; this is only in the case when this is not being done within the scope of a critical section otherwise there is no reason we will not enter this if statement.

Subsequently, we check at the we check the right clock of a variable. The right clock of a variable should precede or $= C_T$. Here is what this means if no thread has over written the value after the transaction started, then the read is successful. Let me explain this in a different manner.

So, let us say that there was a write to the variable and the program is properly synchronized which means that there was a happens before relationship to a synchronization operation. And, then there was an interaction between synchronization operations, again another happens before relationship and then at this point we are doing the read.

So, at this point. so, when we acquire the lock this time we set our thread clock and the lock and the clock of the lock to be both $= C_T$. So, because of the chain of happens before relationships on the property of the vector clock. If let us say there is no data race between them, then this chain of happens before relationships will hold which means that the right clock should either precede or $= C_T$.

So, we will discuss when it will be equal to. So, that comes later, but it should definitely precede. For this interaction this relationship should definitely hold. If this relationship is not holding, then this means from this relationship I will show you which one from this relationship.

So, if let us say W_v is not preceding C_T this means that between this write and this read which is essentially to the same variable there is no chain of happens before edges and if there is no chain of happens before edges then it is a data race. So, we directly come over here. So, we can declare a data race.

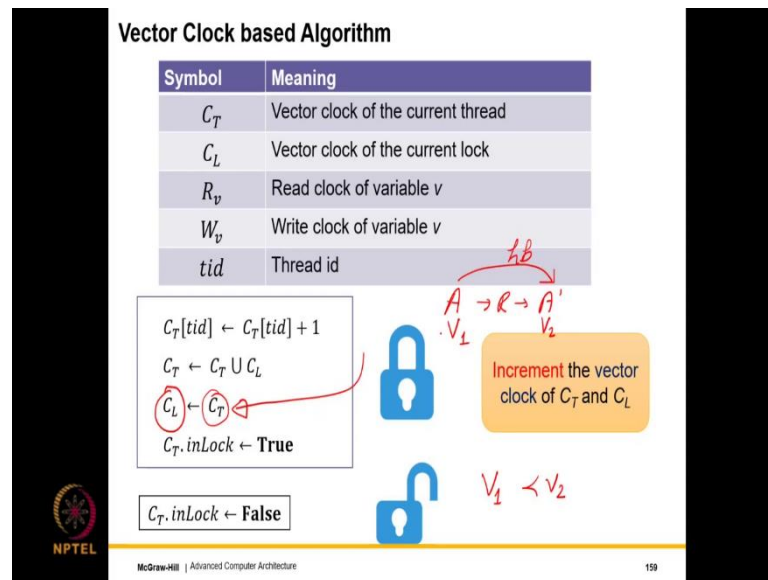
But, if there is a chain of happens before edges there by the property of vector clocks we will have W_v precede C_T there is a case where they will be equal, but I will discuss that later. So, let us assume that if it precedes or it is $= C_T$, then the access is safe, there is no problem.

So, the access is safe we enter the body of the loop. So, in this case, there is no data race. So, in the body of the loop what we do is that we time stamp the value of the variable with the current time of the thread which means that any subsequent access to the variable will definitely see the fact that the current thread had accessed the variable.

So, that is the reason that we compute $R_v \cup C_T$ which is something you always do in this case the union represents the max operation and then we set its value to R_v which ensures that forever this fact is being recorded that this particular access was made.

So, now, if we just come back we can put this discussion in the correct perspective and why we are time stamping the lock?

(Refer Slide Time: 36:25)



So, why we are time stamping the lock here with the time stamp of the thread the reason is that we are basically telling the lock and essentially we are telling the world that a certain thread acquired the lock at this point of time and at that point of time this was the value of the vector clock.

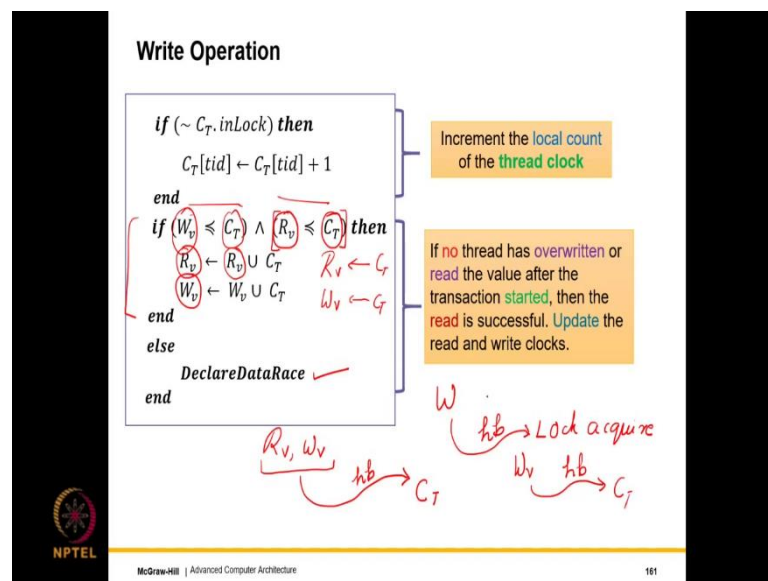
Such, that later on if let us say there is a lock acquire lock release and then the second lock acquire. Then the second lock acquire its vector clock will be strictly greater than the first vector clock. So, if this is V_1 and then this is V_2 , we can happily say that V_1 will precede V_2 .

The reason being that there is acquired there is a set of happens before edges and the set of happens before edges have been recorded by the fact that we are time stamping the lock access. The subsequent lock access the subsequent successful lock access will have the same max operation and that will essentially ensure that V_1 precedes V_2 which means that a happens before edge between lock accesses is automatically guaranteed.

And, if we look at again this relationship over here, you can see the relationship between the precedence of vector clocks and the happens before relationship. So, this is why we time stamp the locks, the acquirer of the lock such that any subsequent acquire will have a greater time stamp because it will have a greater time stamp we can automatically infer the happens before relationship.

Furthermore, what we do is we timestamp every read operation. So, let us say if tomorrow there is a write we can find if a read and write were causally related or not. Like in this case, if we find that the write and let us say that the time stamp of the thread which is when it acquired the lock, if they are not causally related which means that we do not have an interaction like this where there is a synchronization edge and then we read, then we can say that there is no causal relationship there is a data race and we can quickly declare a data race over here.

(Refer Slide Time: 38:34)



We can do something quite similar for writes. So, the idea is again the same if the write is not happening in the context of a critical section, then we increment the time of the that the local time of a thread by one we have already seen this, the new part is this. So, this is what looks at the causal relationship between the write clock of the variable and the current time of the thread.

So, as we have argued that any previous write to the variable has to either be equal to the current time of the thread or preceded, if that is not happening this means there is no happens before relationship between the earlier write and the lock acquire event of the current critical section.

And, given that there is no happens before relationship that is why this is not holding say if this is not holding we quickly declare a data race because it should hold which means that all preceding writes just go back to the lecture on data races all preceding writes have

to have a happens before relationship with the lock acquire operation that started this critical section and that will be captured by this relationship.

If this relationship does not hold then it automatically means that there is no happens before relationship. So, this means there is a data race. Similarly, if I look at the read clock say in the case of writes quickly care about earlier reads as well write. So, basically two reads their ordering does not matter but moment there is a write we care about the ordering of the write with earlier reads as well as earlier writes.

So, similar the read clock also should have happened before relationship with the current clock of the thread if when happens before relationship is not existing then there is a problem and we declare a data race, but if there is happens before relationship then what we do is that we time stamp the variable both of its clocks the read clock and the write clock with the current time and of course, the current time so, we do not discard any updates that are there.

So, essentially we take a union operation a max operation to $R_v \cup C_T$ and $W_v \cup C_T$ thinking. So, union this case is not regular union where it is the vector clock max operation that we have been describing. Say here if you look at this so, having $R_v \cup C_T$ is not technically required even though it has been written to make it look slightly elegant given that R_v precedes C_T anyway. $R_v \cup C_T = C_T$ because every element is less than equal to the corresponding element of C_T .

So, we would have very well written R_v is being set to the current time and W_v is being set to the current time that also would be correct. So, the important point is at any time when you write access the write clock is being set to the current time and subsequently if you do a read within the same thread then $W_v = C_T$ that is the reason here we have the preceding and equal to as well.

So, what is the key idea? Well, the key idea if I were to explain it in yet another different way is basically that wherever I am doing a read or write. So, let us say when I am doing a write I look at it is preceding read clock and write clock which could have been set in a different critical section, both of these clocks have to have a happens before relationship with the current time of the thread.

If they do not have a happens before relationship, then this indicates a data race and if they have one that will be easily captured with these two expressions and then what we do is we obtain the times of the read and write times. And, if we are reading, then we may not check the read clock, but we just check the previous write clock and that needs to have a happens before relationship with the current time.

And, that is easily captured by these expressions over here which let me erase the ink is easily captured by this expression. So, what happens is we can set $R_v = R_v \cup C_T$. So, actually the reason that I have kept the union operation over here and not set R_v as C_T , well, the reason is that in this case we are not really checking if R_v precedes C_T or not because it does not have to.

There is a read or write pattern that allows multiple concurrent readers and if I am reading I can continue to read there is no problem and so, that is the reason there is no need to check if R_v precedes C_T in this case, but we rather have this expression. So, to keep it consistent here also this expression has been used, but here of course, as I have said we could have directly set R_v and W_v in to C_T that also would have been correct.

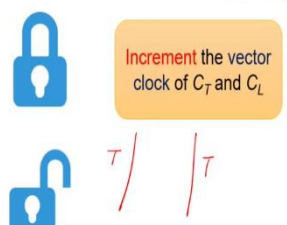
So, this is a fairly simple algorithm and the crux of the algorithm follows from this equation over here where we are linking the precedence of vector clocks to a chain of happens before edges. Say, if is understood the rest is quite easy to understand where you can see that what is being treated as a message over here ok let me maybe explain this slide in a slightly different manner.

(Refer Slide Time: 44:26)

Vector Clock based Algorithm

Symbol	Meaning
C_T	Vector clock of the current thread
C_L	Vector clock of the current lock
R_v	Read clock of variable v
W_v	Write clock of variable v
tid	Thread id

$$C_T[tid] \leftarrow C_T[tid] + 1$$
$$C_T \leftarrow C_T \cup C_L$$
$$C_L \leftarrow C_T$$
$$C_T.inLock \leftarrow \text{True}$$



Increment the vector clock of C_T and C_L

Handwritten notes: $V_1 \leftarrow C_T$, $V_2 \leftarrow C_L$, $V_1 < V_2$

NPTEL

McGraw-Hill | Advanced Computer Architecture

159

So, what is actually being treated as a message over here let us say between two threads is nothing, but the synchronization edge. So, what we are doing is that when we are acquiring a lock we are increasing our current counter. So, let us say there is a thread like this it comes and it acquires a lock over here.

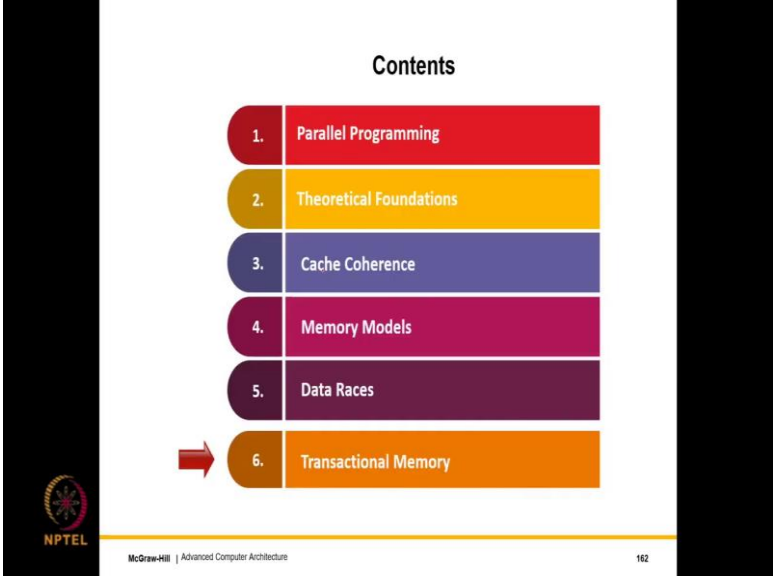
So, after this point let us say its counter is vector clock is C_T and the clocks locks vector clock is C_L . So, what is essentially happening over here is that. For both the lock and the thread it is a rendezvous point and if a thread is successfully acquiring the lock, you can think of this as a message. It is as if the thread and the lock are sending a message to each other.

So, there is a kind of a conversation between the thread and the lock, they are sending a message to each other and both are saying that we need to become equally as up to date. So, that is why they merge their information which is a union right and then both C_T and $C_L = C_T \cup C_L$. So, they come to the same point which means that any subsequent acquire of the lock. If let us say the vector clock of that is V_2 and vector clock of this one is V_1 that is a strict precedence relation, alright.

And, this is being used in both the read part and the write parts. So, we read a variable to only check the previous write clock on the previous read clock and the previous read clock is no doubt updated, but updated in this fashion and let us say for a write operation we

check both the previous locks and there has to be happens before relationship otherwise we declare a data race.

(Refer Slide Time: 46:19)



Contents	
1.	Parallel Programming
2.	Theoretical Foundations
3.	Cache Coherence
4.	Memory Models
5.	Data Races
6.	Transactional Memory

NPTEL

McGraw-Hill | Advanced Computer Architecture

162

So, basically we have discussed a very important component of data race detection. Now, what is left is basically transactional memory. Transactional memory is a new paradigm where it is much easier to actually write parallel programs.

So, we will discuss transactional memory next.